SYSTEMS AND METHODS OF INFORMATION PROTECTION
СИСТЕМИ І МЕТОДИ ЗАХИСТУ ІНФОРМАЦІЇ

УДК 004.056.5 **DOI:10.30837/rt.2026.1.224.01**

І.Д. ГОРБЕНКО, д-р техн. наук, О.Г. КАЧКО, канд. техн. наук,
М.В. ЄСІНА, канд. техн. наук, Я.А. ДЕРЕВ'ЯНКО

ОПТИМІЗАЦІЯ ОПЕРАЦІЙ ОБЧИСЛЕННЯ ТА ПЕРЕВІРКИ ЦИФРОВОГО ПІДПИСУ ДЛЯ СТАНДАРТУ FIPS 205. 3 ЧАСТИНА

Вступ

Аналіз показав, що обчислювальна складність алгоритму FIPS205 [1] та відповідного базового алгоритму SPHINCS+ [2] залишається слабкою стороною.

В попередніх роботах [3, 4] розглянуто можливості збільшення продуктивності алгоритму FIPS 205 за рахунок виключення загальних обчислень, особливого формату вхідних даних для обчислення гешів та паралельних обчислень за рахунок застосування потоків. Дана робота є продовженням попередніх робіт і дозволяє додатково збільшити продуктивність алгоритмів генерації ключів, вироблення та перевірки підпису на прикладі стандарту FIPS 205. Отримані результати можна застосувати для будь-якої схеми підпису, подібної до FIPS 205.

В статті розглядається використання AVX команд для обчислення гешів на прикладі застосування функцій SHA256 та SHA512, а також їх застосування при реалізації різних схем алгоритму, а саме схем одноразового підпису, розширених дерев Мерклі, гіпердерев та лісу дерев.

Автори алгоритму SPHINCS+, який покладено в основу алгоритму FIPS 205 [1], також застосовували AVX операції для реалізації алгоритмів. Їх реалізація з застосуванням цих операцій знаходиться в пакеті з вхідними текстами програм (папка Additional_Implementations [2]). В цій статті порівнюються результати оптимізації отриманої авторами SPHINCS+ (будемо далі називати її базовою реалізацією) та цієї роботи в умовах застосування AVX команд. Ця стаття, на погляд її авторів, має підвищену значимість, оскільки практично усі сучасні процесори спроможні виконувати AVX команди. Команди, для яких є обмеження з застосування, в роботі не розглядаються (наприклад, команди AVX512), або при розгляді вказано засоби для перевірки можливості застосування таких команд.

Авторський код наведено на GitHub у репозиторії [5], для спрощення порівняння він включає код авторів SPHINCS+ (папка OLD). Всі тести, наведені у роботі, проводилися на машині з процесором Intel Core i5 13600KF на ОС Windows 11. Також використовувався компілятор GCC з MinGW.

1. Оптимізація обчислення гешів (функції SHA256, SHA512)

1.1. Апаратна реалізація

Більшість сучасних процесорів мають AVX команди для реалізації криптографічного перетворення щодо алгоритму SHA256. Деякі процесори мають команди для реалізації таких операцій для SHA512. Далі застосування таких операцій будемо називати апаратною реалізацією. Застосування спеціальних пристроїв для реалізації функцій гешування в роботі не розглядається.

Для перевірки можливості застосування цих команд для обчислення SHA256 застосовують команду процесора CPUID, функцію 7, якщо регістр EBX має 1 в біті 29, то команда

підтримується. Команди для обчислення SHA512 підтримуються, якщо функція 7 команди CPUID в регістрі EAX мають 1 в біті 0.

Приклад функції для визначення можливості застосування операцій для обчислення SHA256 (мова C), рис. 1:

```
int check_sha256(){
    uint32_t r[4];              // Регістри для результату, 0 – EAX, 1 – EBX, 2 – ECX, 3 - EDX
    uint32_t mask = 1 << 29;  // Будемо перевіряти біт 29
    __cpuid ((int*)r, 0);                // Застосування функції 0 для команди команди cupid
    return (r[2] & mask) == mask;       // Перевірка значення біту 29
}
```

Рис. 1. Функція для перевірки можливості застосування AVX реалізації обчислення SHA256

Для перевірки можливості застосування операцій SHA512 необхідно замінити номер регістру та номер біту.

Приклад застосування команд процесора для обчислення гешу наведено в [6]. Часові характеристики для застосування апаратної реалізації наведено далі для порівняння з іншими методами оптимізації.

1.2. Програмна реалізація функції SHA256

Ефективність застосування AVX команд основана на можливості одночасного виконання однакових або однотипних операцій над усіма компонентами блоку. Розмір блоку залежить від типу команд і для сучасних процесорів мають значення 128, 256 та 512 бітів. Для AVX команд розмір блоку дорівнює 256 бітів, або 32 байти. Далі передбачається застосування саме AVX команд.

Криптографічне перетворення для SHA256 виконується над даними завдовжки 32 біта, що відповідає 8 компонентам блоку завдовжки 256 бітів. Для SHA512 операції виконуються над даними завдовжки 64 біта, що відповідає 4 компонентам блоку. Але і в SHA256, і в SHA512 виконуються різні операції над суміжними елементами блоку, що робить неефективним застосування AVX команд безпосередньо для криптографічного перетворення блоків, ці операції можна застосовувати тільки для перетворення формату даних з Little Endian в Big Endian та навпаки. Останні операції в порівнянні з багатокроковим криптоперетворенням, на жаль, займають значно менше часу, ось чому пряме застосування AVX команд неефективне при обчисленні гешу. Але, якщо геші треба обчислити одночасно для декількох наборів даних, то, при спеціальному представленні цих даних, такі обчислення можна виконувати паралельно не тільки за допомогою потоків (дивись [3]), а і з допомогою AVX команд. Саме такий спосіб застосовують автори [2] для збільшення продуктивності. Аналогічний спосіб застосовується і в цій роботі.

Як показано в [3], більшість алгоритмів, які застосовують SHA256 або (і) SHA512, застосовують загальний перший блок, який містить відкритий ключ алгоритму і доповнюється нулями до кінця блоку, тому криптографічне перетворення для цього блоку можна виконати заздалегідь (перед обчисленням) і застосувати результати обчислення цього блоку для криптографічного перетворення наступного блоку або блоків. Треба врахувати, що більшість операцій для обчислення гешу потребують обробки тільки двох блоків, тому застосування передобчислень для більшості випадків приводить до практично двократного збільшення продуктивності.

Для ефективного застосування AVX команд необхідно, щоб усі компоненти блоку були заповнені, ось чому в разі застосування AVX команд для SHA256 треба паралельно обчислювати геш одночасно для 8 даних. Розмір блоку, над яким виконується криптографічне перетворення в SHA256, дорівнює 64 байту (2 даних типу AVX). Рядок вхідних даних для одного криптографічного блоку задано на рис. 2 і вхідні дані для 8 наборів – на рис. 3:

in[0], in[1], …, in[7], in[8], in[9], …in[15]

Рис. 2. Рядок вхідних даних для одного криптографічного блоку

in[0][0], in[0][1], …, in[0][7], in[0][8], in[0][9], …in[0][15] in[1][0], in[1][1], …, in[1][7], in[1][8], in[1][9], …in[1][15] … in[7][0], in[7][1], …, in[7][7], in[7][8], in[7][9], …in[7][15]

Рис. 3. Набір вхідних даних для 8 наборів

Для застосування AVX команд необхідно набір даних перетворити в набір, при якому для кожного даного треба виконувати однакові операції (якщо набір даних розглядати як матрицю, то фактично треба виконати транспонування цієї матриці – рис. 4). В подальшому цей формат будемо називати транспонованим форматом.

in[0][0], in[1][0],… in[7][0] in[0][1], in[1][1], … in[7][1] … in[0][7], in[1][7], … in[7][7] in[0][8], in[1][8], … in[7][8] … in[0][15], in[1][15], … in[7][15]

Рис. 4. AVX формат для блоків даних

При такому упорядкуванні даних кожне AVX дане містить один відповідний елемент з усіх вхідних повідомлень. Операції, які треба виконати для елементу блоку, залежать тільки від його номеру в повідомленні, тобто для елементів кожного AVX даного треба виконувати одну й ту ж операцію. Усі необхідні перетворення форматів бажано виконувати за границями обробки блоку, що дає змогу не виконувати зайвих операцій, коли необхідно виконати ланцюжок з обчислення гешів (функція AVX_sha256_compress8).

Фактично функція для обчислення гешу складається з двох частин:

- обчислення w (64 AVX даних), з яких перші 16 – це вхідні дані (функція AVX_sha256_calc_w8);
- обчислення наступного стану з попереднього, попередній стан визначається або початковим станом для sha256, або результатом передобчислень (функція AVX_sha256_calc_state8).

1.3. Особливості програмної реалізації функції SHA512

1. Розмір елементу для криптографічних перетворень дорівнює 64 біта (8 байтів), тому криптографічний блок виконує паралельно операції для 4 замість 8 наборів даних.

2. Усі AVX операції виконуються для елементів блоку завдовжки 8 байтів замість 4 байтів.

3. Розмір блоку для SHA512 дорівнює 128 байтів, що відповідає чотирьом AVX даним замість двох для SHA256.

В зв’язку з цими особливостями транспоновані формати для SHA256 та SHA512 не співпадають.

В разі, якщо n = 16, що відповідає мінімальній криптографічній складності, для усіх схем застосовують SHA256. Для n = 24, 32 функції часто застосовують і SHA256, і SHA512. Результат обчислення геш значення за допомогою SHA256 (транспоноване значення, відповідне функції SHA256) перетворюється в пару транспонованих значень для функції SHA512. Зворотний порядок застосування функцій гешування алгоритм не застосовує. Застосування внутрішніх форматів і їх перетворення дозволяють уникнути зайвих операцій (функції convert…).

1.4. Результати оптимізації функцій гешування

Як і в попередніх роботах цього циклу [3, 4] для виміру часу застосовують такти процесору.

1.4.1. Порівняння апаратної та програмної реалізації функцій гешування (SHA256). Результати порівняння наведено в табл. 1

Таблиця 1

SHA256. Апаратна та програмна реалізація

Функція	Продуктивність (такти)
sha256 (На основі crypto_hash/sha512/ref/ з http://bench.cr.yp.to/supercop.html від D. J. Bernstein. Застосовується в SPHINCS+, в подальшому називаємо sha256(SPHINCS))	6780855
AVX_sha256	7629855
AVX_sha256_device	1473721

Час виміряється для повідомлення завдовжки 1 МБ (104857 байтів).

Друга та третя функції реалізовані авторами [5]. В другій функції застосовуються AVX операції для перетворення форматів, третя функція застосовує AVX команди _mm_sha256msg1_epu32, _mm_sha256msg2_epu32 та _mm_sha256rnds2_epu32 для криптоперетворень блоків.

Два варіанти програмної реалізації практично не відрізняються за продуктивністю, різниця в середньому 10 %, а апаратна реалізація значно ефективніше програмної, майже в 5 разів.

В и с н о в о к . В разі, коли процесор підтримує апаратну реалізацію функцій гешування, її застосування може дати суттєве покращення продуктивності.

Незважаючи на вражаючі результати, для застосування апаратної реалізації обчислення гешу далі в функціях застосовуємо програмну реалізацію, що пов'язано з необхідністю можливості застосування функцій для будь-яких процесорів [7].

1.4.2. Порівняння паралельних обчислень для sha256.

Результати порівняння наведено в табл. 2

Таблиця 2

SHA256. Послідовна та паралельна реалізація

Функція	Продуктивність (такти)
Базова реалізація, послідовне обчислення	2076
Базова реалізація, паралельне обчислення	1395
AVX_sha256_WITH_PREDCALC, паралельне обчислення	1072

Результати показали, що паралельне обчислення для функції thashx8 перевершує послідовний варіант за продуктивністю на 49 %, а функція AVX_SHA256_WITH_PREDCALC8 – на 94 %, тобто останній варіант більш ефективний на 30 %, що дуже важливо, тому що цей блок застосовують практично усі функції, які допускають паралельні обчислення.

2. Оптимізація обчислення для схеми одноразового підпису (WOTS+)

Для цієї схеми виконуються наступні функції:

- WOTS+ Public-Key Generation генерація відкритого ключа. Функцію застосовують при генерації ключів та формуванні підпису;
- WOTS+ Signature Generation**.** Функцію застосовують при формуванні підпису;

- Computing a WOTS+ Public Key From a Signature – функцію застосовують для перевірки підпису.

Усі функції застосовують параметри n (16, 24, 32) в залежності від криптостійкості та параметр len = 2n + 3. Параметр len визначає кількість ключів. Для паралельної обробки даних застосовують блоки, які включають 8 ключів. Для забезпечення максимального паралелізму застосовують — порцій. Це в свою чергу забезпечує паралельне обчислення геш значень за допомогою AVX команд.

Далі розглянуто особливості оптимізації кожної з цих функцій. Результати оптимізації для кожної з цих функцій будуть надані після їх розгляду (п. 2.4)

2.1. Оптимізація функції генерації відкритого ключа (WOTS+ Public-Key Generation)

Вхід:

SK_seed – секретний ключ, рядок байтів завдовжки n байтів (n = 16, 24, 32) в залежності від криптостійкості;

PK_seed – відкритий ключ, рядок байтів завдовжки n байтів (n = 16, 24, 32) в залежності від криптостійкості;

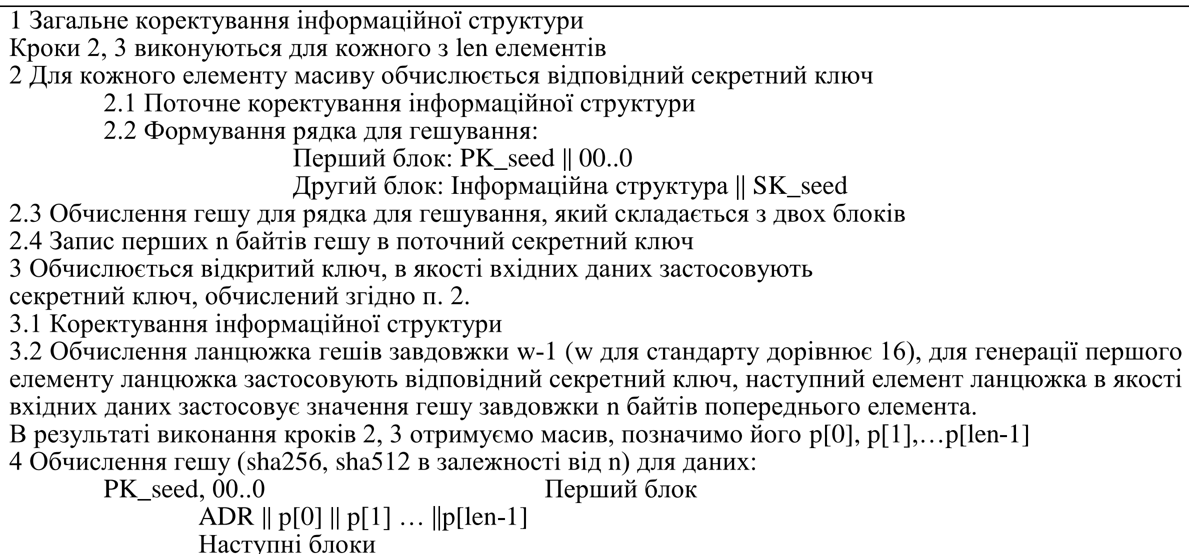
ADR – інформаційний блок завдовжки 22 байти (Інформаційний блок для Стандарту має довжину 32 або 22 байти в залежності від засобів гешування. В разі застосування функцій SHA256, SHA512 застосовують довжину 22 байти).

Вихід:

PK – відкритий ключ для WOTS+, рядок байтів завдовжки n байтів (n = 16, 24, 32) в залежності від криптостійкості.

Алгоритм можна поділити на дві частини:

- обчислення масиву відкритих ключів завдовжки len (len = 2 * n + 3);
- обчислення загального відкритого ключа, для якого обчислюється загальний геш для всього масиву відкритих ключів, отриманих на попередньому етапі (рис. 5).



1 Загальне коректування інформаційної структури
Кроки 2, 3 виконуються для кожного з len елементів
2 Для кожного елементу масиву обчислюється відповідний секретний ключ
2.1 Поточне коректування інформаційної структури
2.2 Формування рядка для гешування:
Перший блок: PK_seed || 00..0
Другий блок: Інформаційна структура || SK_seed
2.3 Обчислення гешу для рядка для гешування, який складається з двох блоків
2.4 Запис перших n байтів гешу в поточний секретний ключ
3 Обчислюється відкритий ключ, в якості вхідних даних застосовують секретний ключ, обчислений згідно п. 2.
3.1 Коректування інформаційної структури
3.2 Обчислення ланцюжка гешів завдовжки w-1 (w для стандарту дорівнює 16), для генерації першого елементу ланцюжка застосовують відповідний секретний ключ, наступний елемент ланцюжка в якості вхідних даних застосовує значення гешу завдовжки n байтів попереднього елемента.
В результаті виконання кроків 2, 3 отримуємо масив, позначимо його p[0], p[1],…p[len-1]
4 Обчислення гешу (sha256, sha512 в залежності від n) для даних:
PK_seed, 00..0 Перший блок
ADR || p[0] || p[1] … ||p[len-1]
Наступні блоки

Рис. 5. WOTS+. Генерація масиву відкритих ключів

Основний час витрачається для виконання кроків 2, 3, тому далі розглянуто оптимізації саме цієї частини.

2.1.1. Оптимізація генерації масиву відкритих ключів.

Для генерації масиву відкритих ключів (рис. 5) спочатку обчислюється відповідний секретний ключ, потім для секретного ключа виконується ланцюжок обчислення гешів завдовжки w-1 (параметр w дорівнює 16 незалежно від криптостійкості). На кожному кроці обчислень коригується інформаційна структура.

Для забезпечення найбільшого рівня паралелізму:

- обчислення секретних (крок 2) та відкритих ключів (крок 3) виконується в окремих циклах, що забезпечує можливість паралельного обчислення секретних ключів та паралельного обчислення відповідних відкритих ключів;
- з урахуванням властивостей AVX операцій розмір порції для паралельного обчислення дорівнює 8 елементів, остання неповна порція доповнюється додатковими елементами, тобто кількість порцій для обчислення ключів дорівнює — .

Спочатку розглянемо оптимізацію кроку 2 з урахуванням того, що він виконується для усіх елементів масиву, а потім – крок 3.

2.1.2. Генерація масиву секретних ключів.

При генерації секретних ключів перший блок для гешування залежить тільки від PK_seed, що залишається постійним після генерації ключів. Значення гешу для цього блоку обчислюється і застосовується в якості параметру функції генерації замість PK_seed. Саме це значення застосовують в якості попереднього стану при обробці другого блоку вхідних даних.

Для забезпечення паралельного виконання робиться транспонування для блоків даних, при цьому необхідно врахувати потребу зміни номеру ключа відповідно функції setChainAddress. Операція транспонування значно спрощується в умовах, коли фактично для різних ключів змінюється тільки значення номеру ключа.

Після обчислення секретних ключів немає потреби переводити їх в звичайний формат, наступний блок застосовує їх у внутрішньому (транспонованому) форматі.

2.1.3. Генерація масиву відкритих ключів (функція FIPS205_wots_gen_pk_new__).

При генерації відкритих ключів в разі застосування транспонованого формату необхідно вирішити наступні проблеми:

- зміна типу для адресної структури (функція setTypeAndClear);
- відновлення та відповідна зміна номерів ключів, номер ключа встановлюються функцією setChainAddress;
- формування номеру ітерації в ланцюжку обчислень геш функції згідно з функцією setHashAddress;
- заміна початкового стану для наступної ітерації в обчисленні ланцюжка гешів.

2.2. Оптимізація функції WOTS+ Signature Generation

Функція WOTS+ Signature Generation генерує підпис, який складається з len байтових рядків завдовжки n байтів.

Вхід.

SK_seed – секретний ключ, рядок байтів завдовжки n байтів;

PK_seed – відкритий ключ, рядок байтів завдовжки n байтів;

ADR – інформаційна структура завдовжки 22 байти в разі застосування SHA функцій для довільної криптостійкості;

Msg – повідомлення, рядок байтів завдовжки n байтів.

Вихід.

Підпис – sign[0], sign[1], …, sign[len – 1].

Алгоритм зі Стандарту представлено на рис. 6

```
1 Кодування Msg за допомогою масиву цілих чисел msg завдовжки len, кожне число в інтервалі
   0..w – 1 (w = 16).
2 Формування адресної структури skADRS  згідно з ADR;
3 Для кожного цілого числа з масиву msg[i]
   3.1 Коректування skADRS згідно з номером числа (функція setChainAddress)
   3.2 Формування секретного ключа (крок 2  попереднього алгоритму);
   3.3 Обчислення ланцюжка гешів (крок 3.2 попереднього алгоритму)
```

Рис. 6. WOTS+. Генерація підпису

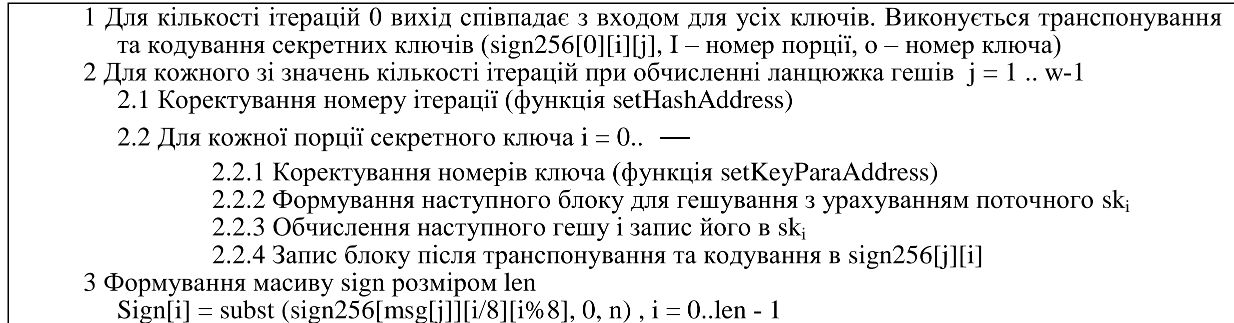
Особливістю цього алгоритму в порівнянні з попереднім є необхідність обчислення для ланцюжка зі змінною довжиною, яка визначається значенням msg_i. Далі розглянуто спосіб оптимізації саме цього кроку.

В х і д .

msg[0], msg[1], …msg[(len + 7)/8 - 1];

sk[0], sk1], …sk[(len + 7)/8 - 1].

Алгоритм для кроку обчислення підписів за секретним ключем наведено на рис. 7.



1 Для кількості ітерацій 0 вихід співпадає з входом для усіх ключів. Виконується транспонування та кодування секретних ключів (sign256[0][i][j], I – номер порції, o – номер ключа)
2 Для кожного зі значень кількості ітерацій при обчисленні ланцюжка гешів j = 1 .. w-1
2.1 Коректування номеру ітерації (функція setHashAddress)
2.2 Для кожної порції секретного ключа i = 0.. —
2.2.1 Коректування номерів ключа (функція setKeyParaAddress)
2.2.2 Формування наступного блоку для гешування з урахуванням поточного sk_i
2.2.3 Обчислення наступного гешу і запис його в sk_i
2.2.4 Запис блоку після транспонування та кодування в sign256[j][i]
3 Формування масиву sign розміром len
Sign[i] = subst (sign256[msg[j]][i/8][i%8], 0, n) , i = 0..len - 1

Рис. 7. WOTS+. Обчислення підпису за результатом кодування та секретним ключем

Такий спосіб обробки дозволяє застосовувати блоки ключів для генерації секретних ключів, що значно спрощує їх застосування без попереднього переформатування.

2.3. Обчислення відкритого ключа за підписом

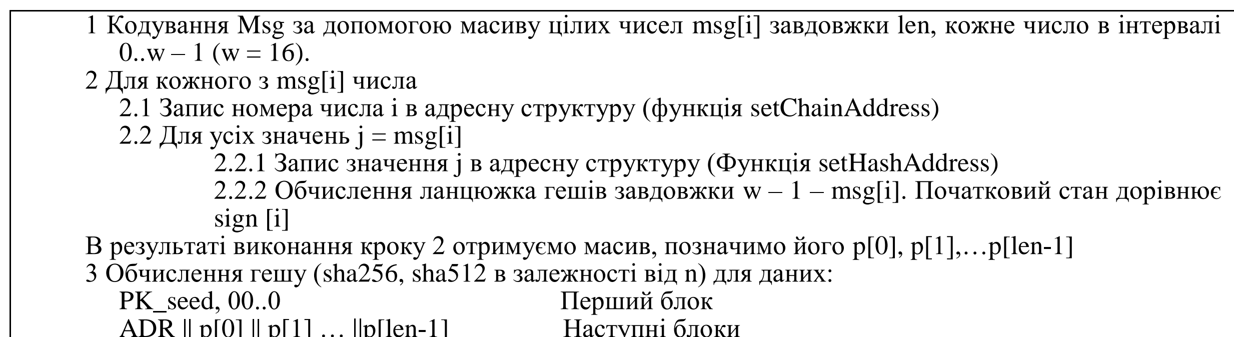
Перший з розглянутих алгоритмів цієї групи обчислював відкритий ключ за допомогою виконання ланцюжка гешів для кожного секретного ключа. Довжина ланцюжка дорівнювала w – 1. Наступний алгоритм обчислював підписи, тобто замість ланцюжка завдовжки w – 1 обчислював геші зі змінною довжиною ланцюжка. Ця довжина дорівнювала значенню msg[i] від 0 до w – 1. Очевидно, якщо виконати ітерації завдовжки w – 1 - msg[i], яких не вистачає, то отримаємо значення відкритого ключа. Саме так обчислюється відкритий ключ в функції перевірки підпису, коли секретний ключ невідомий і не може бути обчисленим. Саме цю операцію виконує поточна функція.

В х і д .

Sign – масив підписів, рядків байтів завдовжки n байтів. Масив має len елементів;

Msg – повідомлення, рядок байтів завдовжки n байтів. Результатом кодування цього повідомлення є масив цілих чисел, який визначає довжини для ланцюжків. Застосовують один і той же масив для функції обчислення підписів і поточної функції.

Алгоритм представлено на рис. 8.



1 Кодування Msg за допомогою масиву цілих чисел msg[i] завдовжки len, кожне число в інтервалі 0..w – 1 (w = 16).
2 Для кожного з msg[i] числа
2.1 Запис номера числа i в адресну структуру (функція setChainAddress)
2.2 Для усіх значень j = msg[i]
2.2.1 Запис значення j в адресну структуру (Функція setHashAddress)
2.2.2 Обчислення ланцюжка гешів завдовжки w – 1 – msg[i]. Початковий стан дорівнює sign [i]
В результаті виконання кроку 2 отримуємо масив, позначимо його p[0], p[1],…p[len-1]
3 Обчислення гешу (sha256, sha512 в залежності від n) для даних:
PK_seed, 00..0 Перший блок
ADR || p[0] || p[1] … ||p[len-1] Наступні блоки

Рис. 8. WOTS+. Генерація відкритого ключа за підписом

Найбільший час при виконанні потребує крок 2 алгоритму.

Неефективним є прийом, коли обчислювалися значення для повного набору ітерацій, а потім обиралися потрібні. В даному випадку змінюється початковий номер ітерацій, саме він записується в інформаційну структуру і впливає на значення гешу, для оптимізації цього кроку виконано наступне.

Виконується упорядкування кодів в порядку зростання номеру ітерації (в даному випадку фактично виконується w – 1 – msg[i], тобто, якщо msg[i] дорівнює 0, треба виконати максимальну кількість ітерацій). Для організації повторного виконання, одночасного для кожного значення msg[i], формується кількість повторів. В зв’язку з особливостями масиву (обмежена кількість елементів – len, обмежено значення елементів 0. w-1) упорядкування такого масиву суттєво спрощено (автори SPHINCS+ застосовували такий же прийом для визначення обсягу робіт в цій функції).

На рис. 9 наведено алгоритм для кроку 2 з урахуванням зауважень та паралельного виконання. Як і для попередніх алгоритмів блок складається з 8 підписів. Для поточного блоку відповідно масиви inumbers та iters містять номери підписів та відповідну їм кількість ітерацій для функції гешування, яка виконана при генерації підпису. Спочатку обробляються записи, для яких виконано мінімальну кількість ітерацій, вони потребують найбільшу кількість ітерацій. Відповідні значення в масиві iters дорівнюють 0.

Визначення first_iter та last_iter дозволяють прискорити роботу за рахунок того, що цикл для first_iter не потребує збереження проміжних даних, їх збереження необхідно, якщо є відповідні коди, наприклад, якщо блок починається з кількості ітерацій 3, і закінчується кількістю 5, то це означає, що при генерації підпису виконано 3 ітерації, тобто нам не потрібні результати виконання для 0, 1, 2 ітерацій. Фактично потрібно запам’ятати результати для випадків, коли виконано 3, 4, 5 ітерацій.

Зберігаються дані тільки для одного блоку, після завершення його обробки результати записуються в масив результатів.

1 Упорядкування кодів msg[0], msg[1], … msg [(len + 7)/8 * 8 - 1] (в доповнені елементи кодів записується значення 15, що відповідає відсутності додаткових ітерацій). В результаті отримаємо масиви упорядкованих кодів (iters) і відповідних номерів підписів (inumbers).
2 Для кожної порції підписів (I = 0, 1, … (len + 7)/8)
 2.1 Коректуються номери підпису для поточної порції (записуються поточні 8 елементів з масиву inumbers)
 2.2 В блок завантажуються підписи з номерами, які відповідають поточній порції (елементи масиву inumbers)
 2.3 Ці підписи записуються в масив sign256 [0], що відповідає випадку, коли не треба нічого робити, кількість додаткових ітерацій дорівнює 0
 2.4 Ці Підписи записуються в блок для обчислення наступних гешей
 2.5 Коректуються номера ітерацій для блоку (поточні 8 елементів з масиву iter)
 2.6 Визначається мінімальний номер first_iter та максмальний last_iter (перший та останній елемент для номерів ітерацій)
 2.7 Цикл ітерацій. Кількість ітерацій дорівнює w – 1 – last_iter
  2.7.1 Завантаження початкового стану
  2.7.2 Обчислення гешу зі зміною початкового стану
  2.7.3 Коректування номеру ітерації
 2.8 Збереження поточного стану для ітерації w – 1 – last_iter
 2.9 Цикл для додаткових ітерацій (кількість дорівнює last_iter - first_iter)
  2.9.1 Завантаження початкового стану
  2.9.2 Обчислення гешу зі зміною початкового стану
  2.9.3 Збереження поточного стану для поточної ітерації
  2.9.4 Коректування номеру ітерації
 2.10 Запис значень відкритого ключа для порції підписів

Рис. 9. Оптимізований алгоритм формуванні відкритих ключів за підписом

2.4. WOTS+. Результати оптимізації

Результати оптимізації наведено в табл. 3.

Три результати, задані в таблиці для кожного варіанту, відповідають значенням n = 16, 24, 32, які визначають криптостійкість. Усі дані отримано для параметрів, які оптимізовані за пам’яттю. Як видно з табл. 3, обчислювальна складність останньої реалізації менше ніж для стандартної.

Таблиця 3

Результати оптимізації WOTS+

Функція	Базова реалізація	Авторська реалізація	Прискорення
Генерація відкритого ключа	90928 151581 169748	71251 100031 131701	1.28 1.51 1.28
Генерація підпису	197173 298574 359236	75849 105140 133201	2.60 2.83 2.69
Генерація відкритого ключа за підписом	46068 73224 80480	38127 51819 63908	1.20 1.41 1.25

Найбільша перевага спостерігається для функції обчислення підпису (не менше ніж в 2 рази). Для решти функцій перевага не менше ніж на 20 %

Отримані результати дуже важливі, тому що ці функції застосовують на всіх етапах генерації ключів, формування та перевірки підпису.

3. Оптимізація обчислення для розширених дерев Мерклі (XMSS)

Розширене дерево є компонентом гіпердерева. Має $2^{h'}$ листів.

Компонентом розширеного дерева є WOTS схеми, ось чому в якості параметрів застосовуються, крім параметру *h'*, параметри *len* та *n*, які застосовуються для WOTS схем.

Підпис для цієї схеми складається:

- з підпису для WOTS схеми (нижній рівень), містить *len * n* байтів;
- шляху аутентифікації для кожного з *h'* рівнів, по одному рядку байтів для кожного рівня.

Загальна довжина підпису $len * n + h' * n = (len + h') * n$.

Для цієї схеми виконуються функції:

- Generating a Merkle Hash Tree – Генерація дерева – фактично застосовується для обчислення кореня дерева для генерації відкритого ключа;
- Generating an XMSS Signature – Генерація підпису – застосовується як компонент для генерації підпису гіпердерева;
- Computing an XMSS Public Key From a Signature – Обчислення відкритого ключа за підписом – застосовують для перевірки підпису для гіпердерева – частина функції перевірки підпису.

3.1. Генерація XMSS дерева

Для генерації дерева ми не застосовуємо рекурсивну функцію, яка пов'язана з додатковими накладними витратами.

Для цього спочатку обчислюються усі листи дерева, а потім поступово обчислюються вузли наступних рівнів. Вузли кожного наступного рівня заміщають відповідні дані для попереднього рівня. Обчислення листів виконується паралельно для блоків розміром 8 листів.

3.2. Генерація підпису

Для генерації підпису необхідно сформувати підпис для WOTS дерева та шлях аутентифікації для кожного рівня. Оптимізовану функцію для генерації підпису для WOTS дерева визначено в п. 2.2.

Для генерації шляху аутентифікації знову обчислюються усі листи, і за вхідним номером листа визначається номер листа для аутентифікації для рівня 0, який записується як відповідний результат. Далі для кожного наступного рівня обчислюються вузли і визначається номер вузла для аутентифікації. Вузли наступного рівня записуються замість листів поточного рівня.

3.3. Генерація відкритого ключа за підписом

Основну частину цієї функції складає функція обчислення відкритого ключа для WOTS схеми, оптимізація якої описана в п. 2.3

3.4. Розширене дерево XMSS. Результати оптимізації

Результати оптимізації наведено в табл. 4, вони містять дані для функцій генерації дерева, підпису та генерації відкритого ключа за підписом для розширених дерев. В стандартній реалізації відсутні функції генерації підпису та обчислення відкритого ключа. Що стосується функції генерації дерева – вона фактично застосовується для генерації кореня дерева (функція генерації відкритого ключа), яка є в стандартній реалізації.

Таблиця 4

Результати оптимізації XMSS

Функція	Базова реалізація	Авторська реалізація	Прискорення
Генерація дерева	49525555	6594493	7.51
	72267555	8006679	9.02
	45668458	6041922	7.55
Генерація підпису	-	6757903	
	-	8790423	
	-	6376797	
Обчислення відкритого ключа за підписом	-	51781	
	-	66734	
	-	87851	

Як видно з табл. 4, для генерації дерева прискорення не менше ніж в 7 разів.

4. Оптимізація обчислень для гіпердерева

Гіпердерево застосовують для підпису відкритих ключів дерев з лісу. Застосовується властивість застосування одного ключа для підпису ключів, якщо в якості цих ключів застосовують відкриті ключі з лісу. Ось чому гіпердерево – це дерево, в якому в якості листів застосовують XMSS дерева. На найнижчому рівні застосовується відкритий ключ дерева лісу, всього гіпердерево має d рівнів, таким чином загальна кількість ключів, які можна підписати, дорівнює = . Саме цей вираз визначає кількість підписів, тобто кількість повідомлень, які можна підписати за допомогою одного секретного ключа.

Підпис для гіпердерева складається з підписів XMSS дерев на кожному з d рівнів, довжина підпису $d * (len + h') * n$.

Для гіпердерева визначені функції:

- Hypertree Signature Generation – генерація підпису для гіпердерева;
- Hypertree Signature Verification – перевірка підпису для гіпердерева.

Ці функції безпосередньо застосовують функції для розширеного дерева, додаткових засобів оптимізації не застосовується.

Результати оптимізації наведено в табл. 5, вони містять дані для функцій генерації підпису та генерації відкритого ключа за підписом для гіпердерева.

Як видно з табл. 5, для усіх функцій базова реалізація поступається за обчислювальною складністю авторській. Оптимізація генерації підпису, запропонована у роботі, дозволяє отримати прискорення не менше ніж у 7 разів.

Таблиця 5

Результати оптимізації обчислень для гіпердерева

Функція	Базова реалізація	Авторська реалізація	Прискорення
Генерація підпису	347873239	43993506	7.91
	504926543	50084671	10.08
	367203659	40967414	8.96
Перевірка підпису	412704	328848	1.25
	604663	430019	1.41
	849879	644794	1.31

5. Оптимізація обчислень для лісу дерев (FORS)

Ліс дерев безпосередньо пов’язаний з повідомленням, що підписується. Для цього повідомлення формується *digest* – рядок байтів постійної довжини, який визначає індекс дерева та індекс листа, які будуть задаватися далі в інформаційній структурі, та склад дерев в лісі (всього кількість дерев визначається параметром *k*). Кожне дерево є двійковим деревом, висотою *a*, тобто на нижньому рівні має 2^a листів.

Для кожного дерева в лісі визначається відповідний відкритий ключ (корінь відповідного двійкового дерева, root [i]) та загальний відкритий ключ за рахунок обчислення загального геш значення:

Pk = hash (root[0] || root[1] ||…).

Саме це значення застосовують в якості початкового значення для роботи з гіпердеревом.

Для лісу визначено функції:

- Generating FORS Secret Values – обчислення секретного ключа для заданого дерева з лісу, ця функція застосовує секретний ключ SK_seed, тому може бути виконана на етапі генерації підпису, включається в склад підпису. Для рядка, який включає SK_seed, обчислюється геш значення, тому це не може бути застосовано в якості додаткових знань про SK_seed;
- Generating a Merkle Hash Tree – Генерація дерева Мерклі – ця функція обчислює вузли дерева. Вона схожа з функцією генерації XMSS дереву, але в якості вузлів застосовують або секретні ключі FORS дерева або відкриті ключі WOTS дерева;
- Generating a FORS Signature – Генерація підпису для FORS дерева. Для кожного з k дерев цифровий підпис містить: лист – відповідний секретний ключ та для наступних рівнів – шлях аутентифікації. Загальна довжина підпису дорівнює $k * (n + a * n)$;
- Computing a FORS Public Key From a Signature – обчислення відкритого ключа за підписом. Функція не потребує знань секретного ключа SK_seed і застосовується на етапі перевірки підпису.

Оптимізація алгоритмів може виконуватись за рахунок незалежних обчислень для кожного дерева з лісу (цикл по k) або за рахунок паралельних обчислень для усіх вузлів дерева на кожному рівні, в цьому випадку обробка рівнів виконується послідовно:

- перший спосіб передбачає застосування потоків, один потік відповідає одному дереву, кожний потік виконує усі операції для різних рівнів дерева;
- другий спосіб забезпечує можливість паралельного обчислення гешів для усіх вузлів на кожному рівні.

В роботі застосовано обидва методи.

Алгоритм в разі застосування паралельної обробки дерев надано на рис. 10. Алгоритм однаковий для усіх дерев в лісі.

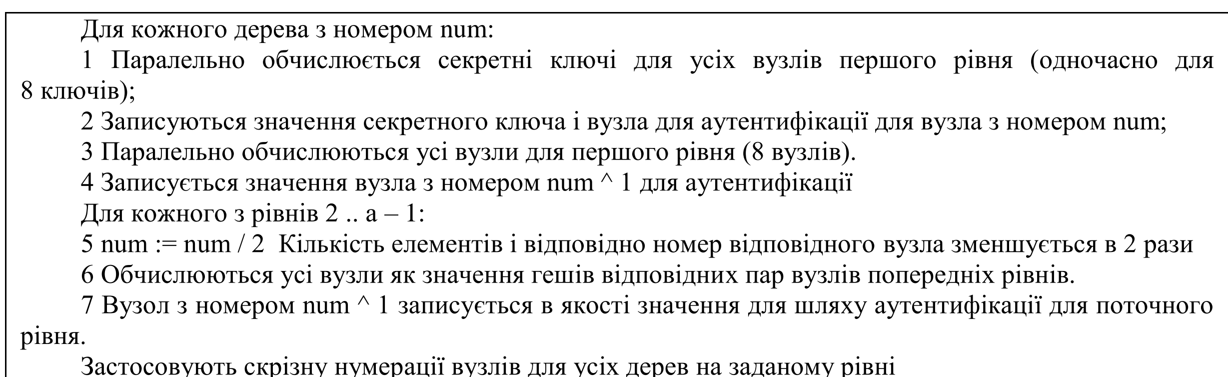

Для кожного дерева з номером num:
1 Паралельно обчислюється секретні ключі для усіх вузлів першого рівня (одночасно для 8 ключів);
2 Записуються значення секретного ключа і вузла для аутентифікації для вузла з номером num;
3 Паралельно обчислюються усі вузли для першого рівня (8 вузлів).
4 Записується значення вузла з номером num ^ 1 для аутентифікації
Для кожного з рівнів 2 .. a – 1:
5 num := num / 2 Кількість елементів і відповідно номер відповідного вузла зменшується в 2 рази
6 Обчислюються усі вузли як значення гешів відповідних пар вузлів попередніх рівнів.
7 Вузол з номером num ^ 1 записується в якості значення для шляху аутентифікації для поточного рівня.
Застосовують скрізну нумерації вузлів для усіх дерев на заданому рівні

Рис. 10. Алгоритм паралельної обробки дерев

Алгоритм в разі застосування паралельної обробки вузлів усіх дерев показано на рис. 11.

```
1 Операції, які не залежать від рівня та для першого рівня
  Обчислення індексу для дерева, листа та номерів дерев в лісі
  Обчислення секретних ключів для усіх дерев (за допомогою одночасного обчислення гешів для
  8 дерев (AVX операції) )
  Запис секретних ключів в підпис з відповідним зміщенням
2 Цикл для кожного рівня (j = 0, 1, 2, a - 1)
  Коректування індексів дерев з урахуванням рівня
  Обчислення вузлів та запис їх в підпис
  If j = 0 then                                    Обробка рівня 0
          Обчислення вузлів для рівня 0  (для усіх дерев паралельно – AVX)
  Else
          Рекурсивна функція обчислення вузла (для усіх дерев паралельно – AVX)
  Endif
  Запис значення вузла як компонент шляху автентифікації
```

Рис. 11. Алгоритм паралельної обробки вузлів усіх дерев

У алгоритмі забезпечено перетворення даних для паралельного обчислення гешів в разі застосування sha256 та sha512. З оптимізацією для решти функцій можна ознайомитись у [5].

В табл. 6 наведено результати оптимізації для функцій, які далі застосовують для генерації та перевірки підпису. Наведено результати для функції з найкращою швидкодією порівняно з базовою реалізацією.

Таблиця 6

Результати оптимізації обчислень для FORS

Функція	Базова реалізація	Авторська реалізація	Прискорення
Генерація підпису	29086178 218074699 229448703	5252033 20503025 26989360	5.53 10.64 8.50
Генерація відкритого ключа за підписом	105825 240167 311711	30750 95141 106555	3.44 2.52 2.93

Як видно з табл. 5, для будь-якого режиму отримані результати кращі ніж результати для базової реалізації. Оптимізація генерації підпису, запропонована у роботі, дозволяє отримати прискорення не менше ніж у 5 разів.

6. Оптимізація основних операцій

Основними операціями алгоритму є операції:

- генерації ключів;
- формування підпису;
- перевірки підпису.

Далі наведено результати оптимізації цих функцій. Результати наведено для режимів оптимізації за пам'яттю – STORE (табл. 7) і за часом – FAST (табл. 8).

Таблиця 7

Результати для основних операцій (STORE)

Функція	Базова реалізація	Авторська реалізація	Прискорення
Генерація ключів	49631526 71945487 45571401	6313459 8621202 5874365	7.86 8.35 7.76
Формування підпису	379581457 736931062 582495158	38172793 73124656 68571217	9.94 10.07 8.49
Перевірка підпису	513915 799941 1131844	355175 520785 764040	1.44 1.54 1.48

Таблиця 8

Результати для основних операцій (FAST)

Функція	Базова реалізація	Авторська реалізація	Прискорення
Генерація ключів	758001 1090295 2826604	303153 370213 718350	2.5 2.95 3.93
Формування підпису	19244083 31662512 61396594	10279401 13721292 16217608	1.87 2.31 3.79
Перевірка підпису	1283628 1976826 1986563	1005613 1415915 1455489	1.27 1.40 1.36

Для більшої наочності порівняння отриманих прискорень дані з таблиць наводяться у вигляді гістограм на рис. 12 – 17.

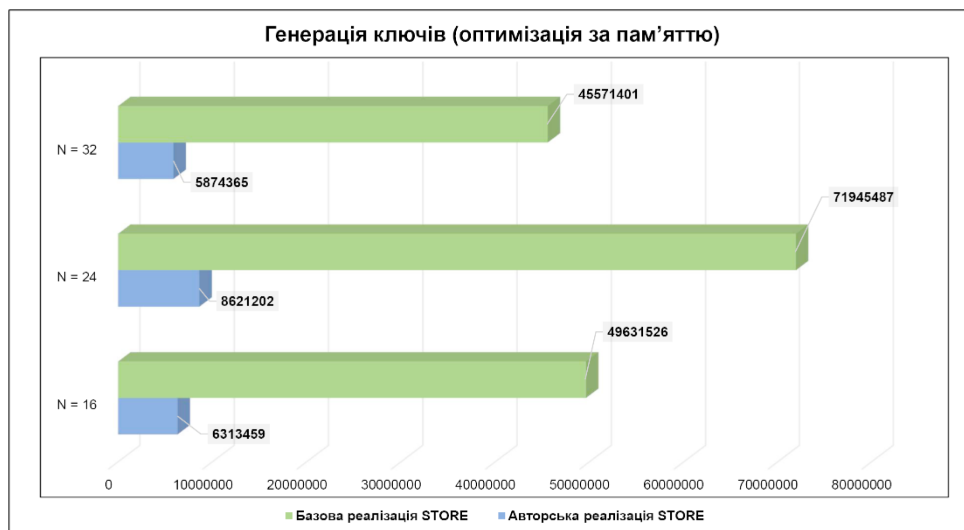


Рис. 12. Кількість циклів процесора на виконання операції генерації ключів для режиму STORE

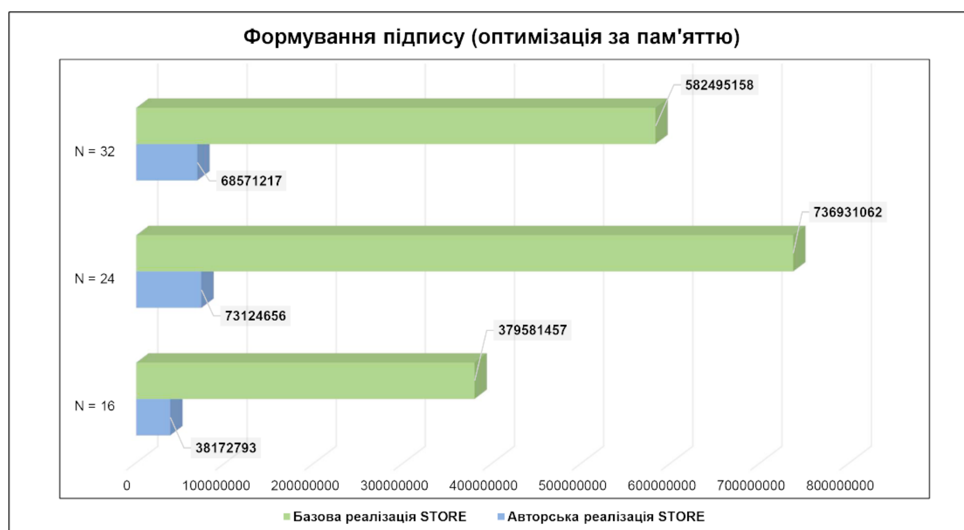


Рис. 13. Кількість циклів процесора на виконання операції формування підпису для режиму STORE

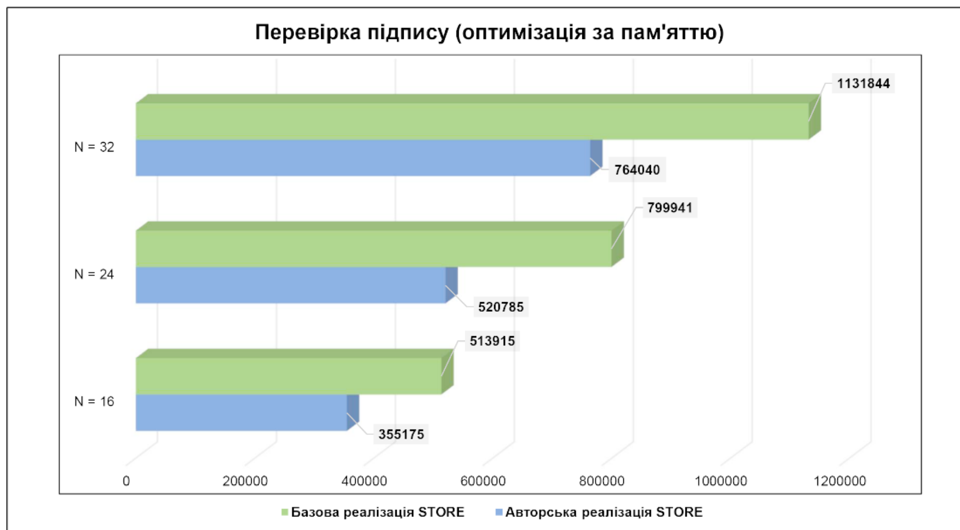


Рис. 14. Кількість циклів процесора на виконання операції перевірки підпису для режиму STORE

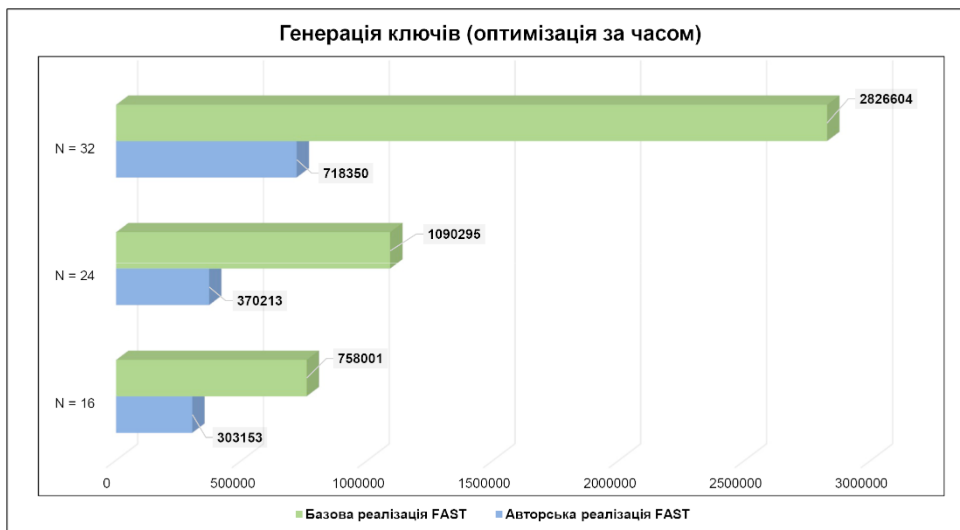


Рис. 15. Кількість циклів процесора на виконання операції генерації ключів для режиму FAST

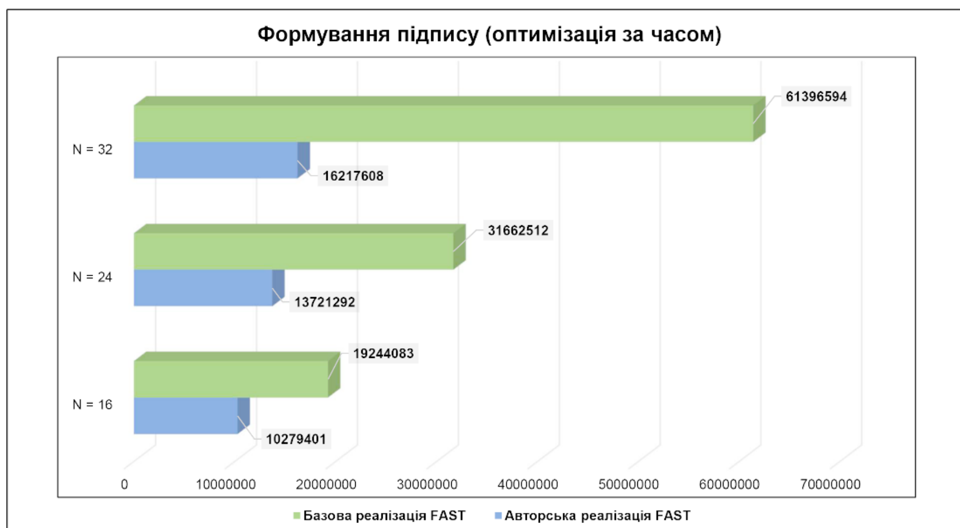


Рис. 16. Кількість циклів процесора на виконання операції формування підпису для режиму STORE

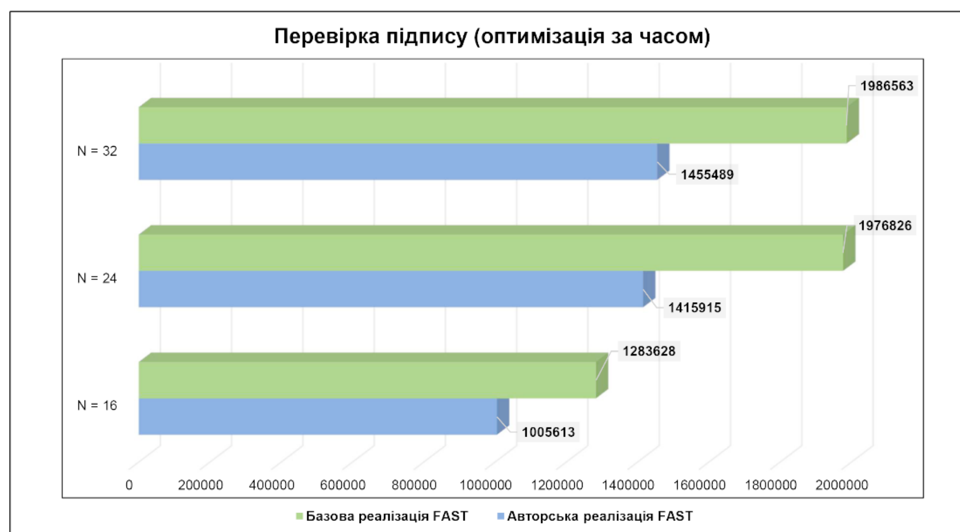


Рис. 17. Кількість циклів процесора на виконання операції перевірки підпису для режиму STORE

Висновки

1. Розглянуто застосування AVX команд для обчислення гешів на прикладі застосування функцій SHA256 та SHA512, а також їх застосування при реалізації різних схем алгоритму. Така оптимізація алгоритмів та застосування AVX операцій дозволяє додатково збільшити продуктивність алгоритмів генерації ключів, вироблення та перевірки підпису для FIPS 205. Крім того, отримані результати можна застосувати для будь-якого підпису, подібного FIPS 205.

2. Завдяки оптимізації алгоритмів та застосуванню AVX отримано суттєві прискорення при виконанні усіх операцій.

3. Для генерації ключової пари отримано прискорення в 7,76 – 8,35 раза для режиму STORE та в 2,5 – 3,93 – для режиму FAST.

4. Для формування підпису було отримано прискорення практично в 10 разів для режиму STORE та в 1.87 – 3.79 – для режиму FAST;

5. Для перевірки підпису отримано прискорення не менше ніж в 1,4 рази для режиму STORE та 27 % – для режиму FAST.

6. Отримані результати є доволі перспективними і показують, що застосування паралельних обчислень на багатоядерних процесорах суттєво збільшує продуктивність функцій для схем WOTS, XMSS та FORS, а також функцій, які їх застосовують.

Список літератури:

1. Stateless Hash-Based Digital Signature Standard, FIPS 205, 2024 [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf

2. NIST PQC. Round 3 Submissions. Алгоритм SPHINCS, Optimized_Implementation. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions

3. Gorbenko I., Kachko O., & Derevianko Y. Optimization of digital signature calculation and verification operations for the FIPS 205 standard // Radiotekhnika. 2025. No 221. P. 7–13. https://doi.org/10.30837/rt.2025.2.221.01

4. Gorbenko I., Kachko Y., & Derevianko Y. Optimization of digital signature calculation and verification operations for the FIPS 205 standard. Part 2 // Radiotekhnika. 2025. No 222). P. 7–21. https://doi.org/10.30837/rt.2025.3.222.01

5. Implementation of FIPS 205 PARALLEL. 2025 [Online]. Available: https://github.com/DereviankoYaroslav/FIPS_205_PARALLEL

6. Saarinen M-J. (2024). Accelerating SLH-DSA by Two Orders of Magnitude with a Single Hash Unit [Online]. Available: https://eprint.iacr.org/2024/367.pdf

7. SLH-DSA development / experiments. 2025 [Online]. Available: https://github.com/slh-dsa/slhdsa-c/

Надійшла до редколегії 09.01.2026
Прийнята до друку після рецензування 23.04.2026
Публікація (оприлюднення) 30.04.2026

Відомості про авторів:

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

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

Єсіна Марина Віталіївна – канд. техн. наук, доцент, Харківський національний університет імені В.Н. Каразіна, завідувачка кафедри кібербезпеки інформаційних систем, мереж і технологій, навчально-наукового інституту комп'ютерних наук та штучного інтелекту; АТ Інститут Інформаційних Технологій", науковий співробітник-консультант; Україна; e-mail: m.v.yesina@karazin.ua; ORCID: https://orcid.org/0000-0002-1252-7606

Дерев'янко Ярослав Андрійович – Харківський національний університет імені 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 співробітник-консультант; Україна; e-mail: yarik0009258@gmail.com; ORCID: https://orcid.org/0000-0002-3290-3373