УДК 004.056.55 **DOI:10.30837/rt.2025.1.220.01**

О.Г. КАЧКО, канд. техн. наук, І.Д. ГОРБЕНКО, д-р техн. наук, С.О. КАНДІЙ

ЗАСТОСУВАННЯ ФІКСОВАНОЇ ТОЧКИ ЗАМІСТЬ ПЛАВАЮЧОЇ ДЛЯ РЕАЛІЗАЦІЇ ЕЛЕКТРОННОГО ПІДПИСУ FALCON

Вступ

Перемога схеми електронного підпису (ЕП) FALCON на конкурсі NIST PQC і розробка стандарту на основі цього алгоритму пов'язана з найкращими показниками по розміру відкритого ключа та електронного підпису [1]. Особливості реалізації алгоритму FALCON з детальним описом засобів оптимізації наведено в роботі [2]. Для ефективного виконання операцій множення та ділення для поліномів застосовують швидке перетворення Фур'є, яке потребує представлення даних в форматі з плаваючою точкою. Для забезпечення можливості застосування алгоритмів на обчислювальних засобах, які не підтримують роботу з плаваючою точкою застосовують емуляцію.

Необхідність застосування плаваючої точки або її емуляції приводить до проблеми залежності точності обчислень від порядку, що може привести до додаткових атак [3]. В роботі [4] наведено застосування фіксованої точки для генерації ключів, що, з одного боку, не потребує від обчислювального пристрою роботи з плаваючою точкою, а по друге, практично не поступається за ефективністю алгоритму генерації з застосуванням плаваючої точки. Але, на жаль, масштаб для фіксованої точки, запропонований в роботі [4], не можна застосовувати для формування електронного підпису згідно з алгоритмом [2]. В роботі [5] запропоновано інший масштаб для завдання даних з фіксованою точкою, який дозволяє застосовувати цей формат для генерації ЕП, але застосування різного формату представлення даних для генерації та обчислення ЕП незручно.

Мета роботи – модифікація алгоритму генерації ключів, запропонованого в роботі [4] для формату даних з роботи [5], що забезпечує застосування загального масштабу для операцій генерації ключів та формування ЕП. Операція перевірки ЕП не потребує застосування даних з плаваючою точкою і, відповідно, операцій з фіксованою точкою. Виконано порівняння продуктивності для основних операцій в разі застосування плаваючої точки, її емуляції (авторська реалізація [2]) та фіксованої точки з різними масштабами [4, 5].

1. Представлення даних

Розглянемо представлення значень з фіксованою точкою. Для зберігання значення відводиться 8 байтів, як і для чисел з плаваючою точкою. Будемо називати масштабом числа (SCALE) кількість бітів, які відводяться для нецілої частини числа.

Формат числа показано на рис. 1.

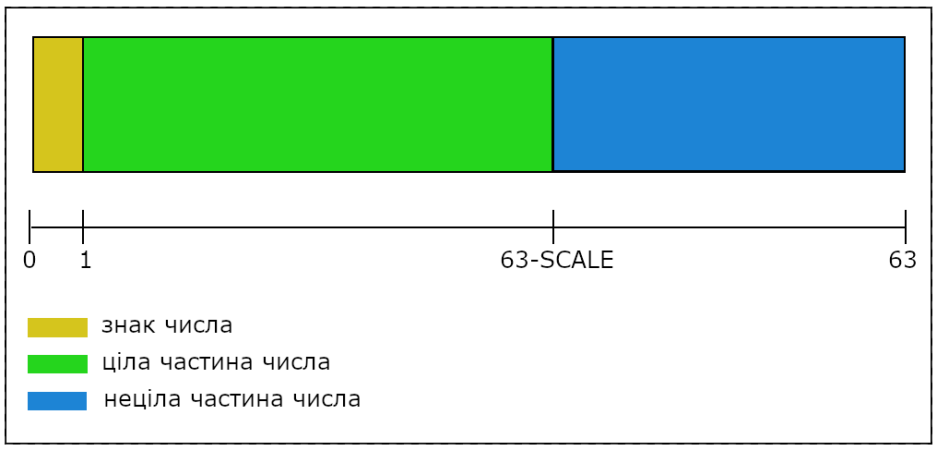

Рис. 1. Формат чисел з фіксованою точкою

Неціла частина має SCALE бітів, де параметр SCALE може приймати значення від 1 до 32. Ціла частина числа має 63 SCALE бітів і знак числа 1 біт (0 для невід'ємного, 1 для від'ємного). Число задається в додатковому коді.

В роботі [4] в якості масштабу застосовують SCALE = 32 біта, тоді для цілої частини залишається 31 біт, що обмежує значення числа. В роботі [5] обчислено масштаб SCALE = 26 біт, який розширює діапазон представлення даних до значень, які дозволяють генерувати ЕП, але, на жаль, зменшує точність даних та потребує суттєво переробити бібліотеку функцій для роботи з фіксованою точкою. У даній роботі була розроблена універсальна бібліотека для довільного масштабу SCALE < 32. Детальний опис функцій наведено у розд. 2.

Для зручності застосування, для функцій, які потрібні при генерації, збережено імена з роботи [4], префікс для імен функцій для реальних даних fxr і для комплексних – fxc. Для обчислення експоненти (під час обчислення ЕП), усі значення менше одиниці, для збільшення точності замість SCALE = 26 застосовують подвійний масштаб SCALE = 2 * 26, і відповідна функція має префікс fxr2.

2. Бібліотека

В табл. 1 представлено набір функцій бібліотеки для роботи з даними з фіксованою точкою, які не залежать від SCALE в умовах загальної довжини числа 64 біта. В якості цих функцій застосовують функції авторів [4]. В табл. 2 представлено набір функцій бібліотеки для роботи з фіксованою точкою, які залежать від SCALE. Туди ж додано функції. які для генерації ключів не застосовуються.

Таблиця 1

Функції, які не залежать від масштабу SCALE

Ім'я функції	Призначення
fxr_add	Додавання даних
fxr_sub	Віднімання даних
fxr_double	Множення на 2
fxr_neg	Для даного x обчислення значення – x
fxr_abs	Обчислення абсолютного значення
fxr_div2e	Ділення числа на 2^n (зсув числа вправо на n біт)
fxr_mul2e	Множення числа на 2^n (зсув числа вліво на n біт)
fxr_lt	Порівняння двох чисел x, y. Якщо x менше y, функція повертає TRUE, інакше FALSE

Таблиця 2

Функції, які залежать від масштабу SCALE

Ім'я функції	Призначення
fxr_of	Перетворення цілого в число з фіксованою точкою
fxr_mul	Множення даних
fxr_sqr	Обчислення квадрату числа
fxr_div	Ділення даних
fxr_inv	Інверсія. Для даного x обчислення 1/x
fxr_sqrt	Корінь квадратний (застосовують для розгортання секретного ключа)
fxr_round fxr_rint	Округлення до найближчого цілого
fxr_trunc	Відсічення нецілої частини
fxr_floor	Ціле значення, яке не перевищує задане
fxr2_exp	Обчислення exp(x), де 0 <=x < 1 (застосовують для обчислення ЕП)

Операції для комплексних чисел (+, -, *, /) застосовують функції для роботи з фіксованою точкою.

Для усіх операцій бібліотеки передбачається, що вхідними даними та результатом виконання є дані з фіксованою точкою, а також відсутність переповнень, тобто ціла частина не перевищує по модулю значення $2^{63\text{-SCALE}}$.

Найбільш ресурсоємними є операції множення, ділення, обчислення кореня та експоненти. Операції множення та ділення застосовують як для реальних так і для комплексних даних. Ці операції застосовують для роботи з секретними ключами, тому для них умова незалежності від часу є обов'язковою.

2.1. Множення даних з фіксованою точкою

Операція множення виконується для невід'ємних чисел. В разі завдання від'ємних чисел, обчислюється їх абсолютне значення. Для кінцевого результату коригується знак.

При множенні даних $x*2^{SCALE}$, $y*2^{SCALE}$ треба отримати результат $x*y*2^{SCALE}$. В результаті множення 64-бітних невід'ємних значень отримаємо 126 бітне дане, яке дорівнює $x*y*2^{2*SCALE}$. Для отримання вірного значення необхідно отриманий результат поділити на 2^{SCALE} з урахуванням округлення. В разі відсутності переповнення отримане значення гарантовано містить не більше ніж 63 біта. Як показує експеримент з генерацією 10000 ключів, формування відповідних ЕП та їх перевірки переповнення не виникають.

При реалізації функції множення не передбачалася підтримка застосування 128 бітних даних, для отримання добутку завдовжки більше ніж 64 біта застосовувалася програмна реалізація за допомогою алгоритму Карацуби (3 множення замість чотирьох).

Спочатку для вхідних даних обчислювалося абсолютне значення. Після множення 64 бітних беззнакових даних отримували добуток завдовжки 126 біт. для якого застосовувався зсув на SCALE бітів вправо та виконувалося округлення. В кінці коригувався знак для результату. Обчислення абсолютного значення та зворотне коригування знаку виконувалося без застосування команд переходу. Квадрат обчислювався множенням однакових значень (функція fxr_sqr). Нижче наведено код функції множення з коментарями основних операцій.

Функція множення чисел у форматі фіксованою точкою.

```
static inline fxr fxr_mul(fxr fx, fxr fy) {
   // Константи множення
   const uint64_t SCALE_MASK = (1ULL << SCALE) - 1;
   const uint64_t ROUND_THRESHOLD = 1ULL << (SCALE - 1);
   // Зчитування значень
   uint64_t x = fx.v;
   uint64_t y = fy.v;
   // Виділення знаків чисел
   uint64_t sign_x = x >> 63;
   uint64_t sign_y = y >> 63;
   uint64_t final_sign = sign_x ^ sign_y;
   // Отримання абсолютних значень
   x = (x ^ -sign_x) + sign_x;
   y = (y ^ -sign_y) + sign_y;
   // Віділення цілої частини
   uint64_t x_high = x >> SCALE;
   uint64_t y_high = y >> SCALE;
   uint64_t x_low = x & SCALE_MASK;
   uint64_t y_low = y & SCALE_MASK;
   // Множення Карацуби
   uint64_t high_product = x_high * y_high;
   uint64_t low_product = x_low * y_low;
```

```
    uint64_t cross_sum = (x_high + x_low) * (y_high + y_low);
    uint64_t middle_term = cross_sum - low_product - high_product;
    // Обчислення результату
    uint64_t result = (low_product >> SCALE) +
        (low_product & SCALE_MASK > ROUND_THRESHOLD) +
        middle_term + (high_product << SCALE);
    // Врахування знаку
    result = (result ^ -final_sign) + final_sign;
    return (fxr){.v = result};
}
```

2.2. Операція ділення

Як і для операції множення для x, y застосовують абсолютне значення, знак результату коригується після обчислень.

При діленні даних $x*2^{SCALE}$, $y*2^{SCALE}$ треба отримати результат $(x/y)*2^{SCALE}$.

Для забезпечення відсутності переповнення x/y повинно не перевищувати $2^{63\ -\ SCALE}$, тобто вхідні дані повинні задовольняти вимозі $x < y * 2^{63\ -\ SCALE}$, або $x/2^{63\ -\ SCALE} < y$. Для x, y застосовують масштабоване значення, тобто для цих значень нерівність залишається.

Фактично застосовують побітовий алгоритм ділення в стовпчик. Починаємо зі старших бітів x, відповідне значення яких не перевищує 2*y, тобто результат ділення x на $2^{62\ -\ SCALE}$, а далі поступово дописуємо по одному біту до тих пір, поки біти x не будуть вичерпані (всього таких бітів 62 – SCALE), які були відкинуті при діленні x на $2^{63\ -\ SCALE}$. В результаті виконання циклу 63 – SCALE разів отримаємо старші 63 – SCALE бітів масштабованого числа. Для отримання решти бітів (нецілу частину) виконуємо цикл SCALE разів число доповнюємо нулями.

Для прикладу роботи алгоритму розглянемо число x з загальною довжиною 8 бітів, масштаб SCALE дорівнює 3 біта. Нехай x = 1011011 і дільник y = 1100.

Згідно з умовою для x, y маємо $x/28 - 3 < y$, тобто значення x можна зсувати на 4 біта і отримаємо значення, яке може перевищувати значення y менше ніж в два рази. На кожному наступному кроці будемо додавати один біт з решти бітів.

Формуємо біти для доповнення, на які ми виконали зсув, доповнюємо нулями до 7 бітів, отримаємо 10110000

Виконуємо 8 кроків для кожного біту результату для ілюстрації роботи алгоритму.

Що порівнюємо	Біт результату	Біти для доповнення
101 < 1100	0	1011000
1011 < 1100	0	011000
10110>1100	1	11000
10101>1100	1	1000
10011>1100	1	000
01110>1100	1	00
00100>1100	0	0
01000>1100	0	
Результат	00111100	

П е р е в і р к а :

X дорівнює 11.2625, y дорівнює 1.5. Результат ділення ≈7.508 і з урахуванням масштабу дорівнює 60. $001111002 = 6010$. Нижче наведено реалізацію функції ділення з коментарями основних операцій.

Функція ділення чисел у форматі фіксованою точкою.

```
static inline fxr fxr_div(fxr fx, fxr fy) {
    const uint64_t MAX_POSITIVE = 0x7FFFFFFFFFFFFFFFULL;
    const int TOTAL_BITS = 64;
    const int SHIFT_BITS = TOTAL_BITS - 2;
    // Зчитування значень та виділення знаку
    uint64_t x = fx.v;
    uint64_t y = fy.v;
    uint64_t final_sign = (x >> 63) ^ (y >> 63);
    // Виділення абсолютних значень
    x = (x ^ -(x >> 63)) + (x >> 63);
    y = (y ^ -(y >> 63)) + (y >> 63);
    // Ініціалізація змінних
    uint64_t quotient = 0;
    uint64_t numerator = x >> (SHIFT_BITS - SCALE);
    uint64_t temp = x << (SCALE + 2);
    // Алгоритм ділення
    #pragma unroll
    for (int i = SHIFT_BITS; i >= 0; i--) {
        uint64_t diff = numerator - y;
        uint64_t bit = 1 - (diff >> 63);

        quotient = (quotient << 1) | bit;
        numerator = ((numerator - (y & -bit)) << 1) | (temp >> 63);
        temp <<= 1;
    }
    // Обчислення результату
    uint64_t final_bit = 1 - ((numerator - y) >> 63);
    quotient = (quotient + final_bit) & MAX_POSITIVE;
    // Врахування знаку
    fx.v = (quotient ^ -final_sign) + final_sign;
    return fx;
}
```

2.3. Операція обчислення кореня квадратного

Застосовують побітовий алгоритм [6], який перетворений для забезпечення незалежності часу виконання операції від конкретних даних. Псевдокод наведено нижче.

Функція обчислення квадратного кореня у форматі фіксованою точкою.

```
static inline fxr fxr_sqrt(fxr fn) {
    const int MAX_STEP = 62;
    const uint64_t INITIAL_D = 1ULL << MAX_STEP;
    const int RESULT_SHIFT = SCALE / 2;
    // Ініціалізація змінних
    uint64_t x = fn.v;
    uint64_t n = x;
    uint64_t c = 0;
    uint64_t d = INITIAL_D;
```

```
    #pragma unroll 4
    for (int step = MAX_STEP; step >= 0; step -= 2) {
        // Ми можемо відняти поточне значення?
        int64_t can_subtract = ~((int64_t)(n - d) >> 63);
        // Обчислення потенційно нового значення
        uint64_t potential = c + d;
        uint64_t trial = can_subtract & potential;
        // Перевірка чи пробне значення в квадраті менше або дорівнює x
        int64_t is_valid = can_subtract & (~((int64_t)(x - trial) >> 63));
        // Оновлення значень
        x -= is_valid & trial;
        c = (can_subtract & (c >> 1)) + (is_valid & d);
        d >>= 2;
    }
    // Врахування масштабу
    fn.v = c << RESULT_SHIFT;
    return fn;
}
```

3. Порівняння продуктивності операцій бібліотеки

Для визначення продуктивності кожної операції виконувався експеримент для 1024 випадкових наборів даних, для яких виключалося переповнення. Для виключення випадкового збільшення часу виконання за рахунок переключення потоків для кожного набору даних виконувалось 256 тестів, з яких обиралося мінімальне значення. Для визначення впливу даних на продуктивність з усіх мінімальних значень обиралося максимальне значення. Продуктивність вимірялася в тактах процесору (12th Gen Intel(R) Core(TM) i5-1240P).

В табл. 3 представлено обчислювальну складність операцій множення та ділення для варіантів: застосування плаваючої точки та її емуляції [1], масштабу SCALE = 32 [3] та масштабу SCALE = 26 [4] відповідно. Для операції обчислення квадратного кореня та експоненти для варіантів застосування плаваючої точки та її емуляції [1] та масштабу SCALE = 26 [4].

Таблиця 3

Продуктивність базових операцій

Операція	Варіант			
	double		SCALE = 32	SCALE = 26
	апаратна	емуляція		
Mul	10 14	10 14	10 14	10 14
Div	10 14	10 14	10 14	10 14
Sqrt	10 14	10 14	-	23 63
Exp	10 14	10 14	-	10 14

Перше число в таблиці визначає мінімальне значення для випадкових 1024 чисел, друге – максимальне. Операція Sqrt виконується при розгортанні секретного ключа і не реалізована для SCALE = 32. Операція exp виконується при генерації ЕП і не реалізована для SCALE = 32. Отримані результати свідчать, що продуктивность окремих операцій практично не залежить від формату.

4. Функції для алгоритму Falcon

4.1. Генерація ключів

Для генерації ключів застосовано авторський алгоритм генерації [4] з деякими змінами.

Для генерації випадкового seed застосовують SHAKE256, функція генерації ключів keygen має інтерфейс, як у авторів falcon [2], наповнення – авторів [4] з масштабом з [5].

Додано генерацію відкритого ключа та видалено перевірку наявності інверсії для поліному f. Наявність інверсії фактично перевіряється при генерації відкритого ключа.

При генерації ключів можливі помилки, пов'язані з невідповідністю згенерованих даних вимогам алгоритму, а саме:

- перевищені максимальні значення для коефіцієнтів або норма для поліномів f, g (fg_error);
- норма ортогонального вектору перевищує максимальну (fg_inv_error);
- помилка при обчисленні відкритого ключа (public_error) або при перевірці наявності інверсії для поліномів f, g (для SCALE = 32);
- NTRU рівняння не має рішення (найбільший спільний дільник не дорівнює 1), gcd_error;
- помилка при редукції поліномів (reduce_error); виникає, якщо в разі редукції для поліномів порядку 2, 4, … n отримаємо значення F, G, які не задовольняють NTRU рівнянню для заданих f, g;
- компоненти поліному F перевищують встановлену границю (limit_error_f);
- компоненти поліному G перевищують встановлену границю (limit_error_g).

В разі наявності будь-якої помилки генерація починається спочатку.

Розглянуто наступні варіанти:

В а р і а н т 1 . Генерація ключів по [2] з застосуванням даних з плаваючою точкою, або емуляції;

В а р і а н т 2 . Генерація ключів по [4] з застосуванням даних з фіксованою точкою, SCALE = 32;

В а р і а н т 3 . Генерація ключів по [5] з застосуванням даних з фіксованою точкою, SCALE = 26;

Варіант 2 не передбачає генерації відкритого ключа, але перевіряє, що вектори f, g мають інверсії, що фактично еквівалентно перевірці наявності відкритого ключа, тому ці дані будуть записані в рядку для public_error.

Для формування випадкових ключів стан генератору встановлюється однаковим для усіх трьох варіантів і задається функціями:

```
inner_shake256_init(&rng);
inner_shake256_inject(&rng, (uint8_t*)buf, strlen(buf));
inner_shake256_flip(&rng);
```

де buf – рядок завдовжки 8 символів, який містить символи:

для n = 512: 0x6b, 0x65, 0x79, 0x67, 0x65, 0x6e, 0x20, 0x39, 0.

Для n = 1024 замість передостаннього символу 0x39 застосовують символ 0x3a.

На рис. 2 представлено дані про кількість помилок при генерації 10000 успішних ключів для Falcon512. На рис. 3 представлено аналогічні дані для Falcon1024.

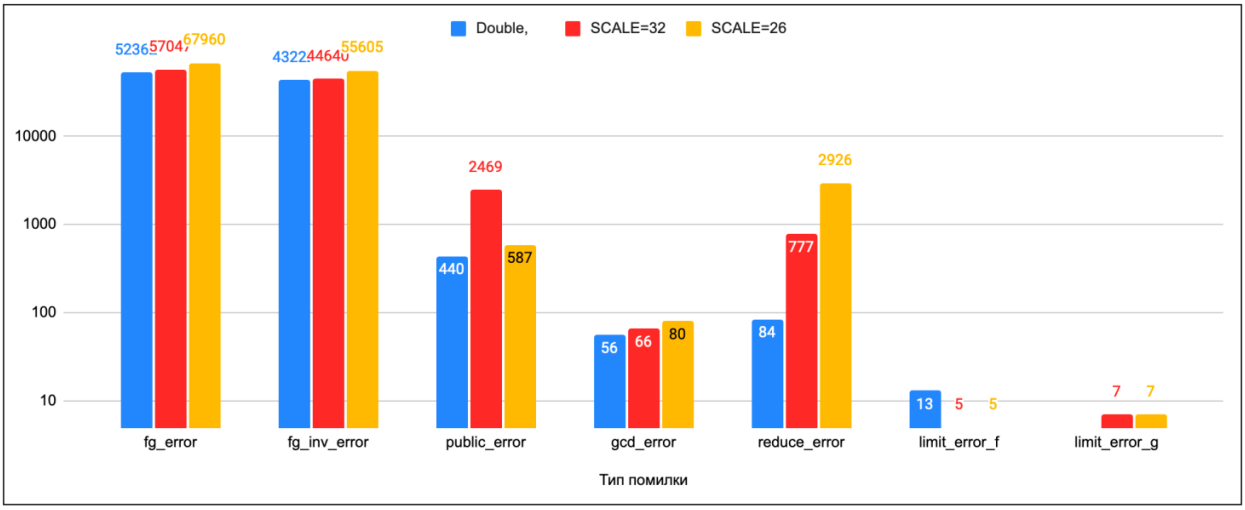

Рис. 2. Дані по помилкам при генерації ключів для Falcon512

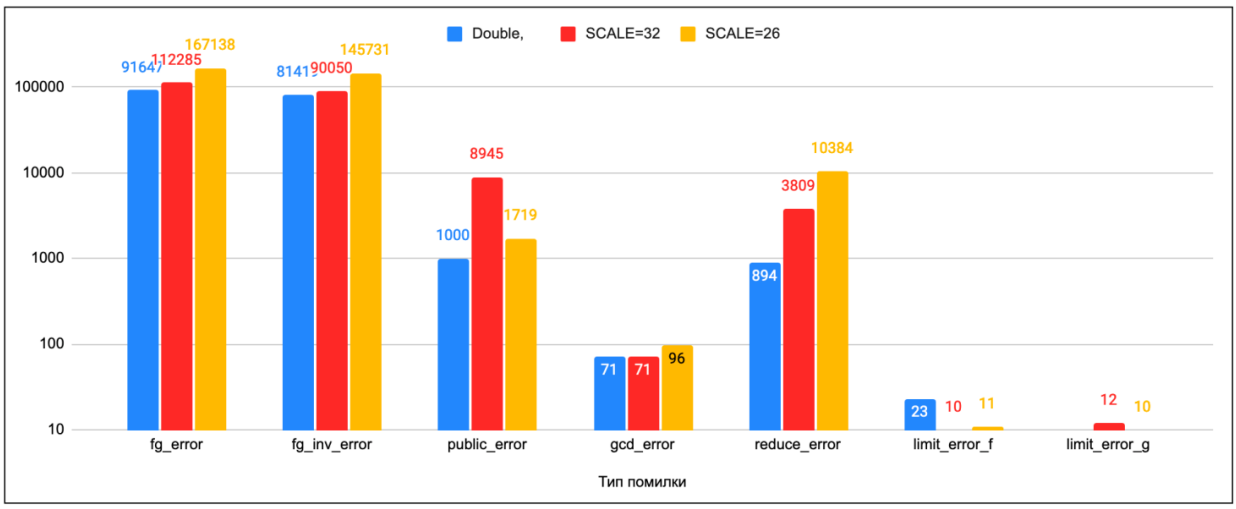

Рис. 3. Дані по помилкам при генерації ключів для Falcon1024

Як показав експеримент, кількість помилок суттєво відрізняється для помилки типу reduce_error. Це пов'язано з суттєвим зменшенням точності в разі застосування фіксованої точки, особливо у варіанті зі SCALE = 26. Збільшення reduce_error впливає на продуктивність операції генерації ключів, тому що ця помилка визначається практично в кінці операції генерації, що потребує починати цю операцію спочатку.

Найбільш "жадібною" серед функцій бібліотеки, які застосовують фіксовану точку, є функція комплексного множення fxc_mul, яка обчислює модуль знаменника (три операції fxr_mul), виконує ділення чисельника на модуль знаменника (дві операції fxr_div) та виконує операцію комплексного множення.

4.2. Розгортання секретного ключа

Розгортання секретного ключа може виконуватись безпосередньо паралельно з генерацією ЕП, а також як окрема функція, яка в якості вхідних даних застосовує тільки набір поліномів для секретного ключа (f, g, F, G).

Варіант з паралельним розгортанням і генерацією ЕП застосовують, якщо обмеження на розмір пам'яті є критичним. В цьому випадку ключи розгортаються кожного разу при створенні ЕП, що збільшує час, необхідний для її генерації. Варіант з окремим розгортанням застосовують, якщо критичним є параметр часу, для одного набору секретних ключів генерують декілька ЕП.

Автори [1, 2] запропонували обидва варіанти, наявність яких може призвести до атаки, за допомогою якої можна обчислити секретні ключі [3]. В [5] визначено умови, при яких дана атака неможлива

Розгортання секретного ключа (побудова дерева Falcon tree) виконується згідно з алгоритмом, представленим в [2] (функція expand_privkey). На основі FFT формату для поліномів f, g, F, G формується дерево, при обчисленні якого застосовують операції з фіксованою точкою. Після обчислення елементів дерева виконується його нормалізація, тобто кожне значення елементу дерева value заміняється значенням $\overline{\overline{\text{value}}}$.

4.3. Електронний підпис

Електронний підпис виробляється згідно з алгоритмом [1, 2] (функція sign_tree), але для роботи з даними з плаваючою точкою застосовується фіксована точка [5]. В якості вхідних даних застосовують повідомлення для підпису, розгорнутий секретний ключ та ініціалізований генератор псевдовипадкових чисел.

Генерація ЕП передбачає генерацію біта, значення якого дорівнює 1 з імовірністю e^x, що потребує обчислення експоненти для аргументу в діапазоні [0.. ln2). Додаткова функція fxr_expm обчислює це значення для аргументів, заданих з фіксованою точкою.

5. Продуктивність функцій алгоритму FALCON

У межах дослідження було виконано порівняння продуктивності функцій генерації ключів, розгортання секретного ключа та вироблення ЕП для варіантів застосування плаваючої точки, емуляції плаваючої точки та фіксованої точки для SCALE = 26.

Як і в попередньому випадку продуктивність визначається в тактах процесору.

Продуктивність визначається для 100 різних ключів.

Для генерації кожного ключа виконується 16 експериментів і визначається мінімальний час (для виключення випадкових переключень потоків). Для 100 різних ключів визначається мінімальна кількість тактів (min), максимальна кількість тактів з мінімальних (max) та середнє значення продуктивності (avg).

Операції розгортання ключів та вироблення ЕП виконується для кожного зі 100 ключів (знову експеримент повторюється для кожного ключа 16 разів, обчислюється мінімальне, максимальне та середнє значення).

Для порівняння застосовується Reference_Implementation без застосування AVX команд авторів (папка falcon-round3, [1]) та відповідна реалізація в разі застосування фіксованої точки з масштабом SCALE = 26.

На рис. 4 наведено результати вимірювання продуктивності функцій алгоритму Falcon для Faclon512. На рис. 5 наведено відповідні дані для Falcon1024.

Для кожного випадку задаються три значення. Перше значення – мінімальне, друге – максимальне з мінімальних, третє – середнє. Для усіх варіантів мінімальне значення для фіксованої точки виграє в порівнянні з мінімальним значенням для режиму емуляції плаваючої точки.

Для усіх варіантів за виключенням генерації ключів для n = 1024 максимальне та середнє значення для фіксованої точки поступається відповідним значенням для режиму емуляції плаваючої точки. Треба зазначити, що застосування фіксованої точки замість плаваючої не вирішує проблему залежності результату від порядку виконання операцій.

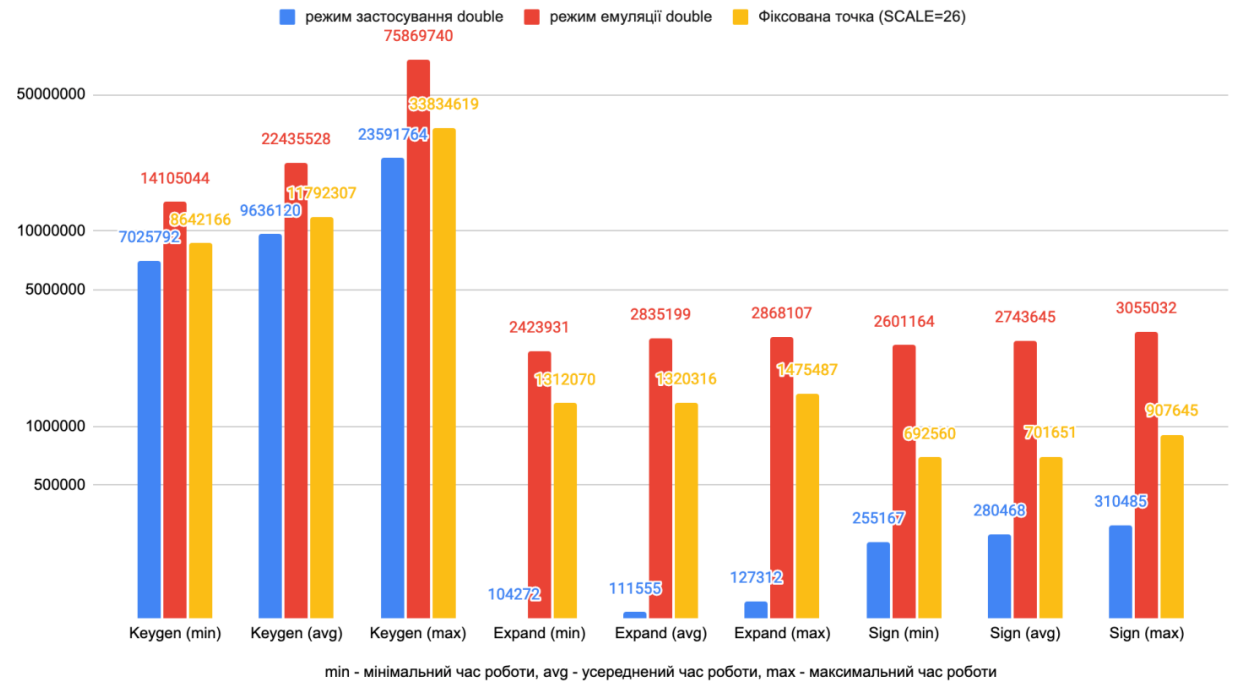

Рис. 4. Оцінка часу роботи функцій Keygen, Expand, Sign для Falcon512

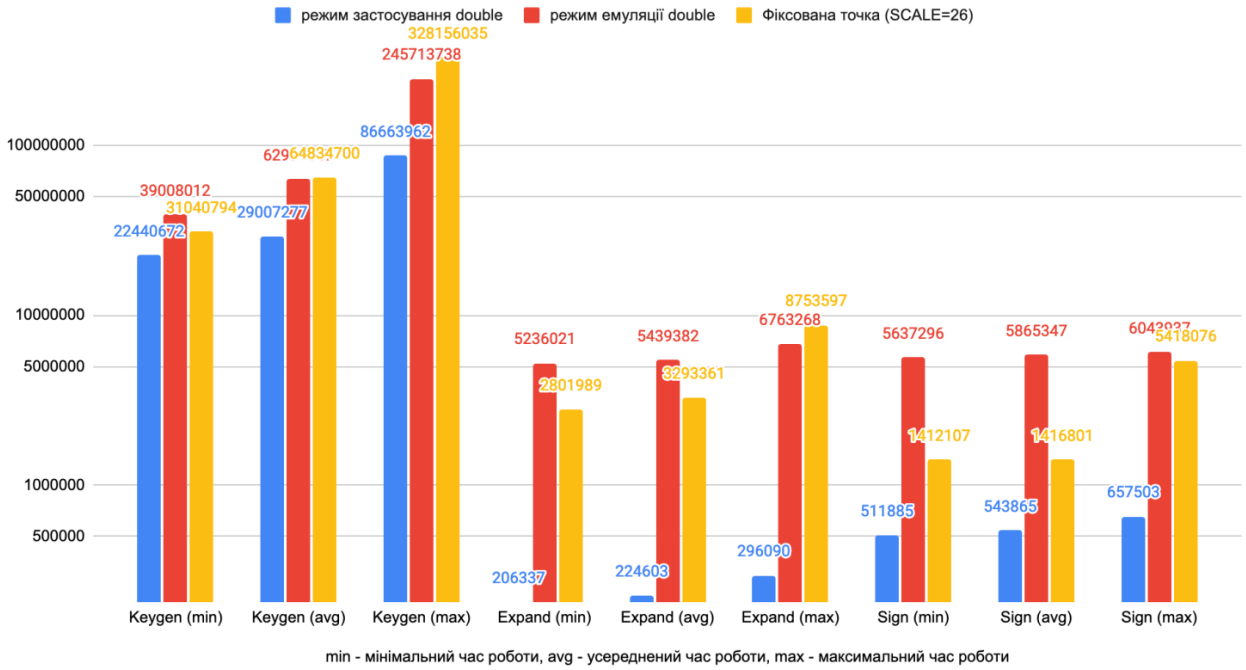

Рис. 5. Оцінка часу роботи функцій Keygen, Expand, Sign для Falcon1024

Висновки

Модифіковано алгоритм генерації ключів та формування електронного підпису з урахуванням формату даних з фіксованою точкою, запропонованого в роботі [5], що дозволило забезпечити єдиний масштаб для усіх операцій генерації ключів та формування електронного підпису. Такий підхід значно спрощує реалізацію криптографічних алгоритмів, зберігаючи при цьому високу продуктивність. Універсальна бібліотека, розроблена в межах дослідження, підтримує масштаб SCALE < 32 та містить реалізації ключових операцій, що враховують специфіку обчислень з фіксованою точкою.

Результати експериментального дослідження показали, що кількість помилок суттєво відрізняється для помилки типу reduce_error. Це пов'язано з суттєвим зменшенням точності в разі застосування фіксованої точки, особливо в варіанті зі SCALE = 26. Збільшення reduce_error впливає на продуктивність операції генерації ключів, тому що ця помилка визначається практично в кінці операції генерації, що потребує починати цю операцію спочатку.

Серед усіх реалізованих функцій найбільш ресурсоємними є операції множення, ділення та комплексного множення. Функція комплексного множення fxc_mul продемонструвала найбільші вимоги до ресурсів через необхідність виконання кількох ключових операцій, таких як обчислення модуля знаменника та ділення. Незважаючи на це, застосування фіксованої точки забезпечує обчислювальну незалежність від часу для операцій із секретними ключами, що є важливою вимогою для забезпечення криптографічної стійкості.

Таким чином, запропоновані модифікації та аналіз продуктивності демонструють, що формат з фіксованою точкою може бути ефективно застосований у криптографічних алгоритмах за умови оптимізації масштабу та врахування впливу на точність. Отримані результати можуть бути використані для подальшого вдосконалення криптографічних протоколів та їх реалізації в обмежених середовищах, де використання плаваючої точки є небажаним або неможливим.

Список літератури:

1. Round 3 submissions – post-quantum cryptography: CSRC, CSRC. Available at: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions (Accessed: 10 November 2024).

2. Pornin T. New efficient, constant-time implementations of Falcon // IACR Cryptology ePrint Archive. Available at: https://eprint.iacr.org/2019/893 (Accessed: 10 November 2024).

3. Potii O. et al. Determining the effect of a floating point on the Falcon Digital Signature Algorithm Security // Eastern-European Journal of Enterprise Technologies. 2024. No 1(9 (127)). P. 52–59. doi:10.15587/1729-4061.2024.295160.

4. Pornin T. Improved key pair generation for Falcon, Bat and hawk // IACR Cryptology ePrint Archive. 2023. Available at: https://eprint.iacr.org/2023/290 (Accessed: 10 November 2024).

5. Kachko O. et al. Improving protection of falcon electronic signature software implementations against attacks based on Floating Point Noise // Eastern-European Journal of Enterprise Technologie. 2024. No4(9 (130)). P. 6–17. doi:10.15587/1729-4061.2024.310521.

6. Woo C. Square root by abacus algorithm, Index of /Martin/Tape/Gos/MISC/personal/MSC/SQRT. Available at: http://freaknet.org/martin/tape/gos/misc/personal/msc/sqrt/ (Accessed: 10 November 2024).

7. Goldwasser S., Micali S. and Rivest R.L. A digital signature scheme secure against adaptive chosen-message attacks // SIAM Journal on Computing. 1988. No17(2). Pp. 281–308. doi:10.1137/0217017.

8. Hövelmanns K., Hülsing A. and Majenz C. Decryption failures and the Fujisaki-Okamoto Transform // Cryptology ePrint Archive. Available at: https://eprint.iacr.org/2022/365.pdf (Accessed: 13 October 2024).

Надійшла до редколегії 06.01.2025

Відомості про авторів:

Качко Олена Григорівна – канд. техн. наук, Харківський національний університет радіоелектроніки, професор кафедри програмної інженерії, факультет комп'ютерних наук; АТ «Інститут інформаційних технологій», начальник відділу програмування; Україна; e-mail: iit@iit.kharkov.ua, ORCID: https://orcid.org/0000-0001-9249-0497

Горбенко Іван Дмитрович – д-р техн. наук, професор, Харківський національний університет ім.В.Н. Каразіна, професор кафедри кібербезпеки інформаційних систем, мереж і технологій, навчально-науковий інститут комп'ютерних наук та штучного інтелекту, АТ «Інститут інформаційних технологій», головний конструктор; Україна; e-mail: i.d.gorbenko@karazin.ua; ORCID: https://orcid.org/0000-0003-4616-3449

Кандій Сергій Олегович – Харківський національний університет ім. В. Н. Каразіна, аспірант кафедри кібербезпеки інформаційних систем, мереж і технологій, навчально-науковий інститут комп'ютерних наук та штучного інтелекту, АТ «Інститут Інформаційних технологій», науковий консультант; Україна; e-mail: sergeykandy@gmail.com; ORCID: https://orcid.org/0000-0003-0552-8341