COMPUTER SYSTEMS AND NETWORKS

Vol. 5, No. 1, 2023

DOI: https://doi.org/10.23939/csn2023.01.103

УДК 004.056:061.68

МІНІМІЗАЦІЯ BITSLICED-ПРЕДСТАВЛЕННЯ 4×4 S-BOXES НА БАЗІ ТЕРНАРНОЇ ЛОГІЧНОЇ ІНСТРУКЦІЇ

Я. Р. Совин, В. В. Хома, І. Р. Опірський

Національний університет “Львівська політехніка”,
кафедра захисту інформації
E-mail: yaroslav.r.sovyn@lpnu.ua, volodymyr.v.khoma@lpnu.ua, ivan.r.opirskyi@lpnu.ua

© Совин Я. Р., Хома В. В., Опірський І. Р., 2024

Розглянуто методи і засоби для генерації програмно-орієнтованих bitsliced-описів бієктивних 4×4 S-Boxes зі зменшеною кількістю інструкцій на базі тернарної логічної інструкції. Згенеровані запропонованим методом bitsliced-описи дають змогу підвищити продуктивність та безпеку програмних імплементацій криптоалгоритмів, що використовують 4×4 S-Boxes на різноманітних процесорних архітектурах та під час проєктування апаратних засобів шифрування.

Розроблено евристичний метод мінімізації, що використовує тернарну логічну інструкцію, яка доступна в x86-64 процесорах із підтримкою розширення системи команд AVX-512 та деяких GPU-процесорах. Завдяки поєднанню у методі різних евристичних технік (попередніх обчислень, вичерпному пошуку на певну глибину, уточнювальному пошуку) вдалося зменшити кількість вентилів у bitsliced-описах S-Boxes, порівняно з іншими відомими методами. Розроблено відповідне програмне забезпечення у вигляді утиліти мовою Python і протестовано її роботу на 225 S-Boxes різноманітних криптоалгоритмів. Встановлено, що розроблений метод у 91,1% випадках генерує bitsliced-опис із меншою кількістю тернарних інструкцій, порівняно із найкращим відомим сьогодні методом, реалізованим в утиліті sboxgates.

Ключові слова: bitslicing; тернарна логічна інструкція; AVX-512; 4×4 S-Box; CPU; логічна мінімізація; програмна імплементація; sboxgates; швидкодія.

Вступ

З огляду на постійне зростання обсягів та швидкості оброблення даних дуже важливою вимогою до криптографічних алгоритмів (КА) є забезпечення достатньо високої продуктивності для широкого класу мікропроцесорних архітектур. Не менш важливим аспектом для програмної реалізації криптографічних алгоритмів 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, а для high-end CPU (x86, ARM Cortex-A) – передусім часові та кеш-атаки.

Щоби забезпечити високу продуктивність крипто алгоритмів, використовують різні підходи до їх програмної імплементації: це і створення передобчислених таблиць (Lookup Tables, LUT) для певних операцій, інтеграція у процесор апаратних криптоакселераторів (напр. AES-NI у x86-процесорах), застосування SIMD-технології для розпаралелення процесу шифрування (напр. векторні інструкції SSE/AVX2/AVX-512 у x86-64 CPU), використання обчислювальних потужностей GPU тощо. Проте всі ці підходи мають низку обмежень і не завжди можуть бути реалізовані в конкретному процесорі.

Bitslicing [1] є одним із перспективних підходів, що забезпечує високопродуктивну constant-time імплементацію КА з імунітетом до часових та кеш-атак [2], максимально використовує можливості сучасних high-end мікропроцесорів щодо збільшення швидкодії внаслідок розпаралелювання як виконання коду, так і оброблення даних, а також допускає адаптацію для low-end CPU і апаратну реалізацію на FPGA і ASIC. Для багатьох КА саме bitsliced підхід забезпечує найбільшу швидкодію в програмній реалізації (якщо не використовуються апаратні криптоакселератори) для різного типу процесорних архітектур [1–7]. Відсутність звернень до передобчислених таблиць у пам’яті та кеші, а також залежних від даних умовних переходів робить bitsliced-реалізації невразливими до часових і кеш-атак та одночасно ускладнює атаки через сторонні канали.

Базова ідея Bitslicing – конвертувати криптографічний алгоритм у послідовність бітових логічних операцій типу AND, XOR, OR, NOT. У процесорах кожну таку логічну операцію можна подати відповідною інструкцією, в апаратних засобах – відповідним вентилем (надалі в роботі поняття “вентиль” та “інструкція” вживатимемо як синоніми). Висока швидкодія програмного Bitslicing досягається завдяки тому, що CPU обробляє багато елементів шифру (байтів, блоків) паралельно, використовуючи швидкі логічні інструкції, та простішому виконанню деяких операцій (наприклад, бітових перестановок, зсувів тощо). Для програмно-орієнтованих bitsliced-реалізацій, крім класичних, можна використати складніші логічні інструкції, які підтримує певний процесор, і зменшити цим їх загальну кількість. Наприклад, багато процесорів підтримують інструкцію AND-NOT (x86-64, ARM), деякі NOR і NAND (ARM) тощо.

Очевидно, щоб забезпечити максимальну швидкодію, потрібно мінімізувати кількість логічних операцій, що входять до bitsliced-опису криптоалгоритму. Більшість криптографічних операцій породжують однозначний опис у разі переходу до bitsliced-опису або не дають багато простору до мінімізації за винятком нелінійних перетворень. У КА операції нелінійної заміни задають у вигляді $n \times m$ LUT-таблиць, так званих S-Boxes, що переважно мають розмір 4×4 ($n = 4$) або 8×8 ($n = 8$) бітів. Таблиці 4×4 біт характерні як для легковагових криптоалгоритмів, спеціально розроблених для ефективної імплементації на обмежених у ресурсах процесорах (напр., блокові шифри PRINCE, LED, Piccolo, хеш-функції PHOTON, Spongent), так і криптоалгоритмів загального призначення (напр. блокові симетричні шифри Serpent, Twofish, Magma, хеш-функції BLAKE, Whirlpool).

Отже, основним ресурсом збільшення швидкодії за bitsliced імплементації КА є представлення S-Boxes мінімально можливою кількістю логічних вентилів/інструкцій. Ця задача є NP-повною й допускає точний розв’язок лише для дуже простих випадків ($n \leq 3$ і деяких $n = 4$), тому більшість сучасних методів та утиліт для генерації bitsliced-опису S-Boxes використовують евристичні підходи, які не гарантують, що отриманий розв’язок є оптимальним, проте забезпечують значно кращий результат порівняно із універсальними методами мінімізації логічних функцій (напр., метод карт Карно чи метод простих імплікант Куайна – Мак-Класкі тощо).

Крім традиційних двохоперандних логічних інструкцій деякі процесори підтримують тернарну логічну інструкцію *ternarylogic(a, b, c, imm8)*, яка дає змогу обчислити довільну булеву функцію від трьох операндів *a, b, c*, задану таблицею істинності у восьмибітній змінній *imm8 (рис. 1)*.

Тернарна інструкція (TI) наявна у таких процесорах:

– x86-64 з підтримкою 512-бітних SIMD-інструкцій із розширення AVX-512F. Оскільки інструкція *vpternlogq zmm_a, zmm_b, zmm_c, imm8* входить у базове розширення AVX-512F (Foundation), то її підтримують усі процесори з технологією AVX-512;

– деяких GPU процесорах. Наприклад, у Nvidia GPU ця інструкція має вигляд *lop3.b32 d, a, b, c immLut*, що обчислює над 32-бітними операндами *a, b, c* логічну функцію, задану таблицею істинності в *immLut*, і зберігає результат у *d*.

Одна тернарна інструкція може замінити декілька двооперандних логічних інструкцій, тому її використання дає змогу згенерувати bitsliced-опис із істотно меншою кількістю операцій, а отже, збільшити швидкодію програмної реалізації. Проте і у цьому випадку навіть для чотирибітних S-Boxes знайти гарантовано оптимальне подання здебільшого неможливо і потрібно використовувати евристичні методи мінімізації. Разом з тим відомі евристичні методи мінімізації здебільшого орієн-

товані на використання двохоперандних інструкцій, тому їх не можна безпосередньо використати для генерації bitsliced-опису на основі TI.

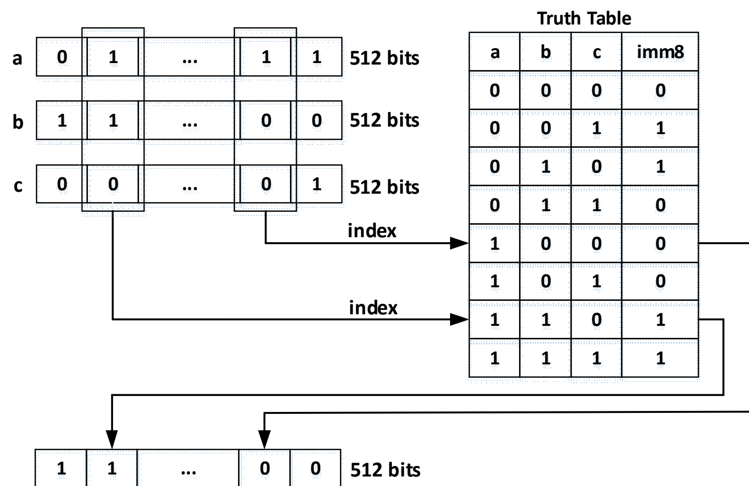

Рис. 1. Принцип роботи тернарної інструкції ternarylogic(a, b, c, imm8)

Отже, проблема знаходження оптимального bitsliced представлення на базі тернарної інструкції навіть для 4×4 S-Boxes далека від вирішення, що потребує пошуку нових евристичних підходів, один із яких висвітлений у нашій роботі.

Огляд літературних джерел

Проаналізуємо методи та засоби для знаходження bitsliced-опису 4×4 S-Boxes за критерієм Bitslice Gate Complexity (BGC), що позначає оптимальне рішення з мінімальною кількістю операцій.

Bitsliced підхід до представлення криптоалгоритму уперше запропонував E. Biham у роботі [1] для пришвидшення програмної імплементації шифру DES. У цій самій роботі E. Biham описав алгоритм bitsliced-представлення DES S-Boxes (6×4) логічними вентилями XOR, AND, OR, NOT. В алгоритмі із шести вхідних змінних вибирають дві вхідні змінні, що формують усі можливі комбінації за допомогою заданих логічних вентилів, з яких потім будують чотири вихідні змінні. У середньому за bitsliced-опису за цим методом один DES S-Box потребує 100 вентилів.

У роботі [8] M. Kwan запропонував значно ефективніший підхід до знаходження bitsliced-представлення на прикладі DES S-Boxes. У ньому кожен вихідний біт S-Box розглядається як функція шести вхідних бітів, що представлена картою Карно й розташовується у 64-бітній змінній. Усі вхідні та проміжні змінні теж можна розглядати як 6-бітні карти Карно, що описуються 64-бітними числами. Тоді задача формулюється так: потрібно скомбінувати наявні вхідні та проміжні карти таким способом, щоб отримати шукану вихідну змінну. Одна вхідна змінна слугує селектором, що об'єднує функції п'яти змінних. Для пошуку подання функцій п'яти змінних із мінімальною кількістю вентилів застосовують вичерпний перебір (brute force) та вентилі, знайдені на попередніх кроках. Залежно від того, в якій послідовності буде здійснюватися пошук, доступні 6! варіантів для вхідних змінних та 4! варіанти для вихідних змінних. Це загалом дає 17280 варіантів пошуку, серед яких вибирають варіант із мінімальною кількістю вентилів. У результаті середня кількість вентилів для bitsliced-опису одного DES S-Box зменшилася зі 100 до 56.

Для мінімізації S-Boxes можна застосувати програми SAT-Solvers, призначені для ефективного вирішення завдання здійсненності булевих формул (SATisfiability problem, SAT). Об'єктом задачі SAT є булева формула, що складається тільки з констант (0/1), змінних і операцій AND, OR,

NOT. Задача полягає в тому, чи можна призначити всім змінним значення False і True так, щоби формула стала істинною. Спеціалізовані програми SAT-Solvers, побудовані на ефективних алгоритмах розв'язання, приймають на вході набір рівнянь і видають результат у вигляді *SAT*, якщо розв'язок знайдено і *UNSAT*, якщо не знайдено. Щоби знайти логічну схему з заданою кількістю вентилів, можна сформувати рівняння, де б змінні задавали всі можливі зв'язки між вентилями та операції, і спробувати розв'язати їх із допомогою SAT-Solvers. Перевагою цього підходу є те, що якщо виявлено рішення з n вентилів (*SAT*) і для $n - 1$ вентиля отримано *UNSAT*, то ми гарантовано знайшли мінімально можливий bitsliced-опис.

SAT-Solvers використано у роботах [9, 10] для знаходження bitsliced-представлення деяких чотирибітних S-Boxes. Загалом, проблемою SAT-Solvers є те, що вони не завжди знаходять рішення для "важких" S-Boxes, що потребують понад 12–13 вентилів. Для порівняно простих S-Boxes з 11–13 вентилями SAT-Solvers не завжди можуть довести, що знайдене подання є мінімальне.

У роботі [11] описано open source утиліту LIGHTER, яка є найефективнішою сьогодні утилітою для пошуку bitsliced-опису 4×4-бітних S-Boxes. LIGHTER має можливість гнучко задавати набір дво- і трвходових вентилів та їх вагові коефіцієнти, що враховують під час мінімізації. Це дає змогу реалістичніше здійснювати оптимізацію за апаратної реалізації, коли різні логічні вентилі відрізняються площею кристала, енергоспоживанням, затримкою тощо, через урахування цих параметрів у вагових коефіцієнтах. Для програмної імплементації, коли логічні інструкції рівноцінні, достатньо встановити однакові вагові коефіцієнти для всіх вентилів.

Сам алгоритми пошуку LIGHTER комбінує два підходи: пошук за допомогою breath-first-search (BFS) алгоритму та meet-in-the-middle (MITM) стратегію. Тобто будують два графи: один починається із базових векторів і здійснює пошук уперед, а інший стартує із шуканих векторів і виконує пошук назад. Обидва графи рухаються один назустріч одному, використовуючи задані логічні операції, поки не зустрінуться. Далі вибирають шлях, який поєднує ці два графи з мінімальною вартістю, яка враховує вагові коефіцієнти для кожного вентиля. Утиліта демонструє високу часову ефективність порівняно із SAT-методами, а її результати, які хоча не можуть вважатися оптимальними, доволі близькі до результатів, отриманих SAT-утилітами.

У роботі [12] описано open source утиліту Peigen (Platform for Evaluation, Implementation, and Generation of S-boxes), що дає змогу знаходити bitsliced-опис S-Boxes у різних логічних базисах, застосовуючи задані критерії мінімізації для апаратних і програмних імплементацій. Алгоритми пошуку bitsliced-опису утиліти Peigen спираються на алгоритми з утиліти LIGHTER, але підвищено їх часову ефективність, зокрема, використано передобчислення та низку додаткових технік. Проте навіть із внесеними удосконаленнями утиліта ефективно працює лише з чотирибітними S-Boxes.

Генерація оптимізованої bitsliced імплементації КА потребує значних затрат часу на написання і налагодження коду та глибокого знання архітектури процесора, низькорівневих інструментів та технік оптимізації на апаратному і програмному рівнях. Тому в роботі [13] подано високорівневу мову Usuba, яка дає змогу описати симетричний криптографічний примітив, а вже сам Usuba компілятор згенерує високооптимізований, розпаралелений і векторизований bitsliced-код. Проте для генерації bitsliced опису S-Box використовують або простий алгоритм мінімізації, який дає далеко не оптимальний результат, або беруть готовий оптимізований опис з бази даних, що входить до складу Usuba, якщо S-Box у ній є. Отже, генерація опису для S-Box є слабким місцем bitsliced-компілятора Usuba.

Розглянуті методи й утиліти формують bitsliced-опис, використовуючи переважно двовходові логічні елементи і не підтримують тернарну логічну інструкцію. Єдина відома нам сьогодні утиліта для генерації bitsliced-описів S-Boxes на основі тернарної інструкції це *sboxgates* [14]. Ця open source утиліта реалізує алгоритм M. Kwan з деякими удосконаленнями і здатна генерувати bitsliced-опис для довільних S-Boxes до 8×8 включно. Утиліта дає змогу задавати довільний набір двовходових вентилів, використовувати тернарну логічну інструкцію, вказувати кількість ітерацій алгоритму пошуку, розпаралелювати пошук між процесорними ядрами тощо. У випадку 4×4 S-Boxes *sboxgates*

видає результати, які, як показано в статті, можна істотно покращити, що є платою за універсальність.

Постановка завдання

Мета статті – подати метод та утиліту для генерації bitsliced-опису бієктивних 4×4 S-Boxes на основі тернарної логічної інструкції, що забезпечують кращі результати порівняно з наявними, а це дасть змогу збільшити швидкодію та безпеку програмно-апаратних реалізацій широкого кола криптографічних алгоритмів, які використовують S-Boxes заданого типу.

Формат представлення S-Boxes для bitsliced імплементації

У специфікаціях криптографічних алгоритмів S-Boxes переважно задають у вигляді LUT-таблиць, наприклад, 4×4 S-Box шифру PRESENT наведено у табл. 1.

Таблиця 1

LUT-таблиця S-Box шифру PRESENT

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S(x)	12	5	6	11	9	0	10	13	3	14	15	8	4	7	1	2

У bitsliced поданні LUT-таблиці розглядають як логічні функції, задані таблицями істинності. Наприклад, вигляд S-Box шифру PRESENT подано у табл. 2.

Таблиця 2

Bitsliced-орієнтоване представлення S-Box шифру PRESENT

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Hex
x_0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	*0xff00*
x_1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	*0xf0f0*
x_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	*0xcccc*
x_3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	*0xaaaa*
S(x)	**12**	**5**	**6**	**11**	**9**	**0**	**10**	**13**	**3**	**14**	**15**	**8**	**4**	**7**	**1**	**2**	
y_0	1	0	0	1	1	0	1	1	0	1	1	1	0	0	0	0	*0x0ed9*
y_1	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0	0	*0x3687*
y_2	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0	1	*0xa74c*
y_3	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	*0x659a*

Отож, компактне представлення цього S-Box у вигляді таблиці істинності матиме вигляд: $S(x) = y$, де $x = \{x_0, x_1, x_2, x_3\} = \{0xff00, 0xf0f0, 0xcccc, 0xaaaa\}$ – вхідні bitsliced змінні, $y = \{y_0, y_1, y_2, y_3\} = \{0x0ed9, 0x3687, 0xa74c, 0x659a\}$ – вихідні bitsliced змінні, що визначають конкретну таблицю заміни, а 16-бітні числа, що задають *x* та *y*, будемо називати векторами.

Завдання пошуку bitsliced представлення S-Box за критерієм BGC можна сформулювати так: задано чотири базові вектори $base = \{x_0, x_1, x_2, x_3\}$, потрібно знайти вектори $y = \{y_0, y_1, y_2, y_3\}$, використовуючи мінімальну кількість тернарних логічних інструкцій *ternarylogic(a, b, c, imm8)*.

Попередні обчислення

На етапі передобчислень одноразово знаходять та зберігають певні дані, які потім багатократно використовують у нашому алгоритмі пошуку bitsliced-опису. Ці дані є двох типів:

1. Для кожного 16-бітного вектора v знаходять BGC(v) – мінімальна кількість тернарних інструкцій, потрібна для його представлення, так звана “складність” вектора.

Оскільки вектори подаються 16-бітними числами, загалом є 65536 векторів, з них чотири це базові вектори $base = \{x_0–x_3\}$ і два це логічні константи $const = \{0x0000, 0xffff\}$ для позначення 0 і 1 для яких BGC дорівнює 0, тому залишається 65530 векторів, складність яких потрібно оцінити.

У табл. 3 подано знайдений розподіл векторів за їх значенням BGC.

Таблиця 3

Розподіл 16-бітних векторів за BGC

BGC	0	1	2	3
Кількість векторів	6	936	34250	30344

Як видно з табл. 3, максимальна складність дорівнює 3, тобто будь-який 16-бітний вектор можна подати за допомогою не більш ніж трьох тернарних інструкцій. Це дає верхню оцінку bitsliced-складності довільного S-Box, що описується чотирма векторами y_0-y_3, яка становить 12-м TI.

Для непрямої попередньої оцінки "складності" S-Box можна використати такий показник, як сумарне значення складності векторів y_0-y_3, що його описують: чим ближче до 12 сумарне значення, тим більше TI слід очікувати у bitsliced-описі й навпаки.

2. Побудова LUT-таблиці для подання всіх графів на глибину ge = 2 інструкцій.

Побудова таблиці здійснюється покроково. На першому кроці будують таблицю $\boldsymbol{q_0}$, що містить усі можливі значення, які можна отримати з базових векторів x_0–x_3 за допомогою однієї тернарної інструкції та які не входять у *base* і *const*. Для цього використовують алгоритм *VECT_SQUARE*, який утворює з вхідних векторів усі можливі комбінації за допомогою TI.

Таких векторів загалом 936. Згенеровані значення вносять у таблицю, а значення x_0–x_3 у таблиці не зберігають для економії пам'яті, хоча вони наявні для кожного рядка неявно. Отже, таблиця $\boldsymbol{q_0}$ = *VECT_SQUARE*($\{x_0, x_1, x_2, x_3\}$) має розмірність 936×1 (рис. 2).

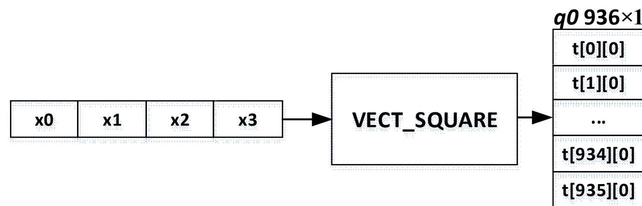

Рис. 2. Формування таблиці $\boldsymbol{q_0}$

Кожен *i*-й рядок таблиці $\boldsymbol{q_0}$ можна розглядати як новий базис $\{x_0$–$x_3, q_0[i]\}$, який використовується для генерації всіх можливих векторів на наступному кроці, а самі рядки LUT таблиць будемо називати графами.

Зокрема, на другому кроці генерується таблиця $\boldsymbol{q_1}$ = *GEN_TABLE*($\boldsymbol{q_0}$), для чого з кожного рядка таблиці $\boldsymbol{q_0}$ знову алгоритмом *VECT_SQUARE* породжуються всі можливі значення, що можна отримати із базових векторів x_0–x_3 та $q_0[i]$ за допомогою однієї тернарної інструкції, формуються рядки з двох векторів та додаються у загальну таблицю. Після цього здійснюється фільтрація логічно еквівалентних графів: якщо рядки таблиці $\boldsymbol{q_1}$ містять однакові значення в довільній послідовності, то з них залишається лише один рядок (рис. 3).

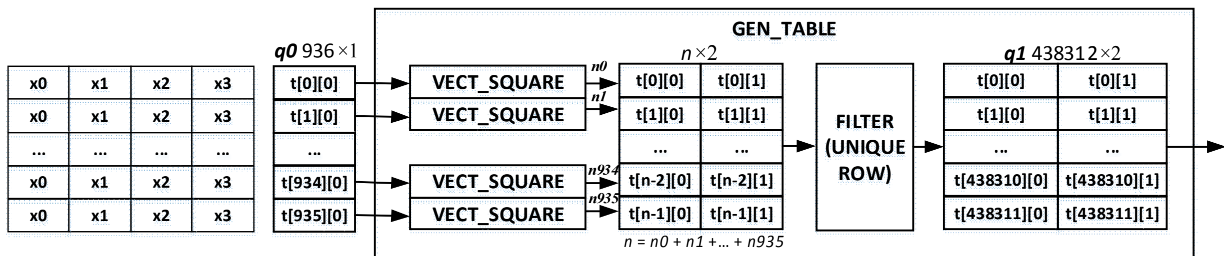

Рис. 3. Формування таблиці $\boldsymbol{q_1}$

Розмірність таблиці $\boldsymbol{q_1}$ становить 438312×2, а коефіцієнт розгалуження *branching* у разі переходу від таблиці $\boldsymbol{q_0}$ до $\boldsymbol{q_1}$ дорівнює 438312/936 = 468,3. Покрокові результати подано в табл. 4.

Таблиця 4

Властивості LUT-таблиць q_0–q_1

Таблиця	q_0	q_1
Розмірність	936×1	438312×2

Подальша побудова таблиць недоцільна, оскільки з огляду на велике значення коефіцієнта розгалуження вони потребують надто багато пам'яті. Таблицю $\boldsymbol{q_1}$ використаємо для формування усіх можливих унікальних графів для представлення векторів *y*.

Алгоритм пошуку bitsliced-представлення

На верхньому рівні алгоритму пошуку здійснюється перебір усіх значень y_0–y_3, генерація для кожного з них із передобчисленої LUT-таблиці ***q1*** матриці графів-кандидатів $\boldsymbol{gr_i}$ = *STEP_0*(y_i) і передавання їх у алгоритм пошуку в глибину *FIND_BS*($\boldsymbol{gr_i}$). Алгоритм *FIND_BS* знаходить решту значень *y*, намагаючись використати мінімум TI, і повертає побудовані доповнені матриці графів $\boldsymbol{gr_0}$–$\boldsymbol{gr_3}$. З отриманих результатів вибрано граф з мінімальним значенням BGC (рис. 4).

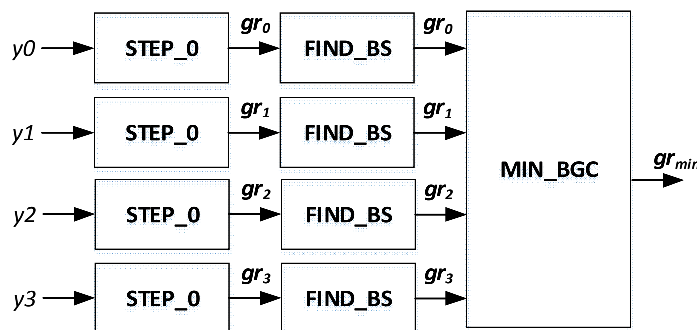

Рис. 4. Узагальнена структура алгоритму пошуку BITSLICED-опису S-Box

Отже, алгоритм пошуку здійснює чотири ітерації, починаючи роботу з різних значень *y*. Позначимо це початкове значення y_{start}. На етапі $\boldsymbol{gr_i}$ = *STEP_0*(y_{start}), використовуючи LUT-таблицю ***q1***, генерується матриця графів $\boldsymbol{gr_i}$, що містить усі можливі графи з вектором y_{start} на певній глибині d_{start} вентилів. Залежно від того, до якої BGC-групи належить вектор y_{start}, евристично підібрано значення d_{start} (табл. 5), щоби забезпечити прийнятні час обчислення та обсяг потрібної пам'яті.

Таблиця 5

**Глибина генерації графів,
що містять y_{start} на кроці *STEP_0***

BGC-група y_{start}	1	2	3
d_{start}	2	3	4

Якщо, наприклад, *BGC*(y_0) = 2, то матриця графів $\boldsymbol{gr_0}$ після *STEP_0* міститиме всі можливі графи із довжиною 3 інструкції (d_{start} = 3), у яких трапляється вектор y_0.

Далі графи кандидати в $\boldsymbol{gr_i}$ сортують на три групи: ***gr_1y***, ***gr_2y***, ***gr_3y*** з однаковою кількістю векторів *y* у кожному графі групи – 1, 2 і 3 відповідно. Позначимо цю кількість *y_find*. Далі пошук ведуться для кожної непорожньої групи окремо відповідно до рис. 5.

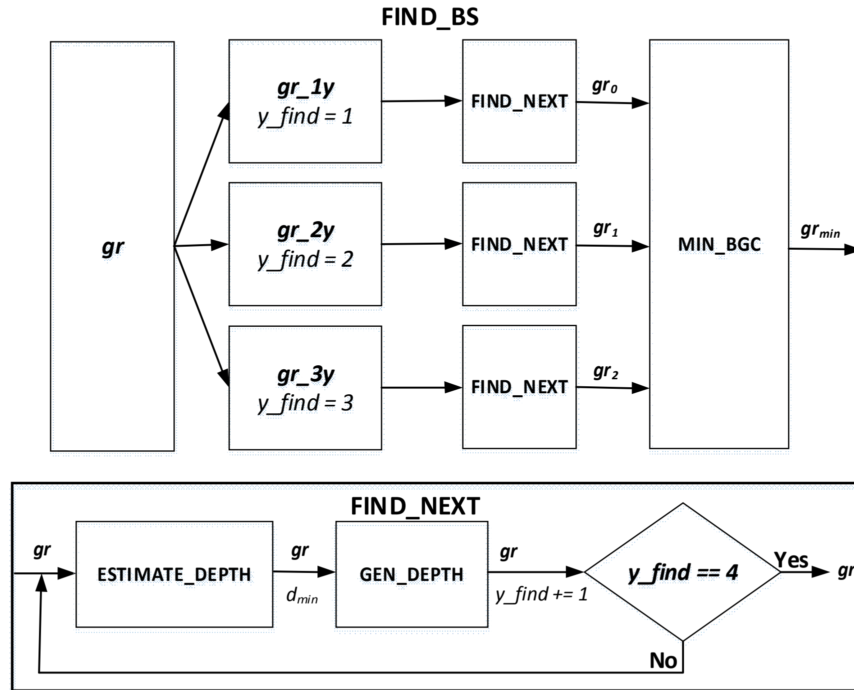

Рис. 5. Узагальнена схема пошуку BITSLICED представлення алгоритмом FIND_BS

Алгоритм *FIND_NEXT* здійснює почерговий пошук y_i, поки не будуть знайдені всі чотири значення y_0–y_3. На вхід подається матриця графів ***gr*** у вигляді таблиці $n \times m$, кожен рядок якої містить *y_find* значень із множини $\{y_0$–$y_3\}$. Кожен рядок таблиці зберігає у явному вигляді m векторів та вектори x_0–x_3 неявно. Спочатку для групи ***gr*** здійснюється оцінка мінімальної віддалі $d_{\min}$, на якій розташоване найближче значення y_x серед усіх графів – *ESTIMATE_DEPTH*. Для цього розроблено швидку функцію *FAST_FIND* вичерпного пошуку вперед на задану глибину 1/2/3 кроки. Пошук і відсікання варіантів здійснюють за допомогою алгоритму пошуку в глибину з ітеративним заглибленням – IDDFS (Iterative Deepening Depth-First Search).

Після виявлення оцінки $d_{\min}$ за допомогою алгоритму *GEN_DEPTH* здійснюється перехід від набору графів з *y_find* = *n_y* до набору графів з *y_find* = *n_y* + 1. Для цього з групи ***gr*** вибирають графи зі знайденим значенням $d_{\min}$ і для цієї групи $\boldsymbol{gr}_{\mathbf{min}}$ роблять крок уперед ***gr*** = = *GEN_TABLE(*$\boldsymbol{gr}_{\mathbf{min}}$*)*. Для згенерованого набору ***gr*** знову вибирають графи зі знайденим значенням $d = d_{\min} - 1$, для них роблять крок уперед і так далі, поки d не дорівнюватиме 0. Після цього в групу відбирають лише ті графи, що містять *n_y* + 1 значень y. Далі ці кроки повторяють, поки не будуть знайдені всі значення y.

Обчислювально найтрудомісткіші процедури *ESTIMATE_DEPTH* та *GEN_DEPTH* реалізують із використанням GPU та технології OpenCL, що дає змогу істотно розпаралелити обчислення і зменшити час роботи алгоритму.

Алгоритм *FIND_BS* на кожному кроці здійснює оцінку мінімальної віддалі $d_{\min}$, на якій розташоване найближче значення y_x, та генерує відповідні графи. Як показано на рис. 6, цей маршрут починається із графів, що містять y_a, згенерованих за допомогою *STEP_0*, від яких найближче значення y_b розміщене на відстані d_{ab} вентилів, далі переходимо до y_c розташованого на мінімальній відстані d_{bc} від y_b та на відстані d_{cd} знаходимо останній вектор y_d.

Проте не завжди рух мінімальними кроками по траєкторії від вектора y_a до y_d дає оптимальний результат загалом (хоча здебільшого це так). Можлива ситуація, коли вибір мінімального значення d на перших кроках, призводить до більших значень d на наступних кроках і в результаті до неоптимального логічного представлення. Наприклад, вважатимемо, що на першому кроці ми

отримали d_{ab} = 1, на другому d_{bc} = 2 і на третьому – d_{cd} = 2, тобто маршрут становить сумарно 5 TI (рис. 6). Проте можливо, що якщо б на першому кроці ми пішли іншим маршрутом і відібрали графи з d_{ab} = 2, то на другому кроці вдалося б знайти значення y_c з d_{bc} = 1 і на третьому y_d з d_{cd} = 1, й одержали б коротший сумарний маршрут із 4 TI. Отже, другий маршрут привів до BITSLICED подання з меншим значенням BGC.

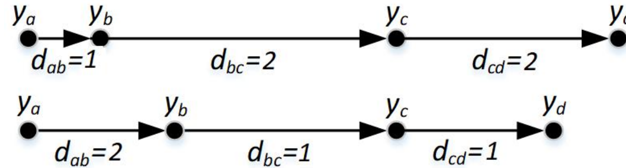

Рис. 6. Знаходження BITSLICED-опису для різних маршрутів

Щоби врахувати різні можливі маршрути, в алгоритмі пошуку здійснюють уточнювальні пошуки за схемою, наведеною на рис. 7. Якщо є набір графів, що містять 3 з 4 можливих значень *y*, то пошук четвертого значення завжди здійснюється на мінімально можливій глибині $d_{\min}$ (*SEARCH_3Y*). Для графів із двома значеннями *y* (*y_find* = 2) пошук третього значення відбувається за двома маршрутами: $d_{\min}$ і $d_{\min}$ + 1, після чого для виявлених графів з *y_find* = 3 застосовується пошук *SEARCH_3Y.* Для графів із одним значенням *y* (*y_find* = 1) пошук другого значення відбувається за трьома маршрутами: $d_{\min}$, $d_{\min}$ + 1 і $d_{\min}$ + 2, після чого для знайдених графів з *y_find* = 2 застосовується пошук *SEARCH_2Y.*

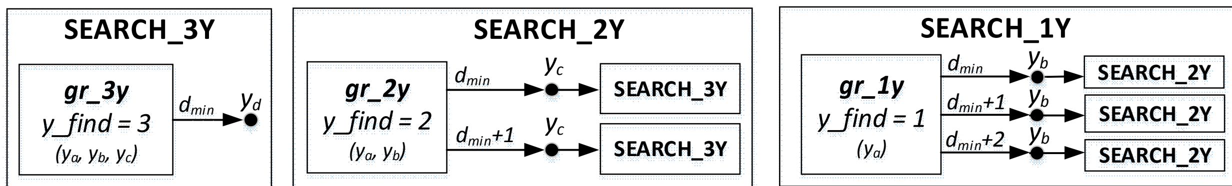

Рис. 7. Схема уточнювального пошуку в алгоритмі FIND_BS

Результати

Запропонований у роботі метод імплементований мовою Python, а для забезпечення швидкодії основні функції опрацювання даних реалізовано на базі бібліотек *numpy* та *pyopencl*.

Для оцінювання алгоритму ми взяли 225 4×4 S-Boxes різноманітних криптографічних алгоритмів. Використано open source проект *sboxgates*, щоб отримати за його допомогою оцінку BGC для вибраних S-Boxes і мати змогу порівняти з нашими результатами. BITSLICED-описи S-Boxes, отримані нашим методом, доступні за посиланням [15]. Також у [15] наведено деталізовану таблицю з результатами, яка через великий обсяг не увійшла у статтю, що містить такі стовпці:

LUT – представлення S-Box у табличному вигляді, де рядок ‘0123456789abcdef’ слід розуміти як *S(x)* = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

BS – представлення S-Box у bitsliced-форматі. Рядок ‘0ed9_3687_a74c_659a’ необхідно розуміти так: y_0 = *0x0ed9*, y_1 = *0x3687*, y_2 = *0xa74c*, y_3 = *0x659a*.

CY – BGC векторів y_0-y_3. Рядок ‘2133’ потрібно інтерпретувати так: BGC(y_0) = 2, BGC(y_1) = 1, BGC(y_2) = 3, BGC(y_3) = 3.

OURS – значення BGC, отримане за допомогою описаного у статті методу.

SG – значення BGC, одержане за допомогою утиліти *sboxgates*; для пошуку задано 1000 ітерацій [9]. Червоним кольором у стовпці ***SG*** позначено S-Boxes із більшим значенням BGC, порівняно з нашим алгоритмом, жовтим – із однаковим з нашим алгоритмом значенням BGC.

Загалом, як свідчать отримані результати, розроблений нами метод продемонстрував значно кращі результати порівняно з найближчим конкурентом, представленим утилітою *sboxgates*. Для 205 S-Boxes з 225 (91,1 %) наш метод забезпечує BITSLICED-опис із меншою кількістю TI. Сумарна кількість тернарних інструкцій для представлення всіх 225 S-Boxes у нашому методі 1578, що на 15,5 % менше порівняно з 1867 інструкціями для утиліти *sboxgates*. Утиліта *sboxgates* не згенерувала BITSLICED-опис з меншою, ніж отримано нашим методом, кількістю інструкцій для жодного S-Box і лише для 20 S-Boxes (8,9 %) змогла згенерувати bitsliced-опис з однаковим із нашим алгоритмом значенням BGC.

Висновки

У роботі описано метод для генерації bitsliced-опису довільних 4×4 бієктивних S-Boxes зі зменшеною кількістю тернарних логічних інструкцій. Отримані описи дають змогу загалом збільшити швидкодію програмних реалізацій відповідних криптоалгоритмів на будь-яких процесорах, що підтримують трьохоперандну тернарну логічну інструкцію *ternarylogic (CPU/GPU)*. На сьогодні запропонований у статті метод є найефективнішим із відомих нам методів за критерієм BGC, що підтверджують наведені в роботі результати досліджень. Метод поєднує в собі евристичні техніки на різних етапах пошуку BITSLICED-представлення, зокрема: передобчислення, вичерпний пошук на глибину до трьох вентилів із використанням GPU, IDDFS-алгоритм для пошуку і відсікання варіантів, уточнювальний пошук, що сукупно забезпечують його ефективність та прийнятну швидкодію.

1. E. Biham, "A fast new DES implementation in software", in International Workshop on Fast Software Encryption, 1997, 260–272. DOI: https://doi.org/10.1007/BFb0052352.

2. E. Kasper and P. Schwabe, "Faster and timing-attack resistant AES-GCM", in Proc. 11th International Workshop Cryptographic Hardware and Embedded Systems, 2009, 1–17. DOI: https://doi.org/10.1007/978-3- 642-04138-9_1.

3. A. Adomnicai and T. Peyrin, "Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V", IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(1), 402–425. DOI: https://doi.org/10.46586/tches.v2021.i1.402-425.

4. P. Schwabe and K. Stoffelen, "All the AES you need on Cortex-M3 and M4" in International Conference on Selected Areas in Cryptography, 2016, 180–194. DOI: https://doi.org/10.1007/978-3-319-69453-5_10.

5. J. Zhang, M. Ma, and P. Wang, "Fast implementation for SM4 cipher algorithm based on bit-slice technology", in International Conference on Smart Computing and Communication, 2018, 104–113. DOI: https://doi.org/10.1007/978-3-030-05755-8_11.

6. N. Nishikawa, H. Amano, and K. Iwai, "Implementation of bitsliced AES encryption on CUDA-enabled GPU", in International Conference on Network and System Security, 2017, 273–287. DOI: https://doi.org/10.1007/978-3-319-64701-2_20.

7. S. Matsuda and S. Moriai, "Lightweight cryptography for the cloud: exploit the power of bitslice implementation", in International Workshop on Cryptographic Hardware and Embedded Systems, 2012, 408–425. DOI: https://doi.org/10.1007/978-3-642-33027-8_24.

8. M. Kwan, "Reducing the Gate Count of Bitslice DES", IACR Cryptology ePrint Archive, 2000 (51). Available from: http://fgrieu.free.fr/Mattew%20Kwan%20-%20Reducing%20the%20Gate%20Count%20of%20Bitslice%20DES.pdf [Accessed: 03 October 2023].

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

10. 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, 165–176, 2013.

11. J. Jean, T. Peyrin, S. Sim, J. Tourteaux, "Optimizing Implementations of Lightweight Building Blocks", IACR Transactions on Symmetric Cryptology, 2017(4), 130–168. DOI: https://doi.org/10.13154/tosc.v2017.i4.130-168.

12. Z. Bao, J. Guo, S. Ling, and Y. Sasaki, "Peigen – a platform for evaluation, implementation, and generation of S-boxes", IACR Transactions on Symmetric Cryptology, 330–394, 2019. DOI: https://doi.org/10.13154/ tosc.v2019.i1.330-394.

13. D. Mercadier, "Usuba, Optimizing Bitslicing Compiler", PhD Thesis, Sorbonne University, France, p. 195, 2020.

14. M. Dansarie, "sboxgates: A program for finding low gate count implementations of S-boxes", Journal of Open Source Software, 6(62), 2021, 1–3. DOI: https://doi.org/10.21105/joss.02946.

15. Ya. Sovyn, "Bitsliced 4x4 S-Boxes Ternary Instruction 2023", 2023. [Online]. Available: https://drive.google.com/drive/folders/1o4GKjb1bIWzHf0H3KmvH--2CxiDNKQmb?usp=drive_link [Accessed: 12 October 2023].

MINIMIZATION OF BITSLICED-REPRESENTATION
OF 4×4 S-BOXES BASED ON TERNARY LOGIC INSTRUCTION

Ya. Sovyn, V. Khoma, I. Opirskyy

Lviv Polytechnic National University,
Information Security Department

© Sovyn Ya., Khoma V., Opirskyy I., 20243

The article is devoted to methods and tools for generating software-oriented bitsliced descriptions of bijective 4×4 S-Boxes with a reduced number of instructions based on a ternary logical instruction. Bitsliced descriptions generated by the proposed method make it possible to improve the performance and security of software implementations of crypto-algorithms using 4×4 S-Boxes on various processor architectures and when designing encryption hardware.

The paper develops a heuristic method of minimization using a ternary logical instruction, which is available in x86-64 processors with support AVX-512 instruction system extension and some GPU processors. Thanks to the combination of various heuristic techniques (preliminary calculations, exhaustive search to a certain depth, refining search) in the method, it was possible to reduce the number of gates in bitsliced descriptions of S-Boxes compared to other known methods. The corresponding software in the form of a utility in the Python language was developed and its operation was tested on 225 S-Boxes of various crypto-algorithms. It was found that the developed method generates a bitsliced description with fewer ternary instructions in 91.1% of cases, compared to the best known method implemented in the sboxgates utility.

Key words: bitslicing; ternary logic instruction; AVX-512; 4×4 S-Box; CPU; logic minimization; software implementation; sboxgates; speed.