Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Безпеки інформаційних технологій
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Програмна оптимізація алгоритму шифрування PRESENT (тема)

Виконав: Бакаєв С.В.
(прізвище, ініціали)
студент 2 курсу, групи БІКСм-19-1
Спеціальність 125 Кібербезпека
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма «Безпека інформаційних і комунікаційних систем»
(повна назва освітньої програми)

Керівник проф. Руженцев В.І.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Халімов Г.З.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Безпеки інформаційних технологій
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 125 Кібербезпека
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна, або освітньо-наукова)

Освітня програма «Безпека інформаційних і комунікаційних систем»
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові *Бакаєву Сергію Володимировичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Програмна оптимізація алгоритму шифрування PRESENT* затверджена наказом по університету від "22" жовтня 2020 р. № 1412Ст
2. Термін подання студентом роботи (проекту) 12. 12.2020
3. Вихідні дані до роботи (проекту) Теоретичні дані про блокчейн
4. Перелік питань, що потрібно опрацювати в роботі (зміст пояснювальної записки)
1. Актуальність легковажних алгоритмів шифрування
2. Алгоритм шифрування PRESENT
3. Метод оптимізації таблицями підстановок
4. Метод оптимізації bitslice
5. Оцінка швидкодії методів оптимізації
5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій *Презентаційний матеріал у вигляді слайдів*
.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів магістерської атестаційної роботи	Термін виконання етапів роботи	Примітка
1	*Отримання завдання*	07.09.2020	
2	*Пошук літератури*	*10.09.20-21.09.20*	
3	*Аналіз алгоритму шифрування PRESENT*	*21.09.20-12.10.20*	
4	*Аналіз методики таблиць підстановок*	*12.10.20-05.11.20*	
5	*Аналіз методики bitslice*	*05.11.20-20.11.20*	
6	*Написання програмної реалізації та оцінка її швидкодії*	*20.11.20-05.12.20*	
7	*Оформлення пояснювальної записки*	*05.12.20-14.12.20*	

Дата видачі завдання ___07__ _серпня_____2020 р.

Студент _____
(підпис)

Керівник роботи (проекту) _____ __проф. Руженцев В.І._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 72 с., 30 рис., 11 табл., 1 додаток, 40 джерел.

ARX АЛГОРТИМ, ARX, ІНТЕРНЕТ РЕЧЕЙ, PRESENT80, PRESNET128, TABLE, BITSLICE, ТАБЛИЦЯ ПІДСТАНОВОК, ОПТИМІЗАЦЯ

Предмет дослідження – блочні шифри та методи оптимізації їх реалізації.

Мета роботи – аналіз та реалізація методів оптимізації таблиць підстановок та bitslice для шифру PRESENT.

Методи дослідження – реалізація програмного коду для кожного з методів. Аналіз та порівняння реалізацій згідно наступним метрикам: розмір коду, час виконання, використання пам'яті , пропускна здатність та можливі сфери використання.

У роботі розглянуто опис алгоритму шифрування PRESENT, таблиць підстановок та bitslice.

Основні результати – створено та проаналізовано програмні реалізації методів оптимізації шифру згідно метрикам оцінювання.

ABSTRACT

Explanatory note of attestation work: 72 p., 30 rice, 11 tables, 1 application, 40 sources.

ARX ALGORITHM, ARX, INTERNET OF THINGS, PRESENT 80, PRESET 128, TABLE, BITSLICE, SUBSTITUTE TABLE, OPTIMIZATION

The subject of research - block ciphers and methods for optimizing their implementation.

The purpose of the work is the analysis and implementation of methods for optimizing substitution tables and bitslice for the PRESENT cipher.

Research methods - the implementation of program code for each of the methods. Analysis and comparison of implementations according to the following metrics: code size, execution time, memory usage, bandwidth and possible areas of use.

The paper considers the description of the PRESENT encryption algorithm, substitution tables and bitslice.

Main results - software implementations of cipher optimization methods according to evaluation metrics are created and analyzed.

ЗМІСТ

ПОЗНАЧЕННЯ ТА СКОРОЧЕННЯ

AES – Advanced Encryption Standard

ISO – Міжнародна організація зі стандартизації

RFID – Радіочастотна ідентифікація

GE – Умовні логічні елементи

LWC – Легковагова криптографія

CP – Обраний відкритий текст

KP – Відомий відкритий текст

CLK – Кількість тактів

MA – Доступ до пам'яті

PE – Обчислення зменшеної версії шифру

XOR – Додавання за модулем два

SIMD – Single Instruction, Multiple Data

SSE – Streaming SIMD Extensions

Table – таблиця підстановок

Bitslice – Бітові зрізи.

SPN – Мережа підстановок та перестановок

EOM – Електронна обчислювальна машина

IOT – Інтернет речей

ВСТУП

Багато криптографічних алгоритмів створюються, використовуючи операції додавання по модулю 2^n, циклічного зсуву, і складання по модулю 2 (XOR). Криптографічні алгоритми, що використовують тільки ці операції, називаються ARX шифри. Перевагою використання цих операцій є те, що вони дуже швидкі при програмній реалізації. У той же час при правильної композиції, вони мають гарні криптографічні властивості. Додавання за модулем 2^n вносить нелінійність, циклічний зсув забезпечує розсіювання в межах одного слова, а XOR вносить розсіювання між словами.

PRESENT - блоковий шифр з розміром блоку 64 біта, довжиною ключа 80 або 128 біт і кількістю раундів 32. Основне призначення даного шифру - використання в вузькоспеціалізованих приладах, на зразок RFID-міток або мереж сенсорів. Є одним з найбільш компактних крипто-алгоритмів: існує оцінка, що для апаратної реалізації PRESENT потрібно приблизно в 2,5 рази менше логічних елементів ніж для AES або CLEFIA.

Даний шифр був представлений на конференції CHES 2007. У 2012 році організації ISO і IEC включили алгоритми PRESENT разом з CLEFIA в міжнародний стандарт полегшеного шифрування ISO / IEC [5]. Незважаючи на те, що PRESENT оптимізовано для апаратних платформ, програмна реалізація добре працює в апаратному і програмному забезпеченні.

Актуальність роботи обумовлена потребую у швидкодіючих шифрів для інтернету речей .У роботі розглянуто шифр PRESENT та методи оптимізації bitslice та таблиці підстановок. Також наводяться можливі приклади використання шифру.

Метою атестаційної магістерської роботи є дослідження та аналіз реалізації програмних оптимізованих методів шифрування bitslice та таблиці підстановок. У результаті роботи буде отримано час шифрування для кожного методу та описано недоліки та переваги кожного з них.

1 БЛОЧНІ АЛГОРИТМИ ШИФРУВАННЯ

1.1 Побудова блочного алгоритму шифрування

Блочний шифр – різновид симетричного шифру, що оперує групами біт фіксованої довжини – блоками, характерний розмір яких змінюється в межах 64 – 256 біт. Якщо вихідний текст (або його залишок) менше розміру блоку, перед шифруванням його доповнюють. Фактично, блочна підстановка може бути моно або поліалфавітна. Блочний шифр є важливим компонентом багатьох криптографічних протоколів і широко використовується для захисту даних, що передаються по мережі.

Блочні симетричні шифри мають ряд переваг: швидкість шифрування інформації, простота застосовуваних операцій, відносно невеликий розмір ключа, схожі функції шифрування та розшифрування, гнучкість[2]. Ці переваги дозволяють використовувати БСШ у повсякденному житті для забезпечення практично всіх критеріїв безпеки (безпосередньо або побічно, наприклад, в алгоритмі електронного цифрового підпису або алгоритмі автентифікації): конфіденційності та цілісності.

На відміну від шифроблокнота, де довжина ключа дорівнює довжині повідомлення, блочний шифр здатний зашифрувати одним ключем одне або кілька повідомлень, сумарною довжиною більше, ніж довжина ключа. Передача малого в порівнянні з повідомленням ключа по зашифрованому каналу – завдання значно простіша і швидша, ніж передача самого повідомлення або ключа такої ж довжини, що робить можливим його повсякденне використання[8].

Блочний шифр складається з двох парних алгоритмів: шифрування та розшифрування. Обидва алгоритми можна представити у вигляді функцій. Функція шифрування E на вхід отримує блок даних M розміром n біт і ключ K розміром k біт, а на виході віддає блок шифротекста C розміром n біт[2]:

$$E_k(M) := E(K, M) : \{0,1\}^k \times \{0,1\}^k \rightarrow \{0,1\}^k \quad (1.1)$$

Криптоалгоритм іменується ідеально стійким, якщо прочитати зашифрований блок даних можливо тільки перебравши всі можливі ключі, до тих пір, поки повідомлення не виявиться змістовим. Так як по теорії ймовірності шуканий ключ буде знайдено з ймовірністю 1/2 після перебору половини всіх ключів, то на взлом ідеально стійкого криптоалгоритму з ключем довжини N потрібно в середньому 2N−1 перевірок. Таким чином, у загальному випадку стійкість блочного шифру залежить тільки від довжини ключа і зростає експоненціально з її зростанням[8]. Загальна схема алгоритму шифрування наведена на рисунку 1.1.

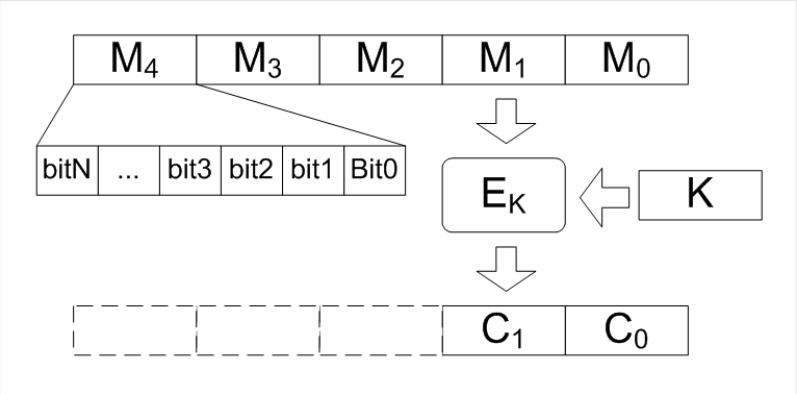

Рисунок 1.1 – Загальна схема блочного шифру

Навіть припустивши, що перебір ключів проводиться на спеціально створеній багатопроцесорній системі, в якій завдяки паралелізму на перевірку 1 ключа йде тільки 1 такт, то на злом 128 бітного ключа сучасній техніці буде потрібно не менше 1021 років. Звичайно, все сказане стосується тільки ідеально стійких шифрів.

Для забезпечення безпеки даних до блочних алгоритмів висувається ряд вимог[6]:

- Високий рівень захисту інформації проти дешифрування та можливих модифікацій.

- Захищеність даних повинна ґрунтуватися лише на знанні ключа та не залежати від того, чи відомий зловмиснику алгоритм (правило Кіркхоффа).
- Невелика зміна вхідного тексту або ключа повинна приводити до значної зміни зашифрованого тексту (тобто забезпечувати «лавинний ефект»).
- Потужність множини значень ключа повинна не допускати можливості дешифрування методами грубої сили.
- Економічність реалізації алгоритму при достатній швидкодії.
- Вартість криптоаналізу даних при відсутності ключа повинна значно перевищувати вартість цих даних.

Найбільш широко використовуваним на сьогоднішній день алгоритмом БСШ є AES (Advanced Encryption Standardабо Rijndael) – національний стандарт блочного шифрування США, який став переможцем однойменного конкурсу в 2000 році.

Блочні шифри є основою, на якій реалізовані практично всі криптосистеми. Методика створення лянцюгів із зашифрованих блочними алгоритмами дозволяє шифрувати пакети інформації необмеженої довжини. Таку властивість блокових шифрів, як швидкість роботи, використовується асиметричними криптоалгоритмами, повільними за своєю природою. Відсутність статистичної кореляції між бітами вихідного потоку блочного шифру використовується для обчислення контрольних сум пакетів даних і в хешуванні паролів[1].

Головним недоліком блочного симетричного алгоритму шифрування є той факт, що якщо ключ був перехопленим, то це гарантує можливість розшифрування інформації, що передається.

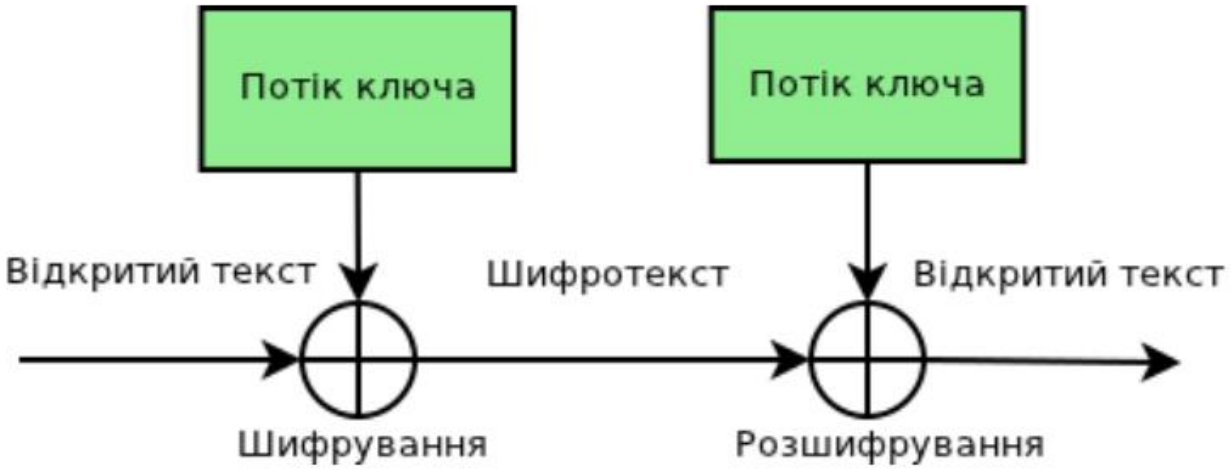

Рис. 1.2 – Принцип функціювання блочних шифрів

1.2 Леговажні алгоритм шифрування

Більшість сучасних алгоритмів захисту інформації і, зокрема, шифрування, розраховані на застосування в ЕОМ в складі програмних комплексів без урахування оптимізації на апаратному рівні забезпечення. Цей факт робить неможливим застосування більшості існуючих криптографічних алгоритмів в пристроях з обмеженою обчислювальною потужністю, малим обсягом і малим енергоспоживанням. Методи криптографічного захисту в системах з низькою вартістю стали основою «легковажної» криптографії.

Особливої актуальності «легка» криптографія набуває в розвитку ідеї «інтернету речей», який являє собою бездротову самоналагоджувальну мережу між об'єктами різного класу, прикладами яких можуть бути побутові прилади, транспортні засоби, інтелектуальні датчики і мітки радіочастотної ідентифікації (RFID)[3].

Найчастіше, розробники легковажних алгоритмів змушені вибирати між трьома, часом взаємовиключними, вимогами до алгоритмам: безпекою, вартістю і продуктивністю.

На практиці не складає труднощів оптимізувати будь-які дві з трьох цілей розробки: безпека і вартість, безпеку і продуктивність або вартість і продуктивність, однак дуже важко оптимізувати всі три цілі розробки одночасно.

У зв'язку з цим існує досить багато реалізацій легковажних алгоритмів криптографії: як програмних, так і апаратних. У них різні, а іноді й протилежні характеристики. Стандартними рішенням проблеми створення ефективних методів і засобів «легковажної» криптографії є:

- Модифікація класичних алгоритмів з адаптацією до апаратних особливостей і обмежень систем з низькою вартістю.
- Розробка нових спеціалізованих рішень в методологічному, алгоритмічній і програмно-апаратному плані.

Кожен з цих підходів має свої недоліки. До сих пір більшість рішень в цій галузі відноситься до третього підходу, і показують непогані результати. При цьому, слід пам'ятати, що при інтеграції криптографічного алгоритму до особливостей апаратного пристрою в умовах обмеженості ресурсів, можуть бути небажані наслідки. Вони можуть виражатися в появі додаткових слабкостей алгоритму або в ослабленні їх загальної стійкості[3].

Деякі важливі властивості блокових шифрів:

- Будь-яка функція над n-бітними рядками $\{0, 1\}^n$ може бути реалізована за допомогою модульного доповнення, обертання, XOR і однієї константи.
- Блочні шифри ARX ефективні в програмному забезпеченні. У порівнянні програмного забезпечення реалізація блокові шифри найбільш ефективні для невеликих процесорів.
- Блочні шифри ARX прості і зручні для реалізації, що призводить до невеликого розміру коду.

1.2.1 Вимоги до засобів низькорівневої криптографії

Як вже було зазначено, головною особливістю сучасного етапу розвитку інтернету є все зростаюча кількість самих різних інтелектуальних пристроїв, що мають доступ в інтернет.

В силу умов їх функціонування, а також жорстких цінових обмежень, властивих масового виробництва, ці пристрої характеризуються значними обмеженнями на використовувані ресурси пам'яті, обчислювальну потужність, джерела живлення і т.д. Звідси випливають обмеження на використовувані технології і технологічні рішення, що пред'являються до засобів низькорівневої криптографії. Так, наприклад, жорсткі обмеження накладаються на енергоспоживання реалізації криптографічних алгоритмів для пасивних інтелектуальних пристроїв таких, як радіочастотні мітки або безконтактні смарт-карти. Відповідно до стандарту ISO / IEC пасивні RFID-мітки повинні мати рівень енергоспоживання не більше 15 μW для того, щоб гарантувати роботу пристрою в радіусі до 1 м [4].

Інший приклад обмежень дають системи автоматичного здійснення дорожніх зборів (плати за проїзд по платних дорогах): для цих систем автомобіль, що рухається з великою швидкістю, повинен бути ідентифікований (authenticate) зчитувальних пристроїв на значній відстані (10-12 м.) І за досить нетривалий час (менше 10 мс.). Ясно, що в цьому випадку швидкість роботи значно більш істотні, ніж розміри мікросхеми або її енергоспоживання. Таким чином обмеженнями, для шифрів легковажної криптографії є:

- Розмір мікросхеми, споживана енергія, час, витрачений на виконання програми; для програмної реалізації.
- Розмір програмного коду, розмір оперативної пам'яті, час, витрачений на виконання програми. Можуть з'являтися і інші обмеження.

Ефективність реалізації того чи іншого перетворення на програмному або апаратному рівні оцінюється по-різному. Для порівняння програмних реалізацій прийнято розглядати вимоги до пам'яті та час роботи, що

вимірюється в тактах процесора. Для апаратної реалізації критерієм ефективності є перш за все розмір мікросхеми і час роботи в тактах процесора, хоча для багатьох додатків важливим фактором є енергоспоживання пристрою[3].

Багато вимог, що пред'являються до алгоритмів, призначеним для використання в обмежених умовах, були закріплені в рамках міжнародного стандарту ISO / IEC FDIS 29192[5].

1.2.2 Операції в легковажних шифрах

Операція XOR – побітове додавання, додавання за модулем 2.

Властивості операції XOR:

- Замкнутість: якщо xy, - n - бітовие слова, то x ⊕ y =z, де z - n - бітовое слово.
- Асоціативність x ⊕(y⊕z) = (x⊕y) ⊕z.
- Комутативність:x⊕y = y⊕x.

Циклічний правий зсув – зсуває кожен біт в n -бітовом слові на k позицій вправо; найправіші k біт праворуч видаляються і стають крайніми лівими[3].

Аналогічно працює лівий зсув. Циклічний лівий зсув – результат інверсії правого зсуву. Якщо один з них використовується для шифрування, інший може застосовуватися для дешифрування.

Властивості циклічного зсува:

- Зміщення по модулю n. Іншими словами, якщо k= 0 або k= n ніякого зсуву не відбувається. Якщо k >n , то вхідна інформація зсувається на k n mod біт.
- Якщо зміщення робиться неодноразово, то знову може з'явитися вихідне n - бітовое слово (зсув є груповою операцією).

Стійкість ARX - криптосистем ґрунтується на тому, що операція додавання за модулем є складним нелінійним перетворенням над полем F_2 [2]. Будемо

називати диференціалом трійку векторів $(\alpha, \beta \to \gamma)$, які задовольняють рівнянню: $(x \oplus a) \oplus (y \oplus \beta) = (x \oplus y) \oplus y$.

1.2.3 Основні критерії до легковажних шифрів

По-перше, це вічний пошук балансу між надійністю, продуктивністю і ціною.

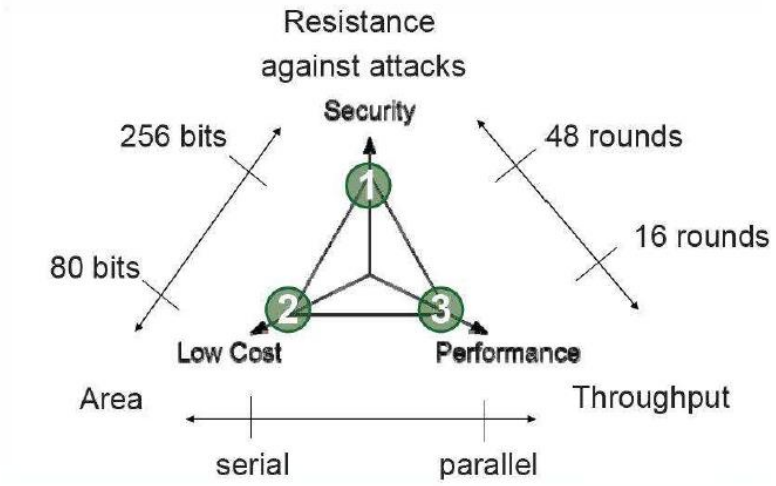

Рисунок 1.3 – Схема взаємодії надійності, продуктивності і ціни

Для блокових шифрів розмір ключа визначає співвідношення надійність / вартість, число раундів шифрування – надійність / продуктивність, а особливості апаратної конструкції – продуктивність / ціна як показано на рисунку 1.3.

Як правило, будь-які дві з трьох цілей розробки можуть бути легко досягнуті, в той час як задоволення всіх трьох вимог – вкрай складне завдання. Наприклад, можна забезпечити прийнятне співвідношення між надійністю і продуктивністю, однак для реалізації подібного алгоритму буде потрібно велика площа на схемі, що призводить до підвищення вартості.

З іншого боку, можна створити надійну і дешеву систему, але з обмеженою продуктивністю.

По-друге, це займає площу мікросхеми.

По-третє, важливо енергоспоживання схеми і, відповідно, вид самої схеми (пасивна або активна), в залежності від якої будуть накладатися додаткові вимоги на схему.

Основні вимоги, що дозволяють криптографії стати невимогливої до ресурсів і при цьому відносно мати стійкі алгоритми шифрування:

- Зменшення (до розумних меж) розмірів основних параметрів алгоритму: блоку шифрованих даних, ключа шифрування і внутрішнього стану алгоритму.
- Спроби компенсації вимушеної втрати стійкості алгоритмів за рахунок проектування на основі добре вивчених, широко застосовуваних операцій, які здійснюють елементарні лінійні / нелінійні перетворення. Такі операції можна уявити як деталі якогось конструктора, з яких криптографи «збирають» алгоритм, що володіє потрібними якостями.
- Зменшення обсягів даних, що використовуються в конкретних операціях. Наприклад, в алгоритмах шифрування часто застосовуються таблиці замін; щоб зберігати таблицю, яка замінює 8-бітові фрагменти даних, необхідно 256 байт, але таку таблицю можна скласти з комбінації двох 4-бітових таблиць, що вимагають всього 32 байти в сумі (даний підхід обрали автори алгоритму Curupira).
- Використання «дешевих» з точки зору споживання ресурсів, але ефективних перетворень, таких як керовані бітові перестановки (в яких вибирається конкретний варіант перестановки в залежності від значення керуючого біта; цим бітом може бути, наприклад, певний біт ключа), зсувні регістри і ін.
- Застосування перетворень, щодо яких можливі варіанти реалізації в залежності від ресурсів конкретного шифратора (наприклад, зменшення вимог до пам'яті, але на шкоду швидкості шифрування, або навпаки)[5].

Слід зазначити, що «полегшені» алгоритми шифрування створюються або для систем з низьким або середнім рівнем безпеки, або для систем, де буде врахована специфіка використовуваних алгоритмів і буде знайдено рішення, що

дозволяє зробити реалізацію алгоритму максимально безпечною для його рівня стійкості.

Одним з основних понять, що використовуються при розгляді «легковажних» алгоритмів криптографії, є GE – Gate equivalent. Ця величина являє собою одиницю виміру, яка дозволяє визначити виробничу складність технології незалежно від складності цифрових електронних схем.

1.2.4 Основні методи та принципи побудови алгоритмів легковажної криптографії.

Як зазначив президент Барт Пренель в своїй доповіді «Stream Ciphers and Lightweight Cryptography» на міжнародному семінарі в Пекіні (червень 2011 р) немає жодного оптимального рішення, що підходить для використання в різних додатках і вбудованих системах - радіочастотних мітках, безконтактних смарт-карт, сенсорах, співпроцесорів для 8-бітових процесорів.

Одним з двох основних напрямків розвитку легковажної криптографії є ефективна реалізація відомих алгоритмів шифрування (можливо, з їх невеликою модифікацією). В останньому випадку можливе часткове зниження безпеки. При реалізації криптографічних алгоритмів розробники пристроїв з обмеженими технічними ресурсами повинні брати до уваги властивості цільової платформи і умови, в яких обладнання буде функціонувати. Так, дуже важливе значення має вибір архітектури. Наприклад, при апаратній реалізації алгоритму послідовні обчислення зменшують розмір схеми і споживану мікросхемою потужність (але, природно, збільшують час роботи).

З іншого боку невеликий час обробки інформації веде до зменшення споживаної енергії. Це важливо, особливо для пристроїв з батарейним живленням , тому що час обробки інформації безпосередньо взаємопов'язане з енергоспоживанням. У сучасні мікроконтролери можна ввести різні режими відключення живлення і енергозбереження після закінчень обчислень. Таким чином, швидке виконання алгоритму може знизити споживання енергії та продовжити час роботи батареї пристрою. Пошук збалансованого рішення є

непростим завданням. Необхідно, наприклад, дослідження залежності швидкості роботи криптоалгоритма від розміру мікросхеми (area for speed - serial / parallel).

Отримання «кращих» реалізацій навряд чи можливо хоча б в силу відсутності методів отримання нижніх оцінок схемної складності навіть для найпростіших перетворень. Однак, за оцінками деяких авторів, для ряду схем досягнута межа в області мінімізації по площі. Іншим напрямком у розвитку легкої криптографії крім ефективної реалізації або невеликої модифікації відомих алгоритмів шифрування, є розробка нових шифрів, орієнтованих на оптимальну реалізацію на апаратному рівні.

Як зазначалося вище, завданням проектування засобів легковагій криптографії є знаходження компромісу між наявними обмеженнями на використовувані ресурси і криптографічного стійкістю розробляється алгоритму (з урахуванням умов, в яких буде функціонувати обладнання, для якого цей алгоритм розробляється). Навіть вибір основних параметрів алгоритму (розмір інформаційного блоку, розмір ключа, розмір внутрішнього стану у алгоритмі потокового шифрування, розмір кінцевого поля у разі асиметричних криптоалгоритмів) потрібно зробити в межах позначених обмежень. Так, обмеження на використовувані ресурси роблять привабливим розробку криптоалгоритмів з малими розмірами інформаційного блоку і ключа. Однак в цьому випадку криптографічний алгоритм схильний до різних атак (наприклад, атаки, пов'язані з парадоксом про дні народження).

Перевагу у використанні отримують перетворення, що вимагають меншого розміру пам'яті обчислювального пристрою (або меншого числа логічних елементів для їх реалізації). Так вибір S - блоків розміру 4x4 краще S-блоків розміру 8x8, так, як 8-бітові S-блоки вимагають для своєї реалізації в середньому аж ніяк не менше 120 GE в той час як 4-бітові можуть бути реалізовані зі складністю 21 - 39 GE, що в середньому веде до зменшення розмірів мікросхеми в десятки разів. Однак 4-бітові S-блоки повинні бути обрані ретельно, так як вони криптографічно слабкіші, ніж 8-бітові. Проте, за

рахунок ретельного відбору, можливо досягти відповідного рівня безпеки. Алгоритм вироблення раундових ключів повинен породжувати їх in-place, тобто такі ключі не рекомендується використовувати в режимі ECB.

1.2.5 Застосування легковажних алгоритмів

Стрімкий розвиток зазначених технологій робить надзвичайно актуальними питання, пов'язані з їх інформаційною безпекою. Так, директор ЦРУ Девід Петреус заявив, що дані з підключених до Інтернету побутових приладів можна використовувати для складання максимально докладного досьє на будь-яку людину. У презентації для MIT Media Lab, зробленої Дослідницької групою по довіреним системам (Trusted Systems Research Group) Агентства Національної Безпеки США (National Security Agency) 30 січня 2013 говориться: «RFID-технології розвиваються надзвичайно швидко. Входячи до складу систем визначення точного місця розташування, мають вихід в глобальні мережі зв'язку, радіочастотні мітки стали надзвичайно потужним засобом для ідентифікації, визначення положення і стеження за окремими людьми або об'єктами ».

Все це, звичайно, дозволить ефективніше виявляти терористів, але, разом з тим, збір даних торкнеться і переважної більшості законослухняних громадян. Потрібно додати до цього численні додатки, пов'язані з обробкою біометричних даних, персональних даних медичного характеру, важливою фінансової інформації та ін. В зв'язку з цим особливо актуальним стає завдання ефективної реалізації алгоритмів захисту інформації, що забезпечують конфіденційність і цілісність даних. Очевидно, що основу такої безпеки повинні утворювати криптографічні методи захисту інформації. І основним засобом забезпечення інформаційної безпеки в світі Інтернету Речей є так звана «легка криптографія» (lightweight cryptography, LWC)[148].

1.2.6 Визначення переваг і недоліків лековажних шифрів

Оскільки «легковісні» алгоритми розробляються під конкретні вимоги, то з них безпосередньо випливають всі переваги і недоліки даного сімейства алгоритмів.

При цьому головною перевагою є вкрай низькі вимоги як до ресурсів, так і щодо споживання енергії, що робить «легковагі» алгоритми вкрай швидкими в роботі і «невибагливими» до середовища, в якій буде здійснюється їх робота. Крім того, це робить «легковагі» алгоритми вкрай дешевими у впровадженні та використанні.

Однак, оскільки «легковагі» алгоритми призначені для обробки малого обсягу інформації, вони не володіють високою пропускною спроможністю. Сам факт наявності обмеження в 1000 GE говорить про те, що «полегшені» шифри, в першу чергу, націлені не так на програмну, а на апаратну реалізацію, а оскільки для реалізації, наприклад, більшої кількості s-боксів в алгоритмах блокового шифрування, або використання ключів великої довжини, потрібно більше GE, то перед розробниками «легковажних» алгоритмів встає досить непросте завдання [3].

Крім того, дані обмеження роблять вельми важким завдання оптимізації вже існуючих алгоритмів під вимоги «легковажною» криптографії, оскільки багато хто з них сильно втратять в стійкості, будучи обмежені в обчислювальних ресурсах. Це сильно затримує процес розробки «легковажних» алгоритмів, хоча і існує стандарт за полегшеним шифрування серії ISO / IEC 29192[5]. Більш того, все частіше проводяться успішні атаки на фаворитів серед наявних алгоритмів легковагій криптографії. Все це робить застосування легких алгоритмів на практиці досить вузькоспеціалізованою і складним завданням.

У всіх оглядах по легковажній криптографії відзначається, що в даний час немає загальної теорії розробки LWC-алгоритмів (можливо і не буде). Цілий ряд авторів пропонує виділити розробку надлегких криптографічних алгоритмів (ultra-lightweight algorithms) в окремий напрямок криптографії. При цьому посилюється розбіжність між програмно і апаратно-орієнтованими легковагими криптоалгоритмами.

Фундаментальна відмінність у вимогах, що висуваються ресурсними обмеженнями до програмно і апаратно-орієнтованим легковажним криптоалгоритм було продемонстровано в роботі[6].

Там було показано, що блоковий шифр PRESENT надзвичайно зручний для легковагій апаратної реалізації, але вимагає значних ресурсів при програмної реалізації. Оскільки основним завданням легковажної криптографії є мінімізація витрачених ресурсів, важливим напрямком є багатофункціональність - можливість за допомогою однієї мікросхеми здійснювати шифрування, реалізацію вироблення імітовставки (MAC), генерацію випадкової послідовності (PRNG) і т.д. При цьому розроблені алгоритми повинні забезпечити ефективну обробку невеликих обсягів даних, що найбільш характерно для вбудованих систем. Нарешті, важливою проблемою для LWC - алгоритмів є атаки side-channel.

2 АЛГОРИТМ ШИФРУВАННЯ PRESENT

2.1 Перелік скорочень для шифру

$K_i = k^i_{63}k^i_{62}...k^i_0$ 64-раундовий ключ, який використовується на і ітерації

k^i_b — біт и раундового ключа K_i

$K = k_{79}...k_0$ 80 бітний регістр ключа

k_b — біт b регістру K

STATE — внутрішній стан

b_i — біт I поточного стану

w_i — 4 бітне слово, де w_i

2.2 Область використання шифру PRESENT

Під час розробки блочного шифру, який буде використовуватись в системах з обмеженими можливостями, важливо розуміти, що шифр не буде використовуватись усюди. Для цілей широкого використання вже є AES[7]. Шифри в системах з обмеженими можливостями мають наступні характеристики:

- Шифр повинен реалізовуватись апаратно.
- Для програми потребується лише середній рівень безпеки. В наслідок чого, 80-бітної версії буде достатньо для безпеки.
- Програмі не потребується шифрування великого об’єму даних. У зв’язку з цим шифр може буду оптимізований по швидкодії або за використанням пам’яті.
- В деяких випадках можливо, що ключ буде постійним під час використання пристрою. Що в свою чергу позбавляє деяких атак з маніпуляцією ключами.
- В програмах, які потребують ефективного використання пам’яті, шифр реалізується лише у режимі шифрування.

2.3 SP мережа

SP – мережа, або мережа замін-перестановок — це ряд пов'язаних математичних операцій що використовуються в блочних шифрах, наприклад AES. Така мережа приймає блок відкритого тексту і ключ на вході, і застосовує декілька «раундів» S і P-box які чергуються для отримання блоку шифротексту. S-box і P-box перетворюють підблоки вхідних бітів на вихідні біти. Ці операції обираються такими щоб бути зручними для ефективного втілення в залізі, наприклад додавання за модулем 2 (XOR) і побітове обертання. Ключ вводиться в кожному раунді, зазвичай у вигляді ключа раунду похідного від нього. (Іноді самі S-box залежать від ключа.)

Розшифрування здійснюється зворотнім процесом (використанням обернених S-box і P-box із застосуванням ключів раундів у зворотному порядку). S-box замінює блок бітів (дані на вході) на інший блок бітів (дані на виході). Зазвичай довжина блоку 4 або 8 бітів. Ця заміна має бути однозначною, для гарантування оборотності (відповідно розшифрування). Зокрема довжина даних на вході може збігатись із довжиною даних на виході (зображення праворуч містить S-box для 4 біт на вході і виході), що не завжди має місце для S-box, які також можуть змінювати довжину.

Наприклад DES S-box це зазвичай не просто перестановка бітів. Ілюстрація алгоритму SP мережі наведена на рисунку 2.1. Вдалий S-box має властивість змінювати близько половини бітів на виході через зміну одного біту на вході (лавиновий ефект). Також вона матиме властивість залежності кожного біту на виході від кожного біту на вході. P-box – перестановка всіх бітів: вона отримує вихідні дані усіх S-box поточного раунду, переставляє біти і передає результат S-box наступного раунду. P-box має властивість: вихід будь-якої S – скрині розподіляється між якнайбільшою кількістю S-box наступного раунду.

На кожному раунді ключ раунду (отриманий з ключа за допомогою простих дій, наприклад, із використанням S і P-box) додається через якусь просту групову дію, зазвичай XOR[8].

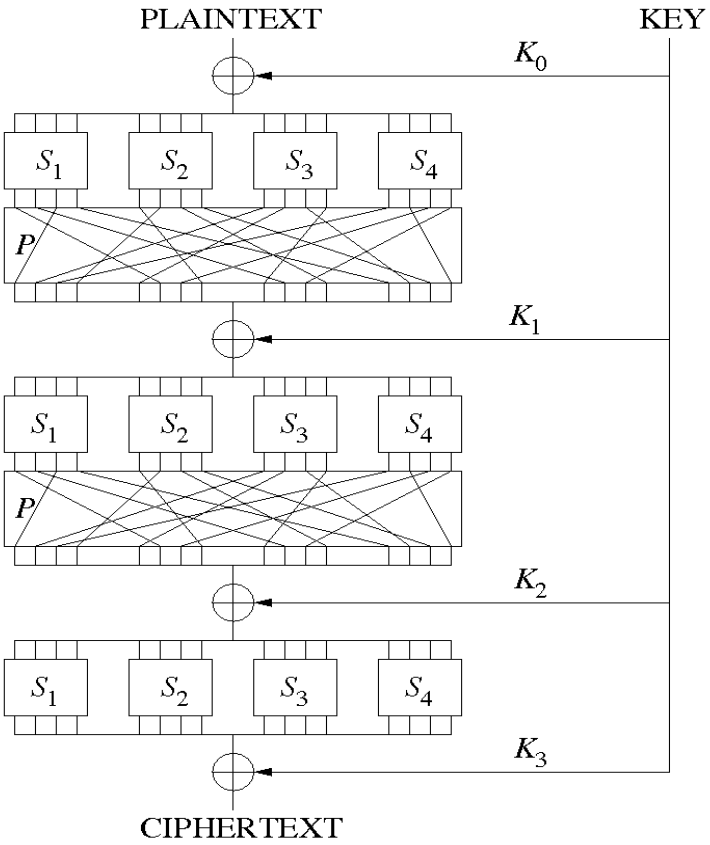

Рисунок 2.1 – Схема трьох раундової мережі замін-перестановок, що шифрує 16 біт вхідного тексту в 16 біт вихідного.

2.4 Опис алгоритму шифрування PRESENT

PRESENT є прикладом SP-мережі і складається з 31 раунда. Довжина блоку - 64 біта, підтримуються дві довжини ключів – 80 та 128 біт. У даній роботі розглядається шифр з довжиною ключа 80 біт. Загальна схема алгоритму наведена на рисунку 2.2.

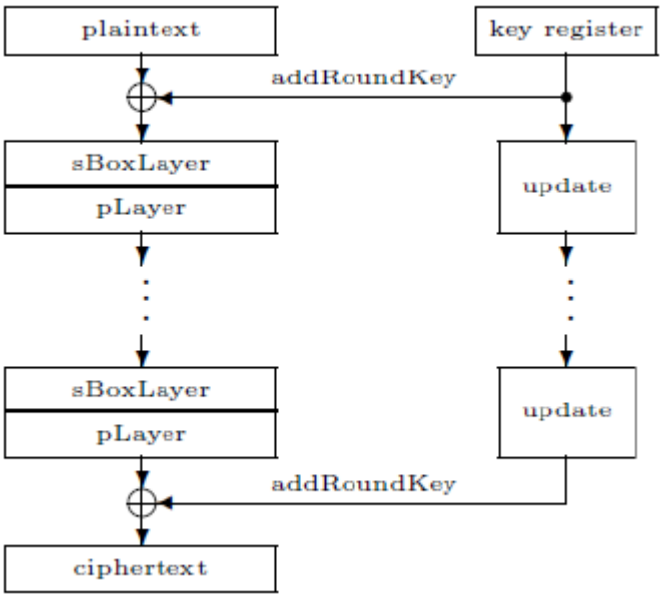

Рисунок 2.2 – Загальна схема алгоритму шифрування PRESENT

Кожен з 31 раундів складається з операції xor для раундового ключа K_i для $1 \le i \le 32$, де K_{32} застосовується для лінійної бітової перестановки та нелінійного рівня заміщення. Нелінійний рівень використовує 4–бітний S-box , який застосовується паралельно 16 разів у кожному раунді.

Шифр описаний у псевдо-коді на рисунку 2.4, і кожен етап виконується по черзі. На рисунку 2.3 для наочних цілей показано два послідовних раунди алгоритму[9].

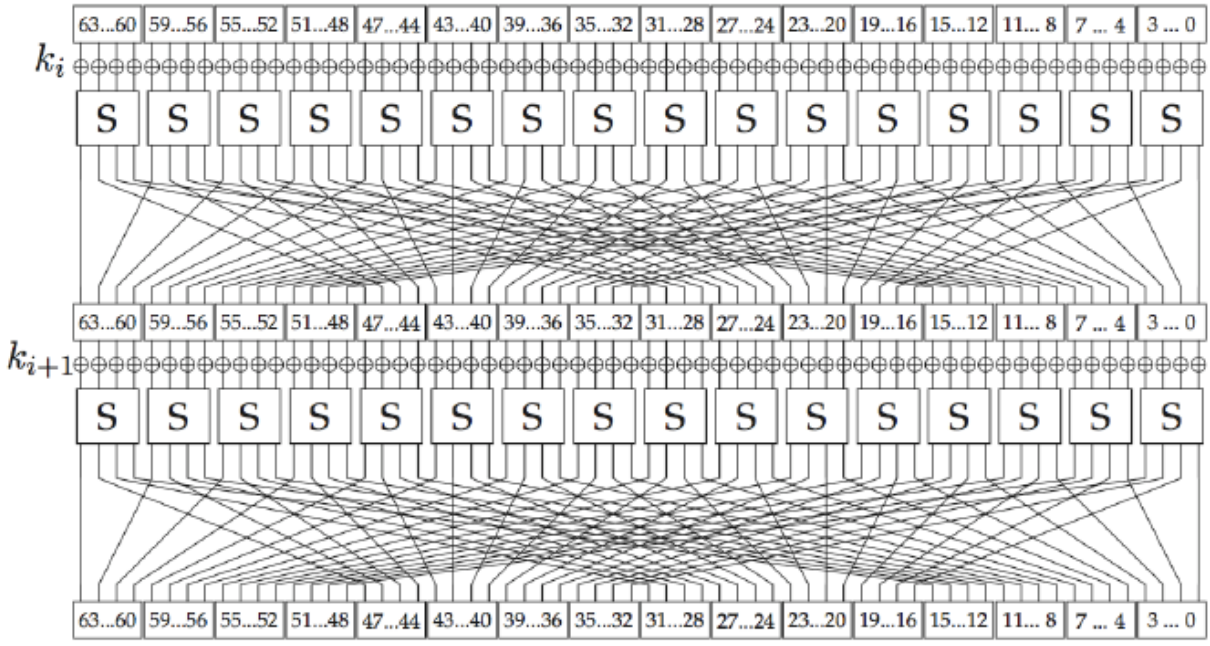

Рисунок 2.3 – Приклад 1 раунда шифру PRESENT

На рисунку 2.4 приведено псевдокод алгоритму PRESENT

```
generateRoundKey()
for(i = 0; i < 32; i++)
{
addRoundKey()
sBoxLayer(STATE)
pBoxLayer(STATE)
}
addRoundKey()
```

Рисунок 2.4 – Псевдокод алгоритму PRESENT

2.4.1 Підстановка S-боксів

Функція підстановки складається з 4 бітових входів і 4 бітових виходів кожного S-box. Цей процес можна представити як операцію відображення і позначати як: $S : F_2^4 \to F_2^4$. Поточний STATE зберігає з 16 4-бітних слів.

Таблиця підстановки в шістнадцяткових позначеннях наведена у таблиці 2.1.

Таблиця 2.1 – Таблиця підстановки S-box

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Для sBoxLayer поточний стан b_{63} ….b_o розглядається як шістнадцять 4-розрядних слів $w_{15}…w_0$, де $w_i = b_{4*i+3} \| b_{4*i+2} \| b_{4*i+1} \| b_{4*i}$, де $0 \le i \le 15$ та вихідна $S(w_i)$ підстановка забезпечує оновлення $S(w_{15}) \| S(w4) \| ... \| S(w_0)$ [9].

Інверсний S-box, який використовується у процедурі дешифрування являється інверсією описаного 4 в 4 бітного прямого S-box.

Інверсний S-box перетворює вхідну послідовність із 4 бітів X в вихідну $s^{-1}(x)$, послідовність зображена на таблиці 2.2

Таблиця 2.2 – Інверсна таблиця підстановки S-box

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	5	E	F	8	C	1	2	D	B	4	6	3	9	7	9	A

Вибір саме 4 бітного S-box, а не 8-бітного полягає у збільшені швидкодії для апаратних пристроїв. Хоча шифр и базується на алгоритмі AES[7], проте техніка підстановки бітів на етапі дифузійної підстановки, яка лежить в основі AES не можливо використовувати у цьому шифрі. Тому на S-box накладаються додаткові вимоги[9]. Проте блоки відповідають наступним вимогам, де S коефіцієнт Фурьє $S_b^W(a) = \sum_{x \in F_2^4} (-1)^{(b,S(x))+(a,x)}$.

- Для будь-якої фіксованої ненульової різниці вхідних даних $\#_I \in F_2^4$ і будь-якої ненульової ненульової різниці на виході $\#_O \in F_2^4$ вимагається
- $\#\{x \in F_2^4 \mid S(x) + S(x + \#_I) = \#_O\} \le 4$
- Для будь-якої фіксованої ненульової різниці вхідних даних $\#_I \in F_2^4$ і будь-якої ненульової не нульової різниці на виході $\#_O \in F_2^4$, така що $\#_I = wt(\#_O) = 1$ $\{x \in F_2^4 \mid S(x) + S(x + \#_I) = \#_O\} \le \varnothing$
- Для всіх ненульових $a \in F_2^4$ та ненульових $b \in F_2^4$ вважається, що $|S_b^W(a)| \le 8$
- Для всіх ненульових $a \in F_2^4$ та ненульових $b \in F_2^4$ такі , що $wt(a) = wt(b) = 1$ вважається $S_b^W(a) = \pm 4$

2.4.2 Перемішування

Бітова перестановка, що використовується в PRESENT, наведена в таблиці 2.3. Біт і стану переміщується в бітове положення P(i).

Таблиця 2.3 – Таблиця бітової перестановки

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47

Продовження таблиці 2.3

P(i)	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Інверсне перемішування – етап зворотної перестановки використовується під час дешифрування, таблиця 4 описує правила зворотної підстановки. На цьому етапі і біт стану переміщується у $P^{-1}(i)$ позицію[9].

Таблиця 2.4 – Інверсна таблиця бітової перестановки

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	1	5	8	13	17	21	25	29	33	37	41	45	49	53	57	61
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63

2.4.3 Розгортка ключа

PRESENT може мати ключі довжиною 80 або 128 біт. Введений ключ зберігається в регістрі K і представлений як $k_{79}\, k_{78} \ldots\, k_0$ На раунді i 64-розрядний раундовий ключ складається з 64 лівих бітів поточного вмісту регістра K.

Таким чином, у раунді ми маємо, що: $K_i = k_{63}k_{62}...k_0 = k_{79}k_{78}...k_{16}$ [10]. Регістр ключів "K" повертається ліворуч на 61 біт позицій, 4 біти лівої позиції надсилають до PRESENT S-box, а значення "n" XOR з бітами $K_{19}, K_{18}, ..., K_{15}$ разом з раундовим лічильником LSB праворуч. Схема розгортки наведена на рисунку 2.4. Алгоритм розгортки ключа наведено на рисунку 2.5.

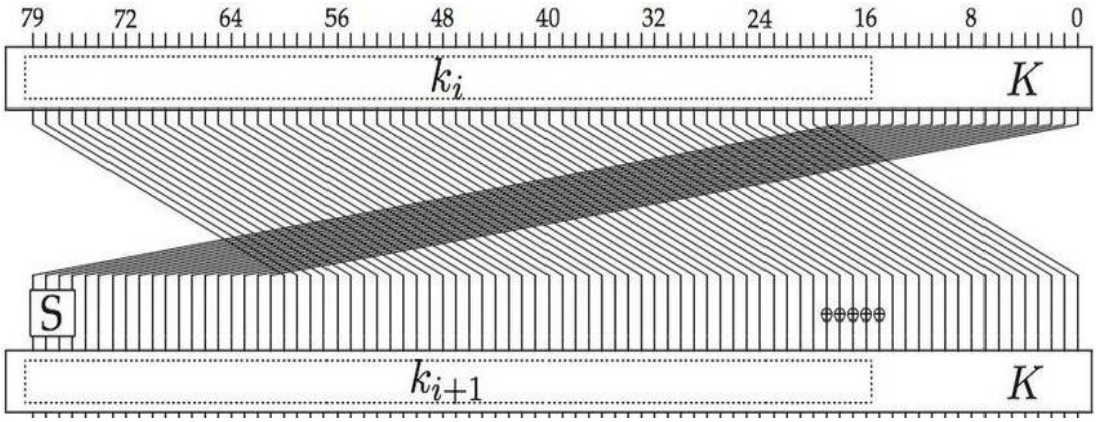

Рисунок 2.4 – Розгортка ключа PRESENT

Крок 1 $[k_{79}k_{78}...k_1k_0] = [k_{18}k_{17}...k_{20}k_{19}]$

Крок 2 $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$

Крок 2 $[k_{19}k_{18}k_{17}k_{16}] = [k_{19}k_{18}k_{17}k_{16}] \oplus$ round counter

Рисунок 2.5 – Алгоритм розгортки ключа для одного раунда

2.5 Атаки на шифр PRESENT

2.5.1 Алгебраїчно диференційна атака

У роботі [11] пропонується 3 атаки, які об’єднують у собі алгебраїчні та диференційні методи. Для 2 із 3 атаки автори пропонують експериментальні дані для атак на PRESENT-80 із 16 та PRESENT-128 зі 17, 18 та 19 ранудами. Проте вони заявляють, що атака на та PRESENT-128 зі 19 раундами потребує 2^{113} циклів ЦП, що у свою чергу не практично. Нижче у таблиці наведено результати однієї атаки. Проте для іншої при кількості раундів більше 13 час атаки також більше 4 годин. Для виконання алгебраїчної атаки використовується базисні алгоритми Гребнера або SAT вирішувач. Реалізація цих алгоритмів надається у Singular 3-0-4-4[12], PolyBoRi 0.5rc6[13] та MiniSat 2.0 beta[14]. У таблиці 2.5 наведені результати однієї з 3 атак.

Таблиця 2.5 – Час атаки у секундах.

N	K_s	r	p	Singular	PolyBoRi	MiniSat2
4	80	4	2^{-16}	0.07 − 0.09	0.05 − 0.06	N/A
4	80	3	2^{-12}	6.69 − 6.79	0.88 − 1.00	0.14 − 0.18
4	80	2	2^{-8}	28.68 − 29.04	2.16 − 5.07	0.32 − 0.82
4	80	1	2^{-4}	70.95 − 76.08	8.10 − 18.30	1.21 - 286.40
16	80	14	2^{-62}	123.82 − 132.47	2.38 − 5.99	N/A
16	128	14	2^{-62}	N/A	2.38 − 5.15	N/A
16	80	13	2^{-58}	301.70 − 319.90	8.69 − 19.36	N/A
16	128	13	2^{-58}	N/A	9.58 − 18.64	N/A
16	80	12	2^{-52}	N/A	> 4 годин	N/A
17	80	14	2^{-63}	318.53 − 341.84	9.03 − 16.93	0.70 − 58.96
17	128	14	2^{-62}	N/A	8.36 − 17.53	0.52 − 8.87
17	80	13	2^{-58}	N/A	> 4 годин	> 4 годин

2.5.2 Диференційна атака на версії PRESENT зі зменшеною кількість раундів

Відразу після публікації шифру PRESENT автор WANG відразу опублікувала свої відкриття о диференційних властивостях шифру PRESENT[15]. У роботі продемонстровано, що 16-ранудова версія вразлива до диференційного криптоаналізу. Для цього потрібно 2^{64} відкритих ключів. У результаті з ймовірністю у 0.999999939 можливо відновити ключ. Недоліком цієї атаки є велика кількість кодової книги, тобто 2^{64} відкритих текстів та їх шифротекстів. Проте атака дійсна лише на 16 раундів зі 32.

2.5.3 Атака статичного насищення

Першою атакою на PRESENT була атака статичного насищення. Суть цієї атаки в аналізі інформації про ключі та відстеженням за нерівномірним розподілом у зашифрованому тексті[16]. Її реалізація можлива лише на 16 раундів шифру. Перша атака використовує 2^{36} пар відкритий текст – зашифрований текст, 2^{28} звертань до пам'яті та потребує збереження 2^{16} раундових лічильників.

Друга вимагає меншу кількість відомих пар відкритого тексту 2^{33} проте потребує зберігання 2^{32} лічильників та 2^{57} звертань до пам'яті . Ця атака використовує властивості дифузії у блочних шифрах, а саме те що 8 із 16 вихідних бітів направляється у інші S-box. Рисунок 2.6 ілюструє дану поведінку.

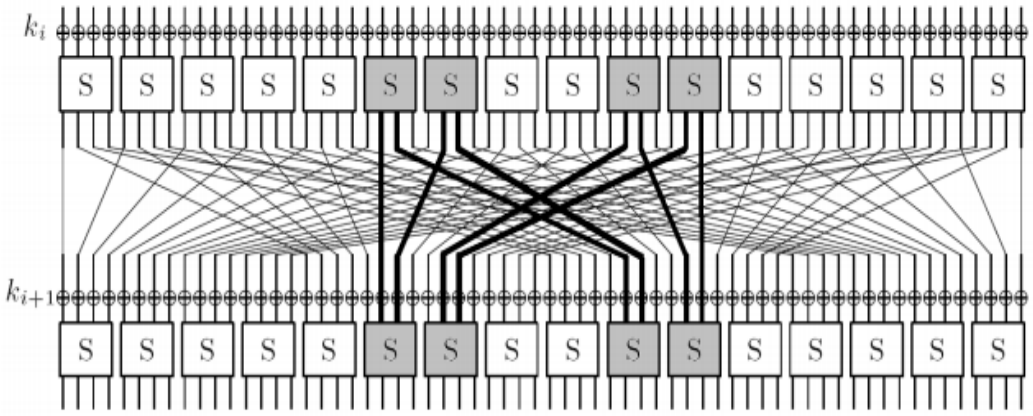

Рисунок 2.6 – Етап перестановки в шифрі PRESENT. Виділенні рівні підкреслюють недолік властивості дифузії.

Також атака даного типу була представлена Nakahara[17]. У результаті цієї атаки було відновлено половину бітів ключа менш ніж за 3 хвилини.

2.5.4 Лінійний криптоаналіз

Автор [18] досліджував комбінацію слабих ключів та лінійного криптоаналізу на версію шифру PRESENT зі зменшеною кількістю раундів. Атака використовує версію шифру з 24 раундами, та може відновити 40 біт ключа з ймовірністю 0.95. Інші 40 біт відновлювались шляхом перебору на основі відомого тексту.

У роботі [19] було запропоновано багатовимірну лінійну атаку на 25 та 26 раундів. Для її виконання потрібно знати повну кодову книгу або $2^{62,4}$ пар відомого відкритого тексту та об'єм пам'яті в 16 гігабайт.

Наступна атака була наведена у роботі[20] та іменується як Linear hull та використовує 26 раундів на 128-бітну версію шифру. Використовує повну кодову книгу та 2^{40} пам'яті .

2.5.5 Структурні атаки

Структурні атаки так інтегральна атака[15] та атаки bottleneck[21] добре підходять для щифрів AES[7], SQUARE[23], та SHARK[24].Ці шифри мають стійку структуру, де слова – це байти. Проте структура PRESENT виключно складається із байтів.

В [25] пропонується атака шифру PRESENT на базі бітових шаблонів. Автори аналізують шифр використовуючи 5, 6 та 7 раундів. Для атаки із 5 раундами потрібно лише 80 відкритих текстів. Проте для атаки із 7 раундами потрібно вже 2^{24}, та потребуючий час зростає до 2^{100}. Також автори заявляють, що інтегральні атаки мають свою кінцеву зупинку, бо після збільшення кількості раундів зростає витрата часу. На основі цього зроблено висновок, що даний тип атака не несе погрози для PRESENT. У таблиці 2.6. підсумовується відомі атаки на шифр з урахуванням розміру ключа та кількості раундів.

Таблиця 2.6 – Опис криптоаналітичних результатів відомих на даний час

Розмір ключа	Раунди	Тип атаки	Данні	Час	Пам'ять
80	16	Алгебраїчно-диференційна	2^{62} *KP*	2^{46} *CLK*	2^{30} *Байтів*
	16	Диференційна	2^{64} *KP*	2^{65} *MA*	$2^{31,6}$ *Байтів*
	16	Статистичного насищення	$c*2^{36}$ *CP*	2^{28} *MA*	2^{16} *Лічільників*
	16	Статистичного насищення	$c*2^{33}$ *CP*	2^{57} *MA*	2^{32} *Лічільників*
	24	Лінійна атака з використанням слабих ключів	$2^{63,5}$ *KP*	не вказано	не вказано
	25	Багатовимірна лінійна	$2^{62,4}$ *KP*	2^{65} *PE*	2^{34} *Байтів*
	26	Багатовимірна лінійна	2^{64} *KP*	2^{72} *PE*	2^{34} *Байтів*
128	7	Бітові шаблони	$2^{24,3}$ *CP*	2^{100} *MA*	2^{77} *Байтів*
	17	Алгебраїчно диференційна	2^{63} *CP*	2^{104} *MA*	2^{30} *Байтів*
	18	Алгебраїчно диференційна	2^{63} *CP*	2^{98} *CLK*	2^{30} *Байтів*
	19	Алгебраїчно диференційна	2^{63} *CP*	2^{113} *CLK*	2^{30} *Байтів*
	25	Linear Hull	2^{64} *KP*	$2^{98,6}$ *PE*	2^{40} *Блоків*

3 МЕТОДИ ОПТИМІЗАЦІЇ ШИФРУ PRESENT

RFID-мітки та обмежені обчислювальні пристрої стають все більш важливими додатком для галузей. Зростання обчислювальних пристроїв та комунікаційних зв'язків, призводить до збільшення кількості можливостей для потенційного збитку зловмисників. Безпека має вирішальне значення для багатьох ситуацій, але часто залишається осторонь через обмеження вартості. Для того, щоб задовольнити потребу в криптографічних примітивах, які можуть бути впроваджені та виконані в дуже обмежених середовищах (площа, енергоспоживання тощо), відомих як легка криптографія, дослідницьке співтовариство нещодавно досягло значного прогресу, зокрема в області криптографії симетричних ключів.

Поточні стандарти NIST для блочного шифру AES [7] або хеш-функції (SHA-2 [26] або SHA-3 [27]) не підходять для обмежених середовищ, і запропоновано кілька альтернатив, таких як PRESENT [9], KATAN [28], LED [29], Piccolo [30] для блочних шифрів та QUARK [31], PHOTON [32], SPONGENT [33] для хеш-функцій. Слід зазначити, що блоковий шифр PRESENT тепер є частиною стандарту ISO [5]. Усі ці пропозиції значно покращили знання щодо легковажних шифрів, і багато хто вже досягає майже оптимальної продуктивності для певних показників, таких як споживання площі.

На практиці обмежені пристрої будуть або взаємодіяти з іншими обмеженими пристроями, або, швидше за все, із сервером. В останньому випадку серверу, ймовірно, доведеться обробляти багато пристроїв, і хоча криптографія може використовуватися в протоколі для захисту комунікацій, інші операції з додатками повинен виконувати сервер. Тому дуже важливо, щоб сервер не витрачав занадто багато часу на виконання криптографічних операцій, навіть коли спілкується з багатьма клієнтами, і, отже, продуктивність програмного забезпечення має значення для легковажної криптографії. На виставці CHES 2012 Matsuda and Moriai[34] вивчали застосування реалізації bitslice до блочних

шифрів PRESENT та Piccolo, дійшовши до висновку, що нинішні легкі шифри можуть бути конкурентоспроможними для хмарних додатків. Реалізація Bitslice дозволяють отримати вражаючі швидкісні результати, а також цінні для властивого їм захисту.

Обмежені пристрої зазвичай шифрують, щоб замінити штрих-коди, використовуючи недорогі пасивні RFID-мітки, використовуючи 64, 96 або 125 біт послідовність, як унікальний ідентифікатор будь-якого фізичного елемента. Малий розмір шифрований даних робить витрати на перетворення даних у bitslice форму та процес розгортки ключів дуже дорогими (ці витрати зазвичай опускаються, якщо припустити, що буде зашифровано досить велика кількість блоків). Тому варто дослідити ефективність програмного забезпечення легковажних шифрів не лише для хмарних додатків, а й для класичних додатків. Крім вбудованих 8-розрядних архітектур, вони насправді не призначені для ефективної роботи в програмному забезпеченні на процесори середнього та високого класу. Наприклад, найкраща реалізація AES, досягає швидкості приблизно 14 циклів на байт (c / B) на 32-розрядному процесорі Pentium III[35], тоді як найкращі реалізації методу bitslice , працюють лише за 130 c/B на тому ж процесорі.

Основна ідея оптимізації алгоритмів полягає у зміні структури шифру. У структурі повільні операції замінюються на більш прості та швидші, що у свою чергу дають такий же результат. Найбільш використовувані методи:

- Table based implementations.
- Bit-slice implementations.

Основна ідея цих технік – це збільшення швидкодії шифру.

3.1 Методи оптимізації

Розробники шифру PRESENT надають лише ASIC імплементацію алгоритму[9]. Оптимізація за використовувану пам'ять потребує 1570 GE та 32 тактових цикли для виконання одного шифрування. У середньому PRESENT-80 потребує лише 32 тактових циклів та 1886 GE.

Метод формування таблиці підстановки. Це техніка відома вже достатньо давно. Вперше вона була використана у [36] для ефективного виконання операцій шифру AES на 32-бітних процесорах. За допомогою даної методики 1 раунд шифру може бути розрахований з використанням 16 таблиць пошук,. 4 таблиць 8-бітних вхідних даних і 32-бітних вихідних даних. Таким чином при використанні цього методу оптимізації декілька операцій будуть замінені пошуком у таблиці підстановки, що у свою чергу збільшить швидкодію шифру. У результаті 1 раунд буде складатися з наступних операцій:

- Етап додавання ключів(може виконуватись до або після пошуку у таблиці, в залежності від структури шифру).
- Виділення зрізів у стані шифру за допомогу операцій зсуву та маски.
- Етап раундового перетворення через декілька таблиць пошуку.
- Комбінування результатів підстановок з таблиць для отримання оновленого стану шифру.

Цей підхід реалізується досить тривіально у шифрах, які представляють собою структуру SPN. Проте найбільш продуктивно цей підхід реалізується шифрах схожих на AES. В цих шифрах раунди основані на етапах підстановки S-box та етапу перестановки з блоками множення (M-Box). У [37] пропонується загальних підхід для реалізації на основі таблиць.

Метод bitslice. Основна концепція bitslice моделювання апаратної реалізації у програмному забезпеченні. Весь алгоритм представляється у вигляді послідовності логічних операцій.

На процесорі з n-бітовими регістрами логічна інструкція відповідає одночасному виконанню n апаратних логічних операцій. Такі шифри як

Fantomas / Robin [30], RECTANGLE [38], NOEKEON [39] було спроектовані з урахуванням бітового зрізу. Використання даного методу до полегшених шифрів призводить до створення шифрів з невеликим розміром коду.

3.2 Метод оптимізації таблицями підстановок

Підстановка операцій для підвищення ефективності один із відомих способів. Основна мета методу полягає у зведені до мінімуму кількості операцій одного раунду за рахунок зведення операцій вигляду таблиць підстановок. Таким чином розрахунок 1 раунду складається з наступних операцій:

- Виділення зрізів внутрішнього стану операціями зсуву та маски.
- Виконання декілька пошуків у таблиці для отримання раундового перетворення.
- Агрегування вхідних даних таблиці для пошуку оновленого внутрішнього стану.
- Виконання рівня додавання ключів.

Даний підхід був вперше запропонований Daemen and Rijmen для ефективного виконання операцій AES на 32-бітних процесорах. Таким чином раунд AES може бути розрахований з 16 пошуками у таблицях зі 8-бітним входом та 32-бітним виходом(кожна таблиця має розмір близько 1 КБайт).

Більшість полегшених шифрів мають 64-бітні блоки та 4-бітну таблицю S-box, це у свою чергу дозволяє зробити багато компромісів при 32-бітних або 64-бітних процесорах.

Розглядаючи шифри, які базуються на SPN, функція раунда може бути виконана на основні таблиць наступним чином:

- Внутрішній стан ділиться на 64/m секцій по m бітів кожна, $T_0, T_1, \ldots$ таблиці із m-бітним входом та 64-бітним виходом.
- Маскування MASKm – маска з m послідовними найменше значущими бітами.

```
// Computation of a generic SPN lightweight cipher round
// Input: 64-bit state  ---  Output: updated 64-bit state
t0 = T0[ state        & MASKm];
t1 = T1[(state >>  m) & MASKm];
t2 = T2[(state >> 2m) & MASKm];
...
state = t0 ^ t1 ^ t2 ^ ...;
```

Рисунок 3.1 – Загальна функція раунду шифрування для SPN шифрів

Варто звернути, що вибір вхідних даних таблиць відповідає розміру внутрішнього стану шифру. За допомогою цього можливо включити рівень перемішування при пошуку у таблиці. Після цього результат одного раунду буде розрахований за допомогою зсуву, маски, пошуку у таблиці та операції XOR. Кількість операцій буде залежить від розміру таблиць(кожна таблиця T має розмір $S_T = 2^m \times 8$ байтів).

Головними питанням залишається вибір найкращого m. Чим більша кількість, а отже розмір таблиці тим менш часу потрібно для розрахунку 1 раунду, но тим більш затримка при пошуку у таблиці. Проте набір інструкцій Intel дозволяє використовувати інструкції маскування та переміщення, що призведе до зниження кількості операцій.

На рисунку 3.2 наведений псевдокод функції раунда з урахуванням різних m (синтаксис Intel).

```
// m=4 or 12 (1 round)
shr state, m
mov tmp, state
and tmp, MASKm
(mov/xor) accumulator, [T+8*tmp]
...
```

```
// m=8 (2 rounds)
shr state, 16
movzbl tmp1, state
movzbh tmp2, state
(mov/xor) accumulator, [T+8*tmp1]
(mov/xor) accumulator, [T+8*tmp2]
...
```

```
// m=16 (1 round)
// The state is in rax
shr rax, 16
mov tmp, ax
(mov/xor) accumulator, [T+8*tmp]
...
```

Рисунок 3.2 – Кількість асемблерних інструкцій в залежності від m

3.2.1 Модель затримки кешу

Для ефективного доступу к пам'яті сучасні процесори використовують різні рівні кешів ($L1$, $L2$ та іноді L3). Відповідно правилам політики кешування найбільш використовувані данні зберігаються в $L1$, потім в $L2$ и т. д. При розгляді таблиці T розміру $|T|$, ймовірність P_{L1} того, що елемент T знаходиться у $L1$ (розмір кешу $|L1|$) дорівнює $P_{L1} = \frac{|L1|}{|T|}$ якщо $|T| > |L1$ та $P_{L1} = 1$ в інших випадках. Крім цього ймовірність того, що в P_{L2} елемент T знаходиться в $L2$, а не у $L1$(з урахуванням що елемент знаходиться або в $L1$ або в $L2$) тобто $|T| \leq |L1| + |L2|$. Таким чином можливо вивести середню затримку l_T для завантаження елементу T під час довільного доступу:

$$l_T = l_{L1} \times P_{L1} + l_{L2} + P_{L2} = l_{L1} \times P_{L1} \times (1 - P_{L1}) \quad (3.1)$$

де l_{L1} та l_{L2} позначають затримку кешів L1 та L2. Оскільки в даній роботі розглядається архітектура x86, а точніше Intel, потрібно зважаючи на надану вище формулу (3.1) враховувати розмір та час очікування на різних кешах. Значення для різних архітектура наведені нижче у таблиці 3.1.

Таблиця 3.1 – Час затримки для архітектури Intel

Архітектура	L1 розмір(кіло-байти)	L1 затримка(цик-ли)	L2 розмір(кіло-байти)	L2 затримка(цикли)
Intel P6	16 або 32	3	512	8
Intel Core	32	3	1500	15
Intel/Hehalen	32	4	256	10
Intel Santy Bridge	32	5	256	12

У результаті аналізу псевдокоду раунду SPN шифру, який було надано на рисунку 3.2 можливо обчислити кількість потребуючих інструкцій для одного раунду з урахуванням m зрізу блока та характеристиками кеш-пам'яті архітектури. Припускається, що регістр для зсуву, переміщення та XOR має затримку в один цикл, а пошук у таблиці має середню затримку в l_T наведену у формулі (3.1). Середня теоретична затримка для різної архітектури наведена у таблиці 3.3 та 3.4(для кожної архітектури середня затримка обчислюється як сума затримок усіх операцій).

Таблиця 3.2 – Теоретична кількість інструкцій для одного раунда(для різних розмірів вхідних таблиць m)

Архітектура	m = 4 бітів	m = 8 бітів	m = 12 бітів	m = 16 бітів
shift	15	3	5	3
move/xor	15	8	5	3
mask	16	0	5	0
пошук у таблиці	16 (32)	8 (16)	6 (12)	4 (8)

Число у дужках позначає вартість для 32-разрядної архітектури, де пошук у таблиці виконується двічі для отриманнях 64-бітного виходу. Проте це вірно лише для регістрів загального призначення. У разі використання регістрів SMID потребується лише один пошук у таблиці, за рахунок додаткових завантажень для збереження стану зі SSE в регістри загального призначення.

Таблиця 3.3 – Середня затримка раунда(для різних розмірів вхідних таблиць m)

Архітектура	m = 4 бітів	m = 8 бітів	m = 12 бітів	m = 16 бітів
Intel P6	142	59	992	93
Intel Core	94	35	91	264
Intel/Hehalen	110	43	68	186
Intel Santy Bridge	126	51	79	114

Варто звернути увагу, що для m = 16 потрібно також враховувати затримку L3 або RAM в залежності від розміру L2 та відповідно розширити рівняння (3.1). Данні були отримані експериментально, реалізувавши та запустивши раунд для шифру з різними значеннями m. В результаті m = 8 кращій компроміс для ефективного використання.

3.3 Опис методу оптимізації таблиці підстановок

З початку створюється 8 таблиць, кожна з них приймає у якості вхідних даних 2 сусідніх 4-бітних S-box(це 8-бітні вхідні данні) та надає 64-бітний вихідні таблиці з урахуванням рівня перемішування. У якості прикладу на рисунку 3.3 наведено опис першої таблиці T_0, котра приймає на вхід 2 найменш значимі 4-бітні слова у стану шифру.

```
// Input: 64-bit state St, round number i  ---  Output: updated 64-bit state St

t0 = T0[ St        & 0xff];    t1 = T1[(St >>  8) & 0xff];    t2 = T2[(St >> 16) & 0xff];    t3 = T3[(St >> 24) & 0xff];
t4 = T4[(St >> 32) & 0xff];    t5 = T5[(St >> 40) & 0xff];    t6 = T6[(St >> 48) & 0xff];    t7 = T7[(St >> 56) & 0xff];
St = t0 ^ t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7;
```

Рисунок 3.3 – Псевдокод раунду шифрування PRESENT

Один раунд шифру виконується 7 зсувами, 8 масками, 8 пошуками у таблиці та 7 операцій XOR. Також потребується 8 таблиць по 2048 байтів, тобто усього 16384 байтів. Такий малий розмір таблиць майже гарантує те, що вони будуть знаходись майже або повністю у кеші L1. Приклад побудування таблиць наведено на рисунку 3.4.

```
// Computation of table T0 for PRESENT
for(i = 0; i < 256; i++)
{
     t   = (SB[(i & 0xf0) >> 4] << 4) | SB[i & 0x0f];
     T0[i]  = ((t >> 0) & 0x01) <<  0;       T0[i] |= ((t >> 1) & 0x01) << 16;
     T0[i] |= ((t >> 2) & 0x01) << 32;       T0[i] |= ((t >> 3) & 0x01) << 48;
     T0[i] |= ((t >> 4) & 0x01) <<  1;       T0[i] |= ((t >> 5) & 0x01) << 17;
     T0[i] |= ((t >> 6) & 0x01) << 33;       T0[i] |= ((t >> 7) & 0x01) << 49;
}
```

Рисунок 3.4 – Приклад функції генерування таблиці T0

Розгортка ключа для шифру потребує значних затрат у програмному забезпечені за рахунок того, що потребується 61-бітний зсув основного ключа(особливо для 80-бітної версії ключа, котра не вміщується в кратний розмір регістру загального призначення x86).

За рахунок використання двох таблиць. Trctr та Tsboxks з 31 та 16-64 бітних слів, можливо вичислити додавання раундового лічільника та пошук S-BOX для розгортки ключа завдяки 1 пошуку у таблиці підстановки та операції XOR. Псевдокод першого раунду наведено на рисунку 3.5, де keyH – найбільш значимі біти головного ключа, keyL – останні 16 біт.

```
// Computation of the PRESENT 80-bit round key i+1
// Inputs: keyL and keyH two 64-bit words, round number i  ---  Outputs: updated keyL and keyH (round-key i+1)
keyH ^= Trctr[i];               t  = keyH;                    keyH <<= 61;        keyH  |= (keyL << 45);
keyH |= (t >> 19);              keyL = (t >>  3) & 0xffff;    t = keyH >> 60;
keyH &= 0x0fffffffffffffff;     t = Tsboxks[t];               keyH  |= t;
```

Рисунок 3.5 – Псевдокод розгортки ключа

3.4 Метод оптимізації bitslice

Основна концепція bitslice моделювання апаратної реалізації у програмному забезпеченні. Весь алгоритм представлений у вигляді послідовності логічних операцій. На процесорі з n-бітовими регістрами логічна інструкція відповідає одночасному виконанню n апаратних логічних операцій.

У реалізації bitslice S-box обчислюється за допомогою бітових логічних інструкцій, а не пошуку у таблицях підстановки. Оскільки час виконання цих конструкцій не залежить від вхідних та ключових значень, реалізація bitslice, як правило стійка до атак часу.

Реалізація Bitslice часто призводить до вражаючих результатів, як показано, наприклад, у [34] для PRESENT та Piccolo. Однак варто врахувати вартість розкладу ключів, яка може бути незначною у кількох типових випадках використання легкої криптографії:

- Невеликі дані.
- Незалежні ключі для різних блоків даних .

Як наслідок, вивчення можливостей bitslice для різних форм розгортання ключа. Для шифрування може бути використано багато різних ключів, а розклади none-bitslice ключів можуть призвести до знищення виграшу паралелізму, якщо упаковувати кожен раундовий ключ (упаковка / розпакування займає ту саму кількість циклів, як і для шифрування у більшості випадків). Перш ніж пояснити метод оптимізації, потрібно зрозуміти як працюють функції SSE(Intrinsic). Важливою особливістю реалізації являється bitslice форма даних. Приклад побудови наведено на рисунку 3.12.

3.4.1 Опис SSE технології

Набір інструкцій, розроблених Intel, і вперше представлений у процесорах серії Pentium III як відповідь на аналогічний набір інструкцій від AMD, який був представлений роком раніше. Початкова назва цих інструкцій була KIN, що розшифровувалася як Katmai New Instructions (Katmai — назва першої версії ядра процесора Pentium III).

Технологія SSE дозволяла вирішити 2 основні проблеми MMX — при використанні MMX неможливо було одночасно використовувати інструкції співпроцесора, оскільки його регістри використовувалися для MMX і роботи з дійсними числами.

У загальному випадку до архітектури процесора додається ряд інструкцій та декілька 128-бітних регістрів. Регістр трактується як два значення з рухомою точкою подвійною точністю (2*64-біт). Але на практиці операції можуть застосовуватись до всіх типів, котрі поміщуються у 16 байтів. SSE містить 8 регістрів, які наведені на рисунку 3.6 та можуть зберігати данні таких типів як вказано на рисунку 3.7.

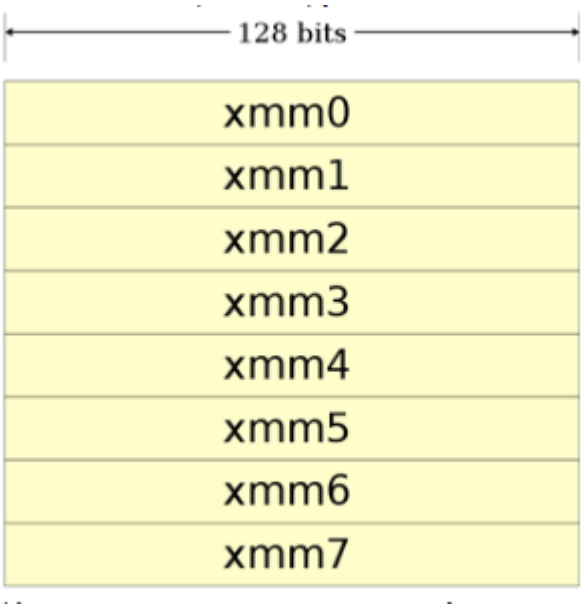

Рисунок 3.6 – Кількість реєстрів SSE

Рисунок 3.7 – Графічне зображення змісту регістрів

За рахунок такої структури з’являється можливість одночасно додати та перемножити операнди з чотирьох чисел за допомогою однієї інструкції. Таким чином для отримання максимальної швидкодії SSE потрібно використовувати такі структури даних щоб вони максимально упаковувались в 128-бітні

регістри. Якщо це не можливо, то існують спеціальні інструкції для цих операцій. На рисунку 3.8 наведено приклад алгоритму упакування операндів.

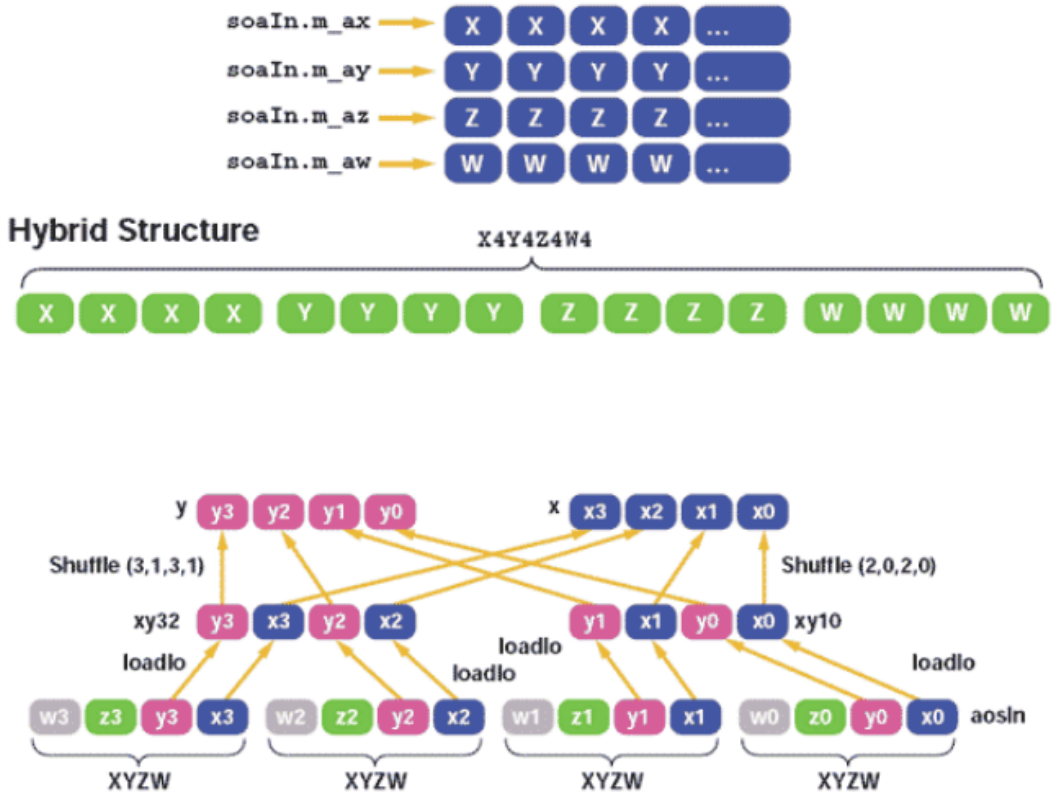

Рисунок 3.8 – Алгоритм упакування операндів

На рисунку 3.9 зображено як зростала кількість реєстрів та операції над ними.

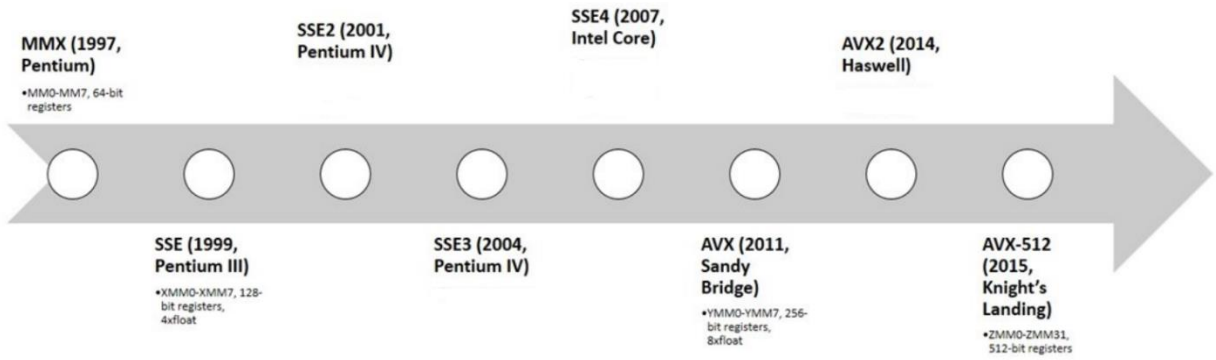

Рисунок 3.9 – Еволюція набору SIMD інструкцій

SSE 2 включає в архітектуру процесора вісім 128-бітних регістрів (xmm0 до xmm7), кожен з яких трактується, як послідовність 4 значень із рухомою крапкою одиничної точності. SSE містить набір інструкцій, які виконують операції зі скалярними і упакованими типами даних.

Перевага у швидкості обчислень досягається в тому випадку, коли необхідно виконати одну і ту саму послідовність дій над різними даними.

Реалізація блоків SIMD виконується розпаралелюванням обчислювального процесу між даними. Тобто коли через один блок даних проходить по черзі багато потоків даних.

Також SSE2 містить вказівки для потокової обробки цілочислових даних у тих же 128-бітних xmm регістрах, що робить це розширення більш продуктивним для цілочислових обрахунків, ніж використання набору вказівок MMX, що з'явилися набагато раніше.

SSE3 (Streaming SIMD Extensions 3, потокове SIMD-розширення процесора, також відоме як PNI(Prescott New Instruction) – це SIMD (Single Instruction, Multiple Data, Одна інструкція – багато даних) набір інструкцій, розроблених Intel, і представлених 2 лютого 2004 року у ядрі Prescott процесора[40].

У 2005 AMD представила свою реалізацію SSE3 для процесорів Athlon 64. Набір SSE3 містить 13 інструкцій: FISTTP (x87), MOVSLDUP (SSE), MOVSHDUP (SSE), MOVDDUP (SSE2), LDDQU (SSE/SSE2), ADDSUBPD

(SSE), ADDSUBPD (SSE2), HADDPS (SSE), HSUBPS (SSE), HADDPD (SSE2), HSUBPD (SSE2), MONITOR (аналога у реалізації

Завантаження та збереження. інструкції вібдуваються у xmm регістрі. Перестановки із 16 та 32-бітиними упакованими операднами(Shuffle) та упаковка та розпакування над векторними даними. Логічне и, або, not, xor. Зсув (упаковані операнди 16, 32 та 64-бітні). Лівостороння да правостороння арифметика(копіювання без знака. Зсув всього регістру xmm побайтно.

Інструкції SSE3:

1. ADDSUBPD (Add Subtract Packed Double).
2. ADDSUBPS (Add Subtract Packed Single).
3. HADDPD (Horizontal Add Packed Double).
4. HADDPS (Horizontal Add Packed Single).
5. HSUBPD (Horizontal Subtract Packed Double).
6. HSUBPS (Horizontal Subtract Packed Single).
7. FISTTP – перетворення дійсного числа в ціле з округленням в меншу сторону.
8. LDDQU – завантаження 128-біт не вирівняних даних із пам'яті в регістр xmm.

Також варто відмітити, що підтримка цих інструкцій наявна не на всіх процесорах.

3.5 Реалізація S-box у техніці bitslice

Перш за все проводиться заміна пошуку у таблиці підстановки на невеликі логічні операції, тим самим збільшується ефективність технікі bitslice. Псевдокод підстановки S-box наведено на рисунку 3.10 а програмна реалізація алгоритму наведено на рисунку 3.1

Розрахунок S-box ведеться за допомогою 5 логічних операцій, а саме:

- Логічне і.
- Логічне або.
- Not.
- Mov.

```
// Input:  r3, r2, r1, r0, tmp
// Output: r3, r2, r1, r0
 1. r2  ^= r1;   r3  ^= r1;
 2. tmp =  r2;   r2  &= r3;
 3. r1  ^= r2;   tmp ^= r0;
 4. r2  =  r1;   r1  &= tmp;
 5. r1  ^= r3;   tmp ^= r0;
 6. tmp |= r2;   r2  ^= r0;
 7. r2  ^= r1;   tmp ^= r3;
 8. r2  = ~r2;   r0  ^= tmp;
 9. r3  =  r2;   r2  &= r1;
10. r2  |= tmp;
11. r2  = ~r2;
```

Рисунок 3.10 – Псевдокод алгоритму S-box

```
// Input: r3, r2, r1, r0, t  ---  Output: r3, r2, r1, r0
#define Sbox(r3, r2, r1, r0, t)
    r2 = XOR(r2,r1);    r3 = XOR(r3,r1);    t  = r2;           r2 = AND(r2,r3);
    r1 = XOR(r1,r2);    t  = XOR(t,r0);     r2 = r1;           r1 = AND(r1,t);
    r1 = XOR(r1,r3);    t  = XOR(t,r0);     t  = OR(t,r2);     r2 = XOR(r2,r0);
    r2 = XOR(r2,r1);    t  = XOR(t,r3);     r2 = ~r2;          r0 = XOR(r0,t);
    r3 = r2;            r2 = AND(r2,r1);    r2 = XOR(r2,t);    r2 = ~r2;
```

Рисунок 3.11 – Програмний алгоритм S-box

3.6 Реалізація дифузійної підстановки

Наступною частиною шифрування являється рівень дифузійної підстановки (linear diffusion layer) у наступному pLayer За стандартним алгоритмом реалізації PRESENT виконується slice по 1 біту. Це дозволяє змінювати лише порядок регістрів без додаткових затрат. Приклад псевдокоду наведено на рисунку 3.12.

За допомогою інструкції pshufb, котра вперше була введена в Intel SSE3 (SSSE3), та інструкцій розпаковки для подвійного слова punpck(h/l) qdq можна виконати обробку дифузійного рівня підстановки. Де h - старші біти, а l - молодші біти у регістрі XMM. За допомогою цих інструкцій можливо

ефективне виконання дифузійної підстановки. Так як для реалізації 16 бітних блоків потрібно двічі виконати 8 бітну паралельну реалізацію, то в роботі буде розглядатись саме 8-бітна версія[34].

Перш за все у регістр XMM за допомогою команди pshufb заноситься r_0, де $n_{i,0}$ лежить у інтервалі $i \le 0 \le 15$. Далі позначимо 64-бітний блок як 16 блоків по 4 біти $n_0,...,n_{15}$. На рисунку 3.11 $n_{i,j}$ це j-тий біт n_i блоку:

$$r[0] : \mathbf{n}_{0,0}||\mathbf{n}_{4,0}||\mathbf{n}_{8,0}||\mathbf{n}_{12,0}||\mathbf{n}_{1,0}||\mathbf{n}_{5,0}||\cdots||\mathbf{n}_{10,0}||\mathbf{n}_{14,0}||\mathbf{n}_{3,0}||\mathbf{n}_{7,0}||\mathbf{n}_{11,0}||\mathbf{n}_{15,0}$$

Тим же методом заповнюються інші 3 регістри. Після цього виконується інструкція punpckhdq для r_0 и r_1 котра розпаковує и розділяє старші біти у подвійних слова r_0 та r_1 у r_0.

Наступні punpckhqdq для r_0 и r_1, де r_2 є результатом r r_2 та r_3, дають потрібні данні для регістру r_0 наступним чином[34] :

$$r[0]: n_{0,0} \| n_{4,0} \| n_{8,0} \| n_{12,0} \| n_{0,1} \| n_{4,1} \| ... \| n_{8,2} \| n_{12,2} \| n_{0,3} \| n_{4,3} \| n_{8,3} \| n_{12,3}$$

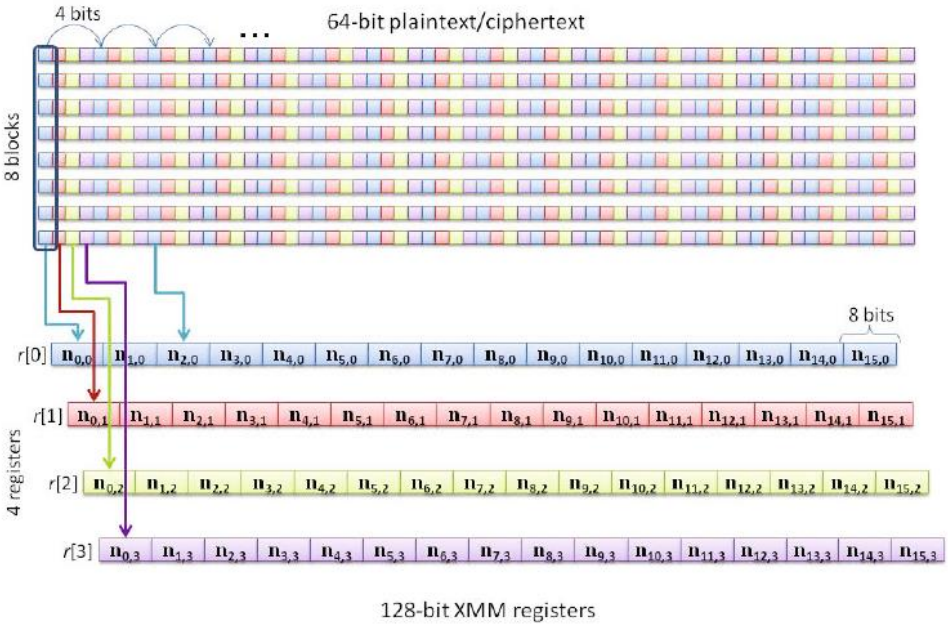

Рисунок 3.12 – Bitslice форма упакованих даних для PRESENT

Для PRESENT функція перестановки бітів pLayer може виконуватися шляхом простої перестановки позицій 16-бітних (або 32-бітних) слів у xmm-регістри у bitslice формі. Під час виконання pLayer за допомогою інструкцій SSE потрібно 20 інструкцій.

```
// Bitslice implementation of pLayer for 8-parallel PRESENT
// Input: r3, r2, r1, r0, t  ---  Output: r3, r2, r1, r0
#define pLayer8(r3, r2, r1, r0, t)
    mask = {0x0f, 0x0b, 0x07, 0x03, 0x0e, 0x0a, 0x06, 0x02, 0x0d, 0x09, 0x05, 0x01, 0x0c, 0x08, 0x04, 0x00};
    r0 = PSHUFB(r0,mask);      r1 = PSHUFB(r1,mask);      r2 = PSHUFB(r2,mask);      r3 = PSHUFB(r3,mask);
    t  = PUNPCKHDQ(r2,r3);     r2 = PUNPCKLDQ(r2,r3);     r3 = PUNPCKHDQ(r0,r1);     r0 = PUNPCKLDQ(r0,r1);
    r1 = PUNPCKHQDQ(r0,r2);    r0 = PUNPCKLQDQ(r0,r2);    r2 = PUNPCKLQDQ(r3,t);     r3 = PUNPCKHQDQ(r3,t);
```

Рисунок 3.13 – Функція перестановки бітів у PRESENT

3.7 Розгортання ключа

Вартість розгортки ключа може бути дуже висока в деяких випадках під час bitslice. Отже потрібно своєчасно спроектувати бітові версії розгорнутого ключа. Це позволить використати паралелізм під час обробки ключів. Також ця процедура підготовить ключі в упакованому виді, отже XOR для кожної bitslice частини буде швидким та простим[34].

У наслідок, формат бітового slice для ключа повинен бути таким як і вхідні дані. Це потрібно для того щоб вартість переупаковки ключа були невелика. Для того щоб звести затрати до мінімуму, упаковка ключа відбувається лише 1 раз для вхідних ключів, на основі котрих формуються підключі за допомогою операцій зсуву та маскування.

Для реалізацій методики bitslice для PRESENT ключ ділиться на 2 блоки (64 та 15 біт для PRESENT-80[144] та два блоки по 64-бітних блока для PRESENT-128) та відображується у бітовий формат за допомогою тієї ж упаковки, що і для вхідних даних

В кінці 61-бітне обертання поділяється на 2 частини. Саме 2 частини, тому що обертання кратне 4 біта може швидко обробити bitslice форма. Спочатку використовується 60-бітне обертання за допомогою інструкцій pshufb (разом з маскування та XOR).

Після чого обертання 1 біта обчислюється шляхом зміни у порядку регістрів xmm (у регістрі xmm знаходяться треті біти Sbox, після зміни будуть знаходитись другі біти Sbox). Після цього необхідно провести редагування, оскільки деякі биті виходять за рамки регістру. Виконати це можливо за допомогою зсувів, масок та XOR як наведено на рисунку 3.13.

```
// Bitslice implementation of the key schedule for 8-parallel PRESENT-80
// Input: k0, k1, k2, k3, k4, k5, k6, k7, t0, t1, t2, t3, t, c0, c1, c2, c3  ---  Output: k0, k1, k2, k3, k4, k5, k6, k7

#define KeySchedule_Bitsliced_Step(k0, k1, k2, k3, k4, k5, k6, k7, t0, t1, t2, t3, t, c0, c1, c2, c3)
    k0 = XOR(k0, c0);    k1 = XOR(k1, c1);    k2 = XOR(k2, c2);    k3 = XOR(k3, c3);
    t1 = PSHIFTR(k1, 1);    k1 = PSHUFB(k1,mask_key1);    t2 = PSHUFB(k5,mask_key2);    k1 = XOR(k1, t2);    k5 = t1;
    t1 = PSHIFTR(k2, 1);    k2 = PSHUFB(k2,mask_key1);    t2 = PSHUFB(k6,mask_key2);    k2 = XOR(k2, t2);    k6 = t1;
    t1 = PSHIFTR(k3, 1);    k3 = PSHUFB(k3,mask_key1);    t2 = PSHUFB(k7,mask_key2);    k3 = XOR(k3, t2);    k7 = t1;
    t1 = k0;                k0 = PSHIFTR(k0, 4);          t2 = PSHIFTL(k4, 12);         k0 = XOR(k0, t2);    k4 = t1;

    t0 = k0; t1 = k1; t2 = k2; t3 = k3;    Sbox(t1, t2, t3, t0, t);

    k0 = AND(k0, (0x0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff));
    t0 = AND(t0, (0xff, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0));        k0 = XOR(k0, t0);
    k1 = AND(k1, (0x0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff));
    t1 = AND(t1, (0xff, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0));        k1 = XOR(k1, t1);
    k2 = AND(k2, (0x0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff));
    t2 = AND(t2, (0xff, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0));        k2 = XOR(k2, t2);
    k3 = AND(k3, (0x0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff));
    t3 = AND(t3, (0xff, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0));        k3 = XOR(k3, t3);

    t0 = k0; t1 = k4;    k0 = k1; k4 = k5;    k1 = k2; k5 = k6;    k2 = k3; k6 = k7;    k3 = t0; k7 = t1;
```

Рисунок 3.13 – Розгортка ключа PRESENT

4 ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДІВ ОПТИМІЗАЦІЇ ШИФРУ PRESENT

4.1 Технології розробки

Програмний код був розроблений на комп'ютері наступної конфігурації:

- Процесор Intel Core i5 7500.
- 8Гб ОЗУ.
- Відеокарта з 1 Гб відеопам'яті.
- Windows 10.

4.2 Побудова програмної реалізації

Для досягнення поставлених задач, необхідно програмно реалізувати блоковий шифр PRESENT. Програма повинна виконувати шифрування вказаного відкритого тексту та виводити час шифрування, який вимірюється у тактах процесору. У даній роботі наведено 3 реалізації шифру: PRESENT, PRESENT Bitslice, PRESENT Table. Всі шифри мають довжину ключа 80-біт. Для перевірки валідності шифротексту було використано тестові вектори, представлені авторами у [9].

4.3 Опис логічної структури

Дана програма складається із 3 програмних модулів. Для спрощення тестування модулів введено інтерфейсну функцію void DoEncrypt(const u64* plaintext_in, const u16* keys_in, u64* ciphertext_out).

У свою чергу ця функція поєднує 2 функції. PRESENT80_key_schedule(const u8* masterKey, u8* roundKeys), котра реалізує логіку розгортки ключа та RESENT80_core(const u8* message, const u8* subkeys, u8* ciphertext) етапу шифрування.

const u8* message – відкритий текст.

const u16* key – мастер-ключ.

u8* ciphertext – результат шифрування.

const u8* masterKey – основний ключ.

u8* roundKeys та const u8* subkeys згенеровані рауднові ключі.

Кожен модуль реалізує свою логіку та у поєднані цих функцій утворює функцію DoEncrypt.

Перший модуль використовується для порівняння швидкості с оптимізованими версіями шифру там містить функцію шифрування.

Другий модуль реалізує метод оптимізації з таблицями підстановок. Модуль реалізує функцію шифрування(DoEncrypt).

Першим етапом генеруються таблиці підстановки. Псевдокод функції було наведено у розділі 3.3 на рисунку 3.4.

У результаті генерації до файлу додаються 8 таблиць: T0_PRESENT, T1_PRESENT, T2_PRESENT, T3_PRESENT, T4_PRESENT, T5_PRESENT, T6_PRESENT, T7_PRESENT та допоміжні таблиці TroundCounters80 та TsboxKS80.

Другий етап це саме функція шифрування. Логіка реалізується завдяки допоміжним функціям:

#define PRESENTROUND(state) – обчислення раунду, де state поточний стан шифру. Псевдокод функції наведено у розділі 4.3.1.

#define PRESENTKS80(keyLow, keyHigh, round) , де keyLow останні головного ключа 16 біт, keyHigh - найбільш значимі біти головного ключа.

Поєднання цих функцій з додаванням ранудової логіки утворю основну функцію DoEncrypt.

Загальний опис алгоритму з урахуванням наведеними вище особливостями його реалізації наведено нижче:

1. Генерація восьми таблиць підстановок .
   1.1. На кожній ітерації подається s-box[i].
   1.2. Виділення старший та молодших біт s-box[i] для і ітерації.
   1.3. Формування T_i таблиці з урахуванням етапу перемішування. Приклад програмної реалізації для генерації T_0 наведено на рисунку 4.1.
2. Генерація допоміжних таблиць Trctr та Tsboxks. Призначення цих таблиць наведено у пункті 3.3. Алгоритм формування Trctr наведений на рисунку 4.2. Алгоритм формування Tsboxks наведений на рисунку 4.3.
3. Для кожного раунду відбувається генерація раундового ключа за допомогою функції PRESENTKS80, яка буде наведена у пункті 4.3.
4. Далі виконується шифрування за алгоритмом наведеним на рисунку 3.4.

```
int MASKm = 0x01;
/* loop over the possible input values to compute for T0, T1, T2, T3, T4, T5, T6 and T7 */
for (i = 0; i < 256; i++)
{
    /* compute low and high parts of i */
    il = i & 0x0f;
    ih = (i & 0xf0) >> 4;

    /* compute two sboxes look-up */
    twoSboxes = (sbox[ih] << 4) | sbox[il];

    /* compute T0 */
    T0_PRESENT[i] = ((unsigned long long)((twoSboxes >> 0) & MASKm)) << 0;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 1) & MASKm)) << 16;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 2) & MASKm)) << 32;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 3) & MASKm)) << 48;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 4) & MASKm)) << 1;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 5) & MASKm)) << 17;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 6) & MASKm)) << 33;
    T0_PRESENT[i] |= ((unsigned long long)((twoSboxes >> 7) & MASKm)) << 49;
}
```

Рисунок 4.1 – Програмний код генерації однієї таблиці підстановки

```
for (i = 0; i < 31; i++)
{
    TroundCounters80[i] = ((unsigned long long)(i + 1)) << 18;
}
```

Рисунок 4.2 – Програмний код генерації однієї таблиці Trctr

```
/* compute TsboxKS80 */
for (i = 0; i < 16; i++)
{
    TsboxKS80[i] = ((unsigned long long)sbox[i]) << 60;
}
```

Рисунок 4.3 – Програмний код генерації однієї таблиці Tsboxks

Другий модуль реалізує метод оптимізації bitslice. У ньому міститься 20 функцій.

Допоміжні функції які використовують SSE (intrinsic функції):

#define CONSTANT(b) установка кожного байту у 128-бітному регістрі значенням b.

#define SET установка байтів у 128-бітному векторі.

#define SET32 установка 32-бітному у 128-бітному векторі.

#define XOR(x,y) XOR x та y, де x та y 128-бітних слова.

#define AND(x,y) логічне множення x та y, де x та y 128-бітних слова.

#define ANDNOT(x,y) логічне множення !x та y, де x та y 128-бітних слова.

#define OR(x,y) логічне і x та y, де x та y 128-бітних слова.

#define LOAD(p) завантаження 16 байтів із адреса пам’яті p, та повернути 128-бітне слово, де p кратне 16.

#define PSHUFB(x,y) порядок байтів упорядкований так як і b.

#define PUNPCKHDQ(x,y).

#define PUNPCKHQDQ(x,y).

#define PUNPCKLDQ(x,y).

#define PUNPCKLQDQ(x,y).

#define PMOVMSKB(x) створює 16-бітну, котра складається з старшого біта кожного байту x.

#define PSHIFTR(x,i) зсув вправо кожне 16-бітне слово x на i позиції.

#define PSHIFTL(x,i) зсув вліво кожне 16-бітне слово x на i позиції.

#define STORE(x,p) зберегти 128-бітний регістр x в пам'ять p.

#define packing8(r0, r1, r2, r3, t0, t1, w0, w1, w2, w3) упаковує біти згідно алгоритму наведеному на рис. 3.11.

#define BitSlice4(a, b, c, d, t0, t1, t2) зріз байтів згідно алгоритму наведеному на рис. 3.1.

#define PRESENT80bitslice8_key_schedule_step(k0, k1, k2, k3, k4, k5, k6, k7, t0, t1, t2, t3, t, c0, c1, c2, c3) розгортка ключа у раунді згідно алгоритму наведеному на рисунку 3.4.

define PRESENT_KEY_SCHEDULE(P, TYPE) розгортка ключа згідно базовому алгоритму PRESENT з використанням PRESENT80bitslice8.

#define pLayer_eight_fast(r3,r2,r1,r0,t) реалізація розгортки ключа згідно алгоритму наведеному на рис. 3.412 Загальний опис алгоритму з урахуванням наведеними вище особливостями його реалізації наведено нижче:

1. Етап розгортки ключа(програмна реалізація функції розгортки наведена на рисунку 3.13.)
   1.1. На вхід подається 8 різних ключів для восьми вхідних текстів.
   1.2. Ключі завантажуються до реєстрів SSE.
   1.3. Упаковуються згідно bitslice форми.
   1.4. Виконується алгоритм розгортки котрий наведений на рисунку 3.13.
2. Етап шифрування
   2.1. На вхід подається 8 різних вхідних текстів.
   2.2. Тексти завантажуються до реєстрів SSE.
   2.3. Упаковуються згідно bitslice форми.
   2.4. Додавання раундового ключа.
   2.5. Підстановка s-box за алгоритмом наведеним на рисунку 3.11.

2.6.Перемішування зашифрованого тексту за алгоритмом наведеним на рисунку 3.13.

4.4 Аналіз швидкодії

Для того щоб порівняти різні методи реалізації, у даному розділі розглядається сервер, що обмінюється даними з пристроями D. Кожен із пристроїв використовує власний унікальний ключ. Для кожного пристрою сервер повинен зашифрувати та розшифрувати В 64-бітних блоків даних. Проте потрібно виділити випадки коли зашифровані данні поступають зі паралельного режиму(CTR) або послідовного(CBC).

- t_E – необхідний час для реалізації процесу шифрування(без розгортки ключа, упаковки та розпаковки вхідних данних)
- P_E – кількість блоків, котрі реалізація шифрує за один раз у процесі шифрування(тобто, кількість використовуваних блоків для реалізації шифру).
- t_{KS} – необхідний час для реалізації процесу розгортки ключа (без урахування упаковки ключових даних).
- P_{KS} – кількість блоків, котрі використовуються у розгортці ключа.
- t_{pack} – необхідний час для упаковки одного блоку.

Потрібно зазначити, що у випадку шифрування кількості блоків менш ніж P_E все одно час потребуючий для розгортки ключа буде дорівнювати t_E. Проте, на відміну від процесу шифрування або розгортки ключів, час упакування та розпакування вхідних / вихідних даних буде залежати від кількості задіяних блоків. З урахуванням того, що t_{pack} час упаковки одного блоку, то час упаковки x блоків буде дорівнювати $x \times t_{pack}$. Відповідно t_{unpack} час для розпаковки одного блоку, та час розпаковки x блоків буде дорівнювати $x \times t_{unpack}$.

Для упакування розгорнутого ключа час позначається як t_{packKS}, а для x ключів відповідно $x \times t_{packKS}$ (для розпакування розгорнутого ключа все так само). Оскільки реалізація на основі таблиць підстановки не пропонує паралельне шифрування блоків то ми маємо: $P_E = P_{KS} = 1$ та $t_{pack} = t_{unpack} = t_{packKS} = 0$.

Таким чином можливо зробити висновок що, час шифрування дорівнює $s(D,B) = t_E + t_{KS} / B$. Напроти реалізація bitslice дозволяє шифрувати декілька блоків паралельно.

Для програмної реалізації цього методу використовується команда __rdtsc[134]. Значення кількості тактів зберігаються у регістрі TSC. Це 64-розрядний регістр процесора, що присутній у всіх процесорах архітектури x86 починаючи з Pentium. Даний регістр є лічильником, що підраховує кількість циклів процесора з моменту перезапуску. Інструкція RDTSC повертає значення TSC у пари регістрів EDX:EAX. У 64-розрядному режимі RDTSC також очищує верхні 32-розрядні половини регістрів RAX і RDX

У свою чергу функція повертає мітку часу процесора. Мітка часу процесора реєструє кількість тактових циклів з моменту останнього скидання. Функція повертає 64-бітне ціле число без знаку, що представляє кількість тактів.

У порівнянні з наданими операційними системами API наприклад WINAPI::QueryPerformanceCounter () або gettimeofday () інструкція __rdtsc може надавати такі переваги:

- Більш висока точність, особливо у випадку застарілих операційних систем, що не мають повноцінної підтримки HPET. Такі ОС використовують системний таймер невисокої точності.
- Менші накладні витрати: інструкції rdtsc виконуються за десятки тактів, що значно менше ніж виконання системних викликів.
- Не вимагають перемикання в привілейований режим Ring0 або в гипервізор в більшості систем (якщо команда дозволена в даній ОС).

4.4.1 Метрики оцінювання програмних реалізацій

Метрика - це спосіб визначення ступеня, в якій реалізація має особливі властивості. Метрики, що використовуються для зважування програмних реалізацій, залежать від цільової архітектури або компілятора / асемблера, використовуваного для генерації двійкового коду.

1. Розмір коду. У той час як розмір апаратної реалізації визначається її областю впровадження, розмір програмної реалізації кількісно визначається його розміром коду. Отже, розмір коду вимірює кількість байтів, необхідних для зберігання двійкового коду в незалежній пам'яті (наприклад, у вигляді оперативної пам'яті, ПЗУ) пристрої[3].

2. Час виконання. Цей показник має ту ж природу, як і час виконання апаратної реалізації. Хоча кількість тактів не залежить від робочої частоти, фактичний час в секундах залежить від частоти тактового сигналу.

3. Споживання ОЗУ. Споживання ОЗУ дає обсяг оперативної пам'яті (в байтах), необхідний для виконання програмної реалізації на процесорі. RAM - це форма енергозалежною пам'яті, яку можна використовувати для зберігання даних програми та її стека виконання.

4. Пропускна здатність. Пропускна здатність є спільною метрикою для апаратних і програмних реалізацій. Як правило, алгоритм може досягати більш високих швидкостей при реалізації в апаратному забезпеченні, ніж коли він реалізований програмно[3].

5. Потужність. Цей показник дає потужність, енергію споживану для операції. Сучасні процесори та мікроконтролери мають кілька режимів споживання, які можна використовувати для оптимізації.

Згідно поданими метриками було оцінено методи оптимізації. Результати наведено у таблиці 4.1. Для вимірювання розміру коду було створено 3 бібліотеки(.lib) та після їх компіляції отримано розмір коду. Споживання ОЗУ було виміряно за допомогою утілити “Performance profiler” яка постачається

разом з Visual Studio 2019. Детальний результат для стандартної реалізації наведено на рисунку 4.4, для table на рисунку 4.5 та для bitslice на рисунку 4.6

Варто зазначити, що еталонний час наведено у роботі[34] і дорівнює 10 тактів процесору на 8.46. Проте варто врахувати що для отримання такого часу було використано AVX розширення.

Пропуска здатність була розрахована без урахування пропускної здатності каналу передачі даних за формулою 4.1, де N = 1000000 та t_{enc} час шифрування одного тексту. У випадку методу bitslice $N = N / 8 = 1000000 / 8 = 125000$.

$$Kpbs = N / t_{enc} \quad (4.1)$$

Таблиця 4.1 – Результати оцінки методів оптимізації згідно метрикам.

	Розмір коду	Час виконання	Споживання ОЗУ	Пропускна здатність	Потужність
Не оптимізована версія	21 кб	202.859	456	4950.496	–
Table	71	113.503	448	8810.334	–
Bitslice	156	60.1683	608	16615.988	–

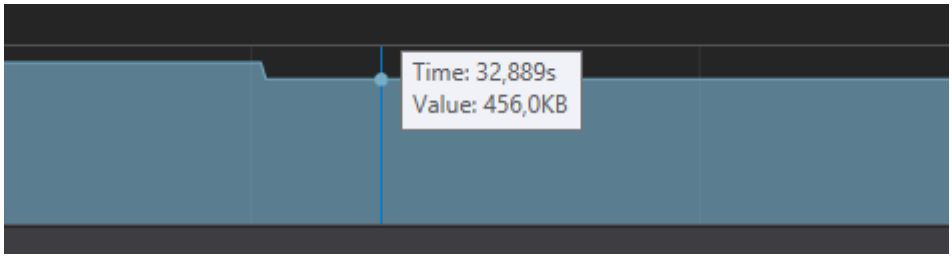

Рисунок 4.4 – Результат профілювання використання пам'яті для не оптимізованої версії

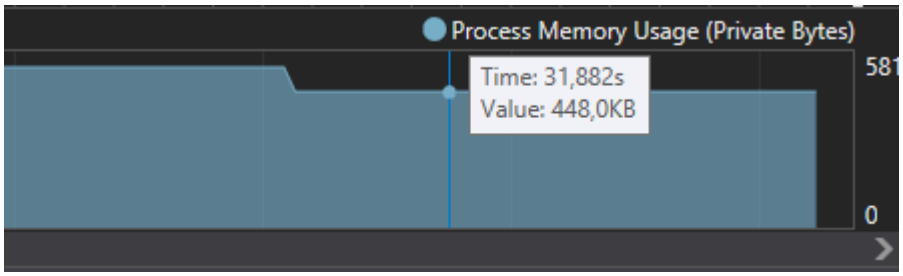

Рисунок 4.5 – Результат профілювання використання пам'яті для table

Рисунок 4.6 – Результат профілювання використання пам'яті для bitslice

4.5 Випадки використання

Для того, щоб мати більш чітке уявлення про різні сценарії, які можуть зустрітись на практиці, пропоную. шість різних та найбільш ймовірних випадків використання, залежно від значення D, B та типу режиму шифрування. Шість ситуацій наведено в таблиці 10 разом із деякими прикладами. Де, S(small) – невелика кількість пристроїв, а B(big) – велика кількість пристроїв.

Можна стверджувати, що перший випадок використання насправді не цікавий, оскільки лише з кількома блоками та кількома пристроями сервер не буде перевантажений роботою шифрування / розшифрування. Однак затримка може бути важливим критерієм саме цей варіант використання перевіряє здатність сервера швидко виконувати криптографічну операцію в програмному забезпеченні.

Можна виділити загальні тенденції випадків використання (див. Таблицю 4.2), і можна зауважити, що реалізації bitslice будуть працювати ефективніше, ніж на основі таблиць, за винятком випадків використання 1 і 3, де задіяно лише кілька пристроїв, а блоки даних можуть обробляються послідовно. Для випадків використання 4 та 6, виграш bitslice від реалізації на основі таблиці стає зрозумілий лише для більш ніж 10 пристроїв.

Для реалізації bitslice перетворення тексту до форми ширфування на сервері може бути видалена, якщо пристрій також шифрує у bitslice форматі. Однак, залежно від типу пристрою, bitslice алгоритм може працювати дуже погано, а вартість зв'язку зросте, якщо використовується послідовний режим або якщо шифрується невелика кількість даних. Більше того, це рішення зменшило б сумісність, якщо іншим учасникам доведеться розшифровувати у bitslice формі.

Таблиця 4.2. – Шість випадків використання пристрою / сервера для легковісного шифрування

	D	B	Режим	Приклад	PRESENT
1	S	S	-	автентифікація та контроль доступу	table
2	S	B	паралельний	захищене потокове спілкування(медичний пристрій, який постійно надсилає конфіденційні дані на сервер, відстежує дані тощо)	bitslice
3	S	B	послідовний	безпечний послідовний зв’язок	table
4	B	S	-	багатокористувацька автентифікація	bitslice

Продовження таблиці 4.2

5	B	B	паралельний	багатокористувацьке безпечне потокове спілкування / хмарні обчислення / мережа датчиків / Інтернет речей	bitslice
6	B	B	послідовний	багатокористувацький захищений послідовний зв’язок	bitslice

4.6 Опис програми

В рамках магістерської дипломної роботи була створена програмна реалізація. PRESENT Bitslice, PRESENT Table. Алгоритми використовують ключ довжиною 80-біт. циклах). В рамках данної роботи нас цікавлять лише час шифрування. Програма має назву «PRESENT.exe» та реалізована у вигляді консольного додатка. Голова функція тестування модулів знаходиться в файлі «PRESENT_IMPL.cpp».

Програма реалізована на мові C++ і може бути запущена з використанням практично будь-якої операційної системи, наприклад, операційних системах сімейства Windows не нижче Windows 98. Для компіляції коду потрібне програмне середовище, яке під тримує мову C++, наприклад, Microsoft Visual Studio 2019. Призначення модулів та можливі приклади їх використання наведено у таблиці 10.

Програма не вимоглива до ресурсів, проте не може бути запущена на будь-якому комп'ютері. Причина такої не універсальності полягає у використанні розширення SSE3, яке підтримується не на усіх ПК.

Можливо зробити висновок, що з точки зору реалізації програмного забезпечення шифр працює досить добре і швидко. Внутрішня раундова

функція цілком підходить для архітектур x86. Табличним реалізаціям допомагає невеликий 64-бітний розмір внутрішнього стану.

4.7 Вхідні та вихідні дані

Для запуску програми необхідно відкрити файл «PRESENT_IMPL.exe». Перед використанням потрібно переконатись, що ПК підтримує розширення SSE3. Далі необхідно вибрати тип оптимізації: Table, Bitslic, Present(шифруваня без оптимізації). Після чого вибрати режим.

Перший режим TEST_VECTOR виконує шифрування тестових векторів, у випадку якщо актуальний шифротекст не буде відповідати очікуваному, то до консолі буде виведено відповідний текст.

Другий OPTIMIZE шифрує випадкові відкриті тексти та виводить час шифрування. Далі по бажанню можливо вказати команди: PERFOMANCE, PHASE_PERFOMANCE. Перша виведе час шифрування функції DoEncrypt. Друга же до загального часу функції також додає час розгортки ключа та час шифрування без розгортки. Команда впливає лише на оптимізовані версії шифру, а саме тому, що для оптимізації функція розгортки ключа виконується спочатку, а потім вже етап шифрування. У звичайній версії етап розгортки виконується під час раунду шифрування.

4.8 Аналіз результатів дослідження

У середовищі C++ з використанням бібліотеки intrin.h розроблено програмну оптимізацію шифру PRESENT. Проведено ряд тестів для шифрування випадкового відкритого тексту на випадкового ключа. У базові версії шифру не використовувалась жодна оптимізація. У випадку оптимізації методом табличних підстановок текст поступав послідовно, а в методі bitslice 8 текстів шифрувались паралельно. У результаті було получено час шифрування для 3 варіантів реалізації шифру наведений на рисунку 4.6.

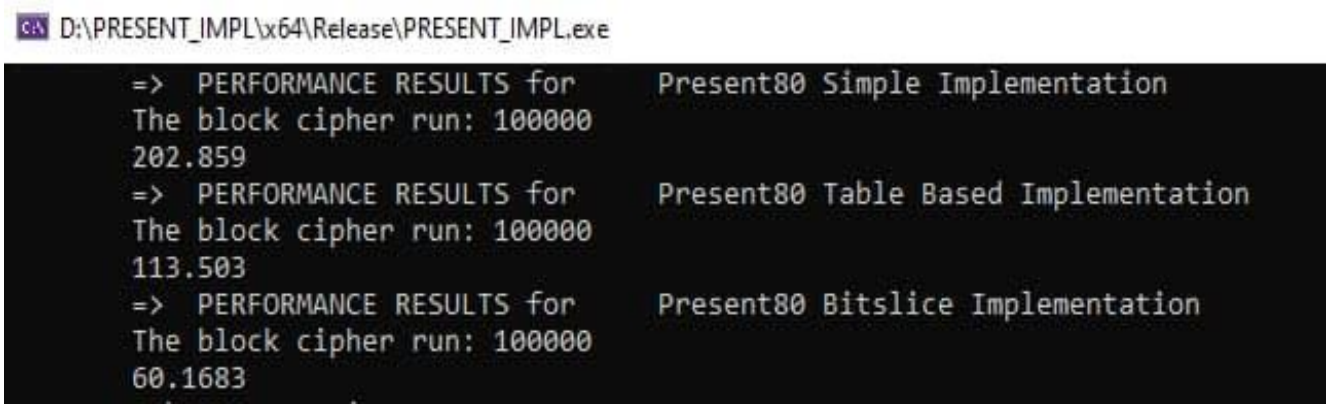

Рисунок 4.6 – Час шифрування для трьох версій шифру

Після отримання таблиць 4.1 та 4.2 можливо зробити висновки щодо кожної реалізації оптимізації.

Для методу таблицями підстановок зменшино час шифрування, та проте збільшили розмір коду втричі. Алгоритм може бути застосований лише для автентифікації, контролю доступу та безпечного послідовного зв’язку. Пропускна здатність алгоритму дозволяє використовувати лише для послідовного шифрування тексту. Дещо зменшилось споживання ОЗУ, проте для пристроїв з обмеженою кількістю пам'яті це може бути значною перевагою.

Для методу bitslice збільшився розмір коду так розмір споживання ОЗУ. Проте ми отримали найліпший результат у час. Прикладів застосування для цього методу значно більше. Вагомим недоліком цієї реалізації являється узгодження форми відкритого тексту. Тобто, або текст буде вже отриманий у bitslice формі та час шифрування ще зменшиться або побудування цієї форми буде займатись сервер. Також важливо своєчасно оновлювати та підтверджувати даний договір.

5 ВИСНОВКИ

У результаті виконання атестаційної роботи проведено дослідницьку роботу, під час якої проаналізовано методи оптимізації bitslice та таблиць підстановок. Розроблено програмну реалізацію цих методів. Оцінено час виконання кожної версії шифру. Запропоновані сфери використання кожного з них. У роботі використано шифрування з 80-бітною довжиною ключа.

Звичайна версія шифру виконується у середньому за 202.859 тактів процесору. Версія з використанням методу таблиць підстановок виконується за 113.503. Версія за використанням методу bitslice з шифруванням восьми тексті виконується за 60.1683. Це в свою чергу вказує що, ми маємо виграш у часі майже більше ніж в 2 рази для версії за таблицями підстановок та в 3.3 раз для bitsice версії.

Кожен метод із двох методів може мати свою сферу використання. У розділі 4.5 наведено ці приклади. Варто відмітити, що техніка btislice має більше сфер застосування.

Висновок зроблено на основі того, що метод дозволяє шифрувати відразу вісім текстів паралельно. Недоліком є побудова bitslice форми для тексту та ключа, проте якщо текст та ключ будуть отримані у даному вигляді, то недолік не тільки зникає, а ще й зростає швидкість шифрування. У подальшому метод bitslice може використовувати замість розширення SSE3 AVX та час шифрування буде ще зменшений.

Метод таблиць підстановок позбавлений цього недоліку, проте й сфера застосування зменшується за рахунок послідовного шифрування.

ПЕРЕЛІК ПОСИЛАНЬ

1. Шнайер, 2002, Типы алгоритмов и криптографические режимы.
2. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. – М.: Триумф, 2002. – 610 с.
3. Жуков А.Е. Легковесная криптография. Часть 1 // Вопросы кибербезопасности. 2015. № 1. С. 26–43.
4. Л. Стасенко Современные технологии радиочастотной идентификации [Текст] / Системы безопасности №2(56), 2004– С. 53. 19. Куроуз Д. Компьютерные сети. Нисходящий подход [Текст] / Росс К., Эксмо – 2016 – С. 912.
5. International Organization for Standardization. ISO/IEC 29192-2:2012, Information technology – Security techniques – Lightweight cryptography – Part 2: Block ciphers, 2012.
6. 147 Баричев С. Г. Основы современной криптографии / С.Г. Баричев, В.В. Гончаров, Р.Е. Серов // – М.: Диалог-МИФИ, 2011. – 176 с.
7. National Institute of Standards and Technology. FIPS 197: Advanced Encryption Standard, November 2001
8. https://uk.wikipedia.org/wiki/SP%D0%BC%D0%B5%D1%80%D0%B5%D0%B6%D0%B0
9. A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In CHES, volume 4727 of LNCS, pages 450–466. Springer, 2007.
10. On the Key Schedule Strength of PRESENT Julio Cesar Hernandez-Castro, Pedro Peris-Lopez pages 5–7. Springer, 2007.
11. M. Albrecht and C. Cid. Algebraic Techniques in Differential Cryptanalysis. In Fast Software Encryption 2009 – FSE 2009, Lecture Notes in Computer Science. Springer-Verlag, to appear., 2009.

12. Gert-Martin Greuel, Gerhard Pfister, and Hans Schonemann. Singular 3.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2005.
13. 115 Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gr¨obner basis computations with Boolean polynomials. In Electronic Proceedings of MEGA 2007, 2007.
14. Nicklas Een and Nicklas Sorensson. An extensible SAT-solver. In Proceedings of SAT '03, pages 502–518, 2003.
15. L. Knudsen and D. Wagner. Integral Cryptanalysis. In Fast Software Encryption — FSE 2002, volume 2365 of Lecture Notes in Computer Science, pages 112–127. Springer-Verlag, 2002.
16. Baudoin Collard and F-X Standaert. A statistical saturation attack against the block cipher present. In Topics in Cryptology–CT-RSA 2009, pages 195–210. Springer, 2009
17. Jorge Nakahara Jr, Pouyan Sepehrdad, Bingsheng Zhang, and Meiqin Wang. Linear (hull) and algebraic cryptanalysis of the block cipher present. In Cryptology and Network Security, pages 58–75. Springer, 2009.
18. K. Ohkuma. Weak keys of reduced-round present for linear cryptanalysis. In Selected Areas in Cryptography, SAC, volume 5867 of LNCS, pages 249-265. Springer, 2009.
19. J.Y.Cho. Linear cryptanalysis of reduced-round present. In Topics in Cryptology - CT-RSA 2010, volume 5985 of LNCS, pages 302
20. J. Nakahara Jr, P. Sepehrdad, B. Zhang, and M. Wang. Linear (hull) and algebraic cryptanalysis of the block cipher present. In Cryptology and Network Security, CANS'09, volume 5888 of LNCS, pages 58.
21. H. Gilbert and M. Minier. A Collision Attack on 7 Rounds of Rijndael. In 3rd AES Candidate Conference, pages 230–241, 2000.
22. M.R. Z'Aba, H. Raddum, M. Henricksen, and E. Dawson. Bit-Pattern Based Integral Attack. In K. Nyberg, editor, Fast Software Encryption — FSE 2008,

volume 5086 of Lecture Notes in Computer Science, pages 363–381. Springer-Verlag, 2008.

23. J. Daemen, L. Knudsen, and V. Rijmen. The Block Cipher Square. In E. Biham, editor, Fast Software Encryption — FSE 1997, volume 1267 of Lecture Notes in Computer Science, pages 149–165. Springer-Verlag, 1997.
24. V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win. The Cipher Shark. In D. Gollmann, editor, Fast Software Encryption — FSE 1996, volume 1039 of Lecture Notes in Computer Science, pages 99–111. Springer-Verlag, 1996.
25. M.R. Z'Aba, H. Raddum, M. Henricksen, and E. Dawson. Bit-Pattern Based Integral Attack. In K. Nyberg, editor, Fast Software Encryption — FSE 2008, volume 5086 of Lecture Notes in Computer Science, pages 363–381. Springer-Verlag, 2008.
26. M.R. Z'Aba, H. Raddum, M. Henricksen, and E. Dawson. Bit-Pattern Based Integral Attack. In K. Nyberg, editor, Fast Software Encryption — FSE 2008, volume 5086 of Lecture Notes in Computer Science, pages 363–381. Springer-Verlag, 2008.
27. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. Submission to NIST, 2008.
28. Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In Clavier and Gaj , pages 272–288.
29. Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In Clavier and Gaj , pages 272–288.
30. Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An Ultra- Lightweight Blockcipher. pages 342–357

31. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A Lightweight Hash. In Stefan Mangard and François-Xavier Standaert, editors, CHES, volume 6225 of LNCS, pages 1–15. Springer, 2010
32. Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON Family of Lightweight Hash Functions. In Phillip Rogaway, editor, CRYPTO, volume 6841 of LNCS, pages 222–239. Springer, 2011
33. Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. spongent: A Lightweight Hash Function. In Preneel and Takagi , pages 312–325.
34. Seiichi Matsuda and Shiho Moriai. Lightweight Cryptography for the Cloud: Exploit the Power of Bitslice Implementation. In Emmanuel Prouff and Patrick Schaumont, editors, CHES, volume 7428 of LNCS, pages 408–425. Springer, 2012.
35. Dag Arne Osvik. Fast assembler implementations of the AES, 2003
36. Joan Daemen and Vincent Rijmen. The Design of Rijndael: AES - The Advanced Encryption Standard. Springer, 2002
37. R. Benadjila, J. Guo, V. Lomner, and T. Peyrin, “Implementing lightweight block ciphers on x86 architectures,” in International Conference on Selected Areas in Cryptography. Springer, 2013, pp. 324–351.
38. Christophe Clavier and Kris Gaj, editors. Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings, volume 5747 of LNCS. Springer, 2009
39. Emilia Kasper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In Clavier and Gaj, pages 1–17.
40. https://uk.wikipedia.org/wiki/SSE