УДК 004.056:061.68
DOI: 10.31673/2409-7292.2025.017435

**Совин Я.Р., Опірський І.Р., Комарницький В.О.**

# ЗНАХОДЖЕННЯ BITSLICED-ПРЕДСТАВЛЕННЯ ВЕЛИКИХ S-BOXES НА БАЗІ ТЕРНАРНОЇ ЛОГІЧНОЇ ІНСТРУКЦІЇ

Статтю присвячено пошуку евристичних методів знаходження bitsliced-описів великих S-Boxes ($n \geq 6$) зі зменшеною кількістю інструкцій на базі тернарної логічної інструкції, яка є доступною в x86-64 процесорах з підтримкою SIMD-технології AVX-512 та деяких GPU-процесорах. Одна тернарна інструкція дає змогу замінити декілька звичайних логічних інструкцій і отже збільшити продуктивність криптографічних алгоритмів, що їх використовують. Згенеровані запропонованим методом bitsliced-описи дають змогу покращити швидкодію програмних імплементацій деяких криптоалгоритмів та зменшити їх вразливості до атак по стороннім каналам. У роботі розроблено евристичний метод мінімізації найпоширеніших 8×8 S-Boxes, в якому завдяки поєднанню різних технік з використанням GPU-обчислень вдалося зменшити кількість вентилів у bitsliced-описах порівнюючи з іншими відомими методами при прийнятній тривалості роботи. Розроблено відповідне програмне забезпечення у вигляді утиліти мовою Python і протестовано її роботу на S-Boxes різноманітних криптоалгоритмів. Встановлено, що розроблений метод генерує bitsliced-описи з на 15% меншим числом тернарних інструкцій, порівнюючи з найкращим відомим на сьогодні методом. Також як частковий випадок розглянуто застосування запропонованого методу для знаходження bitsliced-описів S-Boxes шифру DES (6×4), де також вдалося на 6% зменшити кількість інструкцій порівняно з найкращим існуючим результатом. Ці описи дають можливість збільшити швидкодію DES-шифрування і використовуються в утилітах підбору DES-based хешів Unix-паролів, зокрема, популярній програмі аудиту паролів John The Ripper.

**Ключові слова:** bitslicing, тернарна логічна інструкція, 8×8 S-Box, DES S-Boxes, CPU, GPU логічна мінімізація, програмна імплементація, sboxgates, швидкодія.

## Вступ

Криптографічні алгоритми (КА) стали невід'ємною частиною сучасного цифрового світу у вигляді програмних та/або апаратних засобів. Попри гнучкість і дешевизну, які забезпечують програмні реалізації КА, у них є два серйозних виклики – швидкодія та безпека. Щоби забезпечити високу швидкодію використовуються різні підходи до програмної імплементації криптографічних алгоритмів залежно від конкретної мікропроцесорної архітектури: створення передобчислених таблиць (Lookup Tables, LUT) для трудомістких операцій, інтеграція в процесор апаратних криптоакселераторів (наприклад, AES-NI у x86-процесорах), застосування SIMD-технології для розпаралелення обчислень (наприклад, векторні інструкції SSE/AVX2/AVX-512 у x86-64 CPU) тощо. Проте ці підходи мають низку обмежень, не завжди доступні в конкретному процесорі, а деякі з них потенційно вразливі. Безпека пов'язана з необхідністю забезпечити стійкість до атак через сторонні канали: для low-end CPU (8/16/32-бітні MCU) це насамперед атаки аналізу енергоспоживання, а для high-end CPU (x86, ARM Cortex-A, RISC-V) це передусім часові та кеш атаки.

З огляду на вказані виклики серед різноманітних підходів до програмної імплементації КА виділяється bitslicing [1]. Він забезпечує високопродуктивну constant-time імплементацію з імунітетом до часових та кеш атак [2], максимально використовує можливості сучасних high-end мікропроцесорів щодо розпаралелювання обробки даних, а також допускає адаптацію для low-end CPU і апаратну реалізацію на FPGA. Для багатьох КА саме цей підхід забезпечує найбільшу швидкодію в програмній реалізації (якщо не використовуються апаратні криптоакселератори) для різного типу процесорних архітектур [1-7]. Відсутність звернень до 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 інші сторонні канали.

Bitsliced-опис це опис криптографічного алгоритму у вигляді послідовності логічних операцій AND, XOR, OR, NOT та ін. У процесорах кожну таку логічну операцію можна представити відповідною інструкцією, а в апаратних засобах – відповідним вентилем (тому надалі в роботі поняття вентиль та інструкція будемо використовувати як синоніми). У

випадку програмної bitsliced-реалізації висока швидкодія 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 програмноорієнтованих bitsliced-реалізацій крім традиційних можна використати більш складні логічні інструкції, які підтримує певний процесор, і таким чином зменшити їх загальну кількість. Так, наприклад, багато процесорів підтримують інструкцію AND-NOT (x86, ARM), деякі NOT-OR і NOT-AND (ARM) тощо.

Крім типових двохоперандних логічних інструкцій в деяких процесорах з'явилася тернарна інструкція *ternarylogic(a, b, c, imm8)*, що дає змогу обчислити довільну булеву функцію для трьох операндів *a, b, c* задану таблицею істинності у 8-бітній змінній *imm8* (рис. 1). Тернарна інструкція (TI) присутня у x86-64 CPU з підтримкою 512-бітних SIMD-інструкцій AVX-512 та деяких GPU.

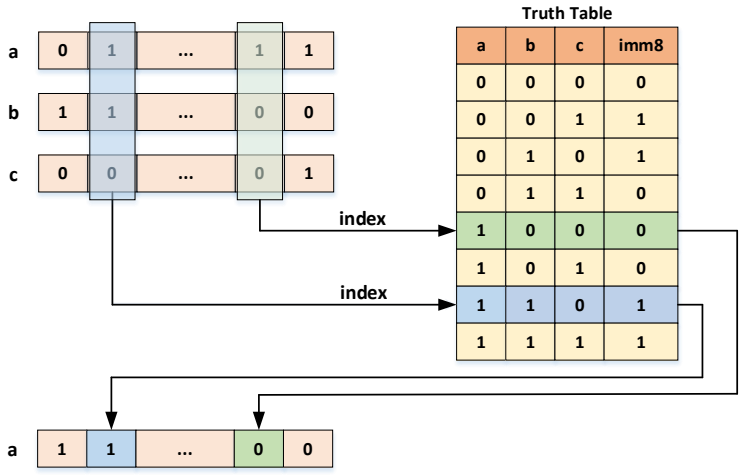


Рис. 1. Принцип роботи тернарної інструкції *ternarylogic(a, b, c, imm8)*

Використання тернарних інструкцій для bitsliced-описів криптоалгоритмів є досить перспективним, оскільки одна тернарна інструкція може замінити декілька двохоперандних логічних інструкцій при однакових latency/throughput, що дозволяє згенерувати опис з суттєво меншим числом операцій, а отже збільшити швидкодію програмної реалізації.

Очевидно, щоб отримати максимальну швидкодію потрібно мінімізувати кількість логічних операцій, що входять до bitsliced-опису криптоалгоритму. Більшість криптографічних операцій породжують однозначний опис при переході до bitsliced-опису (наприклад, XOR) або не дають багато простору для мінімізації за винятком нелінійних перетворень. У КА операції нелінійної заміни задаються у вигляді $n \times m$ LUT-таблиць, так званих S-Boxes, що переважно мають розмір 4×4 ($n$ = 4) або 8×8 ($n$ = 8) біт. Таким чином, основним ресурсом збільшення швидкодії при bitsliced імплементації КА є представлення S-Boxes мінімально можливою кількістю логічних інструкцій. Ця задача 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 використовують евристичні підходи, які не гарантують, що отриманий розв'язок є оптимальним, але забезпечують значно кращий результат порівняно з універсальними методами мінімізації логічних функцій (наприклад, методом Куайна-Мак-Класкі). Слід відзначити, що у випадку використання TI проблема ускладнюється тим, що відомі методи bitsliced-опису S-Boxes здебільшого орієнтовані на стандартні логічні інструкції та не працюють з тернарними. Існуючих рішень для генерації bitsliced-описів великих S-Boxes ($n > 4$) на базі тернарної інструкції досить мало і вони або мають обмежені можливості або недостатньо ефективні.

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

Отже, проблема знаходження bitsliced представлення великих S-Boxes на базі тернарної інструкції є актуальною і потребує пошуку нових евристичних методів, один із яких представлений у нашій роботі.

**Аналіз літературних джерел та формулювання проблеми**

Проаналізуємо методи та засоби для знаходження bitsliced-описів S-Boxes за критерієм Bitslice Gate Complexity (BGC), що позначає пошук рішення з найменшою кількістю операцій.

Bitsliced підхід до представлення криптоалгоритму був уперше запропонований E. Biham у роботі [1] для пришвидшення програмної імплементації шифру DES. У цій ж роботі E. Biham описав алгоритм bitsliced-представлення восьми таблиць заміни DES S-Boxes (6×4) логічними вентилями XOR, AND, OR, NOT. У ньому фіксовані чотири вхідні біти S-Box використовуються для формування селекторів, які вибирають одну з 16-ти можливих логічних функцій двох вхідних змінних. У середньому 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 біт, що представлена картою Карно й розташовується у 64-бітній змінній (у нашій роботі ми називаємо їх векторами). Вхідні та проміжні змінні теж можна розглядати як 6-бітні карти Карно, що описуються 64-бітними числами, які потрібно скомбінувати таким чином, щоб отримати шукану вихідну змінну. Одна вхідна змінна виступає в якості селектора, що об'єднує дві функції п'яти змінних. Для пошуку представлення функцій п'яти змінних із мінімальною кількістю вентилів застосовується вичерпний перебір (brute force) на певну глибину та вентилі знайдені на попередніх кроках. Пошук рекурсивно повторюється. Залежно від того, в якому порядку буде здійснюватися пошук доступні 6! варіантів для вхідних змінних та 4! варіантів для вихідних змінних. Це всього дає 17280 варіантів пошуку серед яких вибирається варіант із мінімальним числом вентилів. Також перебираються всі можливі варіанти функції селекції. У результаті середнє число вентилів для bitsliced-опису одного з восьми DES S-Box зменшилося зі 100 до 56.

У 2011 р. R. Rusakov та A. Peslyak ще удосконалили метод M. Kwan та досягнули середнього показника 44 вентилі/S-Box для набору логічних вентилів AND, OR, XOR, NOT та AND-NOT. Пошук зайняв декілька місяців обчислень на 4-ядерному процесорі з 24 Гбайт RAM. Також ними було знайдено bitsliced-описи DES S-Boxes на базі тернарної інструкції з середнім числом інструкцій 23.5 (табл. 1) [9]. Ці описи до сьогодні мали найменшу відому BGC для DES S-Boxes, через що використовуються у багатьох утилітах підбору DES-based хешів Unix-паролів, зокрема в такій популярній програмі аудиту безпеки паролів як *John The Ripper*. DES-based хеші в Unix-системах формуються шляхом 25-кратного шифрування алгоритмом DES нульового рядка, де в якості ключа виступає пароль користувача. Хешем, який зберігається в системі, є отриманий 64-бітний результат шифрування.

Таблиця 1

Bitsliced-описи DES S-Boxes на базі тернарної інструкції

| **S-Box** | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 |
|---|---|---|---|---|---|---|---|---|
| **TI** | 25 | 24 | 25 | 18 | 25 | 24 | 24 | 23 |

Хоча R. Rusakov та A. Peslyak не надали детальний опис свого алгоритму чи тексту програми, спираючись на інформацію з відкритих джерел та аналіз самих bitsliced-описів можна стверджувати, що ідея була вибрати одну з шести вхідних змінних як селектор і далі вести пошук алгоритмом depth-first search (DFS) до знаходження всіх 8-ми 32-бітних векторів з врахуванням вже знайдених вентилів, після чого знайдені вектори з допомогою відповідної функції об'єднуються в чотири вихідних 64-бітні значення. Також попередньо було побудовано 4 Гбайтну таблицю складності, де кожному 32-бітному вектору ставиться у

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

відповідність його BGC-складність знайдена за алгоритмом breath-first-search (BFS), що дає змогу на перших кроках вибирати найоптимальніше рішення. Алгоритм перебирає всі можливі вхідні біти в якості селектора, а також всі можливі комбінації операцій об'єднання 32-бітних векторів у 64-бітний: OR та XOR, AND та XOR або AND-NOT та XOR.

У роботі [10] описано метод знаходження представлення 8×8 S-Boxes з допомогою тернарної інструкції, що використовує розбиття таблиці істинності на 32-бітні вектори, які є функціями п'ятьох вхідних бітів, а решта три біти використовуються як маски і служать для склеювання 32-бітних векторів у 256-бітний вихідний біт. Метод демонструє достатньо високу швидкодію (2-3 години/S-Box) і забезпечує представлення довільного S-Box з допомогою не більше 210 тернарних інструкцій. Недоліком методу є відносно великі накладні витрати на об'єднання 32-бітних векторів у 256-бітні.

Відомі також інші підходи до мінімізації S-Boxes, які не спираються на метод M. Kwan, а, наприклад, намагаються знайти точний розв'язок за допомогою SAT-Solvers [11], чи використовують meet-in-the-middle алгоритм у поєднанні з алгоритмом пошуку вглибину BFS, як це реалізовано в утилітах LIGHTER та Peigen [12, 13]. Проте вказані підходи потребують інтенсивних обчислень і тому здатні працювати лише з малими S-Boxes (до 4×4), а крім того орієнтовані на використання переважно двовходових логічних елементів і не підтримують тернарну логічну інструкцію.

Генерація оптимізованої bitsliced імплементації КА потребує значних затрат часу на написання і відладку коду та вимагає хорошого знання архітектури процесора, низькорівневих інструментів та технік оптимізації на апаратному і програмному рівні. Тому у роботі [14] представлено високорівневу мову Usuba, що дає змогу описати симетричний криптографічний примітив, а вже сам Usuba компілятор згенерує оптимізований, розпаралелений і векторизований bitsliced-код мовою C. Проте для генерації bitsliced опису S-Box використовується або простий алгоритм мінімізації, який дає далеко не оптимальний результат, або береться готовий оптимізований опис з бази даних, що входить до складу Usuba, якщо S-Box у ній присутній. Таким чином, генерація опису для S-Box є слабким місцем bitsliced-компілятора Usuba.

Єдина відома нам на сьогодні публічна утиліта для генерації bitsliced-описів S-Boxes на базі тернарної інструкції це sboxgates [15]. Ця open source утиліта реалізує алгоритм M. Kwan з деякими удосконаленнями і здатна генерувати bitsliced-опис для довільних S-Boxes до 8×8 включно, але для тернарної інструкції з певними обмеженнями. Утиліта дає змогу задавати довільний набір двовходових вентилів, вказувати кількість ітерацій алгоритму пошуку, розпаралелювати пошук між процесорними ядрами тощо.

Обмеженням є те, що для 8×8 S-Box утиліта не здатна згенерувати bitsliced-опис для всіх 8-ми вихідних біт, а лише для кожного біту окремо через надто великий обсяг обчислень. Тобто вентилі використані для формування одного вихідного біту повторно не використовуються при знаходженні опису іншого вихідного біту, що призводить до не оптимальних результатів.

Один вихідний біт потребує близько 30 TI, відповідно S-Box загалом потребуватиме 240 TI, що є гіршим результатом ніж отримано в [10]. Можна виконувати спільну мінімізацію по два вихідних біти, проте це теж потребує декілька тижнів обчислень, а число вентилів при цьому становить близько 55, що дає оцінку у 220 вентилів – співмірну з результатами [10], проте при значно більшому часу пошуку.

**Мета роботи та цілі дослідження**

Метою статті є представити метод для генерації bitsliced-опису 8×8 S-Boxes на базі тернарної логічної інструкції, що забезпечує як кращі результати порівняно з існуючими, так і прийнятний час роботи, а це дасть змогу збільшити швидкодію та безпеку програмно-апаратних реалізацій широкого кола криптографічних алгоритмів, які використовують великі S-Boxes.

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

**Результати досліджень**

*Алгоритм пошуку bitsliced-представлення.* Двома головними складовими нашого алгоритму пошуку bitsliced-опису великих S-Boxes BSL64 є:

1. Формат представлення S-Box у вигляді масок та векторів.

2. Методи знаходження найменших bitsliced-описів векторів на різних етапах роботи алгоритму.

*1. Формат представлення S-Box.*

Нехай ми маємо 8×8 S-Box $S$ заданий у вигляді LUT-таблиці з 256 значеннями. У нашому алгоритмі $S$ розглядається як логічна функція задана таблицею істинності. Отож, компактне представлення даного S-Box у вигляді таблиці істинності буде мати вигляд: $S(x) = y$, де $x = \{x_0, x_1, ..., x_7\}$ – вхідні bitsliced-змінні, $y = \{y_0, y_1, ..., y_7\}$ – вихідні bitsliced-змінні, що визначають конкретну таблицю заміни. Таблиця розбивається на дві частини: тридцять два 64-бітні вектори, які є функціями молодших шести вхідних змінних $x_2$-$x_7$, та чотири маски, які є функціями старших двох вхідних змінних $x_0$-$x_1$ (табл. 2). Маски служать для об'єднання відповідних 64-бітних векторів у вихідний біт $y_i$. Перехід від 32-бітних векторів, які використовувались в роботі [10], до 64-бітних дає змогу зменшити кількість накладних операцій, необхідних для формування масок і вихідних бітів з 72 до 36, а також краще використовувати проміжні результати, але це досягається за рахунок зростання глибини пошуку і відповідно значного ускладнення процедури пошуку.

Таблиця 2

Розбиття 8×8 S-Box на маски та вектори

| ***mask*** | $x_2$-$x_7$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ | $y_7$ |
|---|---|---|---|---|---|---|---|---|---|
| $m_0 = \overline{x_0}\,\overline{x_1}$ | … | $y_{00}$ | $y_{10}$ | $y_{20}$ | $y_{30}$ | $y_{40}$ | $y_{50}$ | $y_{60}$ | $y_{70}$ |
| $m_1 = \overline{x_0}\,x_1$ | … | $y_{01}$ | $y_{11}$ | $y_{21}$ | $y_{31}$ | $y_{41}$ | $y_{51}$ | $y_{61}$ | $y_{71}$ |
| $m_2 = x_0\,\overline{x_1}$ | … | $y_{02}$ | $y_{12}$ | $y_{22}$ | $y_{32}$ | $y_{42}$ | $y_{52}$ | $y_{62}$ | $y_{72}$ |
| $m_3 = x_0\,x_1$ | … | $y_{03}$ | $y_{13}$ | $y_{23}$ | $y_{33}$ | $y_{43}$ | $y_{53}$ | $y_{63}$ | $y_{73}$ |

Отже, вихідні біти $y_i$ можна представити у вигляді:

$$y_i = m_0 y_{i0} \mid m_1 y_{i1} \mid m_2 y_{i2} \mid m_3 y_{i3}, \text{ де } i = 0, \dots, 7 \quad (1)$$

Обчислення масок $m_0$-$m_3$ і рівнянь (1) з допомогою TI є тривіальним (2) та не залежить від конкретного S-Box, тому вся складність алгоритму полягає у знаходженні векторів $y_{ij}$.

$$\begin{gathered} m_0 = ternarylogic(x_0, x_1, x_0, 0x03) \\ m_1 = ternarylogic(x_0, x_1, x_0, 0x0c) \\ m_2 = ternarylogic(x_0, x_1, x_0, 0x30) \\ m_3 = ternarylogic(x_0, x_1, x_0, 0xc0) \\ \\ y_i = ternarylogic(y_i, m_0, y_{i0}, 0x88) \\ y_i = ternarylogic(y_i, m_1, y_{i1}, 0xf8) \\ y_i = ternarylogic(y_i, m_2, y_{i2}, 0xf8) \\ y_i = ternarylogic(y_i, m_3, y_{i3}, 0xf8) \end{gathered} \quad (2)$$

Варто відзначити, що дане представлення може бути адаптоване і до великих S-Box з розрядністю менше 8. Наприклад, для DES S-Boxes, які мають розмір 6×4, маски будуть відсутні, а таблиця описується чотирма 64-бітними векторами $y_0$-$y_3$ і алгоритм пошуку BSL64 зводиться лише до другої частини.

*2. Методи знаходження найменших bitsliced-описів векторів.*

Завдання на цьому етапі полягає в тому, щоб відштовхуючись від вхідних векторів $x_2$-$x_7$, які ми позначимо як базу $base = [x_2, \dots, x_7]$, потрібно знайти всі 32 вектори $y_{ij}$ використовуючи

мінімум проміжних змінних *t*. При цьому поточні результати пошуку зберігаються у вигляді *траси* – масиву з 64-бітних векторів, що містять проміжні змінні *t* та знайдені $y_{ij}$. Після знаходження всіх векторів $y_{ij}$ траса декодується у відповідні тернарні інструкції.

Кожен елементи траси послідовно додається до бази *base*. Наприклад, якщо під час пошуку отримано такий фрагмент:

*t0 = ternarylogic(x3, x5, x6, 0x29),*
*t1 = ternarylogic(x5, x4, t0, 0x96),*
*t2 = ternarylogic(t1, x3, x6, 0x35),*
*y0 = ternarylogic(t2, x3, t0, 0xd1),*

то він буде представлений у вигляді бази і траси таким чином:

*base = [x2, x3, x4, x5, x6, x7, t0, t1, t2, y0],*
*trace = [t0, t1, t2, y0].*

На кожному з 32-х кроків пошуку нам потрібно знайти один з векторів $y_{ij}$ використовуючи як можна менше тернарних інструкцій. І хоча такий евристичний підхід не гарантує оптимального результату загалом, але допускає знаходження рішення за прийнятний час. На жаль, безпосередній пошук 64-бітних векторів шляхом brute force можливий лише на відносно малу глибину, яка залежить від поточної довжини траси. На рис. 2 показано зростання коефіцієнта розгалуження (branching factor) зі збільшенням довжини траси для 8×8 *Sbox0-Sbox3* шифру «Калина». Як зрозуміло з рис. 2 пошук на глибину *d* рівну 3 здійснений лише на перших кроках, а далі ефективний пошук можливий лише на глибину *d* рівну 1 або 2, але і для цього, по мірі зростання довжини траси, обчислювальної потужності CPU вже не вистачає та потрібно задіювати GPU. Нами було реалізовано функції пошуку найближчого значення $y_{ij}$ методом BFS на глибину від 1 до 3 TI з використанням GPU.

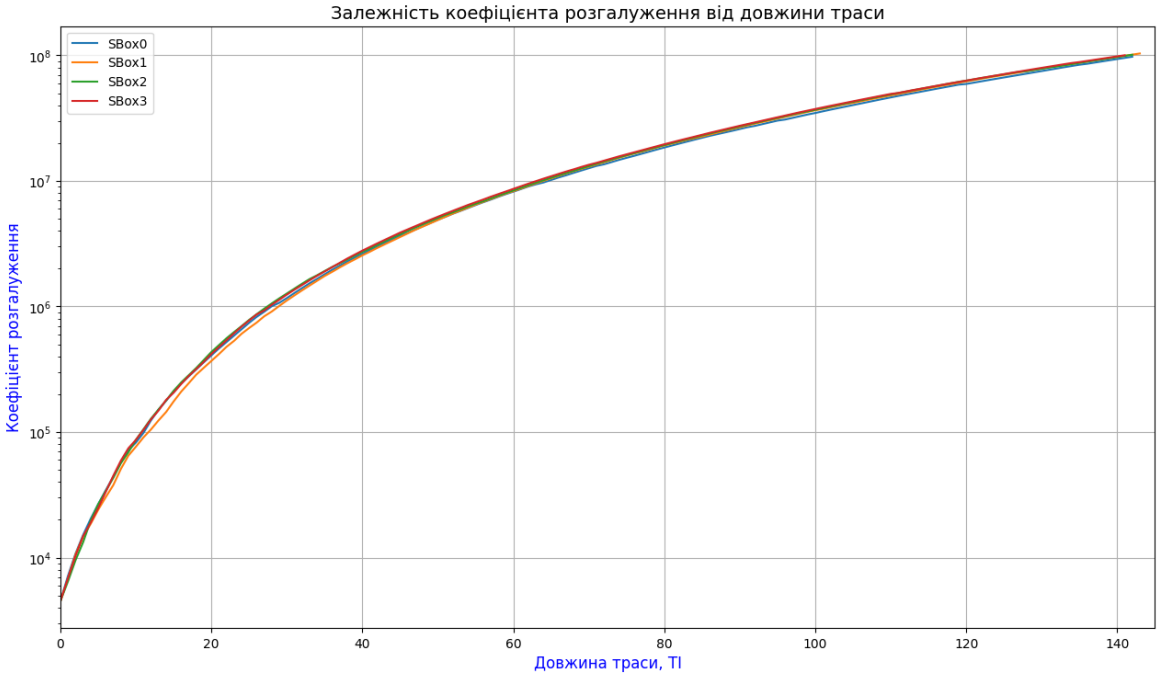


Рис. 2. Ріст коефіцієнта розгалуження зі збільшенням довжини траси

Проте навіть глибини пошуку *d = 3* для 64-бітних векторів є недостатньо, бо як показує практика, на перших кроках коли база невелика, глибина пошуку типово становить 6-8 TI (рис. 3) і зі збільшенням бази поступово спадає до 4 та не перевищує це значення починаючи з десятого кроку, коли довжини трас для *Sbox0-Sbox3* стають рівними 50, 51, 50 і 49 TI

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

відповідно. Тому доводиться так чи інакше розбивати пошук 64-бітного вектора на частини і далі склеювати їх в одне ціле.

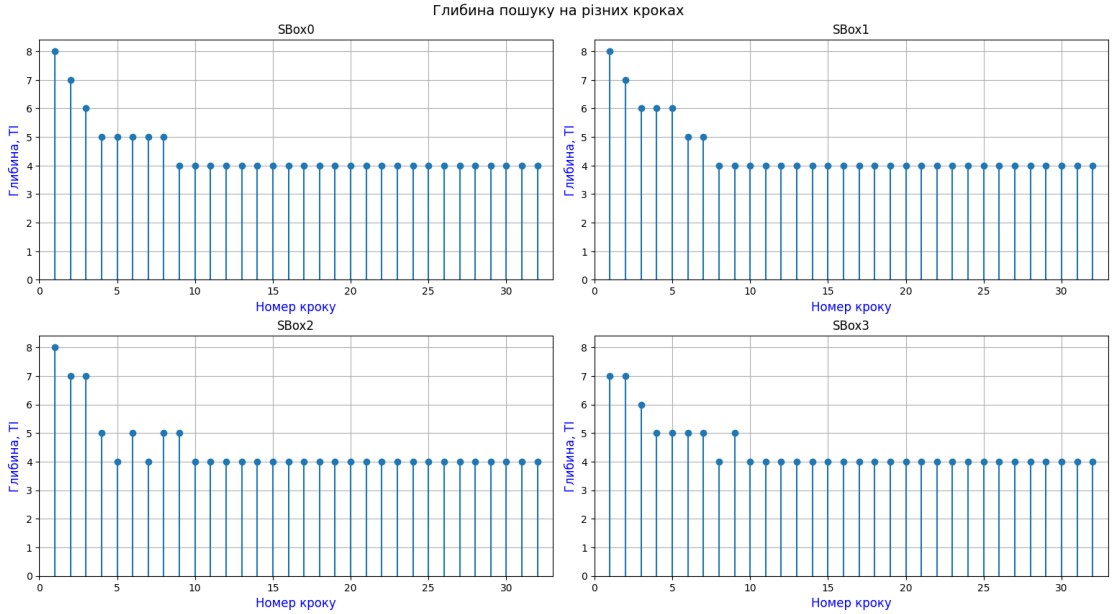


Рис. 3. Залежності глибини пошуку від номера кроку для *Sbox0-Sbox3* шифру «Калина»

В роботах [8, 9, 15] 64-бітний вектор розбивається на два вектори, а для їх об'єднання завжди використовується одна з вхідних змінних *x*, так званий селектор, одиниці і нулі в якому визначають, які біти вибираються з кожної частинки. Оскільки вага Хеммінга *hw* вхідних змінних *hw(x)* завжди рівна 32, то відповідно з кожного вектора відбираються 32-біти, які об'єднуються у 64-бітний вектор.

У нашій роботі селектором можуть виступати довільні вхідні *x*, проміжні *t* чи вихідні *y* змінні, що забезпечує більшу гнучкість і дає змогу знаходити описи з меншою кількістю вентилів, а 64-бітне значення *s* формується TI з двох частин $F_{ab}$ та $F_{abc}$ операцією OR наступним чином:

$$s = ternarylogic(a, b, c, imm8) = \bar{a}b \mid af(b, c) = F_{ab} \mid F_{abc} \qquad (3)$$

Функція *f(b, c)* у (3) задає будь-яку з 16-ти можливих логічних функцій двох аргументів, проте здебільшого це функція XOR (рис. 4). Тернарна інструкція дозволяє також у формулі (3) обирати для $F_{ab}$ форму на основі логічного AND між різними комбінаціями прямого та інверсного представлення *a* і *b*, що враховується в алгоритмі.

Пошук *s* відбувається у два етапи – спочатку знаходиться $F_{ab}$, потім $F_{abc}$. На першому етапі ми здійснюємо пошук на задану глибину $d_{ab}$ і знаходимо всі траси, які забезпечують не менше $h_{ab}$ біт *s* для $F_{ab}$:

$$F_{ab} = \bar{a}b = \bar{a}s, \text{ де } hw(\bar{a}) \geq h_{ab} \qquad (3)$$

Загалом це дає змогу об'єднувати два вектори вибираючи з кожного різну кількість біт, а не тільки по 32, хоча дане значення є найбільш типовим. Будемо позначати символом *ab* проміжний вектор у трасі на глибині $d_{ab}$, який вказує, що $F_{ab}$ знайдено. Наприклад, якщо $d_{ab}$ = 3, то після першого етапу пошуку траса буде мати вигляд:

*trace = [t0, t1, ab]*

На другому етапі здійснюється пошук на глибину $d_{abc}$ шляхом вичерпного перебору алгоритмом BFS для знаходження *c* і відповідно *s*. Наприклад, якщо $d_{abc}$ = 3, то після другого етапу у випадку успішного пошуку траса буде мати вигляд:

*trace = [t0, t1, ab, t2, c, s]*

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

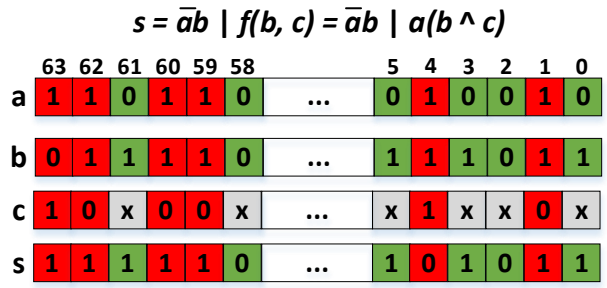


Рис. 4. Об'єднання на основі довільного селектора *a* двох векторів *b* і *c* у вектор *s*

У нашому алгоритмі глибина пошуку $d_{ab}$ на першому кроці становить 3, на наступних кроках 2 і після четвертого кроку 1. Ці значення, а також значення $d_{abc}$ і $h_{ab}$, емпірично підібрані під час тестування алгоритму на реальних S-Boxes, таким чином, щоб знайти компроміс між часом обчислень, який визначається поточною довжиною траси і достатньо високою імовірністю успіху пошуку. У табл. 3 наведено покрокове значення параметрів алгоритму пошуку BSL64: $d_{ab}$, $d_{abc}$, $h_{ab}$. З табл. 3 слідує, що максимальна сумарна глибина пошуку $d$ становить 6 ТІ на першому кроці, $d = 5$ на кроках 2-4 і $d = 4$ на решті кроків.

Таблиця 3

Параметри алгоритму пошуку bitsliced-опису BSL64

| Крок | $d_{ab}$ | $d_{abc}$ | $h_{ab}$ |
|---|---|---|---|
| 1 | 3 | 3 | 28 |
| 2 | 2 | 3 | 28 |
| 3 | 2 | 3 | 29 |
| 4 | 2 | 3 | 32 |
| 5-32 | 1 | 3 | 32 |

На перших кроках, коли база ще мала, може виникнути ситуація, що при вказаних параметрах для відібраних $F_{ab}$ не було знайдено жодного з векторів *s* на глибині $d_{abc}$ *(див. рис. 3)*. Тоді пошук ведеться за спрощеною формулою (3), яка забезпечує значення $d_{abc}$ до 6:

$s = ternarylogic(a, b, c, imm8) = \overline{a_1}b_1 \mid a_1c_1 = \overline{a_1}b_1 \mid a_1[\overline{a_2}b_2 \mid a_2 f(b_2, c_2)]$

*trace = [..., a1b1, a2b2, c2, c1, s]* $d_{abc} = 4$

*trace = [..., a1b1, a2b2, t0, c2, c1, s]* $d_{abc} = 5$

*trace = [..., a1b1, t0, a2b2, c2, c1, s]* $d_{abc} = 5$

*trace = [..., a1b1, t0, a2b2, t1, c2, c1, s]* $d_{abc} = 6$

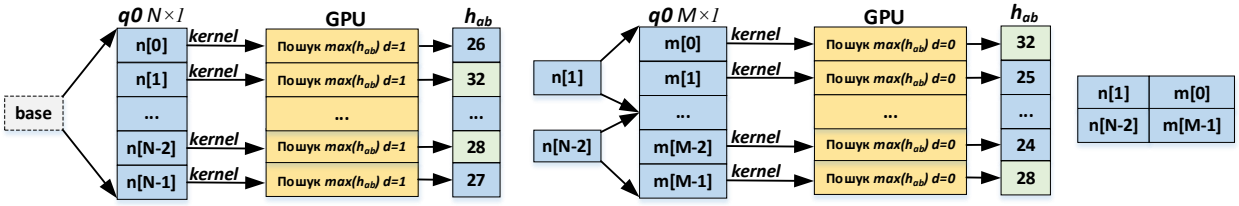


Рис. 5. Пошук $F_{ab}$ за підтримки GPU

Пошук $F_{ab}$ та $F_{abc}$ здійснюється з допомогою GPU. Наприклад, якщо потрібно знайти траси на глибині $d_{ab} = 2$, які забезпечують не менше $h_{ab} = 28$ біт *s* для $F_{ab}$, то спочатку CPU генерує масив $q_0$, який містить $N$ векторів, що можна отримати тернарною інструкцією комбінуючи всіма можливими способами елементи з бази *base*. Далі цей масив $q_0$ разом з *base* і *s* передається в GPU для обробки. Кожне GPU-ядро обробляє відповідний рядок $q_0$ і знаходить максимальне значення $h_{ab}$ на глибині 1 ТІ. Ці значення у вигляді масиву

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

використовуються для фільтрації трас *q0* у яких $h_{ab} \geq 28$. Далі процедура повторюється, тільки GPU-ядра здійснюють тепер пошук на глибину 0, тобто лише серед елементів відповідної траси. Сформований масив значень $h_{ab}$ використовується для остаточного відбору трас довжиною 2 TI, які відповідають заданому критерію (рис. 5).

**Результати**

Запропонований у роботі алгоритм BSL64 був імплементований мовою Python, а для забезпечення швидкодії основні функції обробки даних реалізовані на базі бібліотек *numpy* та *pyopencl*.

Для оцінки нашого алгоритму було взято одинадцять 8×8 S-Boxes різноманітних симетричних шифрів: 4 прямих (*SBox0-SBox3*) і 4 інверсних (*InvSBox0-InvSBox3*) таблиць українського шифру «Калина», пряму та інверсну таблиці американського шифру AES і таблицю китайського шифру SM4. Результати представлені в табл. 4 показують, що завдяки алгоритму BSL64 вдалося сумарно на 15% (31 тернарну інструкцію на S-Box) покращити попередні результати.

Таблиця 4

Порівняння результатів BSL64 для 8×8 S-Boxes

| S-Box | [10] | Ця робота |
|---|---|---|
| *Sbox0 Kalyna* | 209 | 178 |
| *Sbox1 Kalyna* | 208 | 179 |
| *Sbox2 Kalyna* | 209 | 178 |
| *Sbox3 Kalyna* | 209 | 177 |
| *InvSbox0 Kalyna* | 208 | 178 |
| *InvSbox1 Kalyna* | 209 | 179 |
| *InvSbox2 Kalyna* | 209 | 179 |
| *InvSbox3 Kalyna* | 209 | 176 |
| *AES* | 210 | 176 |
| *InvAES* | 210 | 177 |
| *SM4* | 208 | 178 |
| | Σ = 2298 | Σ = 1955 |

Також ми застосували основні ідеї нашого алгоритму до восьми 6×4 таблиць *S1-S8* шифру DES і теж вдалося знайти bitsliced-описи з сумарно на 6% меншою кількістю TI порівняно з найкращими відомими результатами R. Rusakov та A. Peslyak (табл. 5). Додатково у табл. 5 наведено результати отримані утилітою *sboxgates*, що здатна згенерувати повноцінний bitsliced-опис для DES S-Boxes. Ця утиліта демонструє значно слабші результати, які на 21% поступаються нашим.

Таблиця 5

Порівняння результатів BSL64 для 6×4 DES S-Boxes

| S-Box | sboxgates | R. Rusakov, A. Peslyak | Ця робота |
|---|---|---|---|
| *S1* | 30 | 25 | 23 |
| *S2* | 26 | 24 | 22 |
| *S3* | 28 | 25 | 24 |
| *S4* | 30 | 18 | 17 |
| *S5* | 28 | 25 | 23 |
| *S6* | 26 | 24 | 22 |
| *S7* | 27 | 24 | 23 |
| *S8* | 28 | 23 | 23 |
| | Σ = 223 | Σ = 188 | Σ = 177 |

Аномально мале значення BGC для DES S-Box *S4* у наших результатах і у результатах R. Rusakov, A. Peslyak пояснюється тим, що врахована властивість симетричності цієї таблиці,

© Совин Я.Р., Опірський І.Р., Комарницький В.О. (2025). Знаходження bitsliced-представлення великих S-Boxes на базі тернарної логічної інструкції. Сучасний захист інформації, 1(61), 153–164. https://doi.org/10.31673/2409-7292.2025.017435.

коли в якості селектора для всіх вихідних біт обрано вхідний біт $x_5$. У цьому випадку для її опису достатньо знайти два вихідних біти $y_0$ і $y_2$, а $y_1$ та $y_3$ використовують вже знайдені вектори:

$$y_0 = \overline{x_5}\ (0x30c3cf33cf30f03c) \mid x_5(0xc3c3f30cfcc00c3f)$$
$$y_1 = \overline{x_5}\ (0xc3c3f30cfcc00c3f) \mid x_5(\overline{0x30c3cf33cf30f03c})$$
$$y_2 = \overline{x_5}\ (0x0cf33c0fc30cccf3) \mid x_5(0xfcc003cf3c3ccf30)$$
$$y_3 = \overline{x_5}\ (\overline{0xfcc003cf3c3ccf30}) \mid x_5(0x0cf33c0fc30cccf3)$$

Оскільки покращені bitsliced-описи DES S-Boxes можуть бути корисні в багатьох проектах і утилітах, які вже експлуатують результати R. Rusakov та A. Peslyak, ми вирішили навести їх у статті (табл. 6), щоб зробити публічними і доступними для вільного використання.

Таблиця 6

Bitsliced-описи DES S-Boxes на базі тернарної інструкції

| **S1, 23 TI** | **S2, 22 TI** |
|---|---|
| t0 = ternarylogic( x2, x1, x0, 0x29) | t0 = ternarylogic( x4, x3, x0, 0x69) |
| t1 = ternarylogic( x5, x4, t0, 0x96) | t1 = ternarylogic( x5, x2, x1, 0xc1) |
| t2 = ternarylogic( x5, x4, x0, 0x35) | t2 = ternarylogic( x5, x4, x0, 0xc7) |
| t3 = ternarylogic( x3, x2, t2, 0x2e) | t3 = ternarylogic( x3, x2, t2, 0x28) |
| t4 = ternarylogic( x4, x1, t1, 0x9c) | t4 = ternarylogic( x5, x1, t3, 0x94) |
| t5 = ternarylogic( x3, t0, t4, 0x61) | y1 = ternarylogic( t0, t1, t4, 0xd2) |
| y0 = ternarylogic( t1, t3, t5, 0xd1) | t5 = ternarylogic( x4, x0, t1, 0x19) |
| t6 = ternarylogic( x4, x0, t0, 0x94) | t6 = ternarylogic( x4, x1, t0, 0x98) |
| t7 = ternarylogic( x3, t0, t1, 0x6b) | t7 = ternarylogic( x3, x1, t3, 0x21) |
| t8 = ternarylogic( x0, t5, t7, 0xd8) | t8 = ternarylogic( x5, x2, t7, 0x09) |
| t9 = ternarylogic( x5, x3, x2, 0x89) | y0 = ternarylogic( t5, t6, t8, 0xc9) |
| t10 = ternarylogic( t0, t1, t9, 0xe9) | t9 = ternarylogic( x5, x0, y0, 0x74) |
| y1 = ternarylogic( t6, t8, t10, 0xc6) | t10 = ternarylogic( x3, t4, t9, 0x68) |
| t11 = ternarylogic( x5, x1, t7, 0x88) | t11 = ternarylogic( x5, x4, y0, 0x4d) |
| t12 = ternarylogic( x5, x3, y0, 0x01) | t12 = ternarylogic( x1, y1, t11, 0x69) |
| t13 = ternarylogic( x2, x0, t12, 0xb6) | y3 = ternarylogic( t9, t10, t12, 0xc5) |
| t14 = ternarylogic( x2, t0, t1, 0x7c) | t13 = ternarylogic( x4, y0, t10, 0x09) |
| t15 = ternarylogic( x4, t5, t14, 0x79) | t14 = ternarylogic(t12, y3, t13, 0x69) |
| y3 = ternarylogic(t11, t13, t15, 0xc6) | t15 = ternarylogic( t1, t2, y1, 0xb7) |
| t16 = ternarylogic( x3, x0, t2, 0xcd) | t16 = ternarylogic( x1, t3, t9, 0xe2) |
| t17 = ternarylogic( t8, t12, t16, 0xda) | t17 = ternarylogic( t4, t10, t15, 0x92) |
| t18 = ternarylogic( x5, t4, t15, 0x81) | y2 = ternarylogic(t14, t16, t17, 0xf2) |
| y2 = ternarylogic(t14, t17, t18, 0xe3) | |
| **S3, 24 TI** | **S4, 17 TI** |
| t0 = ternarylogic( x4, x2, x0, 0x37) | t0 = ternarylogic( x4, x3, x0, 0xc9) |
| t1 = ternarylogic( x5, x1, t0, 0x96) | t1 = ternarylogic( x2, x1, t0, 0x64) |
| t2 = ternarylogic( x5, x4, x3, 0x63) | t2 = ternarylogic( x4, x3, x2, 0xdb) |
| t3 = ternarylogic( x3, x0, t2, 0x68) | t3 = ternarylogic( x0, t1, t2, 0x6d) |
| t4 = ternarylogic( x1, x0, t2, 0x80) | t4 = ternarylogic( x4, x3, x0, 0x93) |
| y3 = ternarylogic( t1, t3, t4, 0xc9) | t5 = ternarylogic( x2, x1, t4, 0x46) |
| t5 = ternarylogic( x3, t0, t1, 0x62) | t6 = ternarylogic( x0, t2, t5, 0x49) |
| t6 = ternarylogic( x3, x2, x1, 0x29) | t7 = ternarylogic( x0, t3, t5, 0x2c) |
| t7 = ternarylogic( x0, t2, t6, 0x69) | t8 = ternarylogic( x4, t6, t7, 0x16) |
| t8 = ternarylogic( x2, x0, y3, 0xae) | t9 = ternarylogic( x2, x1, t8, 0x6a) |
| t9 = ternarylogic( x3, x1, t8, 0x19) | t10 = ternarylogic( t2, t3, t6, 0xe4) |
| y0 = ternarylogic( t5, t7, t9, 0xc9) | t11 = ternarylogic( x3, x1, t10, 0xd8) |
| t10 = ternarylogic( x5, x4, x1, 0x06) | t12 = ternarylogic( t3, t9, t11, 0x96) |
| t11 = ternarylogic( x3, t7, t10, 0x6c) | y0 = ternarylogic( x5, t6, t12, 0xc6) |
| t12 = ternarylogic( x3, x2, t11, 0xa9) | y1 = ternarylogic( x5, t6, t12, 0x9c) |
| t13 = ternarylogic( t6, y0, t11, 0x83) | y2 = ternarylogic( x5, t3, t9, 0xc6) |
| t14 = ternarylogic( x4, y3, t13, 0x26) | y3 = ternarylogic( x5, t3, t9, 0x63) |

```
y1  = ternarylogic( x2, t12, t14, 0x39)
t15 = ternarylogic( x2,  y0, t10, 0xa7)
t16 = ternarylogic(t13, t14, t15, 0x56)
t17 = ternarylogic( x0,  t9,  y1, 0x0d)
t18 = ternarylogic( x4,  y3,  y1, 0x4e)
t19 = ternarylogic( x1,  y0, t18, 0x96)
y2  = ternarylogic(t16, t17, t19, 0xb8)
```

**S5, 23 TI**

```
t0  = ternarylogic( x5,  x3,  x2, 0x46)
t1  = ternarylogic( x4,  x0,  t0, 0xe9)
t2  = ternarylogic( x5,  x1,  t1, 0x69)
t3  = ternarylogic( x5,  x2,  x0, 0x97)
t4  = ternarylogic( x4,  x1,  t3, 0x27)
y1  = ternarylogic( x3,  t2,  t4, 0xc9)
t5  = ternarylogic( x5,  x1,  t3, 0x8c)
t6  = ternarylogic( x4,  x1,  t0, 0x53)
t7  = ternarylogic( x3,  x0,  t6, 0x94)
t8  = ternarylogic( x2,  x0,  y1, 0x25)
t9  = ternarylogic( x5,  t1,  t8, 0x6e)
y2  = ternarylogic( t5,  t7,  t9, 0xc6)
t10 = ternarylogic( x3,  t0,  t1, 0x29)
t11 = ternarylogic( x3,  x1,  x0, 0x76)
t12 = ternarylogic( x3,  t3, t11, 0xb6)
t13 = ternarylogic( x4,  x1,  x0, 0x1f)
t14 = ternarylogic( x5,  y1, t13, 0x6a)
y0  = ternarylogic(t10, t12, t14, 0xc6)
t15 = ternarylogic( x4,  x1,  y1, 0xb9)
t16 = ternarylogic( x2,  y0, t15, 0xe6)
t17 = ternarylogic( t0,  t1,  t2, 0x6b)
t18 = ternarylogic( t8, t12, t17, 0x24)
y3  = ternarylogic( y2, t16, t18, 0xb4)
```

**S6, 22 TI**

```
t0  = ternarylogic( x5,  x4,  x0, 0x36)
t1  = ternarylogic( x3,  x2,  t0, 0x6c)
t2  = ternarylogic( x1,  x0,  t0, 0x1e)
t3  = ternarylogic( x5,  x3,  t2, 0x69)
t4  = ternarylogic( x5,  x0,  t3, 0x3a)
t5  = ternarylogic( t0,  t2,  t4, 0xd2)
y3  = ternarylogic( t1,  t3,  t5, 0xc6)
t6  = ternarylogic( x4,  t1,  t4, 0x96)
t7  = ternarylogic( x3,  t3,  t5, 0xba)
t8  = ternarylogic( x5,  x3,  t6, 0x09)
t9  = ternarylogic( t0,  y3,  t8, 0x9a)
y2  = ternarylogic( t6,  t7,  t9, 0xb4)
t10 = ternarylogic( x5,  x4,  t3, 0x4a)
t11 = ternarylogic( x1,  t1, t10, 0x9a)
t12 = ternarylogic( x1,  t3,  t4, 0x24)
t13 = ternarylogic( x4,  t1, t12, 0xa9)
y0  = ternarylogic( t7, t11, t13, 0xe1)
t14 = ternarylogic( x5,  x4,  y3, 0x51)
t15 = ternarylogic( x1,  y2, t14, 0x69)
t16 = ternarylogic( t3,  t4,  y2, 0x6b)
t17 = ternarylogic( t2, t10, t16, 0x96)
y1  = ternarylogic( y0, t15, t17, 0x3a)
```

**S7, 23 TI**

```
t0  = ternarylogic( x4,  x3,  x1, 0x59)
t1  = ternarylogic( x5,  x4,  x3, 0x49)
t2  = ternarylogic( x2,  x0,  t1, 0x96)
t3  = ternarylogic( x1,  x0,  t2, 0xb8)
t4  = ternarylogic( x5,  x2,  t3, 0x9c)
y3  = ternarylogic( t0,  t2,  t4, 0x36)
t5  = ternarylogic( x0,  t0,  t4, 0x65)
t6  = ternarylogic( x4,  x1,  x0, 0x74)
t7  = ternarylogic( x5,  t0,  t1, 0x49)
t8  = ternarylogic( x5,  t2,  t5, 0xc2)
t9  = ternarylogic( x5,  t7,  t8, 0x61)
y0  = ternarylogic( t5,  t6,  t9, 0x47)
t10 = ternarylogic( x4,  t0,  t2, 0x69)
t11 = ternarylogic( x3,  x0,  t7, 0xab)
t12 = ternarylogic( x3,  t1,  t3, 0x65)
t13 = ternarylogic( x4,  x0,  t5, 0x25)
t14 = ternarylogic( x5, t12, t13, 0xc6)
y2  = ternarylogic(t10, t11, t14, 0x74)
t15 = ternarylogic( y3,  t5, t11, 0x96)
t16 = ternarylogic( x3,  x1, t12, 0xb9)
t17 = ternarylogic( x5,  x3, t16, 0x8a)
t18 = ternarylogic( x0,  t1,  t4, 0x0e)
y1  = ternarylogic(t15, t17, t18, 0x4b)
```

**S8, 23 TI**

```
t0  = ternarylogic( x4,  x3,  x2, 0x56)
t1  = ternarylogic( x4,  x1,  t0, 0x1b)
t2  = ternarylogic( x5,  x3,  t1, 0x96)
t3  = ternarylogic( x5,  t0,  t2, 0x06)
t4  = ternarylogic( x1,  t0,  t3, 0xa4)
t5  = ternarylogic( x4,  x3,  t4, 0xa6)
y1  = ternarylogic( x0,  t2,  t5, 0x6c)
t6  = ternarylogic( x3,  t5,  y1, 0xc5)
t7  = ternarylogic( x0,  t0,  t6, 0x69)
t8  = ternarylogic( x1,  x0,  t3, 0x17)
t9  = ternarylogic( x4,  t6,  t8, 0xc9)
t10 = ternarylogic( x3,  x1,  t9, 0xa6)
y3  = ternarylogic( x5,  t7, t10, 0x36)
t11 = ternarylogic( t4,  t5,  y1, 0x76)
t12 = ternarylogic( x2,  x1,  t7, 0x9a)
t13 = ternarylogic( x2,  t3,  t8, 0x24)
t14 = ternarylogic( t6, t10, t13, 0x6e)
y0  = ternarylogic(t11, t12, t14, 0xd1)
t15 = ternarylogic( x3,  t1,  y0, 0x91)
t16 = ternarylogic( x2,  t7, t15, 0x69)
t17 = ternarylogic( t3, t11, t13, 0xba)
t18 = ternarylogic(t10,  y0, t17, 0x5b)
y2  = ternarylogic(t11, t16, t18, 0x47)
```

## Висновки

У роботі представлено метод для генерації bitsliced-описів довільних великих S-Boxes ($n > 4$) зі скороченим числом тернарних логічних інструкцій. Метод поєднує в собі евристичні

техніки на різних етапах пошуку bitsliced-представлення, що сукупно забезпечують його ефективність та прийнятну швидкодію Отримані описи дають змогу загалом збільшити швидкодію програмних реалізацій багатьох криптоалгоритмів на будь-яких процесорах, що підтримують 3-операндну тернарну логічну інструкцію (CPU/GPU), а також під час апаратної реалізації на FPGA з відповідними LUT. На сьогодні запропонований у статті метод є найефективнішим із відомих нам методів за критерієм BGC, що підтверджують наведенні в роботі результати досліджень.

**Перелік посилань**

1. Biham E. A fast new DES implementation in software. *International Workshop on Fast Software Encryption*. 1997. С. 260–272. URL: https://doi.org/10.1007/BFb0052352 (дата звернення: 19.02.2025).

2. Kasper E., Schwabe P. Faster and timing-attack resistant AES-GCM. *CHES'09: Proceedings of the 11th International Workshop Cryptographic Hardware and Embedded Systems*. 2009. С. 1–17. URL: https://doi.org/10.1007/978-3-642-04138-9_1 (дата звернення: 19.02.2025).

3. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V / A. Adomnicai, T. Peyrin. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2021, №1. С. 402–425. URL: https://doi.org/10.46586/tches.v2021.i1.402-425 (дата звернення: 19.02.2025).

4. Schwabe P., Stoffelen K. All the AES you need on Cortex-M3 and M4. *SAC'2016: Proceedings of the International Conference on Selected Areas in Cryptography*. 2016. С. 180–194. URL: https://doi.org/10.1007/978-3-319-69453-5_10 (дата звернення: 19.02.2025).

5. Zhang J., Ma. M, and Wang P. Fast Implementation for SM4 Cipher Algorithm based on Bit-Slice Technology. *SmartCom'2018*: *Proceedings of the International Conference on Smart Computing and Communication*. 2018, С. 104–113. URL: https://doi.org/10.1007/978-3-030-05755-8_11 (дата звернення: 19.02.2025).

6. Nishikawa N., Amano H., and Iwai K. Implementation of bitsliced AES encryption on CUDA-enabled GPU. *NSS'2017: Proceedings of the International Conference on Network and System Security*. 2017. С. 273–287. URL: https://doi.org/10.1007/978-3-319-64701-2_20 (дата звернення: 19.02.2025).

7. Matsuda S., Moriai S. Lightweight cryptography for the cloud: exploit the power of bitslice implementation. *CHES'2012: Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. 2012. С. 408–425. URL: https://doi.org/10.1007/978-3-642-33027-8_24 (дата звернення: 19.02.2025).

8. Kwan M. Reducing the Gate Count of Bitslice DES. *IACR Cryptology ePrint Archive.* 2000(51). URL: https://eprint.iacr.org/2000/051 (дата звернення 19.02.2025).

9. Bitslice DES. URL: https://darkside.com.au/bitslice/ (дата звернення 19.02.2025).

10. Bitsliced Implementation of Non-Algebraic 8×8 Cryptographic S-Boxes Using x86-64 Processor SIMD Instructions /Y. Sovyn, V. Khoma, M. Podpora *IEEE Transactions on Information Forensics and Security*. 2023, № 18. С. 491–500. URL: https://doi.org/10.28925/2663-4023.2020.7.131152 (дата звернення 19.02.2025).

11. Stoffelen K. Optimizing S-Box Implementations for Several Criteria Using SAT Solvers. *FSE'2016: Proceedings of the 23rd International Conference on Fast Software Encryption*. 2016. С. 140-160. URL: https://doi.org/10.1007/978-3-662-52993-5_8 (дата звернення 19.02.2025).

12. Optimizing Implementations of Lightweight Building Blocks / J. Jean, T. Peyrin. *IACR Transactions on Symmetric Cryptology*. 2017, №4. С. 130–168. URL: https://doi.org/10.13154/tosc.v2017.i4.130-168 (дата звернення: 19.02.2025).

13. Peigen – a platform for evaluation, implementation, and generation of S-boxes / Z. Bao, J. Guo, S. Ling, Y. Sasaki. *IACR Transactions on Symmetric Cryptology*. 2019, №1. С. 330–394. URL: https://doi.org/10.13154/tosc.v2019.i1.330-394 (дата звернення: 19.02.2025).

14. Mercadier D. Usuba, Optimizing Bitslicing Compiler. *PhD Thesis, Sorbonne University, France*. 2020. С. 195. URL: https://theses.hal.science/tel-03133456v2 (дата звернення: 19.02.2025).

15. Sboxgates: A program for finding low gate count implementations of S-boxes / Dansarie M. *Journal of Open Source Software*. 2021. Т. 62, № 6. С. 1–3. URL: https://doi.org/10.21105/joss.02946 (дата звернення: 19.02.2025).

Надійшла 10.02.2025