Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра математики

Магістерська робота

освітній рівень - магістр

на тему: «**Аналіз алгоритмів і реалізацій цифрового підпису на еліптичних кривих Едвардса**»

Виконав: студент 2-го року навчання,

Спеціальності
121 Інженерія програмного забезпечення

Калінічев Гліб Олександрович

Керівник: Олійник Б. В.,
доктор фіз.-мат наук, професор

Рецензент_____

(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«_____»_____20____р.

Київ - 2025

Тема: Аналіз алгоритмів і реалізацій цифрового підпису на еліптичних кривих Едвардса.

Календарний план виконання роботи:

№ п/п	Назва етапу кваліфікаційної роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на магістерську роботу	30.11.2024	
2.	Огляд літератури	15.01.2025	
3.	Аналіз існуючих реалізацій EdDSA в мові програмування Java	05.02.2025	
4.	Створення стандартних реалізацій EdDSA	01.03.2025	
5.	Створення оптимізованих реалізацій EdDSA	10.04.2025	
6.	Порівняльний аналіз реалізацій EdDSA	01.05.2025	
7.	Оформлення текстової частини роботи, створення презентації та попередній захист кваліфікаційної роботи	29.05.2025	
8.	Коригування роботи за результатами попереднього захисту	30.05.2025	

Студент: Калінічев Г. О.

Керівник: Олійник Б. В. “____”_____2025 р.

Зміст

Вступ

В сучасному світі обсяг інформації, яка зберігається та передається в електронному вигляді, вимірюється зетабайтами. Для розуміння масштабів, один зетабайт дорівнює одному мільярду терабайт. Велика частина цієї інформації є конфіденційною, тобто такою, що вимагає дотримання певних критеріїв безпеки. Серед найбільш поширених з них – гарантії цілісності та аутентифікованості походження даних. Наприклад, коли користувач здійснює грошовий переказ за допомогою банківського застосунку, банк, який обробляє цю операцію, повинен мати підтвердження того, що саме ця особа ініціювала фінансову транзакцію. Крім того, банку необхідно переконатися в тому, що дані переказу, зокрема його сума, не були змінені під час передачі від мобільного застосунку до фінансової установи. Саме цифровий підпис транзакції приватним ключем особи-платника є беззаперечним доказом виконання цих вимог.

Одним із найсучасніших способів створення цифрового підпису є EdDSA – алгоритм цифрового підпису на еліптичних кривих Едвардса, що був вперше запропонований у 2011 році. Він має ряд переваг у порівнянні зі старішими підходами для генерації цифрових підписів, таких як RSA та DSA: менший розмір ключів і підписів, швидше виконання операцій підпису та верифікації, більша стійкість до криптографічних атак. Крім того, EdDSA працює швидше за алгоритм ECDSA, який також використовує еліптичні криві. Завдяки цьому використання EdDSA стає більш поширеним у різних сферах: інтернеті речей, SSH та TLS протоколах тощо.

Існуючі реалізації алгоритму EdDSA в мові програмування Java зі стандартного пакету *java.security*, що є частиною JDK, та однієї з найбільш популярних криптографічних бібліотек Bouncy Castle створені на основі специфікації RFC-8032 [1] від 2017 року. Проте на сьогоднішній день існують способи оптимізації роботи EdDSA, які не описані в даній специфікації. Наприклад, автори D.J. Bernstein та T. Lange в статті [8] запропонували «швидкі» формули додавання та подвоєння точок еліптичної кривої. Автором статті [2] представлений

ефективний алгоритм скалярного множення точки еліптичної кривої. В статті [11] визначається спеціалізоване представлення елементів поля лишків. Тому існує теоретична можливість створити власну реалізацію EdDSA, яка б перевершувала стандартні за швидкістю виконання операцій.

Метою роботи є створення програмних реалізацій алгоритмів Ed25519 та Ed448, що є частковими випадками EdDSA, які б використовували оптимізовані швидкі алгоритми скалярного множення та додавання точок еліптичних кривих Едварса і швидких операцій в скінченному полі, запропонованих в [2], [8], [11]. Визначення переваг і недоліків власних реалізацій порівняно з відповідними стандартними реалізаціями.

Робота складається з чотирьох розділів:

Перший розділ містить теоретичний опис еліптичних кривих та решти супутніх математичних понять.

В другому розділі розглядається загальний алгоритм EdDSA і його часткові випадки: Ed25519 та Ed448.

Третій розділ присвячено технічному опису стандартних та власних реалізацій EdDSA.

В четвертому розділі міститься практичний порівняльний аналіз всіх розглянутих в роботі реалізацій EdDSA.

Постановка задачі:

1. Дослідити теоретичні способи оптимізації EdDSA.
2. Створити стандартну реалізацію Ed25519 та Ed448 відповідно до специфікації RFC-8032 [1].
3. Створити реалізації Ed25519 та Ed448, що імплементують досліджені оптимізації EdDSA.
4. Проаналізувати способи використання стандартних реалізацій EdDSA.
5. Виконати практичний порівняльний аналіз всіх розглянутих в роботі реалізацій EdDSA.

Розділ 1 Теоретичний опис еліптичних кривих

1.1. Базові поняття та визначення

Розгляд поняття еліптичної кривої над полем необхідно почати з визначення базових математичних понять. Спочатку треба дати формальне визначення поняття поля.

Поле – це множина елементів F, над якою визначені бінарні операції, які умовно можна назвати «додавання» (позначається «+») та «множення» (позначається «·»). Для спрощення подальших визначень, введемо допоміжне позначення «∘», яке позначає операцію додавання або множення:

$$\circ = + or \cdot$$

Операції додавання та множення мають задовільняти наступні умови:

1) Замкненість: $\forall a, b \in F: a \circ b = c;\ c \in F$;
2) Асоціативність: $\forall a, b, c \in F: (a \circ b) \circ c = a \circ (c \circ b)$;
3) Комутативність: $\forall a, b \in F: a \circ b = b \circ a$;
4) Існування нейтрального елементу відносно операції додавання (адитивний нейтральний елемент):
   $\exists\ 0 \in F: \forall a \in F, 0 + a = a$;
5) Існування оберненого елементу відносно операції додавання:
   $\forall a \in F\ \exists - a\ \in F: a + (-a) = 0$;
6) Існування нейтрального елементу відносно операції множення (мультиплікативний нейтральний елемент):
   $\exists\ 1 \in F: \forall a \in F, a \cdot\ 1 = a$;
7) Існування оберненого елементу відносно операції множення:
   $\forall a \in F \backslash \{0\}\ \exists\ a^{-1} \in F: a \cdot (\ a^{-1}) = 1$;
8) Дистрибутивність: $\forall a, b, c \in F: a \cdot (b + c) = a \cdot b + a \cdot c$;

Характеристика поля *F* (позначається *char(F)*) – найменше додатнє число *t*, таке що $t \cdot 1 = 0$, тобто сума *t* мультиплікативних нейтральних елементів дорівнює адитивному нейтральному елементу.

Група – множина *G* з бінарною операцією *: $G \times G \to G$, що задовільняє умови замкненості, асоціативності, існування нейтрального елементу відносно операції *, існування оберненого елементу відносно операції *. Позначається, як $(G,*)$.

Комутативна група (або абелева група) – група, для якої виконується умова комутативності.

Порядок групи (позначається $|G|$) – це кількість елементів в групі. Порядок групи може бути скінченний або нескінченний.

Циклічна група – група, в якій існує елемент g (генератор групи), піднесення якого до різних степенів дає решту елементів групи:

$$G = \{g^n : n \in \mathbb{Z}\} = \{\ldots, g^{-2}, g^{-1}, e, g, g^2, g^3, \ldots\}$$

Клас еквівалентності чисел за модулем *p* – множина цілих чисел, що дають однакову остачу від ділення на число *p*. Позначається, як $[a]$.

$$[a] = \{x \in \mathbb{Z} : x \equiv a \ (mod\ p)\}$$

Нехай *p* – просте число. Поле залишків $\mathbb{F}p$ – це множина класів еквівалентності цілих чисел за модулем *p* з визначеними операціями додавання та множення з модулем *p*. Оскільки *p* – просте число, то $\mathbb{F}p$ є полем.

$$\mathbb{F}p = \{[0], [1], [2], \ldots, [p-1]\}$$

Властивості $\mathbb{F}p$:

1) Поле $\mathbb{F}p$ є скінченим, кількість елементів поля дорівнює *p*: $|\mathbb{F}p| = p$;
2) $char(\mathbb{F}p) = p$;
3) Група не нульових елементів $\mathbb{F}p$ з операцією множення (мультиплікативна група $\mathbb{F}p* = \mathbb{F}p \setminus \{0\}$) є циклічною групою порядку p - 1.

4) Існування оберненого елемента: $\forall a \in \mathbb{F}p\ \exists\ a^{-1} \in \mathbb{F}p: a \cdot a^{-1} \equiv 1;$

Для подальшого розгляду операцій в групі точок еліптичної кривої необхідно розуміти, які є способи обчислення оберненого елементу в полі лишків $\mathbb{F}p$. Оскільки модуль поля p – просте число, то для обчислення a^{-1} можна застосувати Малу теорему Ферма (1.1):

$$a^{-1} \equiv\ a^{p-2}(mod\ p), \tag{1.1}$$

Для обчислення оберненого елементу в $\mathbb{F}p$ можна застосувати розширений алгоритм Евкліда, який буде розглянуто пізніше.

Афінний простір розмірності n над полем K – множина точок, що представлені за допомогою n координат.

$$\mathbb{A}^{\mathrm{n}}(K)\ =\ \{(x_1, x_2, \ldots, x\square) :\ x_{\mathrm{i}}\ \in\ K, \forall i \in \{1, 2, \ldots, n\}\}$$

Наприклад, звичайну координатну площину з віссю абсцис та віссю ординат можна подати у вигляді $\mathbb{A}^2(\mathbb{R}) = \{(x, y) : x, y \in \mathbb{R}\}$. Надалі ми будемо розглядати двовимірний афінний простір над полем лишків $\mathbb{F}p$: $\mathbb{A}^2(\ \mathbb{F}p)$.

Проективний простір розмірності n над полем K – множина прямих через початок координат в (n+1)-вимірному просторі. Тобто, якщо в двовимірному афінному просторі точка *(x, y)* – це певна позиція на площині, то у відповідному проективному просторі точка представляє собою пряму, що проходить через початок координат *(0, 0, 0)* та точку *(x, y, z)*. Формальне визначення проективного простору розмірності n наступне (1.2):

$$\mathbb{P}^{\mathrm{n}}(K) = \left\{[X^0 : X^1 : \ldots :\ X\square] : (X^0, X^1, \ldots, X\square) \in\ K^{\mathrm{n}+1} \setminus \{(0, 0, \ldots, 0)\}\right\}, \tag{1.2}$$

Тобто точка A в n-вимірному проективному просторі задається однорідними координатами $[X_0 :\ X_1 : \ldots :\ X\square]$, і ця точка еквівалентна точці B з координатами $[\lambda X_0 :\ \lambda X_1 : \ldots :\ \lambda X\square]$, де $\lambda \in K, \lambda \neq 0$.

Визначимо зв’язок між афінним та проективним координатними просторами. Існує відношення між точками афінного та проективного просторів, таке що:

$$A_1(x_1, x_2, \dots, x_{n-1}) \in \mathbb{A}^n \mapsto A_1'[1, x_1, x_2, \dots, x_{n-1}] \in \mathbb{P}^n.$$

Крім того, можливо обчислити координати точки в афінному просторі, маючи проективні координати. Розглянемо приклад для простору $\mathbb{A}^2$:

$$A_1(x_1, y_1) \in \mathbb{A}^2,\ A_1'[X_1, Y_1, Z_1] \in\ \mathbb{P}^2, x_1 = \frac{X_1}{Z_1}, y_1 = \frac{Y_1}{Z_1}$$

Ці рівності випливають з подібності прямокутних трикутників при геометричному представленні проективного простору. Проективний простір також містить «точки на нескінченності» виду $[0 :\ X_1 : \dots :\ X_n]$, які не мають відповідних точок в афінному просторі. Цікавою властивістю проективного простору є те, що дві паралельні прямі в цьому просторі перетинаються в «точці на нескінченності».

Алгебраїчна крива над полем *K* – множина розв'язків многочлена, залежного від двох змінних *x, y*, у афінному або проективному просторі. Під словом «розв'язки» маються на увазі пари значень змінних *(x, y)*, за яких значення многочлена дорівнює нулю.

Афінна алгебраїчна крива для многочлена $f(x, y), x, y\ \in K$ визначається, як (1.3):

$$C\ =\ \{(x, y) \in\ K^2 : f(x, y) =\ 0\}, \qquad (1.3)$$

Проективна алгебраїчна крива для однорідного многочлена (многочлена, що має рівні степені всіх одночленів) $F(X, Y, Z)$ визначається так (1.4):

$$C'\ =\ \{[X :\ Y :\ Z] \in\ \mathbb{P}^2(K) : F(X, Y, Z) =\ 0\}, \qquad (1.4)$$

Незвідність алгебраїчної кривої над полем *K*: алгебраїчна крива називається незвідною, якщо многочлен, що її визначає, не можливо розкласти на добуток не константних множників.

Для визначення поняття гладкості алгебраїчної кривої необхідно надати визначення градієнта функції в точці. Отже, градієнтом певної функції $f(x, y)$ є вектор часткових похідних, що визначається формулою (1.5):

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right), \tag{1.5}$$

В семантичному розумінні градієнт функції в точці вказує на напрям та швидкість зростання функції в точці. Розглядаючи поняття градієнту в контексті алгебраїчних кривих, можна сказати, що градієнт – це перпендикулярний (нормальний) вектор до лінії кривої в певній точці. Якщо говорити більш чітко, то градієнт – нормальний вектор до дотичної прямої кривої в певній точці.

Отже, алгебраїчна крива над полем називається гладкою, якщо в кожній її точці значення градієнту функції визначального многочлену кривої не рівне нулю.

Рід алгебраїчної кривої – певна числова характеристика кривої, яку семантично можна розуміти як «складність» форми кривої. Для гладких алгебраїчних кривих, що визначаються многочленом степеню *d*, рід *g* обчислюється за формулою роду-степеня (1.6):

$$g = \frac{(d-1)(d-2)}{2}, \tag{1.6}$$

Еліптичні криві, що використовуються в криптографічних системах, мають властивість гладкості та перший рід. Гладкість кривої гарантує застосованість операцій в групі точок до всіх елементів цієї групи без існування виключних випадків, що вимагали б додаткової обробки. Перший рід кривої дозволяє геометрично визначити операції в групі точок кривої. Крім того, група точок кривої першого роду є абелевою групою з операцією додавання. Операції в групі точок еліптичної кривої будуть описані пізніше.

1.2. Загальне поняття еліптичної кривої

Визначення 1. Еліптична крива *E* над полем *K* (позначається *E(K)*) – гладка проективна алгебраїчна крива першого роду, група точок якої містить спеціальну точку $O \in E(K)$, що називається «точкою на нескінченності» («infinity point»). Для того, щоб дати еквівалентне визначення еліптичної кривої

через загальну форму Вейєрштраса, необхідно описати поняття проективного замикання афінної кривої.

Отже, проективним замиканням афінної кривої називається проективна крива, множина точок якої є об'єднанням множини точок початкової афінної кривої з множиною «точок на нескінченності». Процес побудови проективного замикання афінної кривої можна описати декількома кроками: перший крок – гомогенізувати визначальне рівняння $f(x, y) = 0$ для отримання однорідного рівняння $F(X, Y, Z) = 0$ шляхом підстановки $x \to \frac{X}{Z}, y \to \frac{Y}{Z}$ і множення отриманого рівняння на відповідний степінь Z; другий крок – знаходження «точок на нескінченності» виду $[X : Y : 0]$, тобто розв'язків рівняння $F(X, Y, Z) = 0$ при $Z = 0$.

Визначення 2. Еліптична крива $E(K)$ – проективне замикання афінної алгебраїчної кривої, заданої рівнянням Вейєрштраса в загальній формі (1.7):

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1.7}$$

де $a_1, a_2, a_3, a_4, a_6 \in K$.

В проективних координатах існує єдина «точка на нескінченності» для еліптичних кривих, заданих формою Вейєрштраса - $O[0 : 1 : 0]$. Для полів, характеристика яких не рівна 2 або 3: $char(K) \neq 2, 3$, можна використати спрощену форму Вейєрштраса для визначення еліптичних кривих (1.8):

$$y^2 = x^3 + ax + b, \tag{1.8}$$

де $a, b \in K$.

Властивість гладкості еліптичної кривої можна перевірити, порівнявши значення визначника кривої (позначається Δ) з нулем. Еліптична крива є гладкою, коли $\Delta \neq 0$. Для еліптичних кривих, заданих спрощеною формою Вейєрштраса (1.8), визначник обраховується за формулою (1.9):

$$\Delta = -16(4a^3 + 27b^2), \tag{1.9}$$

В геометричному представленні еліптичні криві, задані у формі Вейєрштраса, є симетричними відносно горизонтальної осі абсцис.

1.3. Еліптичні криві над простим полем лишків

Нехай $\mathbb{F}p$ – поле лишків за модулем p, p – просте число. Еліптична крива $E(\mathbb{F}p)$ задається параметрами $a, b \in \mathbb{F}p$, які задовільнять умову (1.10), та є об'єднанням множини точок $(x, y) \in \mathbb{F}_P^2$, що є розв'язками рівняння (1.11), з «точкою на нескінченності» O. Множина точок еліптичної кривої визначається за формулою (1.12). [2]

$$4a^3 + 27b^2 \neq 0, \tag{1.10}$$

$$y^2 \equiv x^3 + ax + b\ (mod\ p), \tag{1.11}$$

$$E(\mathbb{F}p) = \{(x, y) \in \mathbb{F}p \times \mathbb{F}p : y^2 \equiv x^3 + ax + b\ (mod\ p)\} \cup \{O\}, \tag{1.12}$$

де $a, b \in \mathbb{F}p$.

1.4. Група точок еліптичної кривої

Група точок еліптичної кривої $E(\mathbb{F}p)$ – абелева група, що визначається множиною точок (1.12) з операцією $\oplus$, яку можна умовно назвати «додавання»: $(E(\mathbb{F}p),\ \oplus)$. Визначені наступні правила виконання операції $\oplus$ в групі точок еліптичної кривої $E(\mathbb{F}p)$:

1) Результатом додавання двох «точок на нескінченності» є «точка на нескінченності»: $O \oplus O = O$;
2) «Точка на нескінченності» є нейтральним елементом відносно $\oplus$: $(x, y) \oplus O = O \oplus (x, y) = (x, y), \forall (x, y) \in E(\mathbb{F}p)$;
3) Оберненим елементом до точки (x, y) відносно операції $\oplus$ є точка $(x, -y)$: $(x, y) \oplus (x, -y) = O, \forall (x, y), (x, -y) \in E(\mathbb{F}p)$;
4) Додавання двох різних точок кривої (x_1, y_1) та (x_2, y_2), $x_1 \neq x_2$, $(x_1, y_1) \oplus (x_2, y_2) = (x_3, -y_3)$, де:

$x_3 \equiv \lambda^2 - x_1 - x_2 \, (mod\ p)$,

$y_3 \equiv \lambda\,(x_1 - x_3) - y_1 \, (mod\ p)$,

$\lambda \equiv \frac{y_2 - y_1}{x_2 - x_1} \, (mod\ p)$.

5) Подвоєння точки (або додавання точки самої до себе). Нехай є точка кривої $(x_1, y_1) \in E(\mathbb{F}p), y_1 \neq 0$. Тоді результатом подвоєння токи (x_1, y_1) буде точка (x_3, y_3): $(x_1, y_1) \oplus (x_1, y_1) = (x_3, y_3) = 2(x_1, y_1)$, де:

$x_3 \equiv \lambda^2 - 2x_1 \, (mod\ p)$,

$y_3 \equiv \lambda\,(x_1 - x_3) - y_1 \, (mod\ p)$,

$\lambda \equiv \frac{3\,x_1^2 + a}{2y_1} \, (mod\ p)$.

6) Інвертованою точкою до точки кривої $P(x, y)$ є обернений елемент відносно додавання: $-P(x, -y)$. Тобто інверсія точки – це відображення точки відносно горизонтальної осі абсцис у афінних координатах.

Порядок групи точок еліптичної кривої (в літературі позначається $\#E(\mathbb{F}p)$, для зручності використання у формулах введемо позначення $Ord(E(\mathbb{F}p))$) – кількість точок на еліптичній кривій. Група точок еліптичної кривої над полем $\mathbb{F}p$ є скінченною.

Порядок групи точок еліптичної кривої можна оцінити, скориставшись теоремою Гассе про еліптичні криві (1.13):

$$p + 1 - 2\sqrt{p} \leq Ord(E(\mathbb{F}p)) \leq p + 1 + 2\sqrt{p}, \qquad (1.13)$$

В групі точок еліптичної кривої $E(\mathbb{F}p)$ додатково вводиться «допоміжна» операція множення точки на скаляр (scalar multiplication), яка визначається за формулою (1.14):

$$P(x, y) \in E(\mathbb{F}p), d \in \mathbb{Z}: dP = P \oplus P \oplus \ldots \oplus P \; (d\ times), \qquad (1.14)$$

Тобто, поняття множення точки на скаляр *d* еквівалентне поняттю додавання точки *d* раз. Множина точок $\{0, P, 2P, 3P, \ldots, (d-1)P\}$ є циклічною підгрупою

групи точок кривої з генератором *P*. В секції 1.6 будуть розглянуті деякі з алгоритмів множення точки кривої на скаляр.

Оскільки група точок еліптичної кривої є скінченною, то кожен елемент цієї групи має скінченний порядок. Це означає, що для кожної точки *P* еліптичної кривої існує певне найменше значення *n*, таке що результатом скалярного множення точки *P* на значення *n* буде нейтральний елемент групи, тобто «точка на нескінченності» *O*. Формально порядок точки еліптичної кривої *P* (позначається *ord(P)*) визначається так (1.15):

$$\forall P(x,y) \in E(\mathbb{F}p)\ \exists\ ord(P) = n\ \in\ \mathbb{Z}: nP = O, \tag{1.15}$$

За теоремою Лагранжа порядок точки групи ділить порядок групи точок (1.16):

$$\forall P(x,y) \in E(\mathbb{F}p)\ ord(P)\ \equiv 0\ \Big(mod\ Ord\big(E(\mathbb{F}p)\big)\Big), \tag{1.16}$$

1.5. Проблема дискретного логарифму на еліптичних кривих

Для того, щоб зрозуміти, чому еліптичні криві широко використовуються в сучасній криптографії, необхідно описати проблему дискретного логарифму, складність вирішення якої в обчислювальному сенсі і є гарантією захищеності криптографічних систем, побудованих на основі еліптичних кривих.

Отже, нехай є еліптична крива над полем лишків $E(\mathbb{F}p)$, є певна точка цієї кривої $P(x,y) \in E(\mathbb{F}p)$ з порядком *n*: $ord(P) = n$. Також є певне ціле число *d*: $0 \leq d < n,\ d\ \in\ \mathbb{Z}$. Є задача знаходження добутку точки *P* на скаляр *d*: $Q = dP$, яка вирішується відносно легко за допомогою алгоритму скалярного множення точки, наприклад *double-and-add* (буде розглянутий далі), що має логарифмічну складність. Ця задача називається прямою. Водночас, є обернена задача знаходження скаляра *d*, маючи точки *Q, P* в якості вхідних даних. Тобто *d* – це дискретний логарифм *Q* за основою *P*:

$$d = log_P(Q)$$

Геометрична структура групи точок еліптичної кривої не дає можливості напряму обчислити дискретний логарифм. Тривіальним способом обчислення дискретного логарифму є brute-force перебір елементів циклічної підгрупи $\langle P \rangle$, але він має складність *O(n)*, що фактично унеможливлює знаходження розв'язку для еліптичних кривих з великим порядком групи точок. Інші алгоритму знаходження дискретного логарифму (наприклад Baby-Step Giant-Step [3]) мають меншу складність $O(\sqrt{n})$, але все одно не є ефективними для сучасних криптографічних еліптичних кривих.

1.6. Опис алгоритмів скалярного множення точки еліптичної кривої

1.6.1. Стандартний алгоритм скалярного множення точки еліптичної кривої (double-and-add)

Існує багато підходів для обчислення добутку точки еліптичної кривої на скалярне значення $dP, P(x, y) \in E(\mathbb{F}p), d \in \mathbb{Z}$. Тривіальний та найбільш інтуїтивно зрозумілий підхід – застосувати операцію додавання точки до проміжного результату *d* - 1 раз. Попри свою простоту, цей підхід не можна назвати ефективним, адже він вимагає виконання $d - 1$ операцій додавання та має обчислювальну складність $O(d)$. Зазвичай, значення скалярів в алгоритмах цифрового підпису на еліптичних кривих є достатньо великими, що робить обрахунок скалярного добутку точки за допомогою цього методу занадто часозатратною операцією.

Очевидною є необхідність визначення алгоритму скалярного множення точок еліптичної кривої, обчислювальна складність якого є меншою за лінійну. Одним із найбільш відомих таких алгоритмів є алгоритм подвоєння та додавання (Double-and-Add) [4].

Представимо скаляр *k* у двійковому вигляді (1.17):

$$k = k_1 + k_2 \cdot 2^1 + k_3 \cdot 2^2 + \ldots + k_m \cdot 2^m, \quad (1.17)$$

де $k_i = \{0, 1\}, i = 1, .., m.$

Використовуючи властивість дистрибутивності скалярного множення відносно додавання, можемо записати добуток *kP* в такому вигляді (1.18):

$$kP = (k_1 + k_2 \cdot 2^1 + k_3 \cdot 2^2 + \ldots + k_m \cdot 2^m)P =$$

$$Pk_1 + Pk_2 \cdot 2^1 + \cdots + Pk_m \cdot 2^m, \quad (1.18)$$

Таким чином, ми розклали операцію скалярного множення точки в послідовність операцій додавання та подвоєння точки. Цей алгоритм є ітеративним і на кожній його ітерації відбувається подвоєння точки, а додавання – лише коли біт скаляра на певній ітерації дорівнює 1. Якщо число *k* має *m* біт, то *m* можна обрахувати за формулою (1.19):

$$m = \lfloor log^2(k) \rfloor + 1, \quad (1.19)$$

де позначення $\lfloor r \rfloor$ означає найбільше ціле число, що не перевищує *r*.

В цьому алгоритмі ми маємо *m - 1* операцій подвоєння, та максимум *m* операцій додавання (у випадку, коли всі біти числа є 1). В середньому, кількість одиничних бітів числа дорівнює половині всіх бітів, тому в загальному випадку маємо $\frac{m}{2}$ додавань. В найгіршому випадку алгоритм виконає $2\lfloor log^2(k) \rfloor + 1$ операцій подвоєння та додавання, що визначає його обчислювальну складність $O(log(k))$.

1.6.2. Window NAF метод скалярного множення точки еліптичної кривої.

Алгоритм скалярного множення точки з використанням віконної не суміжної форми (wNAF - Window Non-Adjacent Form), представлений в статті Хлєборобова [2], є дещо подібним до розглянутого double-and-add алгоритму в тому сенсі, що він також використовує певне представлення скаляру для ітеративного додавання та подвоєння точки кривої.

В даному алгоритмі скаляр $d > 0$ представлений у формі wNAF, яка визначена формулою (1.20):

$$d = \sum_{i=0}^{l-1} k_i 2^i , \qquad (1.20)$$

де l – довжина wNAF представлення, $|k_i| \leq 2^{w-1}$, $k_{n-1} \neq 0$.

Форма wNAF є узагальненням спеціальної знакової бінарної форми NAF для бітових вікон, шириною $w \geq 2$.

Для скаляру $d \in Z+\backslash\{0\}, d = \{d_{n-1}, \dots, d_0\}_2, d_i \in \{0,1\}, \forall i = 0, \dots, n-1,$ та ширини бітового вікна $w \in Z+\backslash\{0\}$ представлення wNAF (позначається NAF_w) має наступні властивості:

1) Кожне додатнє ціле число *d* має унікальне NAF_w представлення: $NAF_w(d) = \{k_{l-1}, \dots, k_0\}_{NAF_w}, |k_i| \leq 2^{w-1}, \forall i = 0, \dots, l-1;$
2) $NAF_w(d)$ з шириною вікна *w = 2* є $NAF(d)$ представленням: $NAF_2(d) = NAF(d)$. Тобто, NAF_w є узагальненням NAF;
3) Представлення $NAF(d)$ має найменшу кількість не нульових цифр серед усіх знакових представлень *d*;
4) Довжина $NAF(d)$ представлення може перевищувати довжину двійкового представлення максимум на 1 елемент: $l \leq n+1$;
5) В середньому кількість не нульових цифр $NAF_w(d)$ представлення приблизно дорівнює (*w+1*)-ій частині від загальної кількості цифр: $H(NAF_w(d)) \approx \frac{l}{w+1}$, де $H(r)$ – кількість не нульових цифр в представленні числа *r* (Хаммінгова вага).
6) Властивість не суміжності – серед будь-яких *w* послідовних цифр $NAF_w(d)$ представлення не більше однієї може бути не нульовою: $\forall k_i \in NAF_w(d), k_i \neq 0: k_{i+1} = k_{i+2} = \dots = k_{i+w-1} = 0;$

Властивість 5 є наслідком властивості 6. Форма NAF_w є зручною для алгоритмів скалярного множення точки кривої тим, що вона гарантує мінімальну кількість не нульових елементів. Водночас, кожен не нульовий елемент представлення скаляру вимагає виконання операції додавання точки. Тому кількість

дороговартісних операцій, які виконуються в алгоритмі, що базується на NAF_w, буде мінімальною.

Розглянемо алгоритм побудови форми NAF_w з цілого додатнього числа d (рис. 1.1).

Algorithm 1 Computing $\text{NAF}(d)_w$, $d \in \mathbb{Z}^+ \backslash \{0\}$.

Input - Integer $d \in \mathbb{Z}^+ \backslash \{0\}$.
- Window $w \in \mathbb{Z}^+ \backslash \{0\}, w \geq 2$.

Output - Representation of d: $\text{NAF}_w(d) = \{k_{l-1}, \ldots, k_0\}_{\text{NAF}_w}$.

begin
1: $l \leftarrow 0$;
2: **while** $d \geq 1$ **do**
3: **if** $d \mod 2 \neq 0$ **then**
4: $k_l \leftarrow 2 - (d \mod 2^w)$ (signed);
5: $d \leftarrow d - k_l$;
6: **else**
7: $k_l \leftarrow 0$;
8: **end if**
9: $d \leftarrow d/2$;
10: $l \leftarrow l + 1$;
11: **end while**
12: **return** $(k_{l-1}, \ldots, k_0)$**end**

Рисунок 1.1 – алгоритм побудови форми $NAF_w(d)$.

Опишемо даний алгоритм семантично. Ітеруємось по двійковому представленню числа від найстаршого біту до наймолодшого $i = n - 1, \ldots, 0$. Якщо і-тий біт числа є нульовим – додаємо 0 до результуючого списку. Інакше в змінну k_l записуємо значення останніх w бітів числа $\{d_i, \ldots, d_0\}_2$. Якщо значення $k_l > 2^{w-1}$, віднімаємо від k_l значення 2^w і результат знову записуємо в змінну k_l: $k_l = k_l - 2^w$. Це віднімання виконується для того, щоб нормалізувати значення k_l: $-2^{w-1} < k_l < 2^{w-1}$. Також це дозволяє мінімізувати абсолютне значення k_l. Потім k_l додається до результуючого списку. Віднімання $d = d - k_l$ виконується для обнуління останніх (найстарших) w бітів числа d, що гарантує його подільність на 2^{w-1} (це означає, що наступні w елементів NAF_w представлення будуть нулями). Поділ числа d на 2 означає бітовий зсув праворуч на 1 позицію, завдяки чому і відбувається бітова ітерація.

Всі елементи NAF_w є непарними числами. Обчислювальна складність цього алгоритму $O(n)$.

Алгоритм скалярного множення точки кривої з використання NAF_w представлення скаляру продемонстрований на рис. 1.2.

Algorithm 2 Point multiplication based on window NAF method.

Input - Elliptic curve $E(\mathbb{F}_p)$.
- Scalar d representation $\mathrm{NAF}_w(d) = \{k_{l-1}, \ldots, k_0\}_{\mathrm{NAF}_w}$, $d \in \mathbb{Z}^+ \backslash \{0\}$.
- Point $P \in E(\mathbb{F}_p)$ in $\mathcal{A}$ form.

Output - Result of scalar multiplication $dP \in E(\mathbb{F}_p)$ in $\mathcal{A}$ form.

begin

// Precomputations

1: $P_1 \leftarrow P$;
2: $P_2 \leftarrow$ **dbl**(P);
3: **foreach**$(i = 3, 5, \ldots, 2^{w-1} - 1)$ {
4: $P_i \leftarrow$ **add**(P_{i-2}, P_2);
5: }

// Main Computations

6: $Q \leftarrow P_{l-1}$;
7: $i \leftarrow n - 2$;
8: **while** $i \geq 0$ **do**
9: **if** $k_i = 1$ **then**
10: $Q \leftarrow$ **da**(Q, P_{k_i});
11: **else**
12: **if** $k_i = -1$ **then**
13: $Q \leftarrow$ **da**$(Q, \boxminus P_{k_i})$;
14: **else**
15: $Q \leftarrow$ **dbl**(Q);
16: **end if**
17: **end if**
18: $i \leftarrow i - 1$;
19: **end while**
20: **return** (Q)**end**

Рисунок 1.2 – алгоритм скалярного множення точки еліптичної кривої з використанням NAF_w представлення скаляру.

Даний алгоритм складається з двох фаз: попередні обчислення – перша фаза, та основні обчислення – друга фаза. В рамках першої фази нам необхідно обчислити і зберегти всі точки, кратні всім можливим значенням не нульових елементів NAF_w. Першим кроком подвоїмо початкову точку P та збережемо результат в змінну P_2, яка буде константним доданком. Після цього ітеруємось по всім можливим значенням елементів NAF_w: $i = 3, 5, 7, \ldots, 2^{w-1} - 1$, та обчислюємо точки P_i за формулою:

$$P_i = P_{i-1} \oplus P_2$$

Зрештою, отримаємо 2^{w-2} точок, виконавши 2^{w-2} операцій додавання і одну операцію подвоєння. Як можна помітити, обчислювальна та просторова складності фази 1 експоненційно залежать від параметру *w*, тому це значення не повинно бути занадто великим для ефективності обчислень.

Друга фаза алгоритму (main computations) безпосередньо використовує NAF_w представлення скаляру та список попередньо обчислених точок для отримання результату скалярного множення. Ініціалізуємо змінну результату Q точкою з найбільшим коефіцієнтом зі списку попередньо обчислених: $Q = P_{l-1}$. Далі ітеруємось справа - наліво по списку елементів NAF_w. Якщо елемент k_i додатній, то дістаємо точку з відповідним коефіцієнтом зі списку попередньо обчислених, і виконуємо операцію $da(Q,\ P_{k_i})$, що означає атомарну операцію подвоєння точки Q та додавання до результату точки P_{k_i}. Якщо ж елемент k_i є від'ємним, то зі списку попередньо обчислених дістається точка з коефіцієнтом $-k_i$, після чого обчислюється інвертована точка $-P_{k_i}$. Решта дій є аналогічною до випадку $k_i > 0$. В інших випадках (коли $k_i = 0$) виконуємо подвоєння точки Q. Асимптотична обчислювальна складність другої фази є лінійною відносно довжини представлення NAF_w: $O(l)$. При цьому виконується *l* подвоєнь, а кількість додавань приблизно дорівнює $H(NAF_w(d)) \approx \frac{l}{w+1}$, що є значно меншою кількістю, ніж $\frac{n}{2}$ в звичайному алгоритмі *double-and-add*. Крім того, даний алгоритм надає певну гнучкість, що зумовлена можливістю підбору значення *w* під конкретну бітову довжину скаляру. Варто пам'ятати, що збільшення значення *w* призводить до значного збільшення часових та просторових затрат на етапі попередніх обчислень, але при цьому значно зменшує часові затрати фази основних обчислень.

1.7. Еліптичні криві Едвардса над полем лишків

1.7.1. Базові поняття та визначення

[5] Повна крива Едвардса (complete Edwards curve): Нехай F_P– поле лишків за модулем p. Повна крива Едвардса над полем F_P (позначається $Ed(F_P)$) – еліптична крива (гладка алгебраїчна крива першого роду), що задається в афінних координатах рівнянням (1.21).

$$x^2 + y^2 \equiv 1 + dx^2y^2 \ (mod\ p), \tag{1.21}$$

де $x, y, d \in F_P, d \neq 0, d \neq 1$.

Основні властивості повних кривих Едвардса:

1) З геометричної точки зору криві Едвардса є симетричними відносно обох осей координат та діагональних прямих $y = \pm x$. Ця властивість пояснюється тим, що визначальне рівняння для повних кривих Едвардса (1.21) є квадратичним відносно обох змінних x та y.
2) Повні криві Едвардса не вимагають проективного замикання та є природно визначеними в афінних координатах. Нейтральним елементом в групі точок таких кривих є точка (0, 1) в афінних координатах, на відміну від кривих у формі Вейєрштраса, де нейтральний елемент – це «точка на нескінченності», що походить з проективного замикання.
3) «Повнота» формул для операції в групі точок. Це означає, що ці формули є коректно застосованими для всіх пар елементів групи точок без виключень. Через цю властивість криві Едвардса і називаються повними.
4) Повні криві Едвардса та криві у формі Вейєрштраса є біраціонально еквівалентними. Це означає, що існують раціональні відображення з множини кривих Едвардса в множину кривих у формі Вейєрштраса та у зворотному напрямі. Наприклад, точку кривої Едвардса можна відобразити в точку кривої у формі Вейєрштраса $(x, y) \mapsto (u, v),\ (x, y) \in Ed(F_P), (u, v) \in\ E(F_P)$ за допомогою формул (1.22) та (1.23):

$$u = \frac{1 + y}{1 - y}, \tag{1.22}$$

$$v = \frac{1+y}{x(1-y)}, \tag{1.23}$$

Якщо виразити змінні *y* та *x* через змінні *u* та *v* в рівняннях (1.22) та (1.23) відповідно, після чого результуючі рівності підставити у рівняння кривої Едвардса (1.21), то в результаті отримуємо рівняння у спрощеній формі Вейєрштраса (1.24):

$$v^2 = u^3 + au + b, \tag{1.24}$$

де параметри *a, b* виводяться з параметру *d* за формулами (1.25), (1.26).

$$a = \frac{-(d-1)^2(d+1)}{48}, \tag{1.25}$$

$$b = \frac{(d-1)^2(d^2+14d+1)}{864}, \tag{1.26}$$

Точка (0, 1) – нейтральний елемент групи точок повної кривої Едвардса – відображається в «точку на нескінченності» *O* кривої у формі Вейєрштраса.

Операція додавання $\oplus$, що визначена на множині точок в афінних координатах, які є розв'язками рівняння (1.21) над полем F_P утворює абелеву групу точок повної кривої Едвардса:

$$Ed(F_P) = \{(x,y) \in {F_P}^2 : x^2 + y^2 = 1 + dx^2y^2\}$$

Як вже зазначалося раніше, нейтральним елементом відносно операції додавання в групі $Ed(F_P)$ є точка з координатами (0, 1). Інверсія точки в $Ed(F_P)$ відбувається шляхом відображення точки відносно вертикальної осі ординат:

$$\forall P(x,y) \in Ed(F_P): -P(-x,y), \tag{1.27}$$

Операція додавання точок $P_1(x_1,y_1), P_2(x_2,y_2) \in Ed(F_P)$: $P_3(x_3,y_3) = P_1(x_1,y_1) \oplus P_2(x_2,y_2)$ виконується за формулами (1.27), (1.28).

$$x_3 = \frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \tag{1.28}$$

$$y_3 = \frac{y_1 y_2 - x_1 x_2}{1 - d x_1 x_2 y_1 y_2}, \tag{1.29}$$

Визначення операції множення точки на скаляр в групі $Ed(F_P)$ аналогічне до визначення цієї операції в групі $E(F_P)$.

[6] Скручена крива Едвардса (twisted Edwards curve) над полем F_P (позначається $Ed\_\{a, d\}(F_P)$) – еліптична крива, що задається рівнянням (1.30).

$$Ed\_\{a, d\}(F_P) = \{(x, y) \in F_P^2 : ax^2 + y^2 = 1 + dx^2 y^2\}, \tag{1.30}$$

де $x, y, a, d \in F_P, a \neq 0, d \neq 0, a \neq d$.

Як можна помітити, відмінність рівняння (1.29) від рівняння (1.21) полягає в присутності додаткового параметру a. Повні криві Едвардса є частковими випадком скручених кривих Едвардса з $a = 1$. Для повного розуміння взаємозв'язку повних та скручених кривих Едвардса необхідно визначити поняття квадратичного скручення кривої.

Нехай E – це еліптична крива над полем K, d – не нульовий елемент мультиплікативної групи поля K: $d \in K^*$. Квадратичним скрученням кривої E елементом d є нова крива E^d, отримана шляхом заміни однієї зі змінних визначального многочлена кривої E на добуток цієї змінної та $\sqrt{d}$. Наприклад, многочлен $F(x, y)$ визначає криву E. Многочлен $F\left(x, \sqrt{d} \cdot y\right)$ визначає нову криву E^d, яка є результатом квадратичного скручення E елементом d.

Отже, скручена крива Едвардса є результатом квадратичного скручення повної кривої Едвардса елементом a. В геометричному сенсі скручена крива Едвардса «розтягується» або «стискається» по осі абсцис (в залежності від значення параметру a) порівняно з повною кривою Едвардса, але при цьому зберігається симетричність відносно обох осей координат, а також діагональних прямих $y = \pm x$.

Операція додавання $\oplus$, що визначена на множині точок в афінних координатах, які є розв'язками рівняння (1.30) над полем F_P утворює абелеву групу точок скрученої кривої Едвардса (1.31):

$$Ed_{\{a,d\}(F_P)} = \{(x, y) \in F_P^2 : ax^2 + y^2 = 1 + dx^2y^2\}, \quad (1.31)$$

Порядок групи скрученої кривої Едвардса $Ed'(F_P)$ співвідноситься з порядком групи відповідної повної кривої Едвардса $Ed(F_P)$ за формулою (1.32).

$$Ord\big(Ed'(F_P)\big) + Ord\big(Ed(F_P)\big) = 2(p + 1), \quad (1.32)$$

Нейтральним елементом відносно операції додавання в групі точок скрученої кривої Едвардса є точка в афінних координатах (0, 1). Інверсія точки в групі скрученої кривої Едвардса відбувається за тим же правилом, що і в групі повної кривої Едвардса (1.27).

Формула обчислення координати *x* при додаванні пари точок в групі скрученої кривої Едвардса є ідентичною відповідній формулі додавання в групі точок повної кривої Едвардса (1.28). Формула обчислення координати *y* при додаванні пари елементів в групі точок скрученої кривої Едвардса містить параметр *a* (1.33).

$$y_3 = \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}, \quad (1.33)$$

1.7.2. Формули швидкого додавання та подвоєння точок скручених кривих Едвардса

В статті [7] описані формули швидкого додавання та подвоєння точок скручених кривих Едвардса у розширених проективних координатах. Для того, щоб детально розглянути ці формули, необхідно визначити проективне замикання скрученої кривої Едвардса.

Отже, як ми пам'ятаємо, точку в афінних координатах (x, y) можна представити в проективних координатах $(X : Y : Z)$ так, що $x = \frac{X}{Z}, y = \frac{Y}{Z}, Z \neq 0$. Провівши у рівнянні скрученої кривої Едвардса (1.30) заміну афінних координат на

проективні за вказаними раціональними відношеннями, отримуємо рівняння проективного замикання скрученої кривої Едвардса (1.34):

$$(aX^2 + Y^2)Z^2 = Z^4 + dX^2Y^2, \quad (1.34)$$

Нейтральним елементом в цьому проективному замиканні є точка в проективних координатах (0 : 1 : 1). Автори статті [7] використовують проективне замикання скрученої кривої Едвардса у своїй роботі.

Для зменшення кількості обчислень при додаванні точок була запропонована нова система координат $(X : Y : Z : T)$, що є розширенням координатою $T = \frac{XY}{Z}$ класичної проективної системи координат. Нова система отримала назву «розширені скручені координати Едвардса» («Extended Twisted Edwards Coordinates»). Після заміни афінних координат у формулах додавання (1.28), (1.33) на «розширені скручені координати Едвардса» автори статті отримують нові формули додавання (1.35)-(1.38) для $(X_1 : Y_1 : T_1 : Z_1) + (X_2 : Y_2 : T_2 : Z_2) = (X_3 : Y_3 : T_3 : Z_3)$:

$$X_3 = (X_1Y_1 + Y_1X_2)(Z_1Z_2 - dT_1T_2), \quad (1.35)$$

$$Y_3 = (Y_1Y_2 - aX_1X_2)(Z_1Z_2 + dT_1T_2), \quad (1.36)$$

$$T_3 = (Y_1Y_2 - aX_1X_2)(X_1Y_2 + Y_1X_2), \quad (1.37)$$

$$Z_3 = (Z_1Z_2 - dT_1T_2)(Z_1Z_2 + dT_1T_2), \quad (1.38)$$

Крім того, автори запропонували алгоритм для обрахунку $(X_3 : Y_3 : T_3 : Z_3)$, що використовує формули (1.35)-(1.38) та змінні для збереження проміжних результатів обчислень (рис.1.3).

$$A \leftarrow X_1 \cdot X_2, \quad B \leftarrow Y_1 \cdot Y_2, \quad C \leftarrow dT_1 \cdot T_2, \quad D \leftarrow Z_1 \cdot Z_2,$$
$$E \leftarrow (X_1 + Y_1) \cdot (X_2 + Y_2) - A - B, \quad F \leftarrow D - C, \quad G \leftarrow D + C,$$
$$H \leftarrow B - aA, \quad X_3 \leftarrow E \cdot F, \quad Y_3 \leftarrow G \cdot H, \quad T_3 \leftarrow E \cdot H, \quad Z_3 \leftarrow F \cdot G.$$

Рисунок 1.3 – алгоритм обрахунку суми точок в «розширених скручених координатах Едвардса».

Аналогічно, автори статті [7] виводять формули для подвоєння точки в «розширених скручених координатах Едвардса» (1.39)-(1.42) для $2(X_1 : Y_1 : T_1 : Z_1) = (X_3 : Y_3 : T_3 : Z_3)$:

$$X_3 = 2X_1Y_1(2Z_1^2 - Y_1^2 - aX_1^2), \quad (1.39)$$

$$Y_3 = (Y_1^2 + aX_1^2)(Y_1^2 - aX_1^2), \quad (1.40)$$

$$T_3 = 2X_1Y_1(Y_1^2 - aX_1^2), \quad (1.41)$$

$$Z_3 = (Y_1^2 + aX_1^2)(2Z_1^2 - Y_1^2 - aX_1^2), \quad (1.42)$$

На рис. 1.4 представлений алгоритм обрахунку координат подвоєння точки, що використовує формули (1.39)-(1.42) та змінні для збереження проміжних результатів обчислень:

$$A \leftarrow X_1^2, \quad B \leftarrow Y_1^2, \quad C \leftarrow 2Z_1^2, \quad D \leftarrow aA, \quad E \leftarrow (X_1 + Y_1)^2 - A - B,$$
$$G \leftarrow D + B, \quad F \leftarrow G - C, \quad H \leftarrow D - B, \quad X_3 \leftarrow E \cdot F, \quad Y_3 \leftarrow G \cdot H,$$
$$T_3 \leftarrow E \cdot H, \quad Z_3 \leftarrow F \cdot G.$$

Рисунок 1.4 – алгоритм обрахунку подвоєння точки в «розширених скручених координатах Едвардса».

Основною перевагою запропонованих формул додавання та подвоєння точок скручених кривих Едвардса є відсутність необхідності виконувати обчислення оберненого елемента поля, що є дуже дорогою операцією. Використання проективного замикання дозволяє замінити її операціями множення. Додаткова координата T зменшує кількість множень при додаванні точок. Так, обчислення добутку $x_1x_2y_1y_2$ потребує 3 операції множення. Водночас, цей же добуток можна обчислити, виконавши 1 операцію множення T_1T_2. Економія 2 операції множення для кожного додавання за рахунок додаткового використання пам’яті

для зберігання координати T. Крім того, запропоновані формули «швидкого» додавання та подвоєння є повними в групі точок скрученої кривої Едвардса.

1.7.3. Формули швидкого подвоєння та додавання точок повних кривих Едвардса

У статті [8] представлені ефективні формули подвоєння та додавання точок повних кривих Едвардса. У цих формулах також використовується проективне замикання повних кривих Едвардса для усунення необхідності виконувати пошук мультиплікативного оберненого елемента поля. Автори розглядають повні криві Едвардса, задані в узагальненій формі рівнянням (1.43).

$$x^2 + y^2 = c^2(1 + dx^2y^2), \tag{1.43}$$

Виконавши заміну в рівнянні (1.43) афінних координат проективними, автори отримують рівняння проективного замикання (1.44).

$$(X^2 + Y^2)Z^2 = c^2(Z^4 + dX^2Y^2), \tag{1.44}$$

Після цього автори проводять заміну афінних координат проективними у формулах додавання точок (1.28), (1.29) для отримання алгоритму (рис. 1.5), що обчислює проективні координати точки $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3 : Y_3 : Z_3)$.

$$A = Z_1 \cdot Z_2;\ B = A^2;\ C = X_1 \cdot X_2;\ D = Y_1 \cdot Y_2;\ E = d \cdot C \cdot D;$$
$$F = B - E;\ G = B + E;\ X_3 = A \cdot F \cdot ((X_1 + Y_1) \cdot (X_2 + Y_2) - C - D);$$
$$Y_3 = A \cdot G \cdot (D - C);\ Z_3 = c \cdot F \cdot G.$$

Рисунок 1.5 – алгоритм обрахунку суми точок повної кривої Едвардса в проективних координатах.

Аналогічно, автори визначають алгоритм подвоєння точки повної кривої Едвардса (рис.1.6).

$$B = (X_1 + Y_1)^2;\ C = X_1^2;\ D = Y_1^2;\ E = C + D;\ H = (c \cdot Z_1)^2;$$
$$J = E - 2H;\ X_3 = c \cdot (B - E) \cdot J;\ Y_3 = c \cdot E \cdot (C - D);\ Z_3 = E \cdot J.$$

Рисунок 1.6 – алгоритм обрахунку подвоєння точки повної кривої Едвардса в проективних координатах.

Основною перевагою зазначених алгоритмів обчислення суми точок та подвоєння точки є заміна двох операцій пошуку оберненого елементу в полі на операції додавання та множення елементів поля, що значно зменшило обчислювальну складність додавання та подвоєння в групі точок. Крім того, операції суми та подвоєння, реалізовані за допомогою алгоритмів (рис. 1.5) та (рис. 1.6) є повними в групі точок повних кривих Едвардса.

Розділ 2 Теоретичний опис алгоритмів цифрового підпису на кривих Едвардса

2.1. Опис загального алгоритму цифрового підпису на кривих Едвардса (EdDSA)

2.1.1. Параметри загального алгоритму EdDSA

Перед тим, як почати розгляд алгоритмів підпису та верифікації EdDSA необхідно описати всі вхідні параметри алгоритму, а також визначити формати представлення чисел та точок, розглянути алгоритми кодування та декодування точок еліптичної кривої. Стаття, в якій початково представлений EdDSA [9], визначає даний алгоритм над полями лишків, модуль яких ділиться на 4 з остачею 1: $F_q, q \ mod \ 4 \equiv 1$. Автори статті [10] узагальнюють алгоритм EdDSA для всіх полів лишків з простим модулем. Надалі будемо розглядати визначення вхідних параметрів EdDSA зі статті [10]. Отже, EdDSA має 11 вхідних параметрів:

1) Параметр p – просте число в непарному степені. Це число є модулем поля лишків F_p, над яким визначена певна еліптична крива Едвардса. Число p має бути достатньо великим, адже значення p обмежує значення іншого параметру l, який буде описаний нижче. Недостатньо велике значення p негативно впливає на безпеку алгоритму.

2) Параметр b – бітовий розмір ключів та підписів. Розмір публічних ключів дорівнює b, а розмір підписів - $2b$. Десяткове значення двійкового числа, яке має єдиний не нульовий біт на $(b-1)$-ій позиції, має бути строго більшим за обраний модуль поля лишків: $2^{b-1} > p$.
3) Двійкове кодування елементів поля F_p, що видає результат довжиною $(b-1)$ біт. В обох розглянутих далі варіантах EdDSA (Ed25519, Ed448) використовується *little-endian* кодування чисел (молодші біти числа розташовуються спочатку, а старші - вкінці).
4) Криптографічна хеш-функція H з довжиною хешу в $2b$ біт.
5) Параметр c – число 2 або 3. Числа, що є секретними скалярами в EdDSA, мають бути кратними 2^c.
6) Параметр n. Бітова довжина секретних скалярів EdDSA дорівнює $n+1$. Значення n має бути в діапазоні: $c \leq n \leq b$. Кожен секретний скаляр має мати найстарший біт (на позиції n) рівним 1 та перші (наймолодші) c біт нулями. Обране значення n має великий вплив на безпеку алгоритму, тому воно має бути достатньо великим.
7) Параметр a – не нульовий квадратичний елемент поля F_p: $a \in F_p, a \neq 0 : a^2 \in F_p$. Автори статті [10] надають рекомендовані значення a для кращої продуктивності алгоритму: якщо $p \bmod 4 \equiv 1$, то $a = -1$. Якщо $p \bmod 4 \equiv 3$, то $a = 1$. a є параметром скрученої кривої Едвардса в рівнянні (1.30).
8) Параметр d – не квадратичний елемент поля F_p. d є параметром кривої Едвардса (скрученої або повної) в рівняннях (1.21) або (1.30).
9) Параметр B – базова точка, яка належить групі кривої Едвардса (1.31) та не є нейтральним елементом групи $B \neq (0,1)$.
10) Параметр l – непарне просте число, яке є порядком базової точки B в групі кривої Едвардса: $lB = (0,1), 2^c l = Ord(E(F_p))$. Параметр l визначається обраною еліптичною кривою Едвардса.

11) Криптографічна хеш-функція H' для попереднього хешування вхідного повідомлення. В цій роботі розглядається лише «чисті» варіанти EdDSA (PureEdDSA), що не передбачають попереднього хешування повідомлення. Тому далі вважатиметься, що $H'(M) = M$.

2.1.2. Кодування та парсинг точок кривої в EdDSA

Для того, щоб розглянути кодування точок кривої, що використовується в EdDSA, необхідно описати кодування елементів поля F_p. Отже, вводиться поняття «негативного» елемента поля: елемент x є «негативним», якщо його $(b-1)$-бітне двійкове *little-endian* представлення є лексикографічно більшим за $(b-1)$-бітне *little-endian* представлення елементу $-x$. Для поля лишків з простим модулем всі його непарні елементи є «негативними».

Кожна точка еліптичної кривої $(x, y) \in E(F_p)$ кодується у b-бітний рядок. Перші $b-1$ біт цього рядка є кодуванням координати $y \in F_p$, а останній біт є маркером знаку координати x (знаковий біт). Знаковий біт є 1 тоді і тільки тоді, коли координата x є «негативним» елементом поля.

Операція парсингу точки з b-бітного рядка одночасно виконує відновлення координати x та перевірку належності отриманої точки (x, y) до групи точок кривої. Дана операція виконується в декілька кроків:

1) Декодувати координату y з перших $b-1$ біт;
2) Обчислити елемент поля xx за формулою (2.1).

$$xx = \frac{y^2 - 1}{dy^2 - a}, \tag{2.1}$$

3) Обчислити координату x як квадратний корінь з xx: $x = \pm\sqrt{xx}$. Знак x визначається знаковим бітом на позиції b. Якщо елемент поля xx не є квадратичний елементом – процес парсингу точки невдалий.
4) *Автори статті [10] пропонують оптимізацію для відновлення координати x для полів з модулем $p \bmod 4 \equiv 3$: x обчислюється за формулою (2.2):

$$x = \pm(xx)^{\frac{p+1}{4}}, \quad (2.2)$$

Після чого виконується перевірка: $x^2 = xx$. Якщо рівність не справджується – процес парсингу точки невдалий. Цей спосіб заміняє «дорогу» операцію взяття квадратного кореня більш «дешевою» операцією піднесення до степеню $\frac{p+1}{4}$ (яку можна ефективно імплементувати за допомогою ланцюга множень та піднесень до квадрату, про це детально в розділі 4) та операцією піднесення до квадрату.

2.1.3. Секретні та публічні ключі

Секретний ключ в EdDSA – b-бітовий рядок k. Хеш $H(k) = (h_0, h_1, \dots, h_{2b-1})$ визначає число $s = 2^n + \sum_{c \le i < n} 2^i h_i$, що є секретним скаляром. Секретний скаляр використовується для обчислення точки A, яка є публічним ключем в EdDSA: $A = sB, A \in E(F_p)$.

2.1.4. Операція підпису

Операція підпису повідомлення M секретним ключем k виконується в кілька кроків:

1) Отримується хеш другої половини хешу секретного ключа k та повідомлення M: $H_2 = H(h_b, h_{b+1}, \dots, h_{2b-1}, M)$.
2) Отримується число r в результаті *little-endian* інтерпретації $2b$-бітного рядка H_2: $r = interpret(H2), r \in \{0, 1, \dots, 2^{2^b} - 1\}$.
3) Обчислюється точка $R = rB, R \in E(F_p)$.
4) Обчислюється число $S = \left(r + interpret\left(H(R, A, M)\right)s\right) mod\ l$.
5) Підписом повідомлення M секретним ключем k є $2b$-бітний рядок $encode(R)||little\_endian(S)$, де $encode(R)$ – закодована точка R, $little\_endian(S)$ – *little-endian* представлення числа S, $||$ - конкатенація бітових рядків.

2.1.5. Операція верифікації

Операція верифікації підпису повідомлення M публічним ключем pk виконується в кілька кроків:

1) Відбувається парсинг публічного ключа в точку $A = parse(pk), A \in E(F_p)$.
2) В результаті парсингу перших b біт підпису отримується точка $R \in E(F_p)$.
3) В результаті *little-endian* інтерпретації останніх b біт підпису отримується число $S \in \{0, 1, \dots, l-1\}$.
4) Отримується хеш $H_1 = H(R, A, M)$.
5) Виконується перевірка рівняння групи точок кривої $E(F_p)$: $2^c SB = 2^c R + H_1 A$. Якщо рівність не справджується або на якомусь з етапів операція парсингу точки виконується не успішно, то верифікація підпису повідомлення M публічним ключем pk є неуспішною.

2.2. Частковий випадок EdDSA - Ed25519

2.2.1. Визначення Ed25519

Ed25519 є частковим випадком загального алгоритму EdDSA. Можна сказати, що Ed25519 – певний набір значень вхідних параметрів для EdDSA. Тобто це не окремий алгоритм, а одна з інстантинацій (instantination) EdDSA. Розглянемо значення параметрів, що надаються в Ed25519.

Найважливішим параметром Ed25519 є скручена еліптична крива Едвардса, що має назву Edwards25519. Використання цієї кривої в EdDSA пропонується авторами статті [9]. Крива Edwards25519 визначена над полем лишків F_p, з модулем $p = 2^{255} - 19$ і є бірaціонально еквівалентною кривій Curve25519, яка запропонована в статті [11], та задається рівнянням у формі Монгомері:

$$v^2 = u^3 + 486662u^2 + u, \tag{2.3}$$

де $u, v \in F_p$.

У статті [8] вводиться та доводиться теорема 2.1, яка говорить про те, що будь-яка еліптична крива, група точок якої має елемент порядку 4, є біраціонально еквівалентною деякому квадратичному скрученню еліптичної кривої Едвардса. Тому для кривої Curve25519 існує біраціонально еквівалентна скручена крива Едвардса, що здається рівнянням (2.4). Ця крива і є кривою Edwards25519.

$$x^2 + y^2 = 1 + \left(\frac{121665}{121666}\right)x^2y^2, \tag{2.4}$$

де $x, y \in F_p$.

Точка кривої Curve25519 (u, v) відображається в точку кривої Edwards25519 (x, y) за допомогою раціональних перетворень (2.5) та (2.6):

$$x = \frac{\sqrt{486664}u}{v}, \tag{2.5}$$

$$y = \frac{u - 1}{u + 1}, \tag{2.6}$$

Крива Curve25519 забезпечує 128-бітний рівень безпеки. Це означає, що успішне виконання атаки на дану криву потребує в середньому 2^{128} операцій. Далі наведені значення решти параметрів для EdDSA, що наводяться в статті [10] для Ed25519: $b = 256$; кодування – 255-бітне *little-endian* кодування елементів поля F_p та цілих скалярів; $l = 2^{252} +$ $27742317777372353535851937790883648493$; $d = -\frac{121665}{121666}, d \in F_p$; $B(x_b, y_b) \in E(F_p), x_b =$ 15112221349535400772501151409588531511454012693041857206046113283949847762202, $y_b =$ 46316835694926478169428394003475163141307993866256225615783033603165251855960. Значення координат точки B наведені у специфікації Ed25519 від IRTF - RFC-8032 [1]; $H -$ хеш $-$ функція $SHA - 512$.

2.2.2 Спеціалізована арифметика в полі F25519

Задля досягнення високої швидкості роботи алгоритму цифрового підпису необхідно імплементувати ефективні обчислення низького рівня, а саме арифметичні операції над елементами поля лишків $F_p, p = 2^{255} - 19$ (в літературі позначається F25519). В секції 4 статті [11] розглядається проблема імплементації арифметики в полі F25519 на рівні процесора. Оскільки Ed25519 використовує 255-бітне кодування елементів поля, необхідно визначити спеціалізований формат для представлення елементів поля в регістрах процесора, адже 255-бітне число не може вміститися в одне машинне слово. Був запропонований формат limbs-10 у системі числення з основою $2^{25.5}$ (radix $2^{25.5}$). Його суть полягає в розділенні 255-бітного числа на десять 25 або 26-бітних чисел, що мають назву limbs. В середньому, кожен limb має $\frac{255}{10} = 25.5$ біт. В реальності половина limbs має 25 біт, інша половина – 26. В сумі отримуємо 255 біт: $5 \cdot 25 + 5 \cdot 26 = 255$. Це і пояснює, чому така система числення має основу 25.5. Для того, щоб перейти до розгляду математичної основи даного формату, потрібно визначити поняття кільця поліномів. Отже, множина R з визначеними в ній операціями $(+,\ \cdot)$ називається кільцем, якщо $(R, +)$ – абелева група та $(R,\ \cdot)$ – моноїд (виконується умова асоціативності для операції $\cdot$). Також мають виконуватися умови дистрибутивності між операціями $(+,\ \cdot)$ (лівий та правий розподіл):

$$\forall a, b, c \in R: a \cdot (b + c) = a \cdot b + a \cdot c,$$

$$\forall a, b, c \in R: (a + b) \cdot c = a \cdot c + b \cdot c.$$

Кожен елемент поля F25519 представлений поліномом з кільця поліномів R (2.7).

$$R = \{\sum_{i=0}^{9} u_i x^i \mid \forall i = 0, \dots, 9: u_i \in 2\text{^}\lceil 25.5i \rceil \mathbb{Z}\}, \#(2.7)$$

Кожен коефіцієнт u_i полінома з кільця R є цілим числом, кратним $2\text{^}\lceil 25.5i \rceil$. Нотація $\lceil x \rceil$ позначає найменше ціле число, що ж більшим за x (ceiling function).

На практиці елементи поля F25519 представлені значеннями поліномів при $x = 1$, тобто просто сумою десяти коефіцієнтів: $\sum_{i=0}^{9} u_i$. Спробуємо зрозуміти limbs-10 формат семантично. 255-бітне число «розділяється» на 10 бітових діапазонів (limb-ів). Наприклад, перші 26 біт числа – нульовий діапазон, наступні 25 біт – перший діапазон, і так далі. Не існує єдиної математичної формули, що визначала б довжину і-того діапазону. Бітова довжина кожного з 10 limb-ів визначена емпірично. Нехай a_i – це значення і-того бітового діапазону. Тоді для елемента поля F25519 $x \in F_p$ формула конвертації з limbs-10 формату матиме наступний вигляд (2.8):

$$x = \sum_{i=0}^{9} a_i 2^{\lceil 25.5i \rceil} \tag{2.8}$$

У кожного бітового діапазону є певна «вага» $2^{\lceil 25.5i \rceil}$, на яку множиться фактичне значення цього діапазону a_i. В двійковій інтерпретації дане множення – це бітовий зсув ліворуч на $\lceil 25.5i \rceil$ позицій. Це «переміщає» кожні 25 або 26 біт на потрібне місце в 255-бітній двійковій стрічці, відновлюючи таким чином початкове 255-бітне число. Кожне значення a_i є нормалізованим та належить діапазону $[-2^{25}, 2^{25}]$.

Множення двох елементів поля F25519 у форматі limbs-10 відбувається в декілька кроків. Перший крок – поліноміальне множення. Нехай $u(x), v(x) \in R$ – поліноми 9-го степеню. Тоді поліном $w(x) = u(x) \cdot v(x)$ має степінь 18. Хоча $w(x)$ і належить кільцю R (що гарантується властивістю замкненості кільця відносно множення), але цей поліном не відповідає формату limbs-10, який вимагає наявності максимум 10 коефіцієнтів у поліномі. Тому необхідно перетворити поліном $w(x)$ у поліном 9-го степеню без втрати інформації. Для цього виконується другий крок – модульна редукція. Останні вісім коефіцієнтів полінома $w(x)$ (з індексами $t \geq 10$) мають вагу $2^{255} \cdot 2^{\lceil 25.5(t-10) \rceil}$. Використовуючи властивість поля F25519: $2^{255} \equiv 19 (mod\ 2^{255} - 19)$, ваги коефіцієнтів полінома $w(x)$ з індексами $t \geq 10$ можна записати так: $19 \cdot$

$2^{\lceil 25.5(t-10)\rceil}$. Це дозволяє нам зробити наступні дії: коефіцієнти $w_t, t \geq 10$ множимо на 19 та додаємо їхні нові значення до значень коефіцієнтів w_{t-10}, після чого позбуваємося коефіцієнтів w_t (обнуляємо їхні значення). Таким чином, відбувається зменшення степеню полінома $w(x)$ з 18 до 9. Третім кроком необхідно провести нормалізацію значень коефіцієнтів полінома $w(x)$. Цей процес називається перенесенням (carrying). Пояснимо даний процес концептуально. Отже, кожен бітовий діапазон (limb) має визначену бітову довжину. Внаслідок модульної редукції може статися перевищення допустимої бітової довжини певного limb (занадто велике за модулем число). Щоб виправити таку ситуацію, необхідно «високу» частину певного limb перенести до наступного limb. Припустимо, нульовий limb фактично містить 30-бітне число, хоча його допустима довжина – 26 біт. «Високою» частиною цього limb є найстарші 4 біти (від 26-го до 29). Зберігаємо значення цих 4 біт в змінну $carry$: $carry = limb_0 \gg 26$, де операція $\gg$ позначає правий бітовий зсув на 26 позицій. Після цього обнуляємо найстарші 4 біти нульового limb, а значення $carry$ додаємо до наступного limb. Значення коефіцієнтів нормалізуються в допустимих межах $[-2^{25}, 2^{25}]$ шляхом перенесення $carry$ по ланцюгу від нульового limb до дев'ятого. В статті [11] автори вказують на необхідність виконання перенесень $w_8 \rightarrow w_9 \rightarrow w_{10}$ перед виконанням модульної редукції. Це робиться для запобігання переповненню регістра процесора, в якому зберігається w_8, в процесі модульної редукції.

Додавання двох елементів поля F25519 у форматі limbs-10 є більш простою операцією, ніж множення. Сумою двох поліномів 9-го степеню $u(x), v(x) \in R$ є поліном 9-го степеню $r(x) = u(x) + v(x)$. Поліном $r(x)$ є редукованим за модулем, але має не нормалізовані значення коефіцієнтів. Автори статті [11] зазначають, що нормалізація коефіцієнтів суми поліномів не є необхідною, адже в EdDSA будь-яка сума двох елементів поля F25519 використовується лише для подальшого множення. Саме тому нормалізація значень коефіцієнтів так чи інакше відбудеться в рамках операції множення.

2.3. Частковий випадок EdDSA – Ed448

Формальне визначення Ed448 аналогічне до визначення Ed25519. Ed448 – це також інстантинація загального алгоритму EdDSA певним набором значень вхідних параметрів. Ed448 використовує повну еліптичну криву Едвардса під назвою Edwards448-Goldilocks. Ця крива запропонована Майком Гамбургом в статті [12]. Edwards448 визначена над полем лишків F_p з простим модулем $p = 2^{448} - 2^{224} - 1$. Рівняння кривої Edwards448 має вигляд (2.9):

$$x^2 + y^2 = 1 - 39081x^2y^2, \tag{2.9}$$

де $x, y \in F_p$.

Крива Edwards448 забезпечує 224-бітний рівень безпеки. В специфікації [1] наведені такі значення параметрів Ed448: $b = 456$; кодування чисел – 455-бітне формату *little-endian*; $l = 2^{446} -$ 138180668098951153520073867485154268803366924748821786098945475038 85; $d = -39081,\ d \in F_p$; $B(x_b, y_b) \in E(F_p), x_b =$ 224580040295924300187604334099896036246789641632564134246125461686950415467406032909029192869357953282578032075146446173674602635247710, $y_b =$ 298819210078481492676017930443930673437544040154080242095928241372331506189835876003536878655418784733982303233503462500531545062832660; H – хеш-функція SHAKE-256.

Розділ 3 Опис реалізацій EdDSA в мові програмування Java

3.1. Опис інтерфейсів стандартних реалізацій EdDSA в мові програмування Java

3.1.1. Опис реалізацій EdDSA зі стандартного пакету java.security

У 2020 році в рамках оновлення JDK (Java Development Kit) до версії 15 була введена підтримка цифрового підпису EdDSA у вигляді реалізацій Ed25519 та Ed448. Детальний опис даних реалізацій міститься в документі JEP-339 (JDK Enhancement Proposal) [13]. У цьому документів вказано, що реалізації EdDSA розроблені на базі специфікації RFC-8032 [1]. Основною особливістю та перевагою реалізацій EdDSA в JDK є те, що вони повністю інтегровані в JCA (Java Cryptography Architecture) – базовий фреймворк для роботи з криптографічними сервісами в Java [14]. Можна сказати, що JCA надає єдиний API для виконання всіх криптографічних операцій, реалізованих в JDK (в контексті цифрового підпису – це генерація пари ключів, підпис повідомлення приватним ключем та верифікація підпису публічним ключем). Саме тому необхідно розглянути деякі з інтерфейсів JCA, щоб отримати розуміння, як можна використовувати реалізації EdDSA в JDK.

Клас *KeyPairGenerator* [15], що є частиною JCA, використовується для генерації пари приватного та публічного ключів. Екземпляр даного класу отримується за допомогою виклику статичного фабричного методу *getInstance*, вхідним параметром якого є строка-назва потрібного алгоритму (у випадку EdDSA – це строки «Ed25519» або «Ed448»). Безпосередня генерація ключів відбувається через виклик не статичного* методу *generateKeyPair*, результатом якого є об’єкт класу *KeyPair* [16], що і є представленням пари ключів в JCA. (*Надалі в роботі факт того, що метод є не статичним, буде розумітись за замовчуванням). Методи класу *KeyPair getPublic* та *getPrivate* повертають в якості результату об’єкти класів *PublicKey* [17] та *PrivateKey* [18], що є представленнями публічного та приватного ключів в JCA відповідно.

Клас *Signature* [19] є єдиним представленням всіх реалізацій алгоритмів цифрового підпису. Аналогічно до *KeyPairGenerator*, потрібний алгоритм цифрового підпису обирається через відповідну строку-параметр статичного методу *getInstance* (у випадку EdDSA – «Ed25519» або «Ed448»). Для виконання операції підпису в інтерфейсі класу *Signature* передбачений набір методів: 1) *initSign* – перший крок попередньої ініціалізації операції підпису. В якості вхідного параметру отримує об’єкт класу *PrivateKey*, тобто приватний ключ, яким надалі будуть підписуватись повідомлення; 2) *update* – другий крок попередньої ініціалізації операції підпису. Вхідним параметром є повідомлення у вигляді масиву байтів; 3) *sign* – безпосереднє виконання операції підпису, використовуючи надані раніше приватний ключ та повідомлення. Результатом виконання методу *sign* є масив байтів, що представляє собою значення цифрового підпису. Для здійснення операції верифікації цифрового підпису використовується такі інтерфейсні методи класу *Signature*: 1) *initVerify* – перший крок попередньої ініціалізації операції верифікації. Вхідним параметром цього методу є об’єкт типу *PublicKey*; 2) Метод *update* – описаний вище, але в контексті верифікації є другим кроком попередньої ініціалізації даної операції; 3) *verify* – безпосереднє виконання операції верифікації, використовуючи надані раніше публічний ключ та повідомлення. Вхідним параметром даного методу є значення цифрового підпису у вигляді масиву байтів, а результатом – булеве значення, що вказує на успішність або неуспішність верифікації публічним ключем підпису повідомлення.

З точки зору користувача JCA введення підтримки EdDSA в JDK версії ≥ 15 є помітним лише завдяки можливості використання строк «Ed25519» та «Ed448» в статичних методах *getInstance* класів *KeyPairGenerator* та *Signature*. Всі деталі реалізації цих алгоритмів приховані всередині фреймворку JCA. Це, наприклад, дозволяє надзвичайно легко здійснити перехід від DSA (або будь-якого іншого алгоритму цифрового підпису) до EdDSA – достатньо в існуючому коді просто змінити строку «DSA» на «Ed25519» або «Ed448». Така уніфікація і є

найбільшою перевагою JDK реалізацій EdDSA, порівняно з будь-якими іншими Java реалізаціями.

3.1.2. Опис реалізацій EdDSA в криптографічній бібліотеці Bouncy Castle

Починаючи з версії 1.60 (2018 рік), в криптографічній бібліотеці Bouncy Castle для мови Java також була додана підтримка алгоритму EdDSA, а саме створені реалізації Ed25519 та Ed448. На жаль, я не знайшов вичерпного опису цих реалізацій, окрім Java документації для відповідних однойменних класів [20] [21]. Пакет, в якому розміщені класи-реалізації Ed25519 та Ed448, має назву *org.bouncycastle.math.ec.rfc8032*. Зважаючи на дану назву, можна зробити припущення, що ці реалізації також створені на базі специфікації RFC-8032 [1]. Отже, класи-утиліти (класи, що мають лише статичні методи та змінні класу) *Ed25519* та *Ed448* містять повні реалізації відповідних алгоритмів. На відміну від JCA, в бібліотеці Bouncy Castle не використовуються спеціалізовані класи для представлення публічних та приватних ключів. Всі ключі представлені у вигляді масивів байтів. Таким чином, не існує сильної прив’язки реалізацій алгоритмів до способу генерації ключів. Тобто, в якості вхідних даних для операцій підпису та верифікації можна подати власноруч згенеровані ключі правильного розміру в «сирому» вигляді (raw data). В цій роботі ми не будемо розглядати механізм генерації пари ключів в бібліотеці Bouncy Castle, оскільки це не є необхідним для розуміння того, як використовуються відповідні реалізації EdDSA.

Статичний метод *sign* класу *Ed25519* безпосередньо виконує підпис повідомлення приватним ключем. Даний метод приймає значення восьми вхідних параметрів: 1) приватний ключ у вигляді масиву байтів; 2) позиція, починаючи з якої елементи масиву байтів приватного ключа вважаються змістом приватного ключа (private key offset); 3) масив байтів, що є контекстом підпису (використовується в контекстуальній реалізації Ed25519, яка в цій роботі не розглядається); 4) повідомлення у вигляді масиву байтів; 5) зміщення в масиві байтів повідомлення (message offset, аналогічний до private key offset);

6) довжина повідомлення в байтах; 7) порожній масив байтів, в який буде записане значення цифрового підпису; 8) зміщення в масиві байтів для збереження підпису (signature offset). Метод *sign* не повертає результату, а записує значення цифрового підпису в наданий масив.

Статичний метод *verify* класу *Ed25519* виконує верифікацію публічним ключем підпису повідомлення. *Verify* також приймає значення восьми параметрів: 1) значення підпису у вигляді масиву байтів; 2) зміщення в масиві байтів підпису; 3) публічний ключ у вигляді масиву байтів; 4) зміщення в масиві байтів публічного ключа; 5) масив байтів контексту (в цій роботі не розглядається); 6) повідомлення у вигляді масиву байтів; 7) зміщення в масиві байтів повідомлення; 8) довжина повідомлення в байтах. Метод *verify* повертає булеве значення, що вказує на результат операції верифікації.

Статичні методи *sign* та *verify* класу *Ed448* виконують операції підпису повідомлення приватним ключем та верифікації публічним відповідно. Набір параметрів методу *verify* є повністю аналогічним до описаного набору параметрів однойменного методу класу *Ed25519*. Метод *sign* класу *Ed448* приймає значення 10 параметрів: всі описані параметри методу *sign* класу *Ed25519* та два додаткові – публічний ключ у вигляді масиву байтів і зміщення в масиві байтів публічного ключа.

3.2. Опис власних реалізацій EdDSA

3.2.1. Загальний опис інтерфейсів, допоміжних класів та класів-утиліт

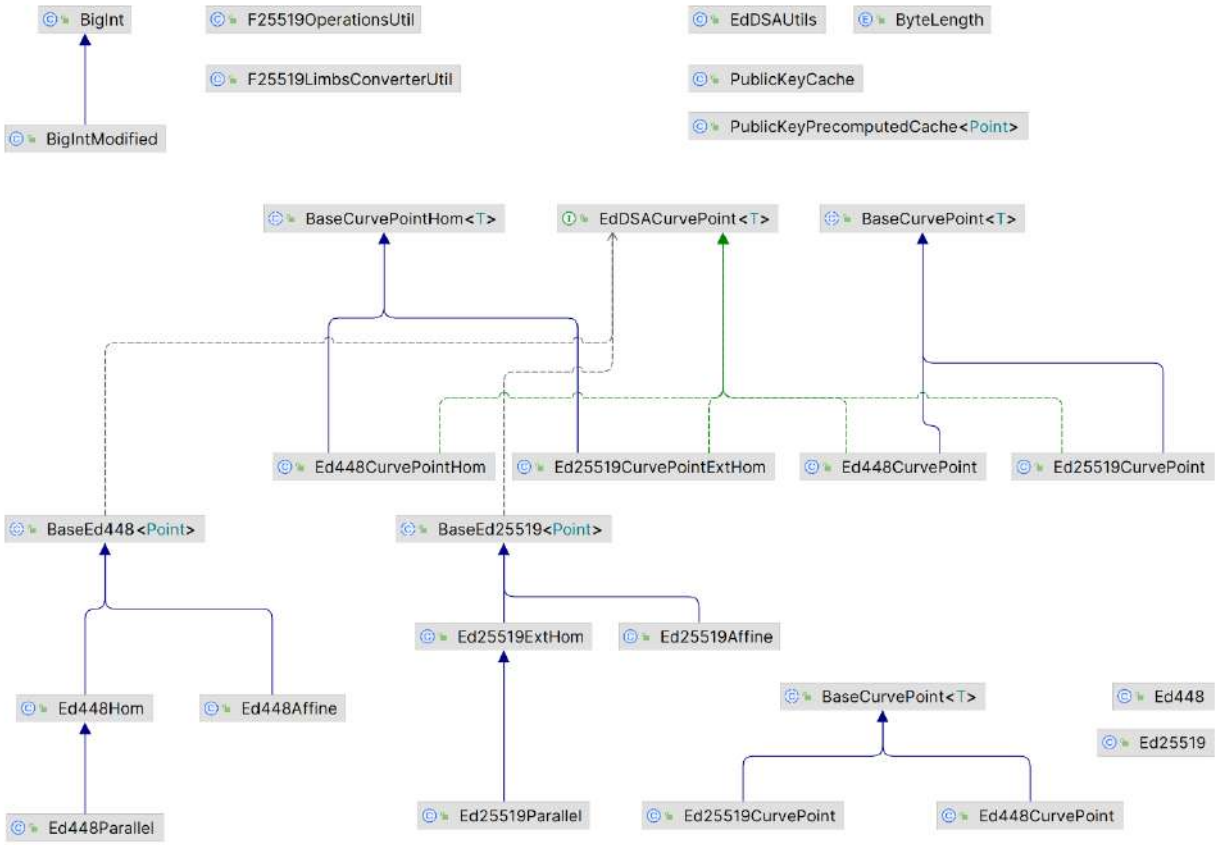

Рисунок 3.1 – діаграма класів проекту реалізацій EdDSA.

На рис. 3.1 представлена діаграма класів власного проекту реалізацій EdDSA. В ході виконання практичної частини роботи були створені декілька типів реалізацій Ed25519 та Ed448, а саме: 1) «стандартні» реалізації EdDSA – реалізації, що повністю використовують стандартне представлення великих чисел *BigInteger* в мові програмування Java та реалізовані на основі специфікації RFC-8032 [1] без додаткових оптимізацій; 2) «mathlib» реалізації – створені на основі RFC-8032 [1] без додаткових оптимізацій та використовують власне представлення великих чисел *BigIntModified*, про яке йтиметься згодом; 3) не паралельні реалізації, створені на основі RFC-8032 [1] з усіма розглянутими в даній роботі оптимізаціями; 4) паралельні реалізації, що використовують паралелізм даних та всі розглянуті оптимізації. Об’єктно-орієнтоване програмування (ООП) було обрано в якості єдиної парадигми програмування для виконання практичної частини роботи. Це дозволило чітко визначити три рівні абстракції, відповідно до яких були створені всі реалізації

EdDSA: рівень елементів поля, рівень елементів групи точок еліптичної кривої та рівень алгоритму. Такий підхід значно полегшив розуміння коду, покращив його розширюваність та підтримуваність, а також зробив кодову базу максимально наближеною до теоретичної доменної області. З іншого боку, слідування парадигмі ООП спонукало до створення чималої кількості допоміжних інтерфейсів, абстрактних класів та класів-утиліт, що певним чином ускладнило загальну архітектуру проекту. В цій секції будуть розглянуті всі допоміжні програмні компоненти, що використовується реалізаціями EdDSA.

Клас *BigIntModified* є власним представленням великих цілих чисел та є альтернативою відповідному стандартному представленню в мові програмування Java – класу *BigInteger* [22]. Основним недоліком *BigInteger* є його незмінність (immutability), через що в ході виконання кожної арифметичної операції над певним об’єктом цього класу створюється новий об’єкт *BigInteger*, який представляє результат відповідної операції. Наслідком цієї особливості є низька швидкість виконання арифметичних операцій та надмірне використання пам’яті, що робить *BigInteger* погано пристосованим до використання у високопродуктивних програмах. Клас *BigIntModified* частково вирішує дану проблему, надаючи можливість напряму змінювати об’єкт, над яким виконується операція, без створення додаткових копій. Це дозволяє зменшити час виконання операцій та оптимізувати використання пам’яті, хоча і збільшує ризик помилок в процесі проведення обрахунків. Даний клас наслідується від класу *BigInt*, який і містить реалізації усіх необхідних математичних операцій: додавання, віднімання, множення, ціла частина від ділення, остача від ділення, бітові зсуви на n позицій праворуч або ліворуч. Клас *BigInt* був створений не мною, я взяв його з публічного репозиторію на платформі GitHub [23] та включив у свій проект. В описі до репозиторію автори прямо вказують на можливість такого варіанту використання їхньої розробки. Для зручності використання класу *BigIntModified* в контексті поля лишків я визначив набір методів, які поєднують операції додавання, віднімання, множення з операцією ділення з остачею на певний модуль (*mod*). Крім того, мною були визначені

допоміжні методи, які спершу копіюють початковий об’єкт, а потім виконують відповідну операцією над копією (аналогічно до *BigInteger*). Це зумовлено тим, що в деяких випадках необхідно зберігати початковий стан об’єкту для проведення подальших обчислень. Інтерфейс класу *BigIntModified* також був розширений спеціалізованими методами, що використовуються у контексті обчислень в полі лишків та скалярного множення точок еліптичної кривої. Першим таким методом є *modInverseETIterative*, що реалізує операцію знаходження оберненого мультиплікативного елементу в полі лишків за допомогою ітеративної версії розширеного алгоритму Евкліда [24]. Додатково визначений метод *modInverseF25519*, який виконує таку ж операцію, але лише в полі F25519. Даний метод конвертує поточний об’єкт *BigIntModified* у формат limbs-10 за допомогою класу-утиліти *F25519LimbsConverterUtil*, делегує безпосереднє виконання операції modular inverse класу-утиліті *F25519OperationsUtil*, та конвертує результат назад в об’єкт класу *BigIntModified*. Класи-утиліти *F25519LimbsConverterUtil* та *F25519OperationsUtil* будуть розглянуті пізніше. Метод *powerMod* використовується для піднесення елементу поля лишків до степеню. Він реалізує алгоритм швидкого піднесення до степеню [25], при цьому виконуючи модульну редукції проміжних результатів на кожній ітерації алгоритму. Метод *wNAF* конвертує об’єкт *BigIntModified* у форму wNAF (описана у розділі 1) та використовується в процесі скалярного множення точок еліптичної кривої. Даний метод реалізує алгоритм, зображений на рис. 1.1.

Клас-утиліта *F25519LimbsConverterUtil* використовується для конвертації об’єкту типу *BigIntModified* у limbs-10 формат та у зворотньому напрямі. Формат limbs-10 описаний у розділі 2 цієї роботи. В якості прикладу для створення даного класу були використані файли *fe_tobytes.c* та *fe_frombytes.c* з публічного GitHub репозиторію реалізації Ed25519 на мові програмування C supercop ref10 [26].

Клас-утиліта *F25519OperationsUtil* містить реалізації операцій над елементами поля F25519 у форматі limbs-10. Прикладом для створення даного класу є клас *Ed25519FieldElement* з Java реалізації Ed25519 str4d [27]. Статичний метод *mulLimbs* виконує множення двох елементів поля F25519. В якості вхідних параметрів в метод передаються два 10-елементних масиви цілих чисел, що є представленнями елементів поля у форматі limbs-10. Результат множення – 10-елементний масив цілих чисел. При цьому, зміст масивів двох множників не модифікується. Статичний метод *mulAndDoubleLimbs* виконує множення двох елементів поля, після чого подвоює результат. Особливістю методів *mulLimbs* та *mulAndDoubleLimbs* є те, що вони використовують спільну реалізацію множення елементів поля, яка міститься у приватному статичному методі *mulLimbs*. Цей метод є перевантаженням публічного статичного методу *mulLimbs* третім параметром – булевим прапорцем. В залежності від значення цього прапорця, після завершення процесу поліноміального множення та модульної редукції та перед початком процесу нормалізації виконується або не виконується поліноміальне додавання. Таке рішення не лише зменшує дублювання коду, але й значно оптимізує операцію multiply-and-double, фактично роблячи її атомарною. Подібний підхід застосований і для операцій square (піднесення елементу поля до квадрату) та square-and-double (піднесення до квадрату з подальшим подвоєнням). Операція square реалізована статичним методом *squareLimbs*, а square-and-double – *squareAndDoubleLimbs*. Статичний метод *modInverseLimbs* виконує операцію знаходження мультиплікативного оберненого елементу в полі. В даному методі реалізований ланцюг послідовних операцій піднесень до квадрату та множення елементів поля для поступового піднесення вхідного елементу до степеню $2^{255} - 21$ (використовується Мала теорема Ферма для поля лишків з простим модулем). Цей ланцюг вперше представлений в реалізації supercop ref10 – файл *pow225521.h* [28]. Статичний метод *pow2p252minus2* використовує частину даного ланцюга для піднесення елементу поля до степеню $2^{252} - 2$. Ця операція використовується для знаходження квадратного кореня елементу в полі. Статичні методи *addLimbs* та

subLimbs виконують додавання та віднімання елементів поля відповідно (реалізують поліноміальне додавання та віднімання без подальшої нормалізації коефіцієнтів полінома-результату). Статичні методи *addOneLimbs* та *subOneLimbs* виконують додавання та віднімання одиниці до або від елементу поля відповідно.

В класі-утиліті *EdDSAUtils* містяться допоміжні статичні методи, що використовуються усіма реалізаціями EdDSA. Метод *generatePrivateKey* генерує масив випадкових байтів, довжина якого задається об'єктом класу-переліку *ByteLength*: 32, 57, 64 або 114 байт. Для генерації випадкового значення використовується об'єкт класу *UniformRandomProvider* з бібліотеки *org.apache.commons* [29]. Статичний метод *concatArrays* використовується для конкатенації довільної кількості масивів байтів. Результатом конкатенації є масив байтів, довжина якого є сумою довжин всіх масивів, що були передані в метод в якості вхідних параметрів. Визначені два перевантажені статичні методи *encodeLittleEndian* для кодування об'єктів класів *BigIntModified* та *BigInteger* у форматі *little-endian*. Результатом кодування є масив байтів. Методи *decodeLittleEndian* та *decodeLittleEndianBigIntModified* декодують масив байтів у форматі *little-endian* в об'єкт типу *BigInteger* та *BigIntModified* відповідно. Метод *sha512* приймає довільний масив байтів та повертає SHA-512 хеш у вигляді масиву байтів. Використовується реалізація хеш-функції SHA-512 з бібліотеки Bouncy Castle. Метод *shake256* створює хеш довільної довжини, яка задається об'єктом класу-переліку *ByteLength*, використовуючи Bouncy Castle реалізацію хеш-функції SHAKE-256. Метод *sha256* хешує вхідний масив байтів, так само використовуючи Bouncy Castle реалізацію SHA-256.

Клас *PublicKeyCache* використовується для кешування публічних ключів. Кеш представлений об'єктом параметризованого класу *Cache* з бібліотеки *caffeine* [30]. Кеш має обмеження розміру, що задається значенням вхідного параметру *maxSize* конструктора класу *PublicKeyCache*. Крім того, кеш реалізований за схемою LRU (least recent used), що означає витіснення найстарших записів в

кеші новими при досягненні максимального розміру. Кеш зберігає пари ключ-значення. Ключами є SHA-256 хеші у вигляді шістнадцяткових строк скалярних значень публічного ключа, а значеннями – закодована у масив байтів точка публічного ключа. Використання цього кешу дозволяє уникнути виконання одної операції скалярного множення точки в процесі операції підпису.

Задача параметризованого класу *PublicKeyPrecomputedCache* – кешування попередньо обчислених точок для точки публічного ключа (призначення попередньо обчислених точок пояснюється в розділі 1). Реалізація кеша аналогічна до відповідної реалізації в класі *PublicKeyCache.* Ключами даного кеша є SHA-256 строкові шістнадцяткові хеші закодованої точки публічного ключа, а значеннями – масиви об’єктів параметризованого типу, що є списками попередньо обчислених точок. Даний кеш використовується в операції скалярного множення точки публічного ключа за допомогою wNAF форми скаляру.

Абстрактний клас *BaseCurvePoint* з пакету *ua.kma.eddsa.mathlib* є базовим представленням точки еліптичної кривої в афінних координатах, що використовується реалізаціями *Ed25519Affine* та *Ed448Affine*. Даний клас параметризується типом класу-нащадку для забезпечення type-safety. Визначені такі числові поля класу: пара координат x, y, модуль поля p та параметр еліптичної кривої d. Всі ці поля представлені об’єктами типу *BigIntModified*. Булеве поле a є індикатором скрученої кривої Едвардса та використовується у загальній формулі додавання точок. Поле типу *ByteLength* вказує на байтову довжину кодування точки в конкретній реалізації EdDSA. Поле-масив байтів *encodedRepresentation* використовується для зберігання закодованого представлення точки. Розглянемо не абстрактні методи класу *BaseCurvePoint*. Метод *encode* – кодування поточної точки. Використовується принцип «lazy» ініціалізації для поля *encodedRepresentation*: якщо дане поле не ініціалізоване, то відбувається процес кодування і результат зберігається в нього, інакше – повертається значення поля *encodedRepresentation*. Метод *addInner* реалізує

операцію додавання іншої точки кривої до поточної за допомогою формул (1.28) та (1.33). В результаті додавання стан поточної точки модифікується. Результатом виконання методу *add* є нова точка, що є сумою поточної та іншої точок кривої. При цьому, стан поточної точки не модифікується. Метод *negate* виконує інверсію поточної точки, створюючи в результаті нову точку. Операція скалярного множення з використанням алгоритму double-and-add реалізована в методі *multiplyOnScalar*. Цей метод також створює нову точку замість модифікації стану поточної. Клас *BaseCurvePoint* визначає наступні абстрактні методи: 1) *belongsToCurve* – перевірка приналежності поточної точки певній еліптичній кривій за рівнянням кривої; 2) *createPoint* – допоміжний метод для створення нового об'єкту певного касу-нащадку *BaseCurvePoint*; 3) *identityPoint* – отримання нейтрального елементу групи точок еліптичної кривої; 4) *modInverse* – пошук оберненого мультиплікативного елементу в полі (даний метод є абстрактним, адже реалізація залежить від конкретного поля лишків). Основна мотивація для створення даного класу є уникнення дублювання коду, адже основні операції в групах точок кривих Edwards25519 та Edwards448 мають однакові реалізації.

Абстрактний клас *BaseCurvePointHom* є базовим представленням точки еліптичної кривої в проективних координатах та використовується реалізаціями *Ed25519ExtHom, Ed25519Parallel, Ed448Hom, Ed448Parallel*, що будуть розглянуті пізніше. Цей клас також параметризується типом класу-нащадку з тих же причин, що і клас *BaseCurvePoint*. Проективні та афінні координати точки представлені числовими полями типу *BigIntModified X*, *Y*, *Z* та *affineX, affineY* відповідно. Булеве поле *isBasePoint* – це індикатор того, чи є поточна точка базовою точкою групи еліптичної кривої (параметр *B* у визначенні EdDSA). Призначення решти полів цього класу є аналогічним до призначення відповідних полів класу *BaseCurvePoint*. Розглянемо не абстрактні методи класу *BaseCurvePointHom*. Метод *add* є аналогічним до однойменного методу в класі *BaseCurvePoint*. Приватний метод *multiplyWNAF* реалізує основну частину алгоритму скалярного множення точки з використанням wNAF форми скаляру

(рис. 1.2). Вхідними параметрами цього методу є список цілих чисел, що є wNAF представленням скаляру, та масив попередньо обчислених точок. Приватний метод *multiplyOnScalarSimple* реалізує алгоритм double-and-add скалярного множення точки. Приватний метод *multiplyWNAFBasePoint* делегує виконання операції скалярного множення методу *multiplyWNAF*, передаючи в нього масив попередньо обчислених точок для базової точки кривої. Є два перевантажених публічних методи з назвою *multiplyOnScalar*: перший має лише один вхідний параметр – скаляр типу *BigIntModified*, другий – скаляр типу *BigIntModified* та об'єкт класу *PublicKeyPrecomputedCache*. Логіка першого методу така: якщо поточна точка є базовою (перевірка індикатору *isBasePoint*), то виклик делегується методу *multiplyWNAFBasePoint*, інакше – виклик делегується методу *multiplyOnScalarSimple*. Другий перевантажений метод має більш складну логіку. Якщо об'єкт типу *PublicKeyPrecomputedCache* не ініціалізований – виклик делегується *multiplyOnScalarSimple*. Інакше – відбувається спроба дістати з кешу масив попередньо обчислених точок для поточної точки. Якщо кеш містить значення для ключа, яким є поточна точка, то подальше виконання операції скалярного множення делегується *multiplyWNAF* з відповідним масивом попередньо обчислених точок. В іншому випадку відбувається попереднє обчислення точок за допомогою виклику статичного методу *precomputePoints* (буде розглянутий пізніше), після чого результат поміщається в кеш під відповідним ключем і так само відбувається делегування виконання операції множення методу *multiplyWNAF*. Вся описана логіка реалізована з єдиною метою – уникнення багаторазового попереднього обчислення точок кривої, що є дуже дорогою операцією як з точки зору часу виконання, так і з точки зору обсягу використовуваної пам'яті. У такий спосіб ми гарантуємо виконання попереднього обчислення точок лише один раз: для базової точки кривої – під час ініціалізації програми, для точки публічного ключа – під час початкового заповнення кешу. Методи *getAffineXCoord* та *getAffineYCoord* використовуються для отримання значень афінних координат *x* та *y*. Якщо поля *affineX* або *affineY* не ініціалізовані, то відбувається обчислення

відповідних афінних координат за допомогою формул переведення проективних координат в афінні, після чого результат зберігається у відповідне поле. Інакше – просто повертаються значення полів *affineX* або *affineY*. Застосований принцип «lazy» ініціалізації. Призначення методу *encode* є аналогічним призначенню однойменного методу в класі *BaseCurvePoint*. Для кодування точки використовуються афінні координати. Статичний метод *precomputePoints* реалізує частину попередніх обчислень алгоритму wNAF скалярного множення точки (рис. 1.2). В класі *BaseCurvePointHom* визначені такі абстрактні методи: *addInner* – додавання точок з модифікацією стану поточної точки; *doublePoint* – подвоєння поточної точки з модифікацією її стану; *negate* – створення новох точки, інвертованої до поточної; *getBasePointPrecomputed* – допоміжний метод для отримання масиву попередньо обчислених точок для базової точки кривої; *allocatePrecomputedPointsArray* – допоміжний метод для отримання порожнього масиву визначеного розміру елементів типу класу-нащадку *BaseCurvePointHom*. Решта абстрактних методів аналогічна до відповідних однойменних абстрактних методів класу *BaseCurvePoint*. Клас *BaseCurvePointHom* має такі статичні поля: *W_SIZE* – розмір вікна у wNAF представленні скаляру; *PRECOMPUTED_ARR_SIZE* – розмір масиву попередньо обчислених точок (визначається за формулою 2^{W_SIZE-2}).

Інтерфейс *EdDSACurvePoint* визначає операції для роботи зі всіма представленнями точок еліптичних кривих, а саме: *encode, multiplyOnScalar, add*. Семантичне призначення даних операцій додаткового пояснення не вимагає. Цей інтерфейс використовується базовими реалізаціями *BaseEd25519* та *BaseEd448*.

Клас *BaseCurvePoint* з пакету *ua.kma.eddsa.standard* є повним відповідником однойменного класу з пакету *ua.kma.eddsa.mathlib* та використовується власними «стандартними» реалізаціями EdDSA. Відмінність між класами *BaseCurvePoint* з цих двох пакетів полягає в тому, що клас з пакету *ua.kma.eddsa.standard* використовує стандартне Java представлення великого

цілого числа *BigInteger*, а інший – власне представлення великих цілих чисел *BigIntModified*.

3.2.2. Опис реалізацій Ed25519

Перед тим, як перейти до безпосереднього опису реалізацій Ed25519, необхідно розглянути класи-представлення точок кривої Edwards25519. Я не розглядав дані класи в секції 3.2.1 цього розділу, адже вони настільки сильно пов’язані з відповідними реалізаціями Ed25519, що фактично є їхньою частиною.

Клас *Ed25519CurvePoint* є представленням точки еліптичної кривої Edwards25519 в афінних координатах. Цей клас наслідується від базового абстрактного класу *BaseCurvePoint* та імплементує інтерфейс *EdDSACurvePoint*. В класі *Ed25519CurvePoint* надаються реалізації всім абстрактним методам батьківського класу *BaseCurvePoint*. Здебільшого, дані реалізації є тривіальними та не заслуговують особливої уваги. Реалізація методу *belongsToCurve* використовує рівняння кривої Edwards25519 для перевірки приналежності поточної точки її групі точок. Клас *Ed25519CurvePoint* не містить власних полів.

Клас *Ed25519CurvePointHom* є представленням точки кривої Edwards25519 в «розширених скручених координатах Едвардса» (опис цього типу координат знаходиться в розділі 1). Даний клас також імплементує інтерфейс *EdDSACurvePoint* та наслідується від класу *BaseCurvePointHom*, реалізуючи його абстрактні методи. Клас *Ed25519CurvePointHom* має власне числове поле типу *BigIntModified* – додаткову координату *T*. Метод *addInner* реалізує алгоритм додавання з використанням «швидких» формул (рис. 1.3). При цьому, спочатку координати обох точок конвертуються у формат limbs-10 за допомогою класу-утиліти *F25519LimbsConverterUtil*, після чого над ними виконуються відповідні операції з класу-утиліти *F25519OperationsUtil*, а результат у вигляді нових значень координат знову конвертується в об’єкти типу *BigIntModified* з формату limbs-10. Метод *doublePoint* реалізує алгоритм подвоєння точки з використанням «швидких» формул (рис. 1.4). Операції над координатами точок

в цьому методі також відбуваються у форматі limbs-10. Реалізації решти абстрактних методів батьківського класу є тривіальними. Клас *Ed25519CurvePointHom* має статичне поле *B_PRECOMPUTED*, що зберігає в собі масив попередньо обчислених точок для базової точки кривої Edwards25519. Значення даного поля повертається реалізацією абстрактного методу *getBasePointPrecomputed* і використовується у реалізаціях wNAF скалярного множення в батьківському класі.

Абстрактний клас *BaseEd25519* є базовою узагальненою реалізацією алгоритму Ed25519 відповідно до специфікації RFC-8032 [1]. Даний клас параметризований загальним типом класів, що імплементують інтерфейс *EdDSACurvePoint*. Такий підхід дозволяє класу *BaseEd25519* оперувати з будь-яким представленням точок еліптичної кривої Edwards25519 (точками в афінних координатах або в «розширених скручених координатах Едвардса»), завдяки чому відбувається уникнення дублювання коду. Крім того, базова реалізація Ed25519 стає придатною до розширення. Тобто, виконуються SOLID принципи програмування – принцип відкритості до розширень та закритості до модифікацій (Open-Closed principle), а також принцип інверсії залежностей (Dependency Inversion principle). Клас *BaseEd25519* є батьківським класом для конкретних реалізацій Ed25519: *Ed25519Affine, Ed25519ExtHom, Ed25519Parallel.* Даний клас містить наступні статичні константні числові поля типу *BigIntModified*: *p* – модуль поля лишків F25519; *d* – параметр кривої Edwards25519; *L* – порядок базової точки в групі точок кривої Edwards25519; *B_AFFINE_X* та *B_AFFINE_Y* – афінні координати базової точки в групі точок кривої Edwards25519. Інші статичні константні поля класу *BaseEd25519* – це: булевий прапорець *a* – параметр скрученої кривої Edwards25519; *i* – допоміжна константа у форматі limbs-10, що визначається за формулою $i = 2^{\frac{p-1}{4}}$ та використовується під час обрахунку квадратного кореня елементу поля; *dLimbs* – значення *d* у форматі limbs-10, *ED25519_PK_SIZE* – байтовий розмір ключів Ed25519. Всі дані значення параметрів Ed25519 наведені в секції опису Ed25519

другого розділу. Не статичними полями класу *BaseEd25519* є базова точка кривої *B*, що має узагальнений тип *Point*, кеш публічних ключів типу *PublicKeyCache* та кеш попередньо обчислених точок для публічних ключів типу *PublicKeyPrecomputedCache*. Публічний метод *sign* виконує операцію підпису повідомлення приватним ключем. Він реалізований відповідно до опису операції *sign* загального алгоритму EdDSA (опис міститься в розділі 2). Вхідними параметрами цього методу є два масиви байтів повідомлення та приватного ключа, а результатом – значення підпису у вигляді масиву байтів. Приватний метод *verifyInner* реалізує операцію верифікації публічним ключем підпису повідомлення згідно з описом даної операції загального алгоритму EdDSA. Публічний метод *verify* є проксі-методом для *verifyInner*. Це означає, що виклик делегується *verifyInner*, а всі помилки, що потенційно виникають в процесі верифікації, оброблюються та трансформується в результат false. Сигнатура методу *verify* містить 3 вхідні параметри у вигляді масивів байтів: повідомлення, підпис та публічний ключ. Публічний метод *generatePublicKey* реалізує відповідно до опису EdDSA операцію генерування публічного ключа на основі приватного ключа. Вхідним параметром цього методу є масив байтів приватного ключа, а результатом – масив байтів публічного. Операція парсингу точки кривої реалізована публічним методом *decode* (реалізація також відповідає опису операції парсингу в EdDSA). Вхідним параметром для *decode* є закодована точка у вигляді масиву байтів, а результатом – об’єкт узагальненого типу *Point*. Всі операції над елементами поля F25519 в методі *decode* виконуються у форматі limbs-10. Приватний метод *sqrtModP* обчислює квадратний корінь для елемента поля F25519 у форматі limbs-10 та використовується в процесі парсингу точки. Приватний метод *getPublicKeyFromCache* використовується методом *sign* для взаємодії з кешем публічних ключів. Логіка цього методу така: якщо для певного скаляру кеш містить відповідну точку публічного ключа, то це значення дістається з кешу та повертається в результаті, інакше – точка публічного ключа обчислюється, зберігається в кеш для відповідного значення скаляру та повертається методом в

результаті. Клас *BaseEd25519* визначає допоміжний абстрактний метод для створення нового об’єкту точки кривої – *createPoint*.

Клас *Ed25519Affine* – реалізація Ed25519 без додаткових оптимізацій на рівні елементів поля та групи точок кривої. Даний клас наслідується від *BaseEd25519* та параметризує його типом *Ed25519CurvePoint*. Це означає, що в даному випадку базова реалізація *BaseEd25519* використовуватиме клас *Ed25519CurvePoint* для представлення точок кривої. Фактично, *Ed25519Affine* не містить власних реалізацій, а лише є «інструкцією» для базової реалізації, що вказує на те, яке представлення точок слід використовувати. Для *Ed25519Affine* реалізована підтримка кешу публічних ключів. Розмір кешу задається параметром *publicKeyMaxCacheSize* в конструкторі класу.

Клас *Ed25519ExtHom* – реалізація Ed25519 з використанням оптимізацій на рівні елементів поля та групи точок. Аналогічно до *Ed25519Affine* даний клас є нащадком *BaseEd25519* та параметризує його типом *Ed25519CurvePointExtHom*. *Ed25519ExtHom* також не містить власних реалізацій. Всі зазначені оптимізації інкапсульовані в класі *Ed25519CurvePointExtHom*. Для даної реалізації Ed25519 існує підтримка кешу публічних ключів (аналогічно до *Ed25519Affine*) та кешу попередньо обчислених точок для публічного ключа, розмір якого задається значенням параметру *publicKeysPrecomputedMaxCacheSize* в конструкторі класу.

Клас *Ed25519Parallel* – паралельна реалізація Ed25519 з використанням оптимізацій на рівні елементів поля та групи точок. Даний клас наслідується від класу *Ed25519ExtHom*. Клас *Ed25519CurvePointExtHom* використовується в якості представлення точки кривої в «розширених скручених координатах Едвардса». Клас *Ed25519Parallel* перевизначає методи *sign* і *verifyInner* базового класу *BaseEd25519* та містить власні реалізації операцій підпису та верифікації. Паралелізм даних досягається за допомогою використання *CompletableFuture* API [31] в поєднанні зі стандартним механізмом управління потоками *ExecutorService* [32]. Для даної реалізації Ed25519 доступна підтримка кешу попередньо обчислених точок публічного ключа. Створення об’єкту класу

Ed25519Parallel відбувається з використанням патерну проектування Builder. Це надає користувачеві інтуїтивно зрозуміле API для ініціалізації *Ed25519Parallel*: спочатку викликається статичний метод *builder*, після чого за допомогою статичного методу *withPublicKeyCache* можна задати максимальний розмір кешу попередньо обчислених точок, а за допомогою виклику статичного методу *withFixedThreadPool* задати кількість потоків. За замовчуванням кількість потоків є рівною до кількості ядер процесора. Для отримання об’єкту *Ed25519Parallel* викликається термінальний метод *build*, що створює об’єкт зі всіма вказаними властивостями. Клас *Ed25519Parallel* містить додатковий публічний метод *terminateExecutor*, що виконує безпечну зупинку *ExecutorService* та знищує створені потоки. Реалізація операцій підпису та верифікації семантично відповідає опису даних операцій в загальному алгоритмі EdDSA (див. розділ 2). Відмінністю від стандартної реалізації (наприклад, в класі *BaseEd25519*) є те, що послідовні процеси підпису та верифікації декомпонуються на низку підзадач, частина яких є незалежною та виконується паралельно. Так, на рис. 3.2 продемонстрована схема паралельного виконання операції підпису.

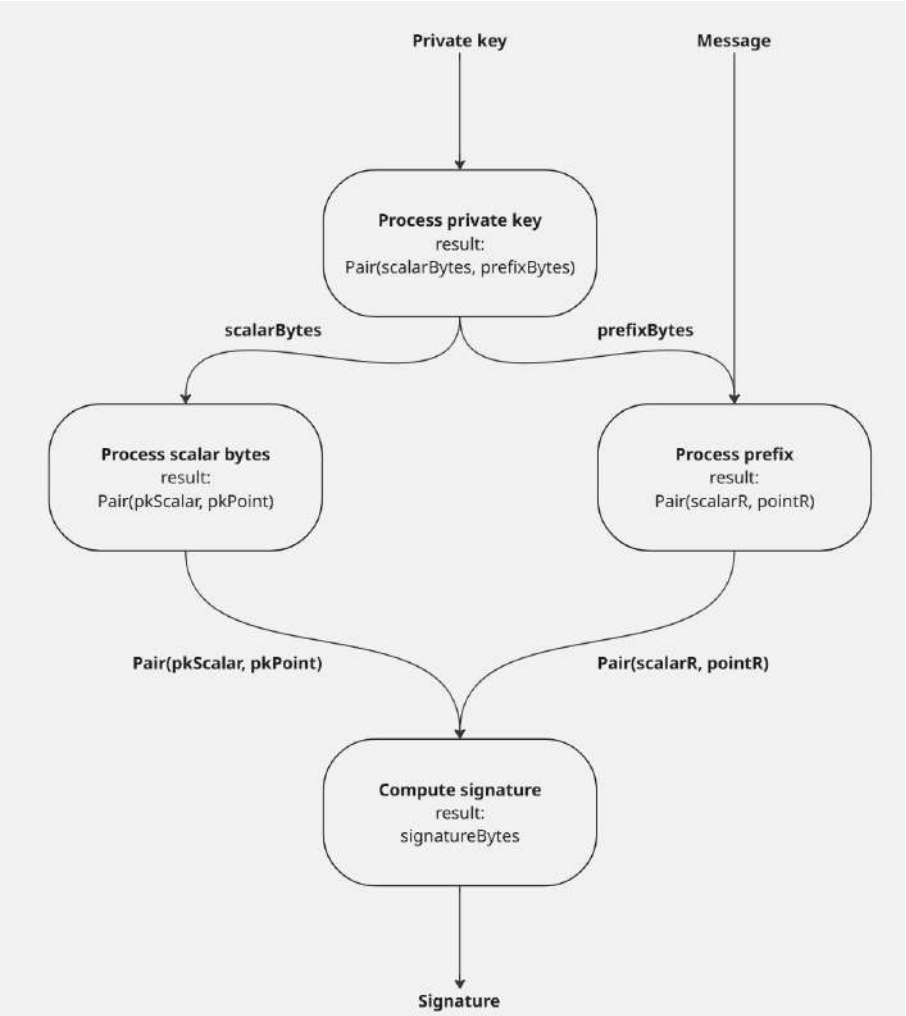

Рисунок 3.2 – схема паралельного виконання операції підпису в реалізації *Ed25519Parallel*.

На рис. 3.2 можна побачити, що спершу виконується блокуюча задача *process private key*, після чого результати її виконання передаються в задачі *process scalar bytes* та *process prefix*, які виконуються паралельно. Результати виконання задач *process scalar bytes* та *process prefix* передаються в кінцеву задачу *compute signature*, яка повертає фінальний результат. На рис. 3.3 продемонстрована аналогічна схема для операції верифікації.

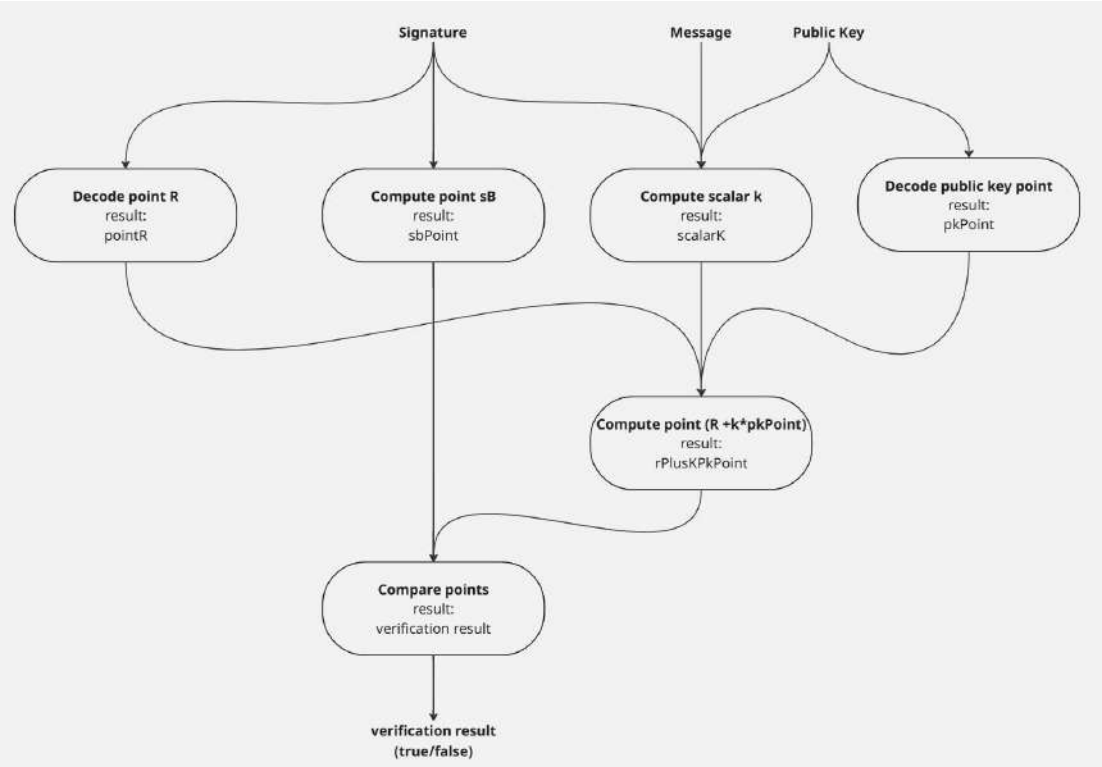

Рисунок 3.3 – схема паралельного виконання операції верифікації в реалізації *Ed25519Parallel*.

Аналізуючи схему на рис 3.3, можна помітити, що рівень паралелізму в операції верифікації є вищим за рівень паралелізму в операції підпису – паралельне виконання чотирьох задачі проти двох.

Клас *Ed25519CurvePoint* з пакету *ua.kma.eddsa.standard* є повним відповідником однойменного класу з пакету *ua.kma.eddsa.mathlib*. Різниця між цими двома класами полягає в тому, що перший (з пакету *ua.kma.eddsa.standard*) використовує *BigInteger* для представлення великих цілих чисел, а другий – *BigIntModified*.

Клас *Ed25519* з пакету *ua.kma.eddsa.standard* є повним аналогом класу *Ed25519Affine*. Знову ж таки, різниця між цими класами полягає у використанні відповідного представлення чисел: *Ed25519* використовує *BigInteger*, а *Ed25519Affine – BigIntModified*.

В таблиці 3.1 продемонстровані властивості усіх власних реалізацій Ed25519, а саме те, які оптимізації були використані в кожній реалізації.

Назва реалізації	*BigIntMod*	Field	Group	Cache1	Cache2	Parallel
Ed25519	-	-	-	-	-	-
Ed25519Affine	+	-	-	+	-	-
Ed25519ExtHom	+	+	+	+	+	-
Ed25519Parallel	+	+	+	-	+	+

Таблиця 3.1 (використані умовні позначення: *BigIntMod* – використання BigIntModified; Filed – оптимізації на рівні елементів поля лишків; Group – оптимізації на рівні групи точок кривої; Cache1 – використання кешу публічних ключів; Cache2 – використання кешу попередньо обчислених точок для точки публічного ключа; Parallel – використання паралелізму даних) – використання оптимізацій власними реалізаціями Ed25519.

3.3. Опис реалізацій Ed448

Клас *Ed448CurvePoint* з пакету *ua.kma.eddsa.mathlib* є представленням точки еліптичної кривої Edwards448 в афінних координатах. Фактично, даний клас майже не відрізняється від класу *Ed25519CurvePoint* за виключенням передачі в батьківський клас *BaseCurvePoint* параметрів еліптичної кривої Edwards448 і поля F448 та іншої реалізації методу *belongsToCurve*, яка використовує рівняння кривої Edwards448.

Клас *Ed448CurvePointHom* є представленням точки кривої Edwards448 в проективних координатах. Він реалізує методи *addInner* та *doublePoint*, використовуючи алгоритми «швидкого» додавання точок та подвоєння точки, що були описані в розділі 1 (рис. 1.5 та рис. 1.6 відповідно). Метод *modInverse* цього класу делегує виконання операції знаходження мультиплікативного оберненого елементу в полі F448 методу *modInverseETIterative* класу *BigIntModified*. Решта публічних методів класу *Ed448CurvePointHom* реалізовані аналогічно до відповідних методів класу *Ed25519CurvePointExtHom*. Клас *Ed448CurvePointHom* також наслідується від базового класу *BaseCurvePointHom* та імплементує інтерфейс *EdDSACurvePoint*.

Абстрактний клас *BaseEd448* є базовою узагальненою реалізацією алгоритму Ed448 відповідно до специфікації RFC-8032 [1]. Призначення статичних константних полів цього класу *p*, *d*, *L*, *B_AFFINE_X*, *B_AFFINE_Y*, *a* є аналогічним до призначення однойменних статичних полів класу *BaseEd25519*. Значення даних константних полів наведені в секції опису Ed448 другого розділу цієї роботи. Допоміжна константа *power* типу *BigIntModified* використовується в операції обчислення квадратного кореню в полі F448 та обчислюється за формулою: $power = \frac{p+1}{4}$. Статичне поле *ED448_PK_SIZE* визначає байтовий розмір ключів в Ed448 (57 байт). Статичний масив байтів *dom4* є результатом застосування функції dom4, описаної в специфікації RFC-8032, на порожній стрічці. Значення цього масиву конкатенується з вхідними даними хеш-функції SHAKE-256 в операціях підпису та верифікації. Методи *decode* та *sqrtModP* оперують об'єктами типу *BigIntModified* для парсингу точки та обчислення квадратного кореня елементу поля F448 відповідно. Реалізації решти методів класу *BaseEd448* не відрізняються від реалізацій відповідних методів класу *BaseEd25519*. Клас *BaseEd448* параметризований загальним типом класів, що імплементують інтерфейс *EdDSACurvePoint*, з тих же причин, що і клас *BaseEd25519*.

Класи *Ed448Affine* та *Ed448Hom* реалізують алгоритм Ed448 з використанням точок в афінних та проективних координатах відповідно (використовуються класи-представлення точок *Ed448CurvePoint* та *Ed448CurvePointHom*). Фактично, реалізації цих класів не відрізняються від реалізацій аналогічних класів-реалізацій Ed25519 *Ed25519Affine* та *Ed25519ExtHom*, крім використання інших класів для представлення точок кривої та наслідування від класу *BaseEd448*.

Клас *Ed448Parallel* являє собою паралельну реалізацію Ed448 з використанням оптимізацій на рівні групи точок кривої та кешу попередньо обчислених точок для публічного ключа. Цей клас є нащадком класу *Ed448Hom* та використовує представлення точок в проективних координатах. Технічні деталі реалізації

класу *Ed448Parallel* не відрізняються від відповідних технічних деталей реалізації класу *Ed25519Parallel*. Схеми паралелізації операцій підпису та верифікації класу *Ed25519Parallel* (рис. 3.2 та рис. 3.3) є актуальними і для реалізацій цих операцій в класі *Ed448Parallel*.

Сенсу детально розглядати технічні деталі стандартної реалізації *Ed448* з пакету *ua.kma.eddsa.standard* немає, адже вона фактично є аналогічною до реалізації *Ed25519* з цього ж пакету.

Властивості кожної власної реалізації Ed448 в контексті використовуваних оптимізацій вказані в таблиці 3.2.

Назва реалізації	*BigIntMod*	Field	Group	Cache1	Cache2	Parallel
Ed448	-	-	-	-	-	-
Ed448Affine	+	-	-	+	-	-
Ed448Hom	+	-	+	+	+	-
Ed448Parallel	+	-	+	-	+	+

Таблиця 3.2 (використані умовні позначення з таблиці 3.1) – використання оптимізацій власними реалізаціями Ed448.

Розділ 4 Порівняльний аналіз реалізацій EdDSA

4.1. Опис методики проведення порівняльного аналізу

В ході порівняльного аналізу проводяться заміри часу виконання операцій підпису повідомлення приватним ключем та верифікації публічним ключем підпису відповідного повідомлення всіма розглянутими в роботі реалізаціями EdDSA (власні реалізації, реалізації з бібліотеки Bouncy Castle та JDK). Визначені три тестові сценарії: підпис та верифікація кожного з 500, 5000 та 10000 слів, випадково згенерованих за допомогою сервісу Lipsum Generator [33]. Під час тестування всіх реалізацій використовуються дві заздалегідь згенеровані пари приватних та публічних ключів для алгоритмів Ed25519 та Ed448 відповідно. Критеріями для порівняння реалізацій є середній час підпису та середній час верифікації. Класи-утиліти *BouncyCastleEdDSABenchmark*, *JdkEdDSABenchmark*, *MathLibEdDSABenchmark*, *StandardEdDSABenchmark* імплементують зазначені тестові сценарії та використовуються для тестування реалізацій EdDSA з бібліотеки Bouncy Castle, з JDK, з власного пакету *ua.kma.eddsa.mathlib* та з власного пакету *ua.kma.eddsa.standard* відповідно. Одиниця виміру часу виконання операцій – мілісекунда.

4.2. Тестування реалізацій Ed25519

Перед проведенням порівняльного аналізу всіх реалізацій Ed25519 необхідно для реалізацій *Ed25519ExtHom* та *Ed25519Parallel* визначити оптимальне значення розміру бітового вікна у wNAF представленні скаляру (параметр *W_SIZE*). Для цього заміряємо середній час виконання операцій підпису та верифікацій для різних значень *W_SIZE*. Дане тестування проведемо для 5000 згенерованих слів. Розмір кешу попередньо обчислених точок для публічного ключа встановимо рівним 1. Результати тестування наведені в таблиці 4.1.

W_SIZE	*Ed25519ExtHom*		*Ed25519Parallel*	
	Середній час підпису, мс.	Середній час верифікації, мс.	Середній час підпису, мс.	Середній час верифікації, мс.
2	0.5125	0.5475	0.4957	0.4691
3	0.4430	0.5043	0.4561	0.4517
4	0.3491	0.4263	0.3572	0.3342
5	0.3399	0.3916	0.3362	0.3159
6	0.3106	0.3726	0.3235	0.2878
7	0.2841	0.3273	0.3186	0.2815
8	0.2737	0.2856	0.3078	0.2606
9	0.2669	0.2886	0.2921	0.2601
10	0.2650	0.2720	0.2877	0.2617
11	0.2634	0.2703	0.2769	0.2601
12	0.2627	0.2699	0.2663	0.2529
13	0.2621	0.2687	0.2645	0.2192
14	0.2620	0.2684	0.2642	0.2085
15	0.2611	0.2678	0.2637	0.1998
16	0.2598	0.2675	0.2618	0.1989
17	0.2590	0.2671	0.2609	0.1982
18	0.2678	0.2692	0.2941	0.2014
19	0.2785	0.2865	0.3013	0.2148
20	0.2914	0.3569	0.3527	0.2561

Таблиця 4.1 – середній час виконання підпису та верифікації реалізаціями *Ed25519ExtHom* та *Ed25519Parallel* в залежності від значення *W_SIZE*.

Дані таблиці 4.1 свідчать про те, що найменший час виконання операцій підпису та верифікації реалізаціями *Ed25519ExtHom* та *Ed25519Parallel* досягається при значенні *W_SIZE* = 17 (рядок виділений червоним). Тому, це значення буде використовуватись при подальшому тестуванні реалізацій *Ed25519ExtHom* та *Ed25519Parallel*. Час виконання операцій підпису та

верифікації зі значенням *W_SIZE > 20* постійно зростає. При значенні *W_SIZE = 22* виникає аварійне завершення програми через нестачу оперативної пам’яті для JVM. Саме тому тестування реалізацій з *W_SIZE > 20* не є доцільним з практичної точки зору.

Результати тестування всіх реалізацій на 500 словах наведені в таблиці 4.2. Тестування реалізації *Ed25519Affine* проводилось у 2 варіантах: з увімкненим (позначення *Ed25519Affine1* в таблиці 4.2) та вимкненим (*Ed25519Affine2* в таблиці 4.2) кешем публічних ключів. Реалізація *Ed25519ExtHom* тестувалась у 4 варіантах: 1) з вимкненими кешами публічних ключів та попередньо обчислених точок для публічного ключа (*Ed25519ExtHom1* в таблиці 4.2); 2) з увімкненим кешем публічних ключів та вимкненим кешем попередньо обчислених точок (*Ed25519ExtHom2* в таблиці 4.2); 3) з вимкненим кешем публічних ключів та увімкненим кешем попередньо обчислених точок (*Ed25519ExtHom3* в таблиці 4.2); 4) увімкненим кешем публічних ключів та увімкненим кешем попередньо обчислених точок (*Ed25519ExtHom4* в таблиці 4.2). Реалізація *Ed25519Parallel* також тестувалась у 2 варіантах: з увімкненим кешем попередньо обчислених точок (*Ed25519Parallel1* в таблиці 4.2) та вимкненим кешем попередньо обчислених точок (*Ed25519Parallel2* в таблиці 4.2).

Назва реалізації	Середній час підпису, мс.	Середній час верифікації, мс.
BouncyCastleEd25519	0.6141	0.6095
JDKEd25519	1.1056	1.0045
Ed25519Affine1	1.1132	1.0997
Ed25519Affine2	1.9541	1.9509
Ed25519ExtHom1	1.1275	1.1396
Ed25519ExtHom2	0.9516	0.9945
Ed25519ExtHom3	0.7204	0.7397
Ed25519ExtHom4	0.6149	0.6313
Ed25519Parallel1	0.6403	0.5910
Ed25519Parallel2	0.9501	0.9238
StandardEd25519	2.1452	2.2379

Таблиця 4.2 (використані умовні позначення: *BouncyCastleEd25519* – реалізація Ed25519 з бібліотеки Bouncy Castle; *JDKEd25519* – реалізація Ed25519 з JDK; *StandardEd25519* – власна реалізація Ed25519 з пакету *ua.kma.eddsa.standard*) – результати тестування реалізацій Ed25519 на 500 словах.

В таблиці 4.2 червоним кольором виділені найменші значення середнього часу підпису та верифікації. Реалізації *BouncyCastleEd25519* та *Ed25519ExtHom4* найшвидше виконують підпис (різницею у 0.0008 мс. можна знехтувати). Реалізація *Ed25519Parallel1* найшвидше виконує верифікацію з-поміж решти реалізацій. Найповільнішою реалізацією за часом виконання всіх операцій виявилась *StandardEd25519*.

Результати тестування реалізацій на 5000 слів наведені в таблиці 4.3 (умовні позначення варіантів реалізацій є аналогічними до позначень в таблиці 4.2).

Назва реалізації	Середній час підпису, мс.	Середній час верифікації, мс.
BouncyCastleEd25519	0.2569	0.2487
JDKEd25519	0.9563	0.9425
Ed25519Affine1	0.9721	0.9685
Ed25519Affine2	1.4394	1.5007
Ed25519ExtHom1	1.1059	1.2719
Ed25519ExtHom2	0.5645	0.6004
Ed25519ExtHom3	0.3751	0.3802
Ed25519ExtHom4	0.2590	0.2864
Ed25519Parallel1	0.2983	0.1982
Ed25519Parallel2	0.5196	0.4998
StandardEd25519	2.1035	2.0956

Таблиця 4.3 (використані умовні позначення з таблиці 4.2) – результати тестування реалізацій Ed25519 на 5000 слів.

Згідно з даними таблиці 4.3 найшвидшими реалізаціями за часом виконання підпису є *BouncyCastleEd25519* та *Ed25519ExtHom4*, а найшвидшою реалізацією за часом виконання операції верифікації є *Ed25519Parallel1*. Найповільнішою реалізацією за всіма критеріями є *StandardEd25519*.

Таблиця 4.4 містить результати тестування реалізацій на 10000 слів.

Назва реалізації	Середній час підпису, мс.	Середній час верифікації, мс.
BouncyCastleEd25519	0.1194	0.1202
JDKEd25519	0.7923	0.7909
Ed25519Affine1	0.8496	0.8513
Ed25519Affine2	1.3891	1.4019
Ed25519ExtHom1	1.2796	1.3841
Ed25519ExtHom2	0.7569	0.7677
Ed25519ExtHom3	0.3045	0.3149
Ed25519ExtHom4	0.1201	0.1307
Ed25519Parallel1	0.1295	0.1152
Ed25519Parallel2	0.6754	0.5891
StandardEd25519	1.9954	2.0045

Таблиця 4.3 (використані умовні позначення з таблиці 4.2) – результати тестування реалізацій Ed25519 на 10000 слів.

Аналізуючи дані таблиці 4.3, можна стверджувати, що найшвидшими реалізаціями за часом підпису є *BouncyCastleEd25519* та *Ed25519ExtHom4*, а найшвидшою за часом верифікації – *Ed25519Parallel1*.

4.3. Тестування реалізацій Ed448

Перед проведенням тестування всіх реалізацій Ed448 згідно з тестовими сценаріями необхідно визначити оптимальне значення параметру *W_SIZE*, яке буде надалі використовуватись реалізаціями *Ed448Hom* та *Ed448Parallel*. Час виконання операцій даними реалізаціями в залежності від значень *W_SIZE* наведений в таблиці 4.4. Тестування проводиться на 5000 слів, розмір кешу публічних ключів та кешу попередньо обчислених точок рівний 1.

W_SIZE	*Ed448Hom*		*Ed448Parallel*	
	Середній час підпису, мс.	Середній час верифікації, мс.	Середній час підпису, мс.	Середній час верифікації, мс.
2	1.3204	2.5781	1.2786	1.2050
3	1.2861	2.1796	1.1831	1.1894
4	1.1276	1.9521	1.1624	1.1589
5	1.0492	1.8677	1.1399	1.1144
6	1.0481	1.8629	1.1378	1.1145
7	1.0247	1.8421	1.1297	1.1087
8	0.9894	1.7632	1.1271	1.0856
9	0.9783	1.7572	1.1157	1.0659
10	0.9741	1.7019	1.1047	1.0570
11	0.9693	1.6427	1.0501	1.0399
12	0.9614	1.6301	1.0471	1.0371
13	0.9601	1.6240	1.0311	1.0264
14	0.9583	1.6235	1.0282	1.0175
15	0.9524	1.6012	1.0178	1.0094
16	0.9001	1.5599	1.0089	1.0076
17	0.8969	1.5524	0.9851	1.0035
18	0.9130	1.5678	1.0004	1.0107
19	1.2461	1.7564	1.3016	1.1964
20	1.5423	1.8421	1.4713	1.2589

Таблиця 4.4 – середній час виконання підпису та верифікації реалізаціями *Ed448Hom* та *Ed448Parallel* в залежності від значення *W_SIZE*.

Як і для *Ed25519ExtHom* та *Ed25519Parallel*, значення *W_SIZE* = 17 є найбільш оптимальним для використання реалізаціями *Ed448Hom* та *Ed448Parallel*, забезпечуючи найменший час виконання операцій підпису та верифікації.

Таблиця 4.5 містить результати тестування всіх реалізацій Ed448 на 500 словах. Сенс умовних позначень *Ed448Affine1*, *Ed448Affine2*, *Ed448Hom1*, *Ed448Hom2*,

Ed448Hom3, *Ed448Hom4*, *Ed448Parallel1*, *Ed448Parallel2* в таблиці 4.5 аналогічний сенсу відповідних позначень *Ed25519Affine1*, *Ed25519Affine2*, *Ed25519ExtHom1*, *Ed25519ExtHom2*, *Ed25519ExtHom3*, *Ed25519ExtHom4*, *Ed25519Parallel1*, *Ed25519Parallel2* в таблиці 4.2.

Назва реалізації	Середній час підпису, мс.	Середній час верифікації, мс.
BouncyCastleEd448	0.4921	0.5287
JDKEd448	2.5728	2.6746
Ed448Affine1	2.6018	2.6799
Ed448Affine2	3.9513	4.0086
Ed448Hom1	2.3854	2.5716
Ed448Hom2	1.6751	1.7524
Ed448Hom3	1.3504	1.4428
Ed448Hom4	1.2781	1.3945
Ed448Parallel1	1.3196	1.1656
Ed448Parallel2	2.1402	1.9976
StandardEd448	4.5972	5.1964

Таблиця 4.5 (використані умовні позначення: *BouncyCastleEd448* – реалізація Ed448 з бібліотеки Bouncy Castle; *JDKEd448* – реалізація Ed448 з JDK; *StandardEd448* – власна реалізація Ed448 з пакету *ua.kma.eddsa.standard*) – результати тестування реалізацій Ed448 на 500 словах.

Згідно з даними таблиці 4.5 найшвидшою реалізацією за всіма критеріями виявилась *BouncyCastleEd448*. Реалізація *Ed448Hom4* стала другою після *BouncyCastleEd448* за часом виконання підпису, а *Ed448Parallel1* – другою за часом виконання верифікації.

Результати тестування реалізацій Ed448 на 5000 слів представлені в таблиці 4.6.

Назва реалізації	Середній час підпису, мс.	Середній час верифікації, мс.
BouncyCastleEd448	0.4562	0.4682
JDKEd448	2.3866	2.3521
Ed448Affine1	2.4149	2.4063
Ed448Affine2	3.9586	4.0184
Ed448Hom1	2.4176	2.5699
Ed448Hom2	1.4051	1.8304
Ed448Hom3	1.1302	1.7228
Ed448Hom4	0.8969	1.5524
Ed448Parallel1	0.9851	1.0035
Ed448Parallel2	2.1322	1.9998
StandardEd448	4.5962	5.0148

Таблиця 4.6 (використані умовні позначення з таблиці 4.5) – результати тестування реалізацій Ed448 на 5000 слів.

Дані з таблиці 4.6 свідчать про те, що найшвидшою за часом підпису та верифікації так само є реалізація *BouncyCastleEd448*. *Ed448Hom4* є другою за часом виконання підпису, а *Ed448Parallel1* – другою за часом виконання верифікації.

Таблиця 4.7 містить результати тестування реалізацій на 10000 слів.

Назва реалізації	Середній час підпису, мс.	Середній час верифікації, мс.
BouncyCastleEd448	0.3357	0.4317
JDKEd448	2.3985	2.3627
Ed448Affine1	2.2059	2.3947
Ed448Affine2	4.0052	4.0163
Ed448Hom1	2.4022	2.5917
Ed448Hom2	1.3099	1.7991
Ed448Hom3	1.1050	1.6322
Ed448Hom4	0.8549	1.4271
Ed448Parallel1	0.9211	0.9675
Ed448Parallel2	2.1467	2.0076
StandardEd448	4.6182	4.9994

Таблиця 4.7 (використані умовні позначення з таблиці 4.5) – результати тестування реалізацій Ed448 на 10000 слів.

Результати тестування на 10000 слів показали, що реалізація *BouncyCastleEd448* є найшвидшою за усіма критеріями, *Ed448Hom4* – друга найшвидша за часом підпису, а *Ed448Parallel1* – друга найшвидша за часом верифікації. Згідно з трьома виконаними тестовими сценаріями найповільнішою за всіма критеріями реалізацією є *StandardEd448*.

4.4. Висновок до тестування

Для реалізацій *Ed25519ExtHom*, *Ed25519Parallel*, *Ed448Hom* та *Ed448Parallel* практичним способом визначене найбільш оптимальне значення параметру *W_SIZE = 17*, використання якого дозволяє досягти найменшого часу виконання операцій підпису та верифікації вказаними реалізаціями. Було проведене тестування всіх розглянутих в роботі реалізацій EdDSA згідно з трьома тестовими сценаріями: замір середнього часу виконання операцій підпису та верифікації кожного з 500, 5000 та 10000 випадково згенерованих слів.

В усіх тестових сценаріях серед реалізацій Ed25519 найшвидшими за часом підпису виявились *BouncyCastleEd25519* та власна реалізація *Ed25519ExtHom* з увімкненим кешем публічних ключів та кешем попередньо обчислених точок. Обидві ці реалізації показали приблизно однаковий час виконання підпису. За часом виконання операції верифікації найшвидшою виявилась власна реалізація *Ed25519Parallel* з увімкненим кешем попередньо обчислених точок. Поясненням повільнішого підпису в паралельній реалізації *Ed25519Parallel* порівняно з *Ed25519ExtHom* є те, що досягнутий рівень паралелізму *Ed25519Parallel* в операції підпису не виправдовує витрату ресурсів на його забезпечення. Наприклад, рівень паралелізму операції верифікації дорівнює 4 (рис. 3.3), а рівень паралелізму операції підпису – 2 (рис. 3.2). Ця відмінність дуже добре пояснює те, чому на практиці верифікація виконується значно швидше за підпис. Для створення в майбутньому найбільш продуктивної реалізації Ed25519 слід врахувати даний факт та відмовитись від паралелізації операції підпису на рівні алгоритму, а операцію верифікації залишити паралельною. Також слід зазначити, що власні реалізації *Ed25519ExtHom* та *Ed25519Parallel* з увімкненими відповідними кешами продемонстрували значно менший час виконання операцій підпису та верифікації, ніж стандартна реалізація *JDKEd25519*.

Після проведення тестування реалізацій Ed448 за трьома сценаріями було виявлено, що найшвидшою реалізацією за усіма критеріями є *BouncyCastleEd448.* Другою найшвидшою реалізацією за операцією підпису є *Ed448Hom* з увімкненим кешуванням публічних ключів та попередньо обчислених точок. За операцією верифікації другою найшвидшою реалізацією є *Ed448Parallel* з увімкненим кешем попередньо обчислених точок. Причиною відставання власних реалізацій Ed448 від відповідного рішення з бібліотеки Bouncy Castle за швидкістю виконання операцій є відсутність оптимізацій у *Ed448Parallel* та *Ed448Hom* на рівні елементів поля F448, тобто використання загального представлення великого цілого числа замість спеціалізованого формату для поля F448. Час виконання підпису реалізацією *Ed448Parallel* є

більшим за час виконання відповідної операції реалізацією *Ed448Hom* з тієї ж причини, з якої *Ed25519Parallel* поступається за швидкістю підпису *Ed25519ExtHom*. Також, цікавим є той факт, що навіть не повністю оптимізовані реалізації *Ed448Parallel* та *Ed448Hom* з увімкненими відповідними кешами значно перевершують стандартну реалізацію *JDKEd448* за часом виконання підпису та верифікації.

Крім того, аналізуючи результати тестування, можна помітити, що середній час виконання операцій підпису та верифікації власними реалізації EdDSA, які використовують відповідні кеші, не є константним, а є обернено пропорційним до кількості слів. Це пояснюється тим, що через початкове заповнення кешів підпис та верифікація першого слова з набору займає значно більше часу, ніж підпис та верифікація наступних слів. У наборах із меншою кількістю слів цей додатковий час, витрачений на перше слово, має більший вплив на кінцевий результат.

Висновки

В даній роботі були розглянуті теоретичні оптимізації EdDSA на рівні елементів групи точок еліптичної кривої, а саме: формули «швидкого» додавання та подвоєння точок для кривих Edwards25519 та Edwards448 та алгоритм скалярного множення точок кривої за допомогою wNAF форми для представлення скаляру запропоновані в [2], [8].

Крім того, був проаналізований запропонований в [11] спеціальний формат для представлення елементів поля лишків F25519 під назвою limbs-10, а також описані в цій роботі арифметичні операції над елементами даного поля в такому форматі. Були запропоновані власні технічні рішення щодо пришвидшення роботи реалізацій EdDSA: використання паралелізму даних, кеш публічних ключів, кеш попередньо обчислених точок для публічних ключів, що використовується в процесі wNAF скалярного множення. До власних технічних рішень можна також віднести створення оптимізованого представлення великих цілих чисел – класу *BigIntModified*.

В магістерській роботі створено низку власних реалізацій Ed25519 та Ed448. На базі стандартної специфікації RFC-8032 [1] без додаткових оптимізацій створені реалізації *Ed25519* та *Ed448* з пакету *ua.kma.eddsa.standard*. На базі стандартної специфікації та з використанням власного представлення великих цілих чисел – *Ed25519Affine* та *Ed448Affine*. Реалізації *Ed25519ExtHom*, *Ed25519Parallel*, *Ed448Hom*, *Ed448Parallel* імплементують «швидкі» формули додавання та подвоєння точок еліптичної кривої, а також алгоритм wNAF скалярного множення точки. Реалізації *Ed25519ExtHom* та *Ed25519Parallel* використовують limbs-10 формат для оптимізації арифметичних операцій в полі F25519. Реалізації *Ed25519Parallel* та *Ed448Parallel* використовують паралелізм даних в операціях підпису та верифікації.

При створенні власних реалізацій було запропоновано власні технічні рішення щодо пришвидшення роботи реалізацій EdDSA: використання паралелізму даних, кеш публічних ключів, кеш попередньо обчислених точок для публічних ключів, що використовується в процесі wNAF скалярного множення. До власних технічних рішень можна також віднести створення оптимізованого представлення великих цілих чисел – класу *BigIntModified*.

В результаті проведення практичного порівняльного аналізу всіх розглянутих реалізацій було з'ясовано, що власна реалізація *Ed25519Parallel* перевершує решту реалізацій Ed25519 (включно з реалізаціями Bouncy Castle та *java.security*) у швидкості виконання операції верифікації. Власна реалізація *Ed25519ExtHom* виконує операцію підпису так само швидко, як і відповідна реалізація Bouncy Castle, а реалізації *Ed448Hom* та *Ed448Parallel* за часом виконання підпису та верифікації перевершують відповідну стандартну реалізацію з пакету *java.security*. Також, були визначені недоліки власних реалізацій та запропоновані потенційні рішення для створення ще більш оптимізованих за швидкістю реалізацій Ed25519 та Ed448.

Скорочення та абревіатури

EdDSA – Edwards-curve Digital Signature Algorithm.

wNAF – window Non-Adjacent Form.

JDK – Java Development Kit.

JEP – JDK Enhancement Proposal.

JCA – Java Cryptography Architecture.

JVM – Java Virtual Machine.

ООП – об’єктно-орієнтоване програмування.

SOLID – single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, dependency inversion principle.

F25519 – поле лишків за модулем 2^{255-19}.

F448 – поле лишків за модулем $2^{448} - 2^{224} - 1$.

Ed25519ExtHom – Ed25519 extended homogeneous.

Ed448Hom – Ed448 homogeneous.

Список літератури

1. IRTF. RFC-8032. [Електронний ресурс] – Режим доступу до ресурсу: https://datatracker.ietf.org/doc/html/rfc8032
2. Denis Khleborodov. Fast elliptic curve point multiplication based on window Non-Adjacent Form method. Applied Mathematics and Computation, 334(C):41-59, 2018.
3. Uma Maheswari, Prabha Durairaj. Modified Shanks' Baby-Step Giant-Step algorithm and Pohlig-Hellman algorithm. International Journal of Pure and Applied Mathematics, 118(10):47-56, 2018.
4. Andrea Corbellini. Elliptic Curve Cryptography: a gentle introduction. [Електронний ресурс] – Режим доступу до ресурсу: https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/
5. Harold M. Edwards. A normal form for elliptic curves. American Mathematical Society, 44(3):393-422, 2007.
6. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, C. Peters. Twisted Edwards Curves. Progress in Cryptology – AFRICACRYPT 2008, Lecture Notes in Computer Science, 5023:389-405, 2008.
7. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson. Twisted Edwards Curves Revisited. Lecture Notes in Computer Science, 5350:326-343, 2008.
8. D. J. Bernstein, T. Lange. Faster Addition and Doubling on Elliptic Curves. Kurosawa, K. (eds) Advances in Cryptology – ASIACRYPT 2007. ASIACRYPT 2007. Lecture Notes in Computer Science, 4833:29-50, 2007.
9. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. Journal of Cryptographic Engineering, 2(2):77-89, 2012.
10. D. J. Bernstein, S. Josefsson, T. Lange, P. Schwabe, and B.-Y. Yang. EdDSA for more curves. Cryptology ePrint Archive, 2015(677), 2015.

11. D. J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, Lecture Notes in Computer Science, 3958:207-228, 2006.
12. M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Paper 2015(625), 2015.
13. Adam Petcher. JEP 339: Edwards-Curve Digital Signature Algorithm (EdDSA). [Електронний ресурс] – Режим доступу до ресурсу: https://openjdk.org/jeps/339
14. Oracle. Java Cryptography Architecture (JCA) Reference Guide. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html
15. Oracle. Class KeyPairGenerator. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/security/KeyPairGenerator.html
16. Oracle. Class KeyPair. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/security/KeyPair.html
17. Oracle. Interface PublicKey. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/security/PublicKey.html
18. Oracle. Interface PrivateKey. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/security/PrivateKey.html
19. Oracle. Class Signature. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/security/Signature.html
20. Bouncy Castle. Class Ed25519. [Електронний ресурс] – Режим доступу до ресурсу: https://javadoc.io/static/org.bouncycastle/bcprov-jdk15on/1.64/index.html?org/bouncycastle/math/ec/rfc8032/Ed25519.html

21. Bouncy Castle. Class Ed448. [Електронний ресурс] – Режим доступу до ресурсу: https://javadoc.io/static/org.bouncycastle/bcprov-jdk15to18/1.70/org/bouncycastle/math/ec/rfc8032/Ed448.html
22. Oracle. Class BigInteger. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html
23. Bwakell. Huldra. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/bwakell/Huldra
24. GeekForGeeks. Euclidean algorithms (Basic and Extended). [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/
25. Baeldung. The Most Efficient Way to Implement an Integer Based Power Function. [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/cs/integer-based-power-function
26. jedisct1. Supercop. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/jedisct1/supercop/blob/master/crypto_sign/ed25519/ref10/
27. str4d. ed25519-java. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/str4d/ed25519-java
28. jedisct1. Supercop pow225521.h. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/jedisct1/supercop/blob/master/crypto_sign/ed25519/ref10/pow225521.h
29. The Apache Software Foundation. Interface UniformRandomProvider. [Електронний ресурс] – Режим доступу до ресурсу: https://commons.apache.org/proper/commons-rng/commons-rng-client-api/javadocs/api-1.4/org/apache/commons/rng/UniformRandomProvider.html
30. Ben Manes. Class Caffeine<K,V>. [Електронний ресурс] – Режим доступу до ресурсу: https://www.javadoc.io/doc/com.github.ben-manes.caffeine/caffeine/2.1.0/com/github/benmanes/caffeine/cache/Caffeine.html

31. Oracle. Class CompletableFuture<T>. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html
32. Oracle. Interface ExecutorService. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html
33. LipsumGenerator [Електронний ресурс]:[Веб сайт]. – Режим доступу до ресурсу: https://www.lipsum.com/