КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№3 (7), 2020
ISSN 2663 - 4023
DOI 10.28925/2663-4023.2020.7.131152
УДК 004.056:061.68

Совин Ярослав Романович
к.т.н., доцент, доцент кафедри захисту інформації
місце роботи: Національний університет «Львівська політехніка», Львів, Україна
ORCID ID: 0000-0002-5023-8442
sovynjarosl@gmail.com

Хома Володимир Васильович
д.т.н., професор, професор кафедри захисту інформації
місце роботи: Національний університет «Львівська політехніка», Львів, Україна
ORCID ID: 0000-0001-9391-6525
volodymyr.v.khoma@lpnu.ua

ПРОГРАМНА BITSLICED-ІМПЛЕМЕНТАЦІЯ ШИФРУ «КАЛИНА» ОРІЄНТОВАНА НА ВИКОРИСТАННЯ SIMD-ІНСТРУКЦІЙ МІКРОПРОЦЕСОРІВ З АРХІТЕКТУРОЮ X86-64

Анотація. Статтю присвячено програмній bitsliced-імплементації шифру «Калина» з використанням векторних інструкцій SSE, AVX, AVX-512 для x86-64 процесорів. Проаналізовано переваги і недоліки різних підходів до ефективної та захищеної програмної реалізації блокових шифрів. Відзначено, що технологія bitslicing поєднує в собі високу швидкодію та стійкість до часових- і кеш-атак, проте наразі відсутні її застосування щодо шифру «Калина». Розглянуто основні підходи до представлення даних і виконання операцій шифру у bitsliced-форматі, особливу увагу приділено ефективній реалізації операції SubBytes, що значною мірою визначає кінцеву швидкодію. Показано, що існуючі методи мінімізації логічних функцій або не дають змогу отримати результат у bitsliced-форматі у випадку 8-бітних неалгебраїчних SBox-ів, або результати далекі від оптимальних. Запропоновано евристичний алгоритм мінімізації логічних функцій, що описують SBox-и «Калини» з використанням операцій AND, OR, XOR, NOT, наявних у системі команд low- та high-end процесорів. У роботі одержані результати, які засвідчили, що для bitsliced-опису одного SBox потрібно близько 520 вентилів, що є відчутно менше ніж забезпечують інші методи. Вказано можливі шляхи збільшення швидкодії завдяки перегрупуванню даних в bitsliced-змінних до і після операції SubBytes, що призводить до ефективнішого використання векторних регістрів. Проведено вимірювання швидкодії bitsliced-реалізацій шифру «Калина» з використанням C++ компіляторів Microsoft та GCC для процесора Intel Xeon Skylake-SP. Одержані у роботі результати bitsliced-реалізації «Калина» можуть бути перенесені і на процесори, які не підтримують SIMD-інструкції, у тому числі low-end, щоб підвищити стійкість до атак через сторонні канали. Також вони дають змогу перейти до апаратної bitsliced-реалізації «Калини» на базі ASIC чи FPGA.

Ключові слова: шифр «Калина»; bitslicing; векторні розширення системи команд; SSE; AVX; AVX-512; SBox; x86-64 архітектура; швидкодія.

1. ВСТУП

Блокові симетричні шифри (БСШ) є основним криптографічним засобом гарантування конфіденційності та цілісності даних в інформаційно-телекомунікаційних системах як у складі протоколів, таких як TLS чи Noise, так і в прикладному чи системному ПЗ. З огляду на постійно зростаючі обсяги та швидкості передачі даних у цих системах дуже важливою властивістю БСШ є забезпечення достатньо високої

© Я. Р. Совин, В.В. Хома, 2020

ISSN 2663 - 4023

продуктивності для широкого класу мікропроцесорних архітектур і, насамперед, для домінуючих на ринку x86-64 платформ.

В Україні до 2015 року діяв стандарт ДСТУ ГОСТ 28147:2009, що визначав як БСШ алгоритм ГОСТ 28147-89 з розміром блоку 64 біти, 256-бітним ключем та 32-ма раундами. Низка виявлених вразливостей, певна моральна застарілість та недостатня швидкодія ГОСТ 28147-89 на сучасних обчислювальних архітектурах загального призначення (порівнюючи з новішими шифрами типу AES та ін.) стали причиною проведення відкритого національного конкурсу з обрання нового стандарту БСШ. Переможцем став шифр «Калина», затверджений із невеликими модифікаціями як стандарт ДСТУ 7624:2014 [1]. Головними вимогам до кандидатів були високий рівень криптографічної стійкості, підтримка 128-, 256- та 512-бітних розмірів блоків і довжин ключів, висока продуктивність у разі програмної реалізації, зокрема передбачалося [2]:

− швидкодія криптоалгоритму повинна бути не меншою, ніж швидкодія ГОСТ 28147-89;

− криптоалгоритм повинен бути орієнтованим для можливості реалізації на 32- або 64-розрядних процесорах;

− операції криптоалгоритму повинні мати ефективну програмну реалізацію;

− давати змогу паралельного виконання декількох операцій (за можливості).

У програмній реалізації БСШ представляється у вигляді програми на мові високого рівня чи асемблера для заданої архітектури процесора. Метою є досягнення максимальної швидкодії за мінімального використання ресурсів (ROM, RAM, стеку, регістрів, енергоспоживання тощо). Не менш важливою для програмної реалізації БСШ є підвищена стійкість до атак через сторонні канали (side-channel attacks): для low-end CPU (8/16/32-бітні мікроконтролери) це насамперед атаки аналізу енергоспоживання, електромагнітні та часові атаки, для high-end CPU (x86, ARM Cortex-A) це передусім часові та кеш-атаки.

Часові атаки можливі коли час виконання програми залежить від даних, що обробляються. Вимірюючи час виконання шифру, зібравши і проаналізувавши відповідну статистику зловмисник може отримати інформацію про секретний ключ. Це призводить до вимоги константного часу виконання (constant-time) криптоалгоритмів. Кеш-атаки враховують різницю в часі доступу до даних, що присутні або відсутні в кеш-пам’яті. Зловмисник маніпулюючи вмістом кешу змінює та вимірює час виконання програми, що дає змогу отримати інформацію про ключ [3] – [4].

Є декілька підходів до програмної реалізації БСШ для low-end та high-end CPU: класичний, табличний, з використанням вбудованих криптоакселераторів, SIMD-інструкцій, bitsliced.

Класичний підхід передбачає покрокове виконання операцій шифру відповідно до специфікації. Ці операції, як правило, реалізуються у вигляді окремих функцій (напр., *SubBytes()*), що викликаються в основному циклі. Перевагою цього підходу є універсальність, тобто можливість реалізації на будь-яких процесорах, невимогливість до ресурсів, простота. Проте це найповільніший підхід, який не використовує можливості сучасних high-end процесорів щодо паралельної обробки даних та інструкцій, крім того такі реалізації здебільшого вразливі як до часових так і кеш-атак.

Багато реалізацій блокових шифрів задля досягнення високої продуктивності замість покрокового виконання операцій використовують передобчислені таблиці (lookup tables), так звані T-таблиці, що реалізують операції раундового перетворення. Робота криптоалгоритму в такому випадку зводиться до пошуку в T-таблиці для кожного байту стану, який виступає індексом, відповідного значення, їх накладанням

між собою та раундовим ключем. Отже, за такого підходу залучені регістри загального призначення (General Purpose Registers, GPR) процесора та базові інструкції системи команд. Табличний підхід використаний у багатьох криптобібліотеках, зокрема й розробниками «Калини» для підтвердження високої швидкодії шифру. Проте такий підхід не позбавлений низки недоліків:

1. Не повністю використовує можливості сучасних high-end процесорних архітектур, щодо паралельної обробки даних (SIMD) та інструкцій (суперскалярність).

2. Є вразливим до кеш-атак (характерно для high-end CPU, які мають кеш).

Т-таблиці в процесі виконання криптоалгоритму кешуються процесором, щоби зменшити час доступу. Раунд полягає в накладанні раундового ключа *key* на матрицю стану *state* і зчитування з Т-таблиць відповідних значень *T[index[i]]*, де *index[i] = state[i] ^ key[i]*. Маніпулюючи кешем і вимірюючи час доступу до даних можливо визначити адреси комірок пам'яті, до яких йде звертання (*index*). Тоді знаючи *state* (наприклад, *state = plaintext* на початку шифрування) можна отримати багато інформації про ключ *key*. Тому такі табличні реалізації є дуже вразливими до кеш-атак, хоча й забезпечують високу швидкодію.

3. Потребує значних об'ємів RAM/ROM для зберігання таблиць, що може бути проблемою для low-end CPU. Наприклад, AES вимагає 8 таблиць по 1 КБайт кожна, а «Калина» 8 таблиць по 16 Кбайт (всього 128 Кбайт). Це може бути проблемою і для high-end CPU, у яких для забезпечення високої продуктивності важливо, щоб таблиці поміщалися у швидкий кеш першого рівня (L1), а він для більшості процесорів має розмір 32-64 КБайт.

Криптоакселератори дають змогу в 10-100 разів прискорити шифрування завдяки наявності спеціалізованих апаратних блоків та розвантажити центральний процесор від інтенсивних криптообчислень. Криптоакселератори присутні як в мікроконтролерах загального призначення у вигляді периферійних модулів, так і high-end CPU, для доступу до яких призначені спеціальні інструкції – AES-NI у x86-процесорах чи інструкції *AESD, AESE, AESIMC, AESMC* у процесорах ARM Cortex-A. Крім збільшення продуктивності, криптоакселератори надають суттєвий виграш у безпеці, оскільки вони виконують операції за константний час і не використовують таблиць, а отже є невразливими до часових і кеш атак. На додаток криптоакселератори спрощують код і зменшують його розмір. Але криптоакселератори БСШ присутні далеко не у всіх процесорах (особливо low-end) та підтримують здебільшого лише алгоритм AES. Також є побоювання наявності закладок, які доступні для спецслужб.

Ще один шлях збільшення швидкодії шифрування в high-end процесорах це використання векторних інструкцій [5]. Векторні інструкції використовують SIMD-технологію (Single Instruction, Multiple Data), коли однією інструкцією одночасно обробляється кілька наборів даних (векторів). Це дозволяє одночасно опрацьовувати однією інструкцією всі байти стану шифру, а в певних випадках і декілька блоків (якщо довжина векторних регістрів більша розміру блоку). Сучасні мікропроцесори з архітектурою x86-64 підтримують кілька наборів векторних інструкцій: SSE, AVX/AVX2, AVX-512 з довжиною вектору 128-, 256- та 512-біт відповідно. Векторні інструкції присутні також у процесорах ARM Cortex-A, зокрема SIMD-розширення NEON оперує 128-бітними регістрами.

Можливість однією інструкцією обробляти від 16 байт і більше, шифрувати одночасно декілька блоків та виконувати операції одночасно різними виконавчими блоками забезпечує високу швидкодію за цього підходу. Також SIMD-інструкції потенційно здатні забезпечити захист від часових і кеш-атак. Проте SIMD-технології

присутні лише в high-end процесорах та й то вибірково (наприклад, технологія AVX-512 тільки з'являється на ринку й наразі підтримується небагатьма CPU). Крім складності імплементації проблемою також є те, що не всі режими роботи шифру можливо розпаралелити.

Постановка проблеми. Отже, є об'єктивна потреба в розробленні таких програмних реалізацій шифру «Калина», які б максимально використовували можливості сучасних high-end мікропроцесорів щодо збільшення швидкодії внаслідок розпаралелювання як виконання коду (суперскалярність), так і обробки даних (векторизація), були невразливі до часових і кеш-атак, а також допускали їх адаптацію для low-end CPU.

Таким підходом є Bitslicing – стратегія імплементації, яка дає можливість швидкої, constant-time імплементації криптографічних алгоритмів з імунітетом до кеш- та часових-атак [6].

Базова ідея Bitslicing це описати шифр у термінах однобітових логічних операцій – AND, XOR, OR, NOT, так як при апаратній реалізації шифру. У процесорах кожну таку логічну операцію можна представити відповідною інструкцією (зокрема і векторною), яка може одночасно обробляти багато біт. Висока швидкодія досягається завдяки тому, що CPU обробляє багато елементів шифру (байтів, блоків) паралельно, використовуючи швидкі логічні інструкції. Чим більша розрядність bitsliced-регістрів, тим більший виграш у швидкодії, тому особливо ефективний цей підхід для векторних інструкцій, які оперують багатобітними регістрами (128-512 біт).

Bitsliced-імплементації володіють потенційно найвищою швидкодією. Наприклад, найшвидша відома x86-реалізація AES, яка не використовує AES-NI, експлуатує саме bitsliced підхід [7]. Для деяких шифрів, таких як DES, AES, Serpent вже є bitsliced-імплементації, проте вони відсутні для шифру «Калина».

Аналіз останніх досліджень і публікацій. Табличні реалізації шифру «Калина» представлені в роботах [8], [9] та криптобібліотеках «Шифр+» v2 [10] і cppcrypto [11].

У [9] наведено результати вимірювання швидкодії шифру «Калина» на x86-64 процесорі Intel Core i5-4670 з тактовою частотою 3,40 ГГц. Для отримання найбільшої швидкодії була обрана мова програмування C++, використано компілятор gcc 4.9.2, тестування здійснювалося на ПК під управлінням 64-бітної ОС Linux (Ubuntu) [8]. Результати тестування швидкодії наведені в таблиці 1. Оскільки в роботі нами для вимірювання швидкодії використовуються такі одиниці як кількість тактів на шифрування одного байту (тактів/байт або cycles per byte, *cpb*), а в [9] як одиниці вимірювання використано Мбіт/с (Mb/s) ми перерахували їх у *cpb* для подальшого порівняння.

У кросплатформенній бібліотеці криптографічних примітивів «Шифр+» v2 [10] досягнуто показників швидкодії представлених у табл. 1. Вимірювання проводилися для x86-64 CPU Intel Core i7-6700HQ 2,6 ГГц з 16 Гбайт ОЗП у середовищі ОС Windows 10 шифруванням блоку 16 Кбайт під час 1 млн повторів.

У документації криптобібліотеки cppcrypto [11] наведено результати вимірювання швидкодії представлені в таблиці 1 (для 64-бітної архітектури). Це кросплатформена C++ бібліотека орієнтована на досягнення максимальної швидкодії. Вимірювання швидкодії здійснювалося шифруванням файлу розміром 130 Кбайт 100000 разів у режимі CBC на процесорі Xeon E5-1650 v3 з тактовою частотою 3,50 ГГц.

Оскільки результати [10], [11] були представлені в Мбайт/с (MB/s), тому для зручності зіставлення їх також перераховано у *cpb*.

ISSN 2663 - 4023

Таблиця 1

Швидкодія табличних реалізацій шифру Калина-128/128

Робота [8]		Бібліотека «Шифр+» v2 [10]		Бібліотека cppcrypto [11]					
				VC++ 2015		gcc 5.2		clang 3.7	
Mb/s	Cpb	MB/s	cpb	MB/s	Cpb	MB/s	Cpb	MB/s	cpb
2611,77	10,41	128,22	20,28	179	19,55	236	14,83	206	16,99

У роботі [12] представлено реалізації шифру «Калина» з використанням SIMD-інструкцій SSE-128, AVX-256, AVX-512. Вимірювання швидкодії велося для процесора Intel Xeon Skylake-SP 2,0 ГГц. Код написано на мові C++, використано компілятори із комплекту Microsoft Visual Studio 2019 (MSVC) та gcc 7.3.0 (GCC). Для зниження впливу переключення контексту процесора виконувалося багатократне шифрування (100 млн повторів). Було оцінено як табличні реалізації шифрів на базі SIMD-інструкцій, які є вразливі до кеш-атак (табл. 2), так і стійкі до кеш-атак реалізації з покроковим виконанням операцій шифру (табл. 3).

Таблиця 2

Табличні реалізації шифру «Калина» (Not Constant Time) [12]

К-ть блоків	GPR				AVX-256				AVX-512			
	GCC, cpb		MSVC, cpb		GCC, cpb		MSVC, cpb		GCC, cpb		MSVC, cpb	
	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC
1-way	9,00	11,25	9,00	10,88	19,50	22,38	19,50	22,00	23,63	26,88	25,38	28,25
2-way	7,75	8,81	7,69	9,00	10,88	12,69	10,63	12,44	13,69	15,94	14,75	16,88
4-way	7,88	8,97	11,06	12,66	9,22	10,41	8,72	9,94	8,25	9,97	8,84	10,59
8-way	-	-	-	-	9,34	10,69	8,42	10,88	5,64	6,95	5,83	7,30
16-way	-	-	-	-	-	-	-	-	5,61	7,09	5,49	7,78

Таблиця 3

Покрокові реалізації шифру «Калина» (Constant Time) [12]

К-ть блоків	SSE-128				AVX-256				AVX-512			
	GCC, cpb		MSVC, cpb		GCC, cpb		MSVC, cpb		GCC, cpb		MSVC, cpb	
	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC
1-way	84,25	107,13	48,75	57,75	38,13	45,38	35,50	43,88	52,38	74,00	57,25	78,63
2-way	50,31	77,88	31,75	44,44	34,75	42,69	24,13	30,94	28,31	43,38	30,00	47,50
4-way	37,94	61,47	24,00	35,88	21,59	29,34	15,09	22,50	18,13	24,69	19,72	25,13
8-way	-	-	-	-	16,42	25,91	10,63	17,19	11,08	16,64	11,73	18,25
16-way	-	-	-	-	-	-	-	-	8,32	15,14	8,78	15,55

З представлених результатів слідує, що використання технології AVX-256 для імплементації шифру «Калини» виправдане за умови необхідності гарантування стійкості до кеш-атак. Це особливо актуально для більшості процесорів, які не мають підтримки AVX-512, тим більше, що перехід до AVX-512 у цьому випадку не дає відчутного приросту швидкодії. Технологія SSE-128 відчутно поступається у швидкодії AVX-256/AVX-512.

Найважчий етап в bitsliced-імплементації, що значною мірою визначає швидкодію загалом, є логічне подання таблиць нелінійної заміни S-Box. Проаналізуємо підходи до представлення S-Box у вигляді комбінаційної логічної схеми з мінімальною кількістю двовходових вентилів AND, OR, XOR, NOT.

Логічні вентилі (Gate Equivalent, GE) у програмній bitsliced-імплементації

ISSN 2663 - 4023

заміняються відповідними інструкціями, які є в наборі практично будь-якого мікроконтролера, low- чи high-end процесора. У деяких процесорах відсутня інструкція NOT, яка емулюється інструкцією XOR, з огляду на те, що $\bar{x} = 1 \oplus x$.

S-Box-и 8×8 *Sbox0-3* та *InvSbox0-3*, присутні в «Калині», можна розглядати як логічні функції задані таблицями істинності. Класичні методи мінімізації логічних функцій заданих таблицею істинності, такі як метод карт Карно чи метод простих імплікант Куайна-Мак-Класкі не підходять у цьому випадку, оскільки здійснюють дворівневу мінімізації використовуючи лише операції AND, OR, NOT (без XOR) та безпосередньо не враховують вимогу двовходовості вентилів.

Світовим стандартом де-факто для мінімізації функцій із великою кількістю змінних є програма Espresso (Espresso logic minimizer), що використовує евристичний алгоритм [13]. Також є похідні від цієї програми, такі як BOOM – має вищу швидкодію, чи Logic Friday – надає графічний інтерфейс до алгоритму Espresso, ABC та інші.

Espresso видає результати, що на практиці дуже близькі до глобального мінімуму, але використовує на декілька порядків менше пам'яті та часу порівнюючи з іншими методами. Також на відміну від інших методів та програм мінімізації вона практично не має обмежень на число вхідних і вихідних змінних. Алгоритм Espresso використовується в багатьох засобах і CAD-пакетах синтезу логічних схем на етапі мінімізації (FPGA, ASIC).

Проте слід зазначити, що використання Espresso теж не є найкращим рішенням, бо зберігається проблема з використанням лише операцій AND, OR, NOT (операція XOR може бути впроваджена вже над результатом мінімізації, але не в процесі) та не враховується обмеження на число входів вентилів.

Результати мінімізації S-Box-ів «Калини» із допомогою Logic Friday представлено в таблиці 4.

Таблиця 4

Мінімізація S-Box шифру «Калина» в програмі Logic Friday

Таблиця	Prime Implicants	Total GE
Sbox0	234	841
Sbox1	231	857
Sbox2	234	854
Sbox3	231	879

Слід враховувати, що в число вентилів входять і елементи NAND, які потребують дві процесорні інструкції в програмному представленні, тому загальне число операцій буде навіть більшим ніж представлене в таблиці. Одержані результати не є задовільними з погляду потенційної швидкодії тому потрібно шукати інші підходи.

Багато робіт присвячено мінімізації S-Box-ів алгоритму AES і тут досягнуто вражаючих результатів: до 128 вентилів на таблицю [14] – [16] (таблиця 5). Основний підхід полягає в розбитті таблиці на лінійну й нелінійну частину та їх мінімізації з використанням евристичних методів. Але ці методи використовують алгебраїчну структуру закладену в AES-таблиці, а мінімізувати формулу значно легше, ніж набір випадкових даних.

ISSN 2663 - 4023

Таблиця 5

Мінімізація S-Box-ів шифру AES

Джерело	Число вентилів
Canright [14]	120 (80 XOR, 34 NAND, 6 NOR)
Boyar [15]	128 (94 XOR, 34 AND)
Reyhani-Light [16]	119 (69 XOR, 43 NAND, 7 NOR, 4 NOT)

Слід зауважити, що представлені результати більше орієнтовані на апаратну, а не програмну реалізацію, тому критеріями мінімізації тут часто виступають не просто мінімальне число вентилів, а площа кристалу, яка необхідна для заданої технології логічних вентилів, або мінімальний шлях проходження сигналу (Critical Path/Depth), мінімальне енергоспоживання та ін. З фізичної точки зору не всі логічні вентилі еквівалентні за цими критеріями, що і враховують відповідні роботи.

У загальному випадку не всі S-Box-и описуються алгебраїчною структурою (як в AES чи китайському стандарті SM4), зокрема це стосується і алгоритму «Калина», тому ці підходи не можуть бути використані безпосередньо в нашому дослідженні.

Для мінімізації невеликих S-Box, які характерні для легковагових шифрів, застосовують програми SAT-Solvers. Підхід до представлення різних критеріїв мінімізації в термінах SAT описано в роботах [17], [18].

Проблемою є те, що такий підхід працює лише для невеликих S-Box-ів, розміром до 4×4 чи 5×5, хоча вже тут виникають складнощі (деякі обчислення потребують до 40 годин). Для 8×8 S-Box цей підхід, на жаль, неможливо реалізувати з погляду обчислювальної складності. Можливо розбивати велику таблицю на менші частини та мінімізувати за допомогою SAT-Solvers кожну частину окремо, проте результат буде далеко не оптимальним. Мінімізовані частини не дадуть глобального мінімуму для всієї таблиці, бо не враховуватимуть залежності між ними.

Отже, жоден з розглянутих методів мінімізації або не дає задовільних результатів або не може бути безпосередньо застосований до шифру «Калина».

Мета статті. Метою статті є вперше представити підходи до ефективної bitsliced-імплементації операцій шифру «Калина», орієнтовані на використання векторних розширень системи команд SSE-128, AVX-256, AVX-512 для процесорів x86-64 та оцінити швидкодію отриманих реалізацій.

2. ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖЕННЯ

2.1. Основні параметри й операції шифру «Калина»

Національний стандарт шифрування ДСТУ 7624:2014 «Калина» [1], [19] належить до SPN, байт-орієнтованих шифрів. Основні параметри шифру, такі як довжина ключа k і блоку даних l, число раундів t та число стовпців матриці стану c пов'язані залежностями представленими в таблиці 6. Довжина блоку і ключа використовуються в позначенні шифру у форматі Калина-l/k.

Таблиця 6

Основні параметри шифру «Калина»

Довжина ключа k, біт	Довжина блоку l, Біт	Число раундів t	Число стовпців матриці стану c
128, 256	128	10	2
256, 512	256	14	4
512	512	18	8

ISSN 2663 - 4023

Під час шифрування операції виконуються над двовимірним масивом байт, названим поточним станом шифру (*State)*. Поточний стан шифру можна представити у вигляді матриці розмірністю $8 \times c$ байтів (вісім рядків по c байт): $State = (s_{i,j})$, де $i = 0,...,7,\ j = 0,...,c-1$.

В алгоритмі використовуються операції арифметичного додавання ($\boxplus$) та віднімання ($\boxminus$) за модулем 2^{64}, додавання за модулем 2 ($\oplus$), нелінійної заміни (*SubBytes*, *InvSubBytes*), циклічного зсуву рядків (*ShiftRows*, *InvShiftRows*) та лінійного перетворення (*MixColumns*, *InvMixColumns*) [1]. Структура шифру «Калина» представлена на рис. 1.

2.2. Векторні розширення системи команд x86-64 процесорів

Сучасні мікропроцесори з архітектурою x86-64 підтримують кілька наборів векторних інструкцій: SSE, AVX/AVX2, AVX-512 [5]. Коротко проаналізуємо їхні можливості та доцільність використання в контексті досягнення поставленої у роботі мети.

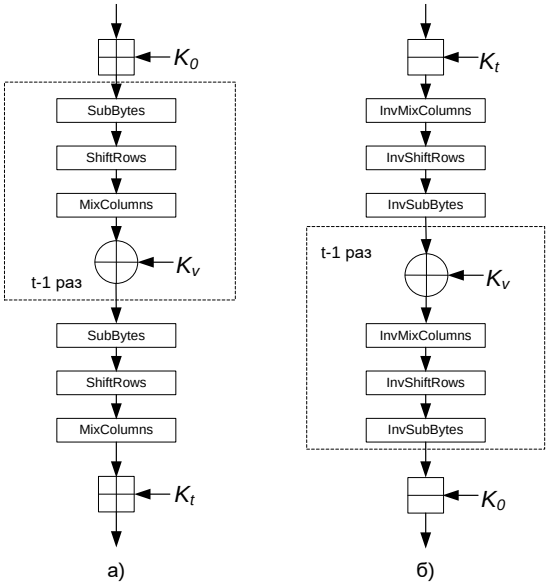


Рис. 1. Структурна схема шифру «Калина» в режимах зашифрування (а) і розшифрування (б)

SSE (Streaming SIMD Extensions) – це набір SIMD-інструкцій, що додає в архітектуру процесора вісім 128-бітних регістрів *xmm0-xmm7* та понад 70 інструкцій для обробки 32-бітних даних типу *float*. У подальшому SSE-технологія доповнилася новими розширеннями, такими як SSE2, SSE3, SSSE3 та SSE4 (у складі SSE4.1 і SSE4.2), які додавали нові інструкції, що значно збільшило її ефективність. З появою 64-бітних процесорів число векторних регістрів у них було збільшено з 8 до 16 (*xmm0-xmm15*). Надалі під SSE чи SSE-128 ми будемо розуміти всі 128-бітні розширення SSE, SSE2, SSE3, SSSE3 та SSE4.

Суттєвим обмеженням SSE-розширень є двоопера́ндний формат SSE-інструкцій ($a = a + b$), за якого вміст одного з операндів втрачається, що вимагає додаткових пересилань даних. Перевагою SSE-технології є те, що вона підтримується фактично

всіма актуальними на сьогодні x86-64 процесорами, оскільки останнє розширення SSE4 з'явилося в процесорах із 2008 року.

AVX (Advanced Vector Extensions) – розширення системи команд x86-мікропроцесорів доступне в процесорах із 2011 року. AVX надає різні удосконалення, зокрема нові інструкції. Розширення AVX2 це подальший розвиток AVX для роботи з цілочисельними операндами, реалізоване в процесорах починаючи з 2013 року. Надалі під позначенням AVX чи AVX-256 ми будемо розуміти всі 256-бітні векторні розширення AVX та AVX2.

У AVX розрядність шістнадцяти SIMD-регістрів збільшується з 128 до 256 біт (*ymm0-ymm15*), з'являються нові інструкції, вводиться неруйнівний формат інструкцій ($c = a + b$), послаблюються вимоги до вирівнювання операндів у пам'яті.

AVX-технологія підтримується більшістю масових процесорів, забезпечує високий рівень паралелізму, надає багатий вибір інструкцій, тому розглядається нами як основна для bitsliced-реалізації шифру «Калина».

AVX-512 є подальшим розширенням 256-бітних інструкцій AVX. Число регістрів збільшується до 32 (*zmm0-zmm31*), а їхня розрядність із 256 до 512 біт, також додано багато нових інструкцій та зросли можливості наявних.

AVX-512 це узагальнена назва для багатьох розширень, які не всі повинні підтримуватися процесорами, що їх впроваджують, за винятком AVX-512F (Foundation). Важливими розширеннями в контексті реалізації криптоалгоритмів є AVX-512 BW (Byte and Word Instructions) – для операцій із 8- та 16-бітними цілими числами і AVX-512 VBMI (Vector Byte Manipulation Instructions) – додає інструкції байтових перестановок *permute*, зокрема *vpermi2b/vpermt2b*, що дають змогу дуже ефективно реалізувати операцію нелінійної заміни *SubBytes*.

Поки що порівняно незначний відсоток процесорів підтримує розширення AVX-512, що почали з'являтися в процесорах лише з 2017 року. Тому технологія AVX-512 розглядається нами наразі для оцінювання потенційної швидкодії шифру «Калина».

3. МЕТОДИКА ДОСЛІДЖЕННЯ

Задля отримання найбільшої швидкодії всі реалізації «Калина» написано на мові C++ з використанням компіляторів із комплекту Microsoft Visual Studio 2019 (надалі MSVC) та mingw-w64 (надалі GCC). Компіляція проводилася за умови максимальної оптимізації з параметрами `/O2` (Maximize Speed) та `-O3` (Optimize fully for speed) для MSVC і GCC відповідно.

Перевірка коректності всіх bitsliced-реалізацій здійснювалася на основі тестових векторів наведених в [19].

Для представлення векторних інструкцій використовувалися intrinsic-функції [20], які є високорівневою C-обгорткою навколо асемблерних команд. Для збільшення швидкодії раундові операції представлялися макросами та здійснювалося розгортання циклів. Під час шифрування використовувалися попередньо обчислені раундові підключі.

Програмна платформа: ОС Microsoft Windows Server (10.0) 64-bit на Google Cloud Platform.

Апаратна платформа: x86-64 процесор Intel Xeon Skylake-SP 2,0 ГГц (*machine types = n1-highcpu-2*) з підтримкою векторних розширень: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, AVX512F, AVX512DQ, AVX512BW, AVX512VL.

ISSN 2663 - 4023

Одиниця вимірювання швидкодії: число тактів на шифрування одного байту – cycles per byte (надалі *cpb*), що дозволяє коректно зіставляти швидкодію CPU з різними тактовими частотами.

Вимірювання числа тактів здійснювалося з допомогою інструкції *rdtsc* для зчитування лічильника *TSC* (Time Stamp Counter), відповідно до методики описаної в [21]. Для зниження впливу переключення контексту процесора виконувалося багатократне шифрування, за кінцевий результат вимірювання приймалося мінімальне значення.

Параметри шифру «Калина»: розмір блоку й довжина ключа 128 біт (Калина-128/128), оскільки такі параметри забезпечують найбільшу швидкодію. Вимірювання швидкодії велося в режимі простої заміни (ECB).

4. РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

4.1. Представлення даних

Наша bisliced імплементація шифру «Калина» значною мірою спирається на підходи запропоновані в роботі [7] для AES, з урахуванням відмінностей у алгоритмах. Порівнюючи з AES bitsliced-реалізація шифру «Калина» ускладнюється низкою чинників:

1. Наявність 4 таблиць нелінійної заміни для прямого і оберненого перетворення (а не однієї як в AES);
2. S-Box-и не мають закладеної алгебраїчної структури, а їх вміст згенеровано як набір випадкових чисел, що, на відміну від AES, не допускає компактного логічного представлення;
3. Матриця перетворення в *MixColumns* має розмір 8×8, а не 4×4 як в AES, та містить коефіцієнти з більшими значеннями: 4-бітні для прямого перетворення і 8-бітні для інверсного (для AES 2-бітні та 4-бітні відповідно), що потребує більшого числа операцій для обчислення модульного множення.

Для початку потрібно визначити, як стан шифру буде представлений у bitsliced-регістрах. Вхідний 128-бітний блок можна розбити таким способом, що кожен біт буде в окремій змінній. Проте такий підхід вимагає надто багато змінних, а крім того, надто зростає розмір повідомлення, яке шифрується, що не завжди можливо чи доцільно на практиці.

Тому нами обрано інше представлення: є 8 bitsliced-змінних *x0*-*x7*, кожна з яких містить біти з відповідним номером – у змінній *x0* зберігаються нульові біти байтів стану, у змінній *x1* – біти з номером 1 байтів стану і т. ін. Відповідно, для одного 16-байтного блоку даних потрібно зберігати в кожній змінній 16 біт. Отож, один 128-бітний регістр *xmm* може вмістити значення бітів для 8 блоків шифру «Калина», *ymm* – для 16 блоків, *zmm* – для 32 блоків (табл. 7).

Позначимо bitsliced стан «Калини» через *x*0-*x*7, де *xi* це значення у векторній змінній. Вісім 16-байтних блоків «Калини» *st0*-*st7* групуються побітово, так що молодший значущий біт кожного байту попадає в *x*0, а старший значущий біт у *x*7.

ISSN 2663 - 4023

Таблиця 7

Обсяг даних при bitsliced обробці

Регістр (розширення)	Біт/байт	Bitsliced-змінні	Число байт	Число блоків
xmm (SSE-128)	128/16	8 (*x*0-*x*7)	8 * 16 = 128	128 / 16 = 8
ymm (AVX-256)	256/32	8 (*x*0-*x*7)	8 * 32 = 256	256 / 16 = 16
zmm (AVX-512)	512/64	8 (*x*0-*x*7)	8 * 64 = 512	512 / 16 = 32

У кожному байті bitsliced-змінної зібрані вісім біт з ідентичних позицій для 8 різних блоків «Калини». На рис. 2 показано порядок біт на прикладі 128-бітних векторних регістрів *xi*.

Для 256/512-бітних регістрів перші 8 блоків розташовуються в перших 128 бітах регістрів *x*0-*x*7 як описано вище, наступні 8 блоків даних у наступних 128 бітах регістрів *x*0-*x*7 і т. ін.

Рис. 2. Перехід від нормального до bitsliced представлення даних

Для упакування і розпакування даних використовуються логічні інструкції та зсуву.

В алгоритмі «Калина» на початку і в кінці перетворення використовується відбілювання на базі арифметичних операцій додавання і віднімання. Оскільки ці операції у bitsliced-представленні досить витратні, бо потребують по 5 вентилів на біт для реалізації повного суматора, перехід до bitsliced-формату здійснюється після виконання початкового відбілювання. Аналогічно наприкінці перетворення перехід від bitsliced до звичайного представлення здійснюється перед кінцевим відбілюванням. Отже, відбілювання виконується традиційним чином на базі векторних операцій додавання і віднімання 64-бітних чисел.

4.2. Bitsliced представлення операції *SubBytes*

Оскільки, як показано вище, наявні методи мінімізації логічних функцій не дають задовільного результату, для bitsliced-представлення S-Box-ів «Калини» нами було розроблено та імплементовано мовою Python власний евристичний метод мінімізації. Розглянемо його на прикладі 8×8 таблиці заміни *Sbox0*.

Таблиці заміни *Sbox0-3* та *InvSbox0-3* наявні в «Калині» можна розглядати як логічні функції 8×8 задані таблицями істинності. Наприклад, фрагмент *Sbox0* має вигляд представлений у табл. 8, де *x0-x7* – вхідні змінні, а *s0-s7* – вихідні.

ISSN 2663 - 4023

Таблиця 8

Фрагмент таблиці істинності заданої для Sbox0

i	x7	x6	x5	x4	x3	x2	x2	x0		s7	s6	s5	s4	s3	s2	s1	s0	Sbox[i]
0	0	0	0	0	0	0	0	0		1	0	1	0	1	0	0	0	0xa8
1	0	0	0	0	0	0	0	1		0	1	0	0	0	0	1	1	0x43
2	0	0	0	0	0	0	1	0		0	1	0	1	1	1	1	1	0x5f
3	0	0	0	0	0	0	1	1		0	0	0	0	0	1	1	0	0x06
4	0	0	0	0	0	1	0	0		0	1	1	0	1	0	1	1	0x6b
5	0	0	0	0	0	1	0	1		0	1	1	1	0	1	0	1	0x75
6	0	0	0	0	0	1	1	0		0	1	1	0	1	1	0	0	0x6c
7	0	0	0	0	0	1	1	1		0	1	0	1	1	0	0	1	0x59
8	0	0	0	0	1	0	0	0		0	1	1	1	0	0	0	1	0x71
9	0	0	0	0	1	0	0	1		1	1	0	1	1	1	1	1	0xdf
10	0	0	0	0	1	0	1	0		1	0	0	0	0	1	1	1	0x87
11	0	0	0	0	1	0	1	1		1	0	0	1	0	1	0	1	0x95
12	0	0	0	0	1	1	0	0		0	0	0	1	0	1	1	1	0x17
13	0	0	0	0	1	1	0	1		1	1	1	1	0	0	0	0	0xf0
14	0	0	0	0	1	1	1	0		1	1	0	1	1	0	0	0	0xd8
15	0	0	0	0	1	1	1	1		0	0	0	0	1	0	0	1	0x09
16	0	0	0	1	0	0	0	0		0	1	1	0	1	1	0	1	0x6d
…	…									…								…

Для кожної з вихідної змінної *s0-s7* довжиною 256-біт її значення послідовно розбиваються на вектори *y_xxxxx* довжиною 16 біт, які кодуються відповідним числом *xxxxx*, що є десятковим представленням цього двійкового вектора. Наприклад, для фрагменту *s7* з табл. 8, який виділено темним кольором, перший вектор буде 0b0110111000000001 = *y_28161*. Оскільки вектори *y_xxxxx* 16-бітні для їх мінімізації використовуються 4 молодші вхідні змінні *x0-x3*: *y_xxxxx = f(x0, x1, x2, x3)*. Змінні *x4-x7* формують 16 масок *m_x*, які накладаються на вектори відповідних вихідних змінних. Для *Sbox0* таблиця істинності набуде вигляду представленого в табл. 9.

Таблиця 9

Представлення *Sbox0* в евристичному алгоритмі мінімізації

Маски *m_x*				*s7*	*s6*	*s5*	*s4*	*s3*	*s2*	*s1*	*s0*
x7	*x6*	*x5*	*x4*								
m_0 = !x7 & !x6 & !x5 & !x4				*y_28161*	*y_25590*	*y_8561*	*y_31652*	*y_49877*	*y_7788*	*y_5662*	*y_40886*
m_1 = !x7 & !x6 & !x5 & x4				*y_3738*	*y_31547*	*y_39875*	*y_36102*	*y_47613*	*y_23781*	*y_13450*	*y_64959*
m_2 = !x7 & !x6 & x5 & !x4				*y_60022*	*y_47114*	*y_2613*	*y_26483*	*y_5067*	*y_34747*	*y_58349*	*y_24334*
m_3 = !x7 & !x6 & x5 & x4				*y_60885*	*y_5626*	*y_59882*	*y_22359*	*y_23236*	*y_11652*	*y_16615*	*y_52792*
m_4 = !x7 & x6 & !x5 & !x4				*y_5927*	*y_10867*	*y_42218*	*y_459*	*y_16193*	*y_62249*	*y_48350*	*y_37900*
m_5 = !x7 & x6 & !x5 & x4				*y_12566*	*y_50958*	*y_14023*	*y_4152*	*y_14473*	*y_24115*	*y_19967*	*y_60098*
m_6 = !x7 & x6 & x5 & !x4				*y_19308*	*y_28310*	*y_12893*	*y_48781*	*y_48694*	*y_33101*	*y_60824*	*y_28360*
m_7 = !x7 & x6 & x5 & x4				*y_11616*	*y_35360*	*y_3807*	*y_48850*	*y_25992*	*y_47924*	*y_10227*	*y_43682*
m_8 = x7 & !x6 & !x5 & !x4				*y_51493*	*y_33726*	*y_40460*	*y_47733*	*y_62327*	*y_37782*	*y_45221*	*y_63233*
m_9 = x7 & !x6 & !x5 & x4				*y_52339*	*y_7141*	*y_42381*	*y_64760*	*y_6173*	*y_7474*	*y_2409*	*y_54645*
m_10 = x7 & !x6 & x5 & !x4				*y_17864*	*y_35113*	*y_56489*	*y_34883*	*y_12385*	*y_40200*	*y_58880*	*y_12558*
m_11 = x7 & !x6 & x5 & x4				*y_32383*	*y_17956*	*y_59067*	*y_23947*	*y_11134*	*y_54046*	*y_57431*	*y_3241*
m_12 = x7 & x6 & !x5 & !x4				*y_38222*	*y_3526*	*y_24983*	*y_22992*	*y_19163*	*y_18831*	*y_35235*	*y_18355*
m_13 = x7 & x6 & !x5 & x4				*y_51491*	*y_58739*	*y_30771*	*y_7409*	*y_39102*	*y_5913*	*y_26263*	*y_17929*
m_14 = x7 & x6 & x5 & !x4				*y_47153*	*y_29750*	*y_18949*	*y_28026*	*y_39585*	*y_45189*	*y_28994*	*y_56928*
m_15 = x7 & x6 & x5 & x4				*y_44365*	*y_23614*	*y_5832*	*y_20482*	*y_14488*	*y_22394*	*y_27856*	*y_9033*

ISSN 2663 - 4023

Наприклад, для *s7* вихідне рівняння буде:

s7 = (m_0 & y_28161) | (m_1 & y_3738) | (m_2 & y_60022) | ... | (m_15 & y_44365).

Вхідними даними для алгоритму є вектори *x0 = 0xaaaa = y_43690, x1 = 0xcccc = y_52428, x2 = 0xf0f0 = y_61680, x3 = 0xff00 = y_65280* (позначені світлим кольором у табл. 8). Також є тривіальні нульовий *0x0000 = y_0* та одиничний вектори *0xffff = y_65535*. Ці вектори шукати не потрібно, бо *m_x & y_0* = 0, а *m_x & y_65535 = m_x*.

Отже, задачу мінімізації можна сформулювати так. Задано набір із 4-х векторів (16-бітних чисел) *x0-x3*. Потрібно використовуючи мінімум операцій AND, OR, XOR, NOT та відповідно проміжних змінних обчислити всі вектори *y_xxxxx*, які входять у задану таблицю нелінійної заміни S-Box.

Вектори *x0-x3* формують базу *base* з якої починається пошук. Для пришвидшення обчислень на першому кроці до бази додаються 22 вектори *r1* (проміжні змінні), утворені з *x0-x3* всіма можливими комбінаціями операціями AND, OR, XOR, NOT. Отже, база *base* складається з 4 + 22 = 26 векторів, які можуть бути використані в наступних обчисленнях.

Далі починає діяти вичерпний пошук (exhaustive search), де спочатку на кроці 1 вектори бази комбінуються між собою всіма доступними способами з використанням операцій *f = AND, OR, XOR, NOT: y = f(base, base)* та перевіряється чи отриманий вектор *y* є в таблиці S-Box. Якщо так, то цей результуючий вектор додається до бази *base* і пошук починається спочатку. Якщо всі можливі комбінації векторів бази перебрано, але залишилося ще не знайдені вектори в таблиці S-Box, тоді починається пошук на кроці 2 з обчисленням проміжних змінних *t_x = f(base, base): y = f(base, t_x) = f(base, f(base, base))*. Якщо знайдено вектор з таблиці S-Box, то він додається до бази *base*, разом із проміжним значенням *t_x* і пошук починається з кроку 1. Пошук продовжується аналогічним чином доти, доки не буде знайдено всі вектори.

Під час кожного занесення в базу *y_xxxxx* відповідні операнди та оператори, а також потрібні для їх обчислення проміжні змінні заносяться в трасу (масив) з якої потім формується система рівнянь.

Оцінимо число вентилів (інструкцій) GE за такого підходу для 8-бітного S-Box. Обчислення масок *m0-m15* потребує 28 GE. Обчислення *r1* потребує 22 GE. Обчислення добутків *m_x & y_xxxxx* та їх сум потребує (16+15)*8 = 248 GE та знаходження 16*8 = 128 векторів із середнім числом операцій на один вектор *x*:

$$Total\ GE\ (8\times8) = 28 + 22 + 248 + 128x = 298 + 128x.$$

Для одночасної мінімізації 2-х таблиць (16 вихідних змінних *s0-s15*) для обчислення добутків *m_x & y_xxxxx* та їх сум потрібно (16+15)*16 = 496 GE та знайти 16*16 = 256 векторів:

$$Total\ GE\ (8\times16) = 28 + 22 + 496 + 256x = 546 + 256x.$$

Для одночасної мінімізації 4-х таблиць (32 вихідні змінні *s0-s31*) для обчислення добутків *m_x & y_xxxxx* та їх сум потрібно (16+15)*32 = 992 GE та знайти 16*32 = 512 векторів:

$$Total\ GE\ (8\times32) = 28 + 22 + 992 + 512x = 1042 + 512x.$$

Для однієї таблиці 8×8 при середньому числі операцій на знаходження одного вектора *x* = 1 отримаємо *Total GE* = 426, при *x* = 2 маємо 554 GE.

Результати застосування розробленого алгоритму до різних S-Box наведено в табл. 10. Записи *Sbox01, Sbox23* означають одночасну мінімізацію двох таблиць заміни алгоритму «Калина» *Sbox0* і *Sbox1* та *Sbox2* і *Sbox3* відповідно (вони використовуються для оптимізації в п. 7.4). Запис *Sbox03* це одночасна мінімізація всіх 4 таблиць *Sbox0-Sbox3*. Таблиці для AES і SM4 наведено для оцінки алгоритму та порівняння їхніх

властивостей з «Калиною», хоча з практичної точки зору застосування запропонованого евристичного алгоритму до них немає сенсу, оскільки вони мають алгебраїчну структуру, що дозволяє значно ефективнішу оптимізацію. Отримані результати значно кращі за ті, що були здобуті з використанням Espresso.

Розроблена програма евристичної мінімізації на Python автоматично формує за знайденими рівняннями С-функції для різних векторних розширень SSE-128, AVX-256, AVX-512, що разом з вихідними кодами доступні за посиланням [22].

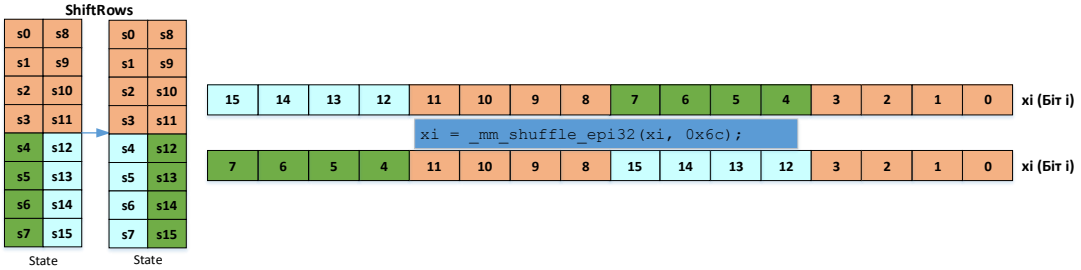
Таблиця 10

Результати мінімізації S-Box-ів розробленим евристичним алгоритмом

S-Box	GE		
	Калина	**AES**	**SM4**
Sbox0	518	513	523
Sbox1	512		
Sbox2	522		
Sbox3	514		
InvSbox0	522	519	-
InvSbox1	523		
InvSbox2	519		
InvSbox3	516		
Sbox01	912	-	-
Sbox23	908		
InvSbox01	926		
InvSbox23	910		
Sbox03	1580		
InvSbox03	1590		

4.3. Bitsliced представлення операцій *ShiftRows, MixColumns* та *AddRoundKey*

Операція *ShiftRows* здійснює ротацію 4-х старших рядків матриці стану на один байт як показано на рис. 3 (а). За вибраного bitsliced-формату, коли кожен байт 0-15 змінної *x* містить біти з ідентичних позицій 8 блоків «Калини», *ShiftRows* потребує перестановок 8 байтів у кожній змінній *x0-x7*, групами по 32-біти. Біти 4-го байту стануть бітами 12-го і навпаки і т. ін. Це виконується для кожного регістру *xi* (*x0-x7*) інструкціями *shuffle* як показано на рис. 3 (б).



а) *б)*

Рис. 3. Операція ShiftRows (а) та її bitsliced-представлення (б)

Операції *MixColumns* та *InvMixColumns* здійснюють множення матриці стану *State* (*s0-s15*) на матрицю 8×8 коефіцієнтів за модулем твірного поліному $poly = x^8 + x^4 + x^3 + x^2 + 1$ *(0x11d)*.

ISSN 2663 - 4023

Розглянемо виконання *MixColumns* на прикладі одного стовпця матриці стану *s0-s7*.

$$\begin{bmatrix} 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 \\ 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 \\ 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 \\ 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 \\ 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 \\ 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 \\ 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 \\ 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 \end{bmatrix} \otimes \begin{bmatrix} s0 \\ s1 \\ s2 \\ s3 \\ s4 \\ s5 \\ s6 \\ s7 \end{bmatrix}$$

Під час модульного множення враховується, що коефіцієнти матриці можна виразити через степені 2: $4=2^2$, $5=2^2+2^0$, $6=2^2+2^1$, $7=2^2+2^1+2^0$, $8=2^3$, тоді:

$$1\otimes\begin{bmatrix} s0\oplus s1\oplus s2\oplus s3\oplus s6 \\ s1\oplus s2\oplus s3\oplus s4\oplus s7 \\ s2\oplus s3\oplus s4\oplus s5\oplus s0 \\ s3\oplus s4\oplus s5\oplus s6\oplus s1 \\ s4\oplus s5\oplus s6\oplus s7\oplus s2 \\ s5\oplus s6\oplus s7\oplus s0\oplus s3 \\ s6\oplus s7\oplus s0\oplus s1\oplus s4 \\ s7\oplus s0\oplus s1\oplus s2\oplus s5 \end{bmatrix} \oplus 2\otimes\begin{bmatrix} s5\oplus s6 \\ s6\oplus s7 \\ s7\oplus s0 \\ s0\oplus s1 \\ s1\oplus s2 \\ s2\oplus s3 \\ s3\oplus s4 \\ s4\oplus s5 \end{bmatrix} \oplus 4\otimes\begin{bmatrix} s2\oplus s5\oplus s6\oplus s7 \\ s3\oplus s6\oplus s7\oplus s0 \\ s4\oplus s7\oplus s0\oplus s1 \\ s5\oplus s0\oplus s1\oplus s2 \\ s6\oplus s1\oplus s2\oplus s3 \\ s7\oplus s2\oplus s3\oplus s4 \\ s0\oplus s3\oplus s4\oplus s5 \\ s1\oplus s4\oplus s5\oplus s6 \end{bmatrix} \oplus 8\otimes\begin{bmatrix} s4 \\ s5 \\ s6 \\ s7 \\ s0 \\ s1 \\ s2 \\ s3 \end{bmatrix}$$

Множення числа *x* на 2 в полі $GF(2^8)$ за модулем *poly = 0x11d* еквівалентно зсуву числа *x* вліво й додавання за модулем два з незвідним поліномом *poly*, якщо старший біт *x* був рівний 1. Отже, у *MixColumns* потрібно здійснити три модульні множення, байтові перестановки та сумування за модулем два, які потрібно перекласти на bitsliced-представлення даних. Розглянемо на рівні байту *b* (біти *b0-b7*), як відбувається множення на 2 за модулем твірного поліному *poly=0x11d*:

B	*b7*	*b6*	*b5*	*b4*	*b3*	*b2*	*b1*	*b0*
b << 1	*b6*	*b5*	*b4*	*b3*	*b2*	*b1*	*b0*	*0*
poly & b7	*0*	*0*	*0*	*b7*	*b7*	*b7*	*0*	*b7*
$2\otimes b$	*b6*	*b5*	*b4*	$b3\oplus b7$	$b2\oplus b7$	$b1\oplus b7$	*b0*	*b7*

Отже, якщо ми маємо 8 змінних *x0-x7* в яких зберігаються відповідно біти 0-7 то множення на 2 для SSE-128 реалізується так:

```
tmp = x7;
x7  = x6;
x6  = x5;
x5  = x4;
x4  = _mm_xor_si128(x3, tmp);
x3  = _mm_xor_si128(x2, tmp);
x2  = _mm_xor_si128(x1, tmp);
x1  = x0;
x0  = tmp;
```

Перестановки здійснюються інструкціями *shuffle* для SSE-128/AVX-256, а для AVX-512 використовується інструкція циклічного зсуву вправо *_mm512_ror_epi64(reg, bits)* [20].

Аналогічним чином виконуються операція *InvMixColumns*, але тепер потрібно обчислювати не тільки 2*s*, 4*s*, 8*s*, але і 16*s*, 32*s*, 64*s*, 128*s*. Отже, операція розшифрування завжди буде повільніша за операцію зашифрування в програмній реалізації.

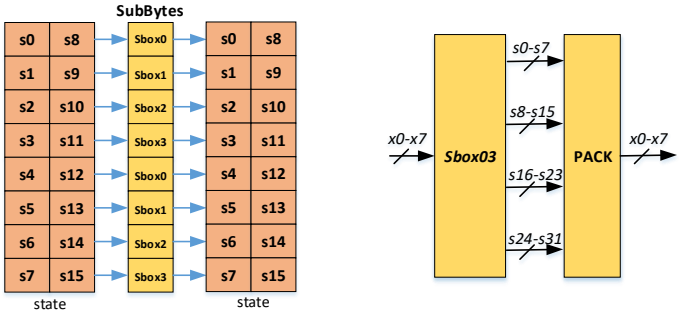
Раундові ключі перетворюються в bitsliced-представлення під час розгортання ключа. Кожен з 128-бітних ключів *rk1-rk9* конвертується у вісім bitsliced-значень (перший і останній раундові ключі *rk0*, *rk10*, що беруть участь у відбілюванні, не

конвертуються), які в раунді операцією XOR накладаються на відповідні регістри x*0-x7*. Отже, загальне число раундових ключів Калини-128/128 становить 2 + 9*8 = 74.

6.4. Оптимізація bitsliced представлення операції *SubBytes*

У шифрі «Калина» використовуються чотири таблиці заміни *Sbox0-Sbox3*, через які проходять байти стану як показано на рис. 4 (а). При bitsliced виконанні операції *SubBytes* вісім регістрів *x0-x7* подаються на вхід функції *Sbox03* (яка забезпечує найменше сумарне число інструкцій) для обчислення 4-х таблиць *Sbox0-Sbox3*. Отримаємо 32 змінні *s0-s31*, що відображають вихідні значення для кожної з таблиці, які потім повинні бути упаковані процедурою *PACK* у регістри *x0-x7* відповідно до рис. 4 (б).

Розглянемо більш детально операцію *PACK* на прикладі 128-бітних *xmm*-регістрів. Як показано на рис. 5 під час упакування більшість байтів у регістрах *s0-s31* не використовуються (позначені червоним кольором) і фактично відкидаються. Так із регістрів *s0-s7* потрібні лише байти з номерами 0, 4, 8 та 12. Отож, ці регістри використовуються неефективно (на 25%), що зменшує швидкодію bitsliced-імплементації.



а) *б)*

Рис. 4. Операція SubBytes (а) та її bitsliced представлення (б)

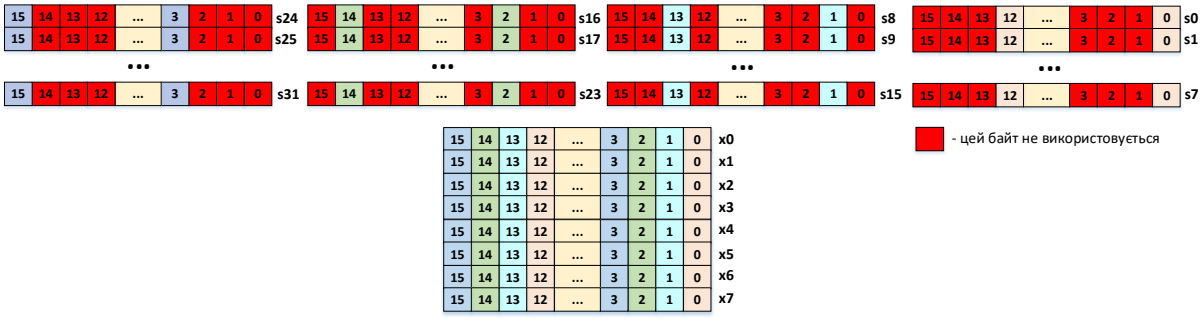


Рис. 5. Упакування результатів проходження 4-х таблиць в bitsliced змінні x0-x7

Щоб позбутися цього недоліка нами пропонується здійснювати bitsliced-шифрування одночасно не 8 блоків даних, а 16 або 32 з відповідним групуванням даних у bitsliced-регістрах перед і після обчислення S-Box. Будемо вважати 8 блоків даних шифру «Калини» набором *v*.

Розглянемо спочатку випадок з двома наборами *v0-v1* по 8 bitsliced-регістрів. Тоді перед операцією *SubBytes* байти, які проходять через *Sbox0-Sbox1* групуються в регістрах набору *v0*, а байти, які пропускаються через *Sbox2-Sbox3* групуються в регістрах набору *v1*. Далі до регістрів набору *v0* застосовується bitsliced-функція

ISSN 2663 - 4023

Sbox01, а до регістрів *v1 – Sbox23*. Схема перегрупування даних та виконання операції *SubBytes* для випадку двох наборів регістрів показана на рис. 6.

У такий спосіб, вдається вдвічі (до 50%) підвищити ефективність використання регістрів (рис. 7) завдяки невеликим накладним витратам на перегрупування перед і після процедури *SubBytes*. Водночас додатково дещо зростає число інструкцій для обчислення самих таблиць *Sbox01-Sbox23*: 912 + 908 = 1820 проти 1580 інструкцій *Sbox03* для зашифрування і 926 + 910 = 1836 проти 1590 інструкцій для розшифрування.

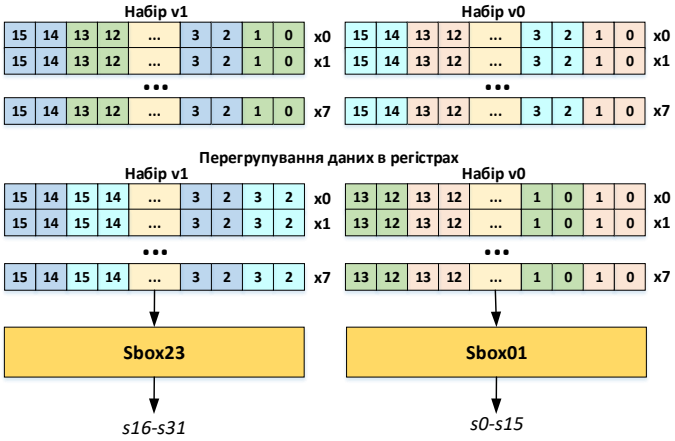


Рис. 6. Перегрупування двох наборів даних v0-v1 в bitsliced-регістрах перед виконанням операції SubBytes

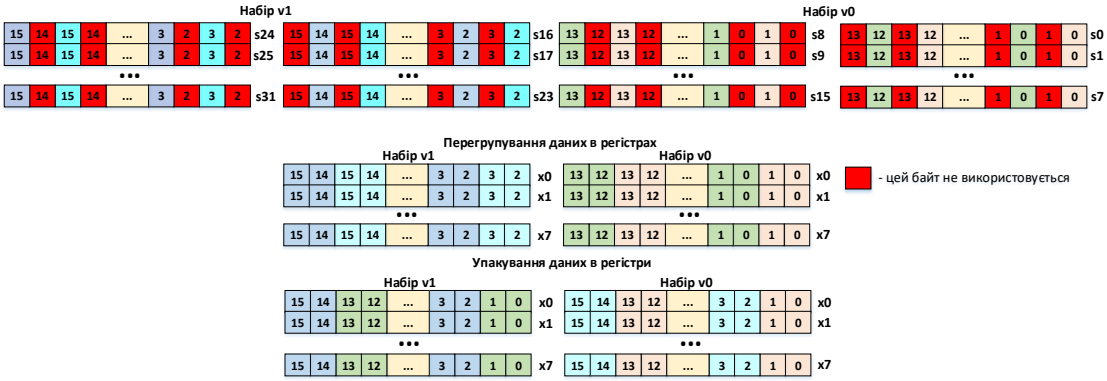


Рис. 7. Упакування результатів проходження 4-х таблиць в bitsliced змінні x0-x7 для випадку двох наборів даних v0-v1

Ще більшої оптимізації можна досягнути у випадку 4-х наборів регістрів *v0-v3*, що забезпечує майже чотирикратне зменшення числа операцій незважаючи на складнішу процедуру перегрупування. Схема перегрупування та виконання операції *SubBytes* для випадку чотирьох наборів регістрів показана на рис. 8. Досягається 100% використання регістрів, але одночасно зростають як накладні витрати на перегрупування, так і число інструкцій на обчислення *Sbox0-Sbox3*, оскільки згідно з даними в табл. 10 сумарно вони становлять 518 + 512 + 522 + 514 = 2066 інструкцій проти 1580 інструкцій у випадку *Sbox03*. Для розшифрування будемо мати 522 + 523 + 519 + 516 = 2080 проти 1590 інструкцій *InvSbox03*.

ISSN 2663 - 4023

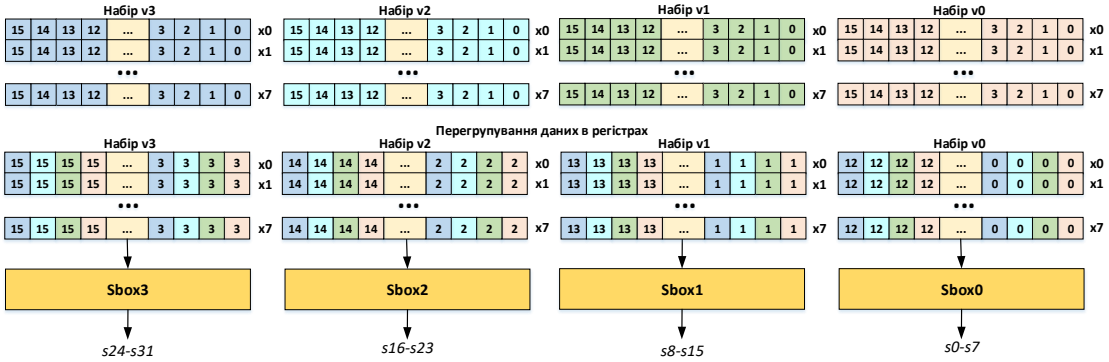


Рис. 8. Перегрупування чотирьох наборів даних v0-v3 в bitsliced-регістрах перед виконанням операції SubBytes

У табл. 11 представлено результати вимірювання швидкодії за різного числа наборів даних *v*, які шифруються: 1 набір з 8 bitsliced-змінних (**8-way**), 2 набори з 16 bitsliced-змінних (**16-way**), 4 набори з 32 bitsliced-змінних (**32-way**).

Таблиця 11

Програмні bitsliced-реалізації шифру «Калина»

Число змінних	SSE-128				AVX-256				AVX-512			
	GCC, cpb		MSVC, cpb		GCC, cpb		MSVC, cpb		GCC, cpb		MSVC, cpb	
	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC
8-way	91,25	111,81	83,25	94,05	40,26	49,95	39,58	43,88	27,01	33,30	26,29	30,11
16-way	56,19	76,19	49,65	58,26	24,23	31,70	22,87	29,00	15,10	19,24	15,45	17,84
32-way	37,55	55,86	31,55	47,22	16,14	24,04	16,00	19,77	9,79	14,31	9,83	12,20

Загалом максимальна швидкодія bitsliced-реалізацій шифру «Калина», відображена в табл. 11, поступається швидкодії стійких до кеш-атак програмних реалізацій представлених у роботі [12] (табл. 3). Перехід від технології SSE-128 до AVX-256 майже вдвічі збільшує швидкодію, проте перехід від AVX-256 до AVX-512 збільшує її не так відчутно, що можна пояснити тим, що нові інструкції ще недостатньо оптимізовані на апаратному рівні.

5. ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

У роботі представлено стійкі до кеш-атак програмні bitsliced-реалізації шифру «Калини» на базі векторних інструкцій SSE-128, AVX-256 і AVX-512 та здійснено оцінювання швидкодії для однакової програмно-апаратної платформи. Їхня швидкодія виявилася дещо нижчою за результати отримані в роботі [12], що пояснюється наявністю в «Калині» 4-х неалгебраїчних таблиць нелінійної заміни та складністю їх bitsliced-представлення. Проте, на відміну від [12], bitsliced-реалізації шифру «Калина» можуть бути перенесені і на процесори, які не підтримують SIMD-інструкції, у тому числі мікроконтролери, щоб підвищити стійкість до атак через сторонні канали. Також можливий перехід від програмної bitsliced-реалізації до апаратної на базі ASIC чи FPGA.

Потенційним ресурсом збільшення швидкодії bitsliced-реалізацій шифру «Калини» є подальша мінімізація опису таблиць нелінійної заміни, наприклад, шляхом

переходу від 16-бітних векторів до 32-бітних у рамках запропонованого в цій роботі евристичного методу мінімізації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] ДСТУ 7624:2014. Інформаційні технології. Криптографічний захист інформації. Алгоритм симетричного блокового перетворення. – К.: Мінекономрозвитку України, 2015. [Електронний ресурс]. Доступно: http://ukrndnc.org.ua/downloads/new_view/?i=dstu-7624-2014&pz=%C4%D1%D2%D3+7624%3A2014.

[2] О. О. Кузнецов, Р. В. Олійников, Ю. І. Горбенко, А. І. Пушкарьов, О. В. Дирда, та І. Д. Горбенко, "Обґрунтування вимог, побудування та аналіз перспективних симетричних криптоперетворень на основі блочних шифрів", *Вісн. Нац. ун-ту "Львів. політехніка"*, № 806, с. 124-140, 2014.

[3] C. Rebeiro, D. Mukhopadhyay, S. Bhattacharya, *Timing Channels in Cryptography. A Micro-Architectural Perspective*. Cham, Switzerland: Springer, 2015. DOI: https://doi.org/10.1007/978-3-319-12370-7

[4] Q. Ge, Y. Yarom, D. Cock, G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2016. DOI: https://doi.org/10.1007/s13389-016-0141-6

[5] D. Kusswurm. *Modern x86 Assembly Language Programming 32-bit 64-bit SSE and AVX*. N.-Y., USA: Apress, 2014. DOI: https://doi.org/10.1007/978-1-4842-0064-3

[6] E. Biham, "A fast new DES implementation in software". In: Biham E. (eds) *Fast Software Encryption*. FSE LNCS, vol. 1267, pp. 260-272, 1997. DOI: https://doi.org/10.1007/BFb0052352

[7] E. Käsper, P. Schwabe, "Faster and Timing-Attack Resistant AES-GCM", in *Proc. 11th International Workshop Cryptographic Hardware and Embedded Systems*, Lausanne, Switzerland, 2009, pp. 1-17. DOI: https://doi.org/10.1007/978-3-642-04138-9_1

[8] Р. Олійников, І. Горбенко, О. Казимиров, В. Руженцев, та Ю. Горбенко, "Принципи побудови і основні властивості нового національного стандарту блокового шифрування України", *Захист інформації*, Том 17, № 2, с. 142-157, 2015. DOI: https://doi.org/10.18372/2410-7840.17.8789

[9] R. Oliynykov, O. Kazymyrov, O. Kachko, R. Mordvinov, et al. Source code for performance estimation of 64-bit optimized implementation of the block ciphers Kalyna, AES, GOST, BelT, Kuznyechik. [Електронний ресурс]. Доступно: https://github.com/Roman-Oliynykov /ciphers-speed/.

[10] В. Ковтун, А. Охрименко. Особенности построения кроссплатформенной библиотеки криптографических примитивов "Шифр+" v2. [Електронний ресурс]. Доступно: https://cipher.com.ua/media/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D1%8B/%D0%A8%D0%B8%D1%84%D1%80%2Bv2.1/Presentation_Cipher_Plus.pdf.

[11] cppcrypto library. Encryption performance. [Електронний ресурс]. Доступно: http://cppcrypto.sourceforge.net/.

[12] Я. Р. Совин, В. В. Хома, Ю. М. Наконечний, та М. Ю. Стахів, "Ефективна імплементація та порівняння швидкодії шифрів «КАЛИНА» та ГОСТ 28147-89 за використання векторних розширень SSE, AVX та AVX-512", *Захист інформації*, Том 21, № 4, с. 207-223, 2019. DOI: https://doi.org/10.18372/2410-7840.21.14266

[13] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*. Hingham, USA: Kluwer Academic Publishers, 1984. DOI: https://doi.org/10.1007/978-1-4613-2821-6

[14] D. Canright, "A very compact S-Box for AES", in *Proc. 7th International Workshop Cryptographic Hardware and Embedded Systems*, Edinburgh, UK, 2005, pp. 441-455. DOI: https://doi.org/10.1007/11545262_32

[15] J. Boyar, and R. Peralta, "A small depth-16 circuit for the AES S-Box", *IFIP Advances in Information and Communication Technology*, vol. 376, pp. 287–298, 2012. DOI: https://doi.org/10.1007/978-3-642-30436-1_24

ISSN 2663 - 4023

[16] A. Reyhani-Masoleh, M. Taha, and D. Ashmawy, "Smashing the implementation records of AES S-Box", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, no. 2, pp. 298–336, 2018. DOI: https://doi.org/10.13154/tches.v2018.i2.298-336

[17] K. Stoffelen, "Optimizing S-Box Implementations for Several Criteria Using SAT Solvers", in *Proc. 23rd International Conference on Fast Software Encryption*, Bochum, Germany, 2016, pp. 140-160. DOI: https://doi.org/10.1007/978-3-662-52993-5_8

[18] N. Courtois, T. Mourouzis, and D. Hulme, "Exact logic minimization and multiplicative complexity of concrete algebraic and cryptographic circuits", *International Journal On Advances in Intelligent Systems*, vol. 6, no. 3 and 4, pp. 165–176, 2013.

[19] R. Oliynykov, I. Gorbenko, O. Kazymyrov et al. "A New Encryption Standard of Ukraine: The Kalyna Block Cipher", in *Proc. Norwegian Information Security Conference*, Ålesund, Norway, 2015, 113 p.

[20] Intel Intrinsics Guide. [Електронний ресурс]. Доступно: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[21] "Using the RDTSC Instruction for Performance Monitoring. Application Note", Intel, 1998.

[22] Bitsliced SBox (Kalyna). [Електронний ресурс]. Доступно: https://drive.google.com/open?id=1e7iC_uADd8dDwdgaBgmN3OGiW4vyuLrr.

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 3 (7), 2020
ISSN 2663 - 4023

Yaroslav R. Sovyn
PhD of Technical Sciences, ass. professor, ass. Professor at the Department of Information Security
Lviv Polytechnic National University, Lviv, Ukraine
ORCID ID: 0000-0002-5023-8442
sovynjarosl@gmail.com

Volodymyr V. Khoma
Dr. Sci. Tech., Professor, Professor at the Department of Information Security
Lviv Polytechnic National University, Lviv, Ukraine
ORCID ID: 0000-0001-9391-6525
volodymyr.v.khoma@lpnu.ua

SOFTWARE BITSLICED IMPLEMENTATION OF KALYNA CIPHER IS ORIENTED TO USE SIMD INSTRUCTIONS FOR MICROPROCESSORS WITH X86-64 ARCHITECTURE

Abstract. The article is devoted to software bitsliced implementation of the Kalyna cipher using vector instructions SSE, AVX, AVX-512 for x86-64 processors. The advantages and disadvantages of different approaches to efficient and secure block cipher software implementation are shown. It is noted that bitslicing technology combines high speed and resistance to time and cache attacks, but its application to the Kalyna cipher is not available at the moment. The basic approaches to data representation and bitsliced encryption operations are considered, special attention is paid to the effective implementation of SubBytes operation, which largely determines the final performance. Existing methods for minimizing logical functions have been shown to either fail to produce the result in bitsliced format in the case of 8-bit non-algebraic SBoxs, or far from optimal. A heuristic algorithm for minimizing logic functions describing Kalyna SBoxes using the operations of AND, OR, XOR, NOT available in the instruction set of low- and high-end processors is proposed. The results show that a bitsliced description of one SBox requires about 520 gates, which is significantly less than other methods. Possible ways to increase performance by regrouping data into bitsliced variables before and after the SubBytes operation are indicated, which results in more efficient use of vector registers. The bitsliced implementations of Kalyna cipher were measured using C++ compilers from Microsoft and GCC for the Intel Xeon Skylake-SP processor. The results of the bitsliced Kalyna implementation can also be transferred to processors that do not support SIMD instructions, including low-end, to increase resistance to attacks through third-party channels. They also enable switching to ASIC or FPGA-based bitsliced implementation of Kalyna.

Keywords: cipher "Kalyna"; bitslicing; vector instructions; SSE; AVX; AVX-512; SBox; x86-64 architecture; performance

REFERENCES

[1] Standards Ukraine 2014, DSTU 7624:2014 Information Technologies. Cryptographic Data Security. Symmetric block transformation algorithm. [online] Available: http://ukrndnc.org.ua/downloads/new_view/?i=dstu-7624-2014&pz=%C4%D1%D2%D3+7624%3A2014.

[2] Kuznetsov O., Oliynykov R., Gorbenko Y., Pushkaryov A., Dyrda O., and Gorbenko I., "Requirements justification, construction and analysis of perspective symmetric cryptographical transformations on the base of block cipher codes", *Academic Journal of Lviv Polytechnic. Series of Computer Systems and Networks*, № 806, pp. 124-140, 2014.

[3] C. Rebeiro, D. Mukhopadhyay, S. Bhattacharya, *Timing Channels in Cryptography. A Micro-Architectural Perspective*. Cham, Switzerland: Springer, 2015. DOI: https://doi.org/10.1007/978-3-319-12370-7

[4] Q. Ge, Y. Yarom, D. Cock, G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware", *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2016. DOI: https://doi.org/10.1007/s13389-016-0141-6

[5] D. Kusswurm. *Modern x86 Assembly Language Programming 32-bit 64-bit SSE and AVX*. N.-Y., USA: Apress, 2014. DOI: https://doi.org/10.1007/978-1-4842-0064-3
[6] E. Biham, "A fast new DES implementation in software". In: Biham E. (eds) *Fast Software Encryption*. FSE LNCS, vol. 1267, pp. 260-272, 1997. DOI: https://doi.org/10.1007/BFb0052352
[7] E. Käsper, P. Schwabe, "Faster and Timing-Attack Resistant AES-GCM", in *Proc. 11th International Workshop Cryptographic Hardware and Embedded Systems*, Lausanne, Switzerland, 2009, pp. 1-17. DOI: https://doi.org/10.1007/978-3-642-04138-9_1
[8] R. Oliynikov, I. Gorbenko, O. Kazimirov, V. Ruzhentsev, & Y. Gorbenko, "Design principles and main properties of the new Ukrainian national standard of block encryption", *Ukrainian Information Security Research Journal*, vol. 17, no. 2, pp. 142-157, 2015. DOI: https://doi.org/10.18372/2410-7840.17.8789
[9] R. Oliynykov, O. Kazymyrov, O. Kachko, R. Mordvinov, et al. Source code for performance estimation of 64-bit optimized implementation of the block ciphers Kalyna, AES, GOST, BelT, Kuznyechik. [online] Available: https://github.com/Roman-Oliynykov /ciphers-speed/.
[10] V. Kovtun, & A. Okhrimenko. Features of construction of a cross-platform library of cryptographic primitives "Cipher+" v2. [online] Available: https://cipher.com.ua/media/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D1%8B/%D0%A8%D0%B8%D1%84%D1%80%2Bv2.1/Presentation_Cipher_Plus.pdf.
[11] cppcrypto library. Encryption performance. [online] Available: http://cppcrypto.sourceforge.net/.
[12] Y. Sovyn, V. Khoma, Y. Nakonechny, & Y. Stakhiv, "Effective implementation and performance comparison of «Kalyna» and GOST 28147-89 ciphers witch the use of vector extensions SSE, AVX and AVX-512", *Ukrainian Information Security Research Journal*, vol. 21, no. 4, pp. 207-223, 2019. DOI: https://doi.org/10.18372/2410-7840.21.14266
[13] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*. Hingham, USA: Kluwer Academic Publishers, 1984. DOI: https://doi.org/10.1007/978-1-4613-2821-6
[14] D. Canright, "A very compact S-Box for AES", in *Proc. 7th International Workshop Cryptographic Hardware and Embedded Systems*, Edinburgh, UK, 2005, pp. 441-455. DOI: https://doi.org/10.1007/11545262_32
[15] J. Boyar, and R. Peralta, "A small depth-16 circuit for the AES S-Box", *IFIP Advances in Information and Communication Technology*, vol. 376, pp. 287–298, 2012. DOI: https://doi.org/10.1007/978-3-642-30436-1_24
[16] A. Reyhani-Masoleh, M. Taha, and D. Ashmawy, "Smashing the implementation records of AES S-Box", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, no. 2, pp. 298–336, 2018. DOI: https://doi.org/10.13154/tches.v2018.i2.298-336
[17] K. Stoffelen, "Optimizing S-Box Implementations for Several Criteria Using SAT Solvers", in *Proc. 23rd International Conference on Fast Software Encryption*, Bochum, Germany, 2016, pp. 140-160. DOI: https://doi.org/10.1007/978-3-662-52993-5_8
[18] N. Courtois, T. Mourouzis, and D. Hulme, "Exact logic minimization and multiplicative complexity of concrete algebraic and cryptographic circuits", *International Journal On Advances in Intelligent Systems*, vol. 6, no. 3 and 4, pp. 165–176, 2013.
[19] R. Oliynykov, I. Gorbenko, O. Kazymyrov et al. "A New Encryption Standard of Ukraine: The Kalyna Block Cipher", in *Proc. Norwegian Information Security Conference*, Ålesund, Norway, 2015, 113 p.
[20] Intel Intrinsics Guide. [online] Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[21] "Using the RDTSC Instruction for Performance Monitoring. Application Note", Intel, 1998.
[22] Bitsliced SBox (Kalyna). [online] Available: https://drive.google.com/open?id=1e7iC_uADd8dDwdgaBgmN3OGiW4vyuLrr.

(cc) BY-NC-SA
This work is licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.