Т. 1, № 1, 2019

https://doi.org/

УДК 004.056:061.68

ПОРІВНЯННЯ AEAD-АЛГОРИТМІВ ДЛЯ ВБУДОВАНИХ СИСТЕМ ІНТЕРНЕТУ РЕЧЕЙ

Я. Р. Совин, В. В. Хома, В. І. Отенко

Національний університет "Львівська політехніка",
кафедра захисту інформації

Виконано порівняння за швидкодією і вимогами до пам'яті реалізацій AEAD-шифрів AES-GCM та ChaCha20-Poly1305 для типових 8/16/32-бітних вбудованих low-end процесорів у складі пристроїв Інтернету речей за різних підходів до забезпечення стійкості до часових атак і простих атак на енергоспоживання. Особливу увагу приділено низькорівневій реалізації множення в полях GF(2^{128}) із константним часом виконання як ключовій операції GCM, оскільки у low-end процесорів немає готової інструкції для carry-less множення. Для кожного процесорного ядра AVR/MSP430/ARM Cortex-M3 відповідно запропонована реалізація carry-less множення з константним часом виконання, яка за ефективністю близька до алгоритмів із неконстантним часом виконання.

Ключові слова: AEAD, AES-GCM, ChaCha20-Poly1305, часові атаки, атаки через сторонні канали, IoT, поліноміальне множення, мікроконтролери.

Вступ

Ми живемо в глобальному інформаційному просторі, який забезпечує ефективну взаємодію людей, їх доступ до світових інформаційних ресурсів, продуктів і послуг. У зв'язку з цим виникає проблема захисту інформації, що передається і обробляється електронними засобами, яку вирішують за допомогою криптографічних методів.

Бурхливий розвиток Інтернету речей (Internet of Things, IoT) ще більше підкреслює важливість криптографії та ставить перед нею нові виклики. Концепція Інтернету речей передбачає, що смарт-пристрої з допомогою вбудованих сенсорів збирають та вимірюють параметри навколишнього середовища та передають їх через IoT-шлюзи на віддалений сервер у хмарі. Зрозуміло, що всі аспекти такої взаємодії повинні бути надійно захищені, особливо це важливо для об'єктів критичної інфраструктури, від яких залежить життя і здоров'я людей. IoT-пристрої побудовані на обмежених у ресурсах обчислювальних засобах, якими переважно є мікроконтролери (МК). Для мікроконтролерів, крім малої обчислювальної потужності, характерні порівняно невеликі обсяги ROM і RAM та вимоги низького енергоспоживання і ціни. Отже, криптографічні алгоритми повинні бути ефективно (з використанням мінімуму ресурсів) реалізовані на IoT-пристроях та протистояти широкому спектру атак, зокрема атак через сторонні канали. Складність одночасної оптимізації рівня безпеки, ціни та продуктивності в готовому пристрої становить основну проблему впровадження криптографії у вбудовані системи (ВС).

Історично криптоалгоритми розробляли насамперед для імплементації в інформаційних системах, побудованих на універсальних високопродуктивних мікропроцесорах, через що вони

погано адаптовані до застосування у ВС на основі 8/16/32-бітових процесорів із малими обчислювальними ресурсами. Шифрування чи хешування у разі програмної реалізації на мікроконтролерах доволі повільне та енерговитратне і потребує значних об'ємів пам'яті. Пошуком нових та адаптацією відомих алгоритмів для малоресурсних платформ займається легковагова криптографія [1, 2]. Інший підхід до вирішення проблеми складності криптоалгоритмів – застосування вбудованих криптоакселераторів [3], що в десятки і сотні разів прискорюють обчислення.

IoT-аплікації функціонують на основі стандартних чи власних протоколів, дані в яких передаються у вигляді пакетів, що містять і секретну інформацію, яка повинна бути захищена від перехоплення, модифікації та підміни, і несекретні дані. Несекретною інформацією може бути заголовок пакета (header), в який входять адреси, номера портів, версії протоколу та інші відомості, необхідні для його оброблення. Ці приєднані дані повинні бути автентифіковані і водночас залишатися відкритими, щоб мережеві пристрої могли ними оперувати.

Криптографічним примітивом, який дає змогу комплексно вирішити вказані завдання є автентифікуюче шифрування з приєднаними даними (Authenticated Encryption with Associated Data, AEAD), за якого частина повідомлення шифрується, частина залишається відкритою, і все повідомлення повністю автентифікується. AEAD-шифрування є режимом симетричного шифрування пакетизованих даних, який одночасно забезпечує і конфіденційність, і автентифікацію даних, гарантуючи їх цілісність, з використанням єдиного програмного інтерфейсу.

Автентифікуюче шифрування є ефективнішим і простішим, ніж використання окремих методів, та зазвичай потребує меншу кількість ресурсів. Також воно дає змогу уникнути критичних помилок під час поєднання шифрування та автентифікації, які стали причиною низки практичних атак на протоколи та додатки, зокрема SSL/TLS [4].

Типовий програмний інтерфейс для реалізації AEAD-режиму забезпечує такі функції:

- Зашифрування
  Вхідні дані: ключ, одноразовий код, відкритий текст і, опціонально, заголовок.
  Вихідні дані: шифртекст і тег автентифікації – рис. 1.
- Розшифрування
  Вхідні дані: ключ, одноразовий код, шифртекст, тег і, опціонально, заголовок.
  Вихідні дані: відкритий текст або помилка, якщо обчислений тег не відповідає наданому.

Оскільки для IoT-протоколів сьогодні безпека має вирішальне значення [5, 6], то існує об'єктивна потреба у AEAD-алгоритмах. Проблемою є те, що AEAD-алгоритми передбачають поєднання симетричного шифру та MAC-алгоритму, а це істотно збільшує споживання ресурсів (пам'яті, тактів, енергії) вбудованих процесорів порівняно з простим шифруванням. Водночас на цей момент майже відсутні публікації щодо способів високоефективної реалізації AEAD-шифрів у вбудованих системах.

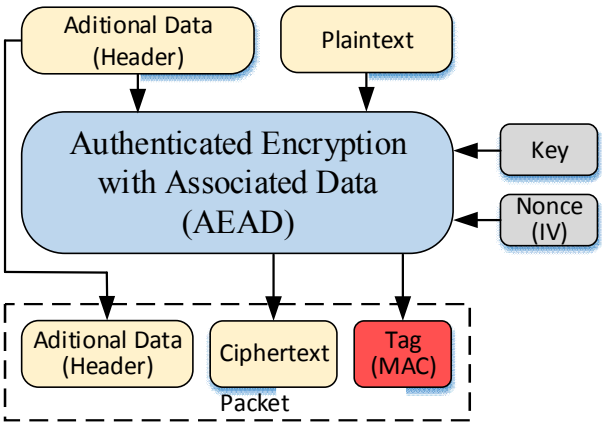


Рис. 1. Принцип функціонування автентифікуючого шифрування з приєднаними даними

Аналіз останніх досліджень і публікацій

Оскільки безпека каналів зв'язку є серйозною проблемою для Інтернету речей, то доцільним виглядає застосування стандартизованих та апробованих у комп'ютерних мережах AEAD-алгоритмів, зі складу захищених протоколів, таких як TLS 1.2 та його нової версії TLS 1.3. Протокол TLS 1.2 реалізований у багатьох бібліотеках, зокрема і легковагових, що орієнтовані на вбудовані системи та IoT, найвідоміші з яких WolfSSL, GUARD TLS Tiny, mbed TLS, CycloneSSL. Попри все протокол TLS доволі важкий для світу IoT, тому як альтернативу можна розглядати порівняно новий протокол Noise, який зменшує складність і накладні витрати TLS.

З огляду на це для дослідження обрано два найпоширеніші сьогодні алгоритми: AES-GCM та ChaCha20-Poly1305. Обидва алгоритми підтримуються у протоколах TLS 1.2/1.3 та Noise і закріплені в стандартах NIST SP 800-38D, RFC5116 [7, 8] та RFC8439, RFC7905 [9, 10] відповідно.

Потребу в AE-шифрах підтвердив оголошений в 2013 р. конкурс CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [11] для вибору портфелю придатних для широкого використання алгоритмів, які б перевершували AES-GCM за швидкодією і захищених від атак типу nonce reuse/misuse. Станом на вересень 2018 р. у фінал вийшло 7 кандидатів, проте остаточні переможці ще не названі.

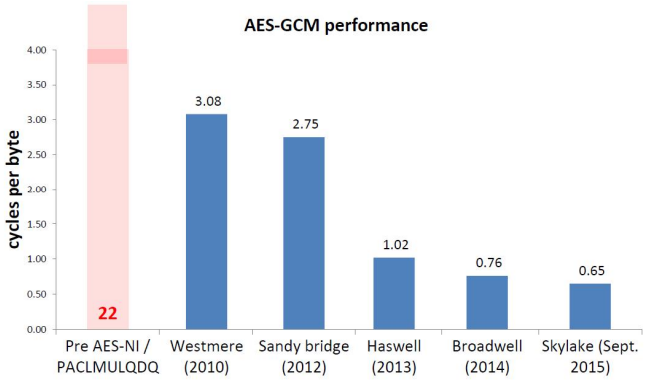
З моменту затвердження у 2007 р. стандарту AES-GCM навіть високопродуктивні мікропроцесори загального призначення (Intel, AMD) з високими тактовими частотами, великими обсягами оперативної та кеш-пам'яті і потужною системою команд зіткнулися з проблемою недостатньої продуктивності під час його реалізації. Алгоритм AES-GCM складається з двох операцій: шифрування даних AES у режимі лічильника та алгоритму GHASH на основі множення в полях Галуа з модульною редукцією, який є значно повільніший за шифрування. Тому в 2010 р. у x86-процесори додали інструкції шифрування AES-NI, а також спеціальну команду *PCLMULQDQ* (Carry-Less Multiplication) для множення в полях Галуа $GF(2^{128})$ [12, 13], які у міру свого вдосконалення та застосування різних оптимізуючих технік обчислень (особливо модульної редукції) [14] істотно збільшили продуктивність AES-GCM (рис. 2).

У процесорах з архітектурою ARMv8 для підтримки AES-GCM теж додали спеціальні чотири SIMD-інструкції, які виконують шифрування AES (*AESE/AESD* – AES single round encryption/decryption, *AESMC* – AES mix columns, *AESIMC* – AES inverse mix columns) та команду множення 64×64 у полях Галуа *PMULL* [15].

З апаратною підтримкою AES-GCM у вбудованих процесорах ситуація значно гірша. Лише деякі моделі 8/16-бітних МК мають криптоакселератори алгоритму AES-128, зокрема в 8-бітних AVR він шифрує блок за 375 тактів, а в 16-бітних MSP430 за 167 тактів. Частіше AES-криптомодулі зустрічаються в 32-бітних МК з ядром ARM Cortex-M, в яких залежно від виробника шифрування займає від 12 до 168 тактів [3]. Проте у МК відсутня апаратна підтримка carry-less множення в операції GHASH, яка за складністю в 2–4 рази перевершує AES-шифрування, а крім того частіше виконується, оскільки їй потрібно обробляти і приєднані дані, які не шифруються. Тому в статті головну увагу приділено технікам оптимізації саме операції GHASH із врахуванням особливостей архітектури процесорів та їх системи команд.

Рік	CPU	PCLMULQDQ		AESENC/AESDEC	
		Latency	Reciprocal throughput	Latency	Reciprocal throughput
2010	Intel Westmere	12	8	5	2
2011	Intel Sandy Bridge	14	8	8	4
2012	Intel Ivy Bridge	14	8	4	1
2013	Intel Haswell	7	2	7	1
2014	Intel Broadwell	5	1	7	1
2015	Intel Skylake	7	1	4	1
2011	AMD Bulldozer	12	7	5	2
2012	AMD Piledriver	12	7	5	2
2014	AMD Steamroller	11	7	5	1
2013	AMD Jaguar	3	1	5	1
2017	AMD Ryzen	4	2	4	0.5

а



б

Рис. 2. Еволюція криптографічних інструкцій в поколіннях процесорів Intel та AMD [16] (а) та її вплив на продуктивність AES-GCM [17] (б)

Що стосується програмних реалізацій алгоритму AES, то тут є добре відомі ефективні техніки обчислень, з яких найшвидкодіючою є використання так званих *T*-таблиць [18] – це наперед обчислені чотири Lock-up таблиці *T0-T3* розміром 1 Кбайт (256×32 біти), які містять 32-бітний результат виконання операцій *SubBytes(), ShiftRows()* та *MixColumns()* для заданого байта матриці стану. Шифрування AES у цьому разі зводиться до пошуку відповідного значення в таблиці та додавання за модулем 2 з раундовим ключем.

Алгоритм ChaCha20-Poly1305 є комбінацією шифру ChaCha20 та MAC-функції Poly1305, що виробляє тег арифметичним множенням ключа з акумулятором даних за модулем $2^{130} – 5$.

Шифр ChaCha20 належить до ARX-шифрів (Add, Rotate, Xor) і використовує найпростіші арифметичні та логічні операції, які входять у систему команд більшості процесорів. Завдяки цьому для досягнення високої продуктивності він не потребує спеціальної апаратної підтримки, як це є в AES. Тому для багатьох 32/64-бітних процесорів без спеціалізованих інструкцій ChaCha20-Poly1305 є швидшим за AES-GCM. Оскільки основні операції ChaCha20 реалізуються за допомогою типових інструкцій процесора, тут мало простору для оптимізації. Значно складнішою є в реалізації операція Poly1305, швидкодія якої і буде визначати загальну швидкодію ChaCha20-Poly1305.

У відкритому доступі майже відсутні оцінки реалізацій AEAD-алгоритмів для типових вбудованих процесорів, за винятком декількох публікацій, основні результати яких подано у табл. 1. У цих роботах розглянуто впливи і архітектури, і різних способів оптимізації алгоритму на продуктивність шифрування, виміряну в тактах/байтах (cycles per byte – cpb), і обсяг необхідної пам’яті.

У роботі [19] критичні функції реалізовано на асемблері, а під час підрахунку кількості тактів у кінцевий результат не введено операції породження залежних від ключа змінних. Ця робота цікава також тим, що в ній показано, що використання вбудованого криптоакселератора AES-128 дало змогу підняти швидкодію AES-GCM в 2,2 раза. Дані [20] взяті з бібліотеки криптопримітивів Cifra, орієнтованої на вбудовані системи. Відповідно її пріоритетами є простота, протидія стороннім каналам витоку інформації властивим деяким алгоритмам, помірні вимоги до розміру коду і даних, і, як наслідок, невисока швидкодія. У роботі [21] використано мову асемблера та різні прийоми оптимізації для досягнення максимальної швидкодії алгоритму ChaCha20-Poly1305.

Таблиця 1

Параметри програмних реалізацій AEAD-алгоритмів для вбудованих процесорів

AEAD	CPU	Enc/Dec, cpb	ROM/RAM, байт	Plaintext/header, байт
AES128-GCM [19]	MSP430 (16 біт)	863/862	3169/192	16/0
AES128-GCM [20]	ARM Cortex-M3 (32 біти)	2769/-	2644/812	16/16
ChaCha20-Poly1305 [21]	ARM Cortex-M4 (32 біти)	210/-	1946/332	16/16

Постановка завдання

Мета статті – продемонструвати способи ефективної реалізації найпоширеніших у IoT-аплікаціях AEAD-алгоритмів: AES-GCM та ChaCha20-Poly1305, а також оцінити і порівняти їх вимоги до ресурсів типових вбудованих процесорів.

Вбудовані 8/16/32-бітні процесори для IoT

Зважаючи, що в IoT відсутня домінуюча платформа, важливо спрогнозувати поведінку AEAD-алгоритмів у різноманітних сегментах вбудованих процесорів: low-end (8/16 біт) та high-end (32 біти). Для досліджень обрано по одній типовій 8-, 16- і 32-бітній MCU-архітектурі.

AVR-мікроконтролери (8-бітові). Як 8-бітову платформу обрано родину мікроконтролерів AVR. Цей вибір обумовлений вдалою системою команд цих МК, що орієнтована на максимальну ефективність виконання програм, написаних мовами високого рівня.

Серед особливостей AVR-ядра, важливих у контексті криптографії, варто виокремити, що пам’ять має гарвардську організацію з розділеними 8-бітовою пам’яттю даних та 16-бітовою пам’яттю програм, що збільшує продуктивність. Регістровий файл містить 32 регістри загального

призначення (РЗП) безпосередньо під'єднаних до АЛП, в якому виконуються арифметичні, логічні та бітові операції. Система команд достатньо розвинена і складається з понад 130 інструкцій, більшість з яких завдяки дворівневому конвеєру виконуються за один такт. У ядрі наявний помножувач 8×8, який виконує інструкцію *MUL* за 2 такти [22].

AVR-мікроконтролери підтримують безпосередню пряму та непряму адресації. Наявність режимів предекременту, постінкременту і зміщення за непрямої адресації дає змогу ефективно обробляти масиви даних у процесі виконання криптоалгоритму, генеруючи компактний програмний код. Для звертання до даних у Flash-пам'яті (S-Box, Look-Up таблиці) використовують непряму адресацію. Доступ до SRAM здійснюється за 2 такти, до Flash за 3 такти.

MSP430-мікроконтролери (16-бітові). Родина MSP430 завдяки наднизькому енергоспоживанню здобула популярність у IoT і особливо у безпровідних сенсорних мережах.

Компактне 16-бітове RISC-ядро MSP430 побудоване за прінстонською архітектурою та містить 16 регістрів, з яких дванадцять (*R4-R15*) є регістрами загального призначення. Регістри *R0-R3* – виконують спеціальні функції (Program Counter, Stack Pointer, Status Register, Constant Genereator). Набір команд дуже простий і представлений 27 оригінальними і 24 емульованими інструкціями, які оптимізовані для ефективного використання мовами програмування високого рівня. Всі команди 16-бітні й можуть обробляти як 8- так і 16-бітові операнди. Наявний помножувач 16×16. Підтримується сім режимів адресації. Кількість тактів на виконання інструкції залежить від формату команди і режиму адресації та може становити від 1 до 6 [23].

Завдяки однотактовим регістровим операціям та ортогональній архітектурі забезпечується компактність коду та висока продуктивність. Важливою в контексті криптографії є також така особливість процесора MSP430, як прямий обмін даними між комірками пам'яті, минаючи регістри.

ARM Cortex-M3-мікроконтролери (32-бітові). Реалізацію на 32-бітовій платформі виконано на основі процесора ARM Cortex-M3, оскільки ARM-ядра домінують на ринку 32-бітових RISC-мікроконтролерів і наразі за енергоефективністю та ціною наблизилися до 8-бітових моделей, складаючи останнім серйозну конкуренцію у їх традиційних сегментах використання.

ARM Cortex-M3 є 32-бітовим процесором на основі гарвардської архітектури з трирівневим конвеєром, який реалізує системи команд Thumb та Thumb-2. Ядро Cortex-M3 містить 16 регістрів *R0-R15*, з яких регістри *R0-R12* є регістрами загального призначення. АЛП має 32-бітовий блок зсуву, який дає змогу одночасно з виконанням операції здійснювати зсув одного з операндів на задану кількість розрядів. Наявний однотактний помножувач 32×32 [24].

У табл. 2 зібрано основні властивості вбудованих процесорів у контексті криптообчислень.

Таблиця 2

Обчислювальні властивості вбудованих процесорів

CPU	Розрядність, біт	РЗП	Арифметичні й логічні операції (тактів на операцію)	Помножувач
AVR	8	32	AND (1), OR (1), XOR (1), NOT (1), >> 1 (1), 1 << (1), >>>1 (1), <<< 1 (1)	$8\times8\rightarrow16$ (ядро)
MSP430	16	12	AND (1), OR (1), XOR (1), NOT (1), >> 1 (1), 1 << (1), >>>1 (1), <<< 1 (1)	$16\times16\rightarrow32$ (модуль)
ARM Cortex-M3	32	13	AND (1), OR (1), XOR (1), NOT (1) >> 1…32 (1), 1…31 << (1), >>>1…31 (1)	$32\times32\rightarrow64$ (ядро)

AES-GCM

GCM (Galois/Counter Mode) – найпопулярніша схема автентифікованого шифрування, що стандартизована NIST і використовується в протоколах TLS, Noise, IPSec, SSH та ін. Популярність GCM забезпечує відсутність патентів, апаратну підтримку в сучасних мікропроцесорах та можливість конвеєризації і розпаралелювання обчислень. GCM передбачає використання 128-бітного блокового шифру *CIPH*, яким переважно є AES (*CIPH* = AES). Надалі під AES-GCM матимемо на увазі найекономнішний для IoT варіант AES-128 GCM.

Функція автентифікованого шифрування GCM-AE$_K$ (*IV, A, P*) = (*C, T*) шифрує конфіденційні дані та обчислює тег автентифікації для конфіденційних та будь-яких додаткових, неконфіденційних даних. Вхідними даними для АЕ-шифрування є: ключ *K*; відкритий текст *P;* приєднані автентифіковані дані (Additional Authenticated Data, AAD) *A*; вектор ініціалізації *IV*. Вихідними даними є: зашифрований текст *C* і тег автентифікації *T*. Довжина тегу може бути 128/120/112/104/96 бітів. У роботі обрано найпоширеніший варіант 128-бітного тегу. Функція автентифікованого розшифрування GCM-AD$_K$(*IV, A, C, T*) = (*P* або *FAIL*) розшифровує конфіденційні дані, залежно від результату перевірки тегу. GCM-AD на вхід приймає *K, IV, A, C* і *T*, а її виходом є відкритий текст *P*, якщо прийнятий тег *T* відповідає обчисленому *T'*, або спеціальний код помилки *FAIL* у протилежному випадку.

AES-GCM складається з двох частин: AES у режимі лічильника (AES CTR) для шифрування та універсальної функції хешування GHASH (Galois Hash) для обчислення автентифікаційного тегу.

AES CTR. GCM використовує AES у режимі лічильника, для забезпечення конфіденційності відкритого тексту *P*, шифруванням за допомогою секретного ключа *K* початкового значення лічильника AES$_K$(*CTR*), утвореного з вектора ініціалізації *IV*: $Y = CTR = IV||0^{31}1$. Ключовий потік на виході функції шифрування використовується в операції XOR із відкритим текстом для отримання шифртексту *C* (рис. 3). Перший блок ключового потоку Y_0 зарезервований для шифрування виходу GHASH. Для кожного наступного блоку значення лічильника збільшується на 1 функцією інкременту $incr_{32}$. Вектор ініціалізації *IV* може бути довільної довжини, проте рекомендують довжину 96 бітів (прийнято в роботі), з міркувань простоти та сумісності, інакше над ним виконується функція GHASH$_H$({}, *IV*). Враховуючи, що AES CTR є потоковим шифром, то вимога унікальності *IV* є критично важливою для безпеки.

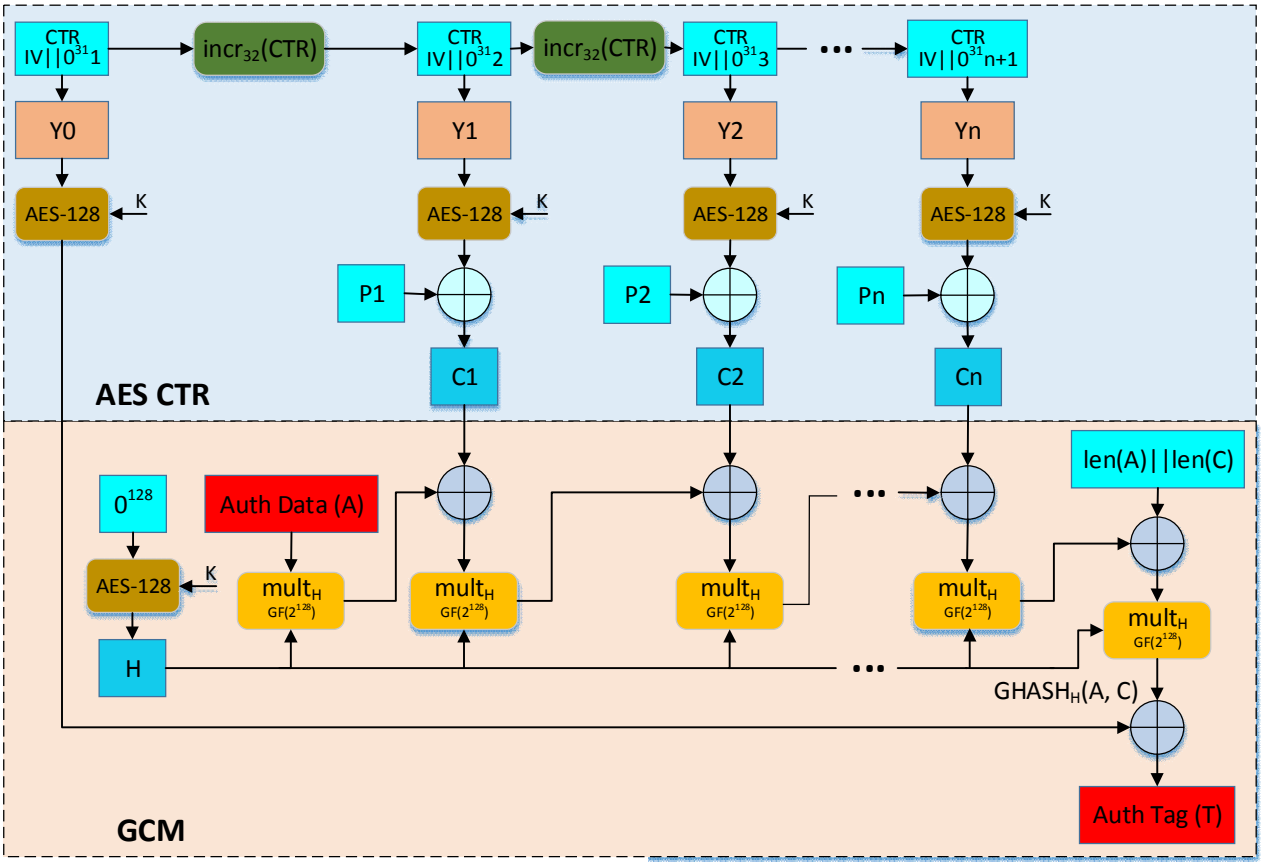


Рис. 3. GCM-AE$_K$ (IV, A, P) = (C, T)

GHASH. GHASH стискає AAD і шифртекст *C* у єдиний блок, який потім зашифровується для створення тегу автентифікації. GHASH використовує обчислення в $GF(2^{128})$ за модулем незвідного

поліному $g = x^{128}+x^7+x^2+x^1+1$ для вироблення HMAC. Дані, які потрібно автентифікувати (AAD і шифртекст) хешуються блоками по 128 бітів функцією *mult*, яка множить 128-бітний блок даних D на 128-бітний підключ хешування H за модулем g. Функція *mult* передбачає два кроки:

1) поліноміальне carry-less множення (128 бітів $\otimes$ 128 бітів $\rightarrow$ 256 бітів): $Z = D \otimes H$;

2) редукція: 256 бітів $\rightarrow$ 128 бітів $mod\ x^{128}+x^7+x^2+x^1+1$.

Останні блоки AAD і шифртексту за необхідності доповнюють нулями до 16 байт.

Підключ хешу H виробляють AES шифруванням 128-бітного блока нулів ключем K: $H = \mathrm{AES_K}(0^{128})$. На вихід функції $\mathrm{GHASH_H}(A, C, \mathrm{len}(A)_{64} \,||\, \mathrm{len}(C)_{64})$ накладають зашифрований блок Y_0 і формують тег: $T = \mathrm{GHASH_H} \oplus \mathrm{AES_K}(Y_0)$.

ChaCha20-Poly1305

Завдяки своїй простоті й швидкодії алгоритм ChaCha20-Poly1305 став дуже популярним останніми роками. Він складається з двох частин: шифрування ChaCha20 і вироблення автентифікатора Poly1305.

ChaCha20. ChaCha20 – потоковий шифр із високою швидкодією у разі програмної реалізації. Він приблизно втричі швидший за AES на платформах, які не мають апаратних криптоприскорювачів AES.

Потоковий шифр ChaCha20 генерує 512 бітів ключового потоку з 256-бітного ключа (K), 96-бітного вектора ініціалізації (IV), 32-бітного лічильника блоків (CTR) та 128-бітної константи: $\mathrm{ChaCha20}_K(IV, CTR, P) = (C)$.

ChaCha20 належить до ARX-шифрів і використовує такі операції:

– арифметичне додавання 32-бітних чисел за модулем 2^{32} (Add);

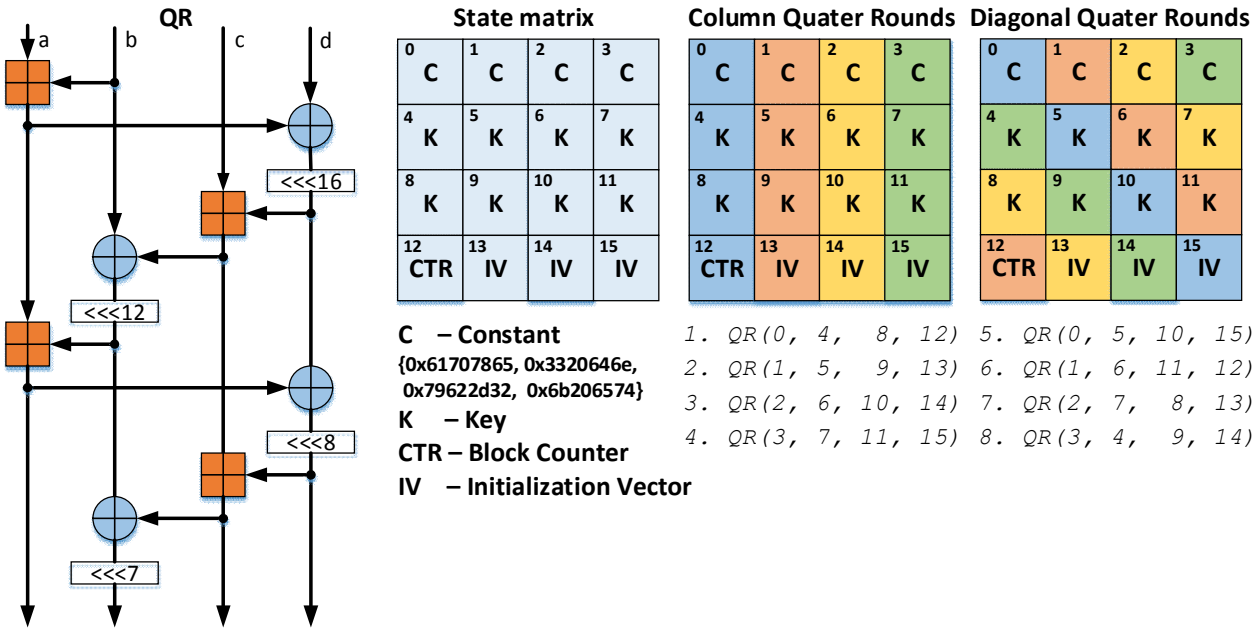
– циклічний зсув 32-бітних чисел вліво на задане число позицій u (<<< u, Rotate);

– побітове додавання за модулем 2 (Xor).

Базовою операцією алгоритму ChaCha20 є *Quater Round* (надалі *QR*). Вона оперує чотирма 32-бітними числами *a, b, c* та *d*, як показано на рис. 4, *а*.

Вхідні дані для шифру ChaCha20 подано у вигляді матриці стану розміром 4×4, яка складається з 32-бітних чисел (512 біт), принцип формування якої показано на рис. 4, *б*.

Кожен із 20 раундів шифрування складається з 4-х операцій *QR*, які обробляють матрицю стану спочатку за стовпцями, а потім за діагоналями – рис. 4, *в*.



а *б* *в*

Рис. 4. Операція QR(a, b, c, d) (а), подання матриці стану (б) та її оброблення функцією QR (в)

Після 20 раунду початкове значення матриці стану арифметично додають до отриманого результату. Після чого матриця байт за байтом (serialize) накладається операцією XOR на відкритий текст для отримання шифртексту розміром до 64 байтів (рис. 5). Якщо розмір відкритого тексту більший за 64 байти, потрібно послідовно викликати функцію шифрування з тим самим ключем і вектором ініціалізації, кожен раз збільшуючи значення лічильника блоків *CTR* на 1.

Розшифрування є повністю аналогічне шифруванню.

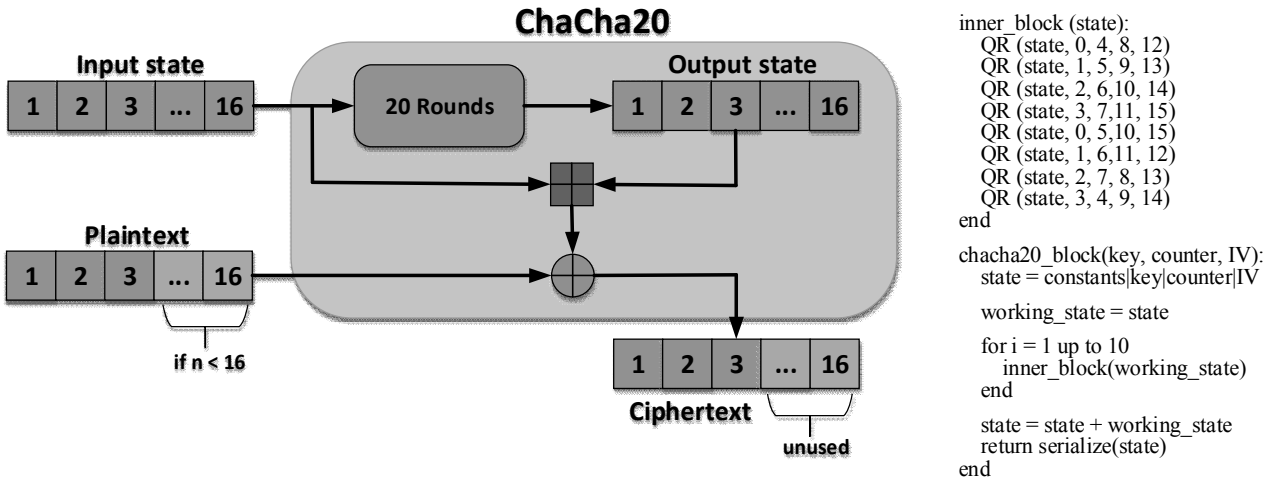


Рис. 5. Принцип роботи шифру ChaCha20

Poly1305. Poly1305 – це одноразовий автентифікатор, який приймає 32-байтовий одноразовий ключ (*r*, *s*) і повідомлення *M* та видає 16-байтовий тег автентифікації повідомлення *T*: Poly1305(*r, s, M*) = (*T)*. Одноразовий ключ автентифікації розділений на дві частини по 16 байт, які позначаються *r* та *s*, а для вироблення такої пари ключів використовується функція ChaCha20 з початковим значенням лічильника блоків *CTR*, що дорівнює 0: $\text{ChaCha20}_K(IV, 0, 0) = (r, s)$. Перші 128 бітів згенерованої гами утворюють *r*, наступні 128 бітів – *s* (рис. 6).

Перед початком роботи алгоритму потрібно обробити *r* функцією *Clamp*, яка скидає певні біти: *r* &= 0x0ffffffc0ffffffc0ffffffc0fffffff. Це допомагає спростити подальші обчислення за незначної втрати стійкості.

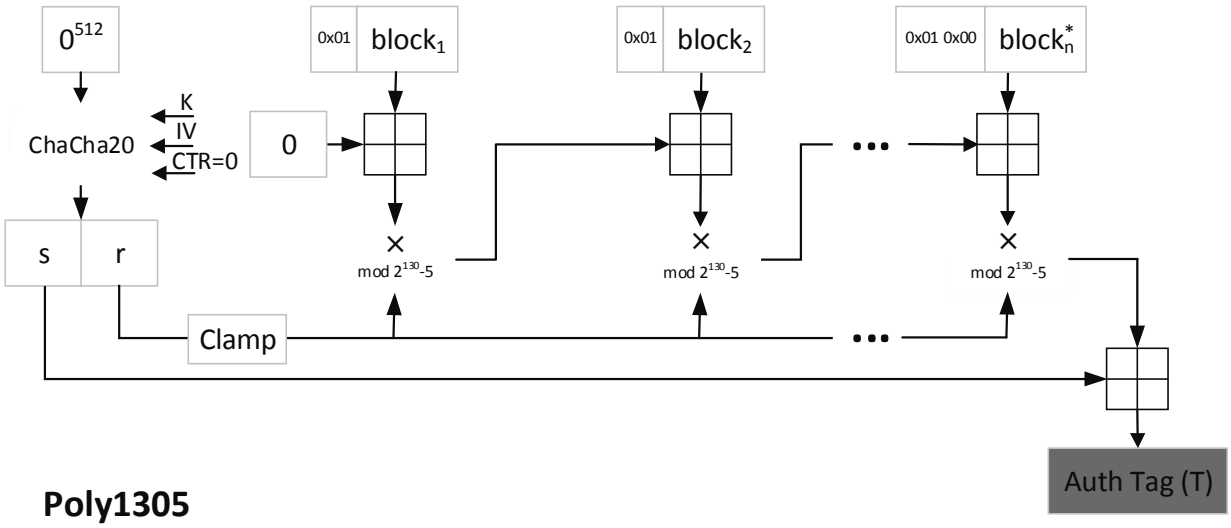


Рис. 6. Обчислення MAC в алгоритмі Poly1305

У Poly1305 використовують поліноміальне множення за модулем простого числа $p = 2^{130} - 5$. Змінну акумулятора *acc* на початку встановлюють рівною нулю (*acc* = 0). Повідомлення *M* розбивається на 16-байтові блоки $block_i$, над якими здійснюють такі операції:

– кожен блок $block_i$ доповнюють байтом 0x01 та трактують як число;

– якщо останній блок $block_n^*$ не кратний 17 байтам, він доповнюється від 0 до 16 байт;

– акумулятор містить результат множення за модулем p: $acc = (acc + block) \times r \bmod p$.

Наприкінці значення секретного ключа s додають до акумулятора за модулем 2^{128} і 128 молодших бітів формують тег: $T = acc + s \bmod 2^{128}$.

Криптопримітиви ChaCha20 і Poly1305 комбінують у AEAD-схему, яка приймає відкритий текст довільної довжини P і приєднані дані (Authenticated Data) AD довільної довжини так (рис. 7):

• генерується одноразовий ключ для Poly1305 із ключа K та вектора ініціалізації IV за допомогою функції ChaCha20: $\text{ChaCha20}_K(IV, 0, 0) = (r, s)$;

• викликається функція шифрування ChaCha20 для відкритого тексту P із використанням K і IV, з початковим значенням лічильника блоків CTR, що дорівнює 1: $\text{ChaCha20}_K(IV, 1, P) = (C)$;

• викликається функція Poly1305 з ключем (r, s), згенерованим вище, і повідомленням M, сформованим так:

– приєднані дані AD доповнено за потреби нулями до розміру кратного 16 байтам;

– шифртекст C доповнений за потреби нулями до розміру кратного 16 байтам;

– *len1* – довжина AD у байтах (64-бітне число у форматі little-endian);

– *len2* – довжина C у байтах (64-бітне число у форматі little-endian).

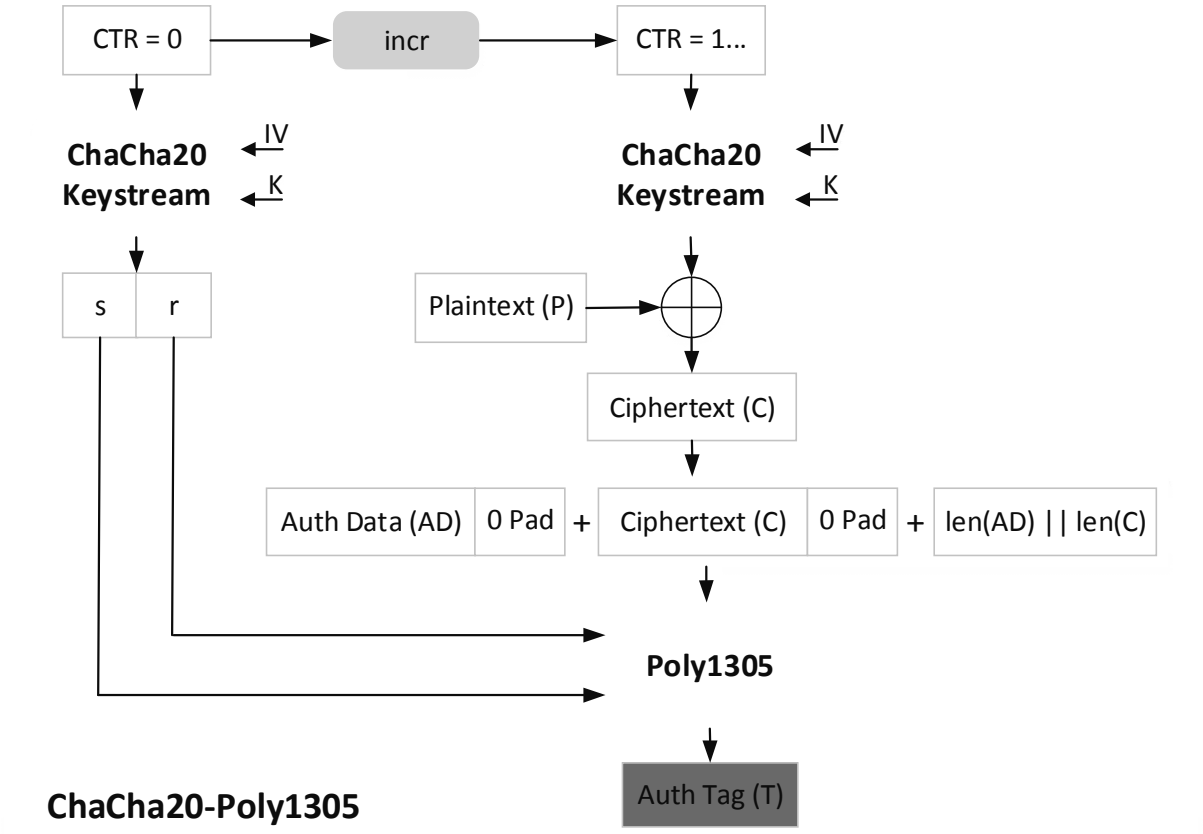


Рис. 7. AEAD-алгоритм ChaCha20-Poly1305

Отримуємо тег автентифікації: $\text{Poly1305}(r, s, AD||\text{pad}_{16}(0)\ C||\text{pad}_{16}(0)\ \text{len}_8(AD)||\text{len}_8(C)) = (T)$.

Виходом є два значення: зашифрований текст C і 128-бітний MAC-тег T.

Розшифрування відбувається аналогічно.

Особливості реалізації

Для кожної обраної родини мікроконтролерів AVR (8 бітів), MSP430 (16 бітів) та ARM Cortex-M3 (32 біти) AE-алгоритми реалізовано мовою *C* у інтегрованих середовищах розробки IAR

Embedded Workbench for AVR (v7.10.5), IAR Embedded Workbench for MSP430 (v7.11.2) та IAR Embedded Workbench for ARM (v8.22.1) відповідно. У цих середовищах також здійснено оцінювання кількості тактів і розміру коду.

Параметрами, які вимірювали, були: швидкодія зашифрування/розшифрування, виражена в тактах/байтах (cpb), обсяг постійної пам'яті (ROM), який складається з розміру програми та таблиць, розташованих у Flash-пам'яті, та обсяг оперативної пам'яті (RAM), представлений таблицями в SRAM та потрібним розміром стека.

Імплементація AES

Для всіх типів вбудованих процесорів використано AES-реалізацію на базі *T*-таблиць, як таку, що має найвищу швидкодію за прийнятних затратах пам'яті. Це потребує 4 таблиці *T0*-*T3* по 1 Кбайту (256×32 бітів), де кожна 1024-байтна Lookup-таблиця містить результат операцій *SubBytes()*, *ShiftRows()* та *MixColumns()*.

Враховуючи, що у разі 8-бітних AVR мікроконтролерів зчитування з пам'яті відбувається побайтно, розмір таблиць та їх кількість можна оптимізувати. Для цього використовують таку їх властивість, що кожне 32-бітне значення в таблиці складається з двох однакових байт, які є лінійною комбінацією інших двох байт. Наприклад, *T0*[0] = *0xc6**6363**a5*, де *0x63=0xc6* ⊕ *0xa5*. Отже, достатньо зберігати два різних байти замість чотирьох і обчислювати необхідні байти під час роботи, що зменшує розмір таблиці до 512 байтів і дає виграш у тактах. Крім того, в таблицях *T0*-*T3* елементи з однаковими індексами містять однакові байти, але в різному порядку (*T0*[0] = =*0xc66363a5*, *T1*[0] = *0xa5c66363* і т. д.). Це теж враховується програмно під час побайтового зчитування і тоді достатньо зберігати одну таблицю *T0*. Отже, розмір *T*-таблиць для випадку AVR-MCU становить 512 байт.

Для 16- і 32-бітних процесорів зчитування 32-бітного значення дає більшу швидкодію, ніж побайтові операції, тому використано традиційний підхід на основі чотирьох 1024-байтних таблиць.

Імплементація GCM

Найкритичнішою операцією GCM з огляду на продуктивність є carry-less множення двох 128-бітних операндів в операції GHASH, оскільки вбудовані процесори не підтримують інструкції множення в бінарних полях $GF(2^{128})$.

Для імплементації GHASH можна використати декілька підходів, які відрізняються і швидкодією, і безпекою. До того ж для програмних реалізацій криптографічних алгоритмів важливо, щоб вони надавали певний рівень захисту від атак через сторонні канали (Side-channel attacks). Що стосується GHASH, то основний рівень захисту передбачає стійкість до атак за часом виконання (timing attacks) і простих атак на енергоспоживання (Simple Power Analysis), що досягається відсутністю в програмі циклів, операцій та умовних переходів, час виконання яких залежить від значень секретних даних. Такі реалізації виконують за константний час (constant-time, CT) незалежно від вхідних даних, що проте зменшує швидкодію. Вимога CT є практично обов'язковою для криптографічних бібліотек, проте за необхідності може бути відключена перед компіляцією. Кеш-атаки неактуальні для більшості ВС, тому ними тут знехтувано.

У роботі залежно від реалізації carry-less множення використано дві різні версії GHASH: повільнішу GHASH-CT і швидшу GHASH-NCT, яка виконується за неконстантний час.

Для GHASH-NCT використано метод безпосереднього множення Shift-XOR або так званий шкільний метод (Schoolbook) [25]. У ньому під час множення $c = a \otimes b$ виділяється кожен i-й біт b, і якщо він дорівнює 1, відбувається XOR акумулятора c зі зсунутим на i-біт значенням a ($a << i$). Перевагою цього методу є простота та можливість одночасно з множенням здійснювати модульну редукцію для незвідного поліному g. Як бачимо з рядків 3–4 Алгоритму 1, час виконання множення залежить від значень бітів H_i одного з операндів.

Алгоритм 1 обчислює значення $Z = D \otimes H \; mod \; g$, де D, H та $Z \in GF(2^{128})$, R = 0xe1.

1 $Z \leftarrow 0$, $V \leftarrow D$
2 **for** i = 0 to 127 **do**
3 **if** H_i = 1 **then**
4 $Z \leftarrow Z \oplus V$
5 **end if**
6 **if** V_{127} = 0, **then**
7 $V \leftarrow$ rightshift(V)
8 **else**
9 $V \leftarrow$ rightshift(V) $\oplus R$
10 **end if**
11 **end for**
12 **return** Z

Можливо забезпечити виконання алгоритму Shift-XOR за константний час заміною залежних від даних умовних операторів на кроках 3 і 6 безумовними операціями з використанням масок. Цей варіант алгоритму позначений GHASH-CTM і дає змогу оцінити втрати в продуктивності порівняно з GHASH-NCT. Наступний фрагмент демонструє основну ідею досягнення константного часу виконання на основі масок залежно від значення нульового біта a:

NCT

```
if (a & 0x01)
{b = b ^ c;}
```

CTM

```
mask = 0x00 - (a & 0x01);
b    = b ^ (c & mask);
```

Типові бібліотечні програмні імплементації множення використовують той факт, що один із множників (H) є константою і це дає змогу будувати Lookup-таблиці різного розміру залежно від бажаної швидкодії [25]. Основна ідея всіх цих алгоритмів полягає у розбитті неконстантного множника D на частини s (переважно 8 або 4 біти) та використанні їх як індексів для таблиць, що зберігають наперед обчислені часткові добутки $s \otimes H$. Lookup-таблиці як і сам ключ хешування (H) генеруються під час офлайн фази. В онлайн фазі обчислення $D \otimes H$ заміняється пошуком у таблицях та XOR зчитаних елементів для формування результату.

Найшвидший варіант такої реалізації передбачає використання 64 Кбайтів для зберігання 16 таблиць T_i (для кожного байта D), де кожна таблиця містить 256 елементів $T_i[j]$ (для кожного можливого значення байта D) по 128-бітів кожен, який дорівнює:

$$T_i[j] = (Hash\ Key \otimes (j << 2^{8i})) \; mod \; g, \text{ для } j = 0, 1, \ldots, 255 \text{ та } i = 0, 1, \ldots, 15.$$

Множення в такому варіанті потребує 16 зчитувань 128-бітних значень із таблиць і 16 128-бітних XOR операцій.

Можна використовувати таблиці меншого розміру (8/4 Кбайт), але тоді істотно знижується ефективність їх використання, оскільки зростає кількість операцій обчислення адреси і читань із пам'яті, накладні витрати від яких перекривають досягнутий виграш в обчисленнях.

У роботі не використано табличного підходу з огляду на декілька міркувань:

1. Він потребує багато високодефіцитної RAM-пам'яті, оскільки таблиці обчислюють у процесі роботи динамічно. Такі обсяги RAM як 64 чи 8 Кбайтів є критичними для вбудованих процесорів, особливо 8/16-бітних.

2. Обчислення таблиці займає багато часу, що збільшує час реакції у разі зміни ключа.

3. З огляду на п.1-2 для IoT-аплікацій, де потрібно використовувати декілька ключів або ж ключі часто міняються, такий підхід є неможливий або недоцільний.

У роботі для GHASH-CT залежно від архітектури і розрядності процесора використано три підходи, які забезпечують множення 8×8, 16×16 і 32×32. Далі на основі цих операцій ієрархічно з використанням алгоритму Карацуби (Karatsuba) формується результат множення $128 \times 128 \rightarrow 256$, що дає змогу зменшити кількість множень на кожному рівні ієрархії з чотирьох до трьох.

Розглянемо, які ефективні техніки запропоновано для CT-множення в полях Галуа.

AVR. Для реалізації базової функції $c = a \otimes b$ $(8\times8\rightarrow16)$ використано вбудований апаратний двотактний помножувач (команда *MUL*). Основна ідея: розбити множники на менші частини так, щоб під час операції множення переноси не впливали на інші біти результату. Для цього спочатку виконується СТ-множення a на два молодші біти b_1b_0:

$$c = ((0x00 - (b \;\&\; 0x01)) \;\&\; a) \;\hat{}\; (a * (b \;\&\; 0x02)).$$

Вираз 0x00 – (b & 0x01) формує значення-маску 0x00, якщо $b_0 = 0$ та 0xff, якщо $b_0 = 1$. Відповідно отримаємо $c = a \;\hat{}\; (a*2b_1)$, якщо $b_0 = 1$ та $c = a*2b_1$ при $b_0 = 0$. Надалі біти b_1b_0 не впливатимуть на результат і будуть маскуватися під час обчислень.

Далі виконують чотири множення, в кожному з яких максимальна кількість елементів в стовпці дорівнює 3, а відстань між ними становить 1 розряд, що гарантує відсутність спотворень внаслідок переносу (рис. 8). Арифметична сума елементів у стовпці без врахування переносу дорівнює їх сумі за модулем 2, а перенос ніколи не вплине на сусідні ненульові елементи. За допомогою відповідних масок виділяють потрібні біти та об'єднують у кінцевий результат.

Як показали експерименти, цей метод швидший за Shift-XOR і забезпечує множення 8×8 за 41 такт.

MPS430. Хоча подані мікроконтролери мають апаратний помножувач $16\times16\rightarrow32$, проте він не належить до ядра процесора, а функціонує як периферійний модуль. Відповідно для доступу до його вхідних і вихідних даних та управління режимом роботи є спеціальні регістри, які повинні налаштовуватися з основної програми. Це потребує значної кількості тактів і робить його використання для множення нерентабельним.

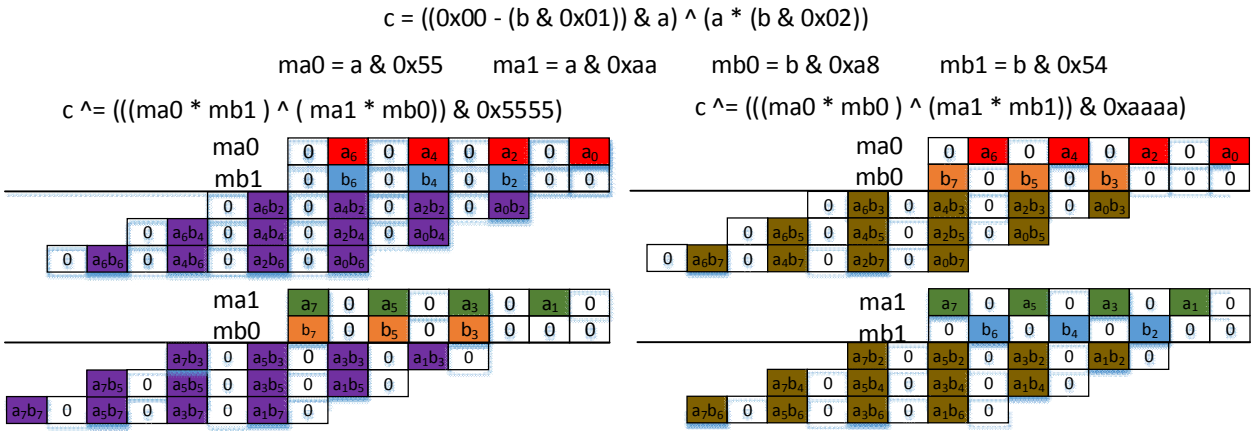


Рис. 8. СТ-множення 8×8

Тому для СТ-множення використано вже розглянутий підхід. А саме, побітно сканується значення a_i, з допомогою виразу ($mask$ & 0x01) = $0b0...00a_i$. Вираз $0x00$ – ($mask$ & $0x01$) набуває значення 0x0000 або 0xffff, яке слугує маскою для множника b. Отриманий частковий добуток накладається на результат із відповідним зміщенням. Високорівневий С-код транслюється компілятором у компактні детерміновані асемблерні інструкції, як показано на рис. 9. Функція множення 16×16 займає 180 тактів.

ARM Cortex-M3. Ядро Cortex-M3 має однотактний помножувач $32\times32\rightarrow64$, проте ефективнішим виявився підхід на основі методу Shift-XOR, враховуючи, що можна поєднувати зсув одного з операндів на задану кількість бітів та операцію XOR в одній інструкції. Хоча метод Shift-XOR мовою *C* містить умовні оператори і, отже, виглядає ніби виконується не за константний час, але його машинний код, згенерований компілятором, є СТ-кодом. Це досягається завдяки використанню компілятором команди *IT* (IF-THEN), яка дає змогу умовно виконувати невеликий фрагмент коду (до 4 інструкцій). У цьому разі не відбувається втрата продуктивності за рахунок

розривів у роботі конвеєра, оскільки порядок виконання програми не змінюється. Залежно від умови, інструкції, що йдуть після *IT*, виконуються або ні (не змінюють стан регістрів та прапорців), проте в будь-якому разі вони проходять через конвеєр, а отже займають однаковий час під час оброблення. Отже, забезпечується константний час виконання фрагмента коду. На рис. 10 показано С-код та його асемблерне подання, які засвідчують відсутність умовних переходів.

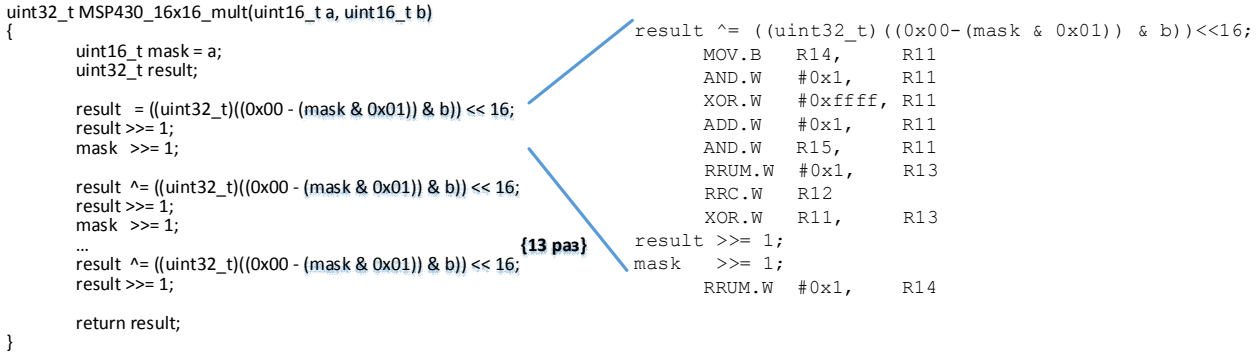


Рис. 9. СТ-множення 16×16

Функція множення 32×32 займає 140 тактів.

Для здійснення модульної редукції використовують Алгоритм 4, запропонований в [14].

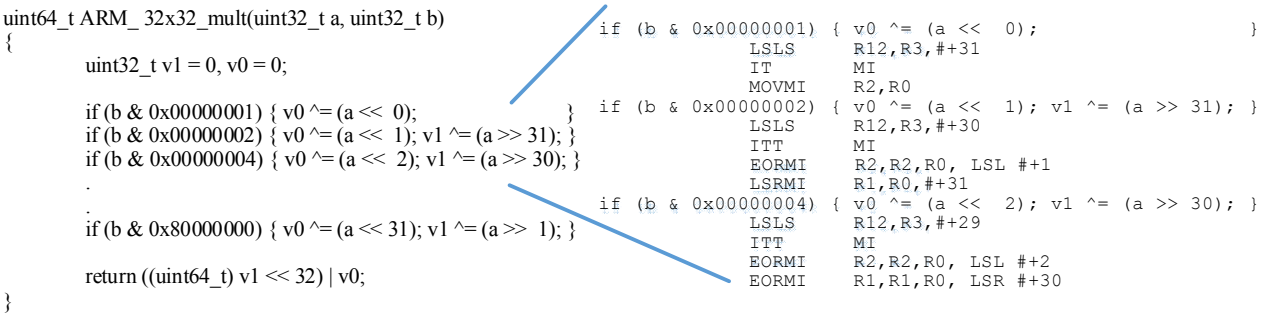


Рис. 10. СТ-множення 32×32

Імплементація ChaCha20

Проста структура шифру ChaCha20 та базової операції *QR* дає змогу *C*-компіляторам генерувати код близький за ефективністю до асемблерного, з константним часом виконання. Тому оптимізація для 8/16/32-бітних вбудованих процесорів полягала в максимальному використанні РЗП для зберігання внутрішнього стану шифру (*S*[0-15] = 16×4 = 64 байти) та зменшення кількості операцій звертання до пам'яті.

Наявних в AVR, MSP430 та ARM мікроконтролерах РЗП (32×8, 12×16 і 13×32 біти відповідно) недостатньо для подання всієї матриці стану, тому їх використовують для зберігання змінних *a, b, c, d* у межах операції *QR* та декількох елементів матриці стану. Експерименти показали, що для AVR у РЗП компілятор може розмістити одне 32-бітне слово матриці стану, для MSP430 – жодне, ARM Cortex-M3 – 4 слова.

Зменшення кількості читань/записів пам'яті використовує ту властивість, що у разі переходу від Column-раундів до Diagonal-раундів та навпаки одна зі змінних матриці стану (*S*[15] та *S*[4] відповідно) є спільна для обох раундів, що дає змогу не зберігати її в пам'яті, а потім зчитувати щоразу, а безпосередньо використовувати в наступній операції *QR*(*a, b, c, d*) у вигляді змінної *d* або *b* з попередньої операції (рис. 11).

Загалом шифрування одного 64-байтного блоку займає 11798, 10417 та 1640 тактів для AVR, MSP430 та ARM мікроконтролерів відповідно.

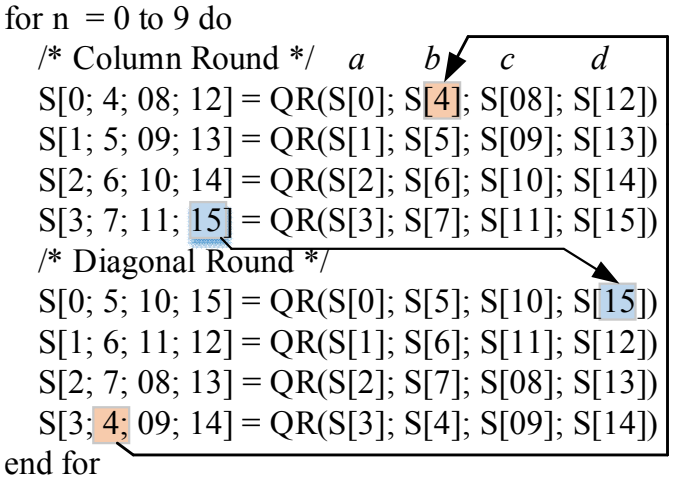


Рис. 11. Перекриття змінних у Column та Diagonal раундах

Імплементація Poly1305

Для обчислення MAC використано підходи, описані в [26], та їх реалізації, оптимізовані для різних розрядностей операції множення (8/16/32/64) з [27]. Встановлено, що у випадку AVR-MCU найкращі результати дає версія з множенням 16×16, для MSP430- і ARM-MCU – з множенням 32×32.

Оцінка та порівняння продуктивності

Вимірювані параметри реалізацій AE-шифрів зібрано в табл. 3.

Як бачимо з табл. 3, залежно від розміру повідомлення та розрядності процесора спостерігаються певні тенденції. Зокрема шифр ChaCha20-Poly1305 на великих розмірах пакетів є майже вдвічі та втричі ефективніший за AES-GCM для 16- і 32-бітних MCU відповідно. Водночас за невеликих розмірів повідомлення (до 64 байтів) ChaCha20-Poly1305 приблизно вдвічі швидший на 32-бітних процесорах і незначно відрізняється від AES-GCM на 16-бітних. Що стосується 8-бітних AVR-процесорів, то тут ситуація інакша і AES-GCM випереджає ChaCha20-Poly1305 для всіх розмірів пакетів, хоча у міру зростання розміру пакета різниця стає менш помітною.

Таблиця 3

AEAD-шифрування за довжини приєднаних даних 16 байт (*AAD* = 16 байт)

Алгоритм	Швидкодія зашифрування/розшифрування повідомлення довжиною *mlen*, тактів/байтів									ROM, байт	RAM, байт
	mlen, байтів										
	8	16	32	64	128	256	512	1024	2048		
1	2	3	4	5	6	7	8	9	10	11	12
CPU AVR (8-bit)											
AES-GCM NCT	4322/ 4345	2100/ 2107	1411/ 1415	1073/ 1075	886/ 887	802/ 803	758/ 759	733/ 733	724/ 724	9844	1083
AES-GCM CTM	9046/ 9069	4529/ 4537	3003/ 3007	2240/ 2242	1859/ 1860	1668/ 1669	1573/ 1573	1525/ 1525	1501/ 1501	9642	1092
AES-GCM CT	4358/ 4381	2185/ 2193	1440/ 1444	1067/ 1069	880/ 881	787/ 788	741/ 741	717/ 717	706/ 706	11012	1514
ChaCha20-Poly1305	7156/ 7181	3580/ 3590	2119/ 2124	1351/ 1354	1080/ 1082	945/ 946	877/ 878	844/ 844	827/ 827	6164	396
CPU MSP430 (16-bit)											
AES-GCM NCT	3058/ 3062	1492/ 1496	994/ 996	749/ 750	616/ 616	555/ 555	523/ 523	506/ 506	499/ 499	11134	350

Продовження табл. 3

1	2	3	4	5	6	7	8	9	10	11	12
AES-GCM CTM	5765/ 5769	2882/ 2886	1906/ 1908	1417/ 1418	1173/ 1174	1051/ 1051	990/ 990	960/ 960	944/ 944	11630	360
AES-GCM CT	3638/ 3642	1819/ 1823	1195/ 1197	883/ 884	728/ 728	650/ 650	611/ 611	591/ 591	581/ 581	11568	522
ChaCha20-Poly1305	3086/ 3086	1559/ 1564	837/ 840	441/ 442	332/ 332	277/ 278	250/ 251	237/ 237	230/ 230	5438	374
CPU ARM Cortex-M3 (32-bit)											
AES-GCM NCT	981/ 985	486/ 488	318/ 319	234/ 234	192/ 192	171/ 171	160/ 160	155/ 155	153/ 153	9738	452
AES-GCM CTM	1535/ 1540	763/ 765	503/ 504	373/ 373	307/ 308	275/ 275	259/ 259	250/ 250	246/ 246	10508	372
AES-GCM CT	1015/ 1020	503/ 505	329/ 330	243/ 243	199/ 199	177/ 177	166/ 167	161/ 161	158/ 158	9108	432
ChaCha20-Poly1305	578/ 585	288/ 290	154 156	87/ 88	66/ 66	55/ 55	50/ 50	47/ 47	46/ 46	3010	376

Використання запропонованих методів CT-множення в AES-GCM дало змогу уникнути істотних втрат продуктивності порівняно з NCT-версіями.

За розміром коду ChaCha20-Poly1305 значно випереджає AES-GCM, що головно є наслідком використання табличних методів у AES.

Висновки

У роботі представлено реалізації перспективних для протоколів IoT AEAD-алгоритмів AES-GCM і ChaCha20-Poly1305 із використанням типових 8/16/32-бітних low-end вбудованих процесорів. Основний акцент зроблено на досягненні максимальної швидкодії, а також забезпеченні константного часу виконання – як базового рівня захисту від атак через сторонні канали. Подані результати дають змогу зробити обґрунтований вибір автентифікуючого шифру на основі аналізу трафіку та доступних ресурсів процесора для конкретної IoT-аплікації.

Список літератури

1. Alex Biryukov and Leo Perrin. State of the Art in Lightweight Symmetric Cryptography. Cryptology ePrint Archive, Report 2017/511, 2017.

2. Sergey Panasenko and Sergey Smagin. Lightweight Cryptography: Underlying Principles and Approaches. International Journal of Computer Theory and Engineering, Vol. 3, No. 4, August 2011, pp. 516–520.

3. Sovyn Ya., Nakonechny Yu., Opirskyy I., Stakhiv M. Analysis of hardware support of cryptography in Internet of Things-devices // Ukrainian Scientific Journal of Information Security, 2018, vol. 24, issue 1, p. 36–48.

4. Eldewahi A. E. W., Sharfi T. M. H., Mansor A. A., Mohamed N. A. F. and Alwahbani S. M. H. SSL/TLS attacks: Analysis and evaluation. 2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE), Khartoum, 2015, pp. 203–208.

5. Schaumont P. Security in the Internet of Things: A challenge of scale. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 674–679.

6. Yang Y., Wu L., Yin G., Li L. and Zhao H. A Survey on Security and Privacy Issues in Internet-of-Things. IEEE Internet of Things Journal, Vol. 4, No. 5, pp. 1250–1258, Oct., 2017.

7. Dworkin M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication, NIST Special Publication 800-38D, November, 2007.

8. McGrew D. An interface and algorithms for authenticated encryption. IETF RFC 5116. January, 2008.

9. Nir Y., Langley A. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439. June 2018.

10. Langley A., Chang W., Mavrogiannopoulos N., Strombergson J., Josefsson S. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905. June 2016.

11. "CAESAR Competition for Authenticated Encryption: Security, Applicability, and Robustness". 2012.

12. Intel Architecture Instruction Set Extensions and Future Features Programming Reference. March, 2018.

13. Shay Gueron. Intel Advanced Encryption Standard (AES) New Instructions Set. Intel White Paper, 2012.

14. Shay Gueron, Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. Intel White Paper, April, 2014.

15. ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile. December, 2017.

16. Agner Fog. Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. 2018.

17. Shay Gueron, Adam Langley, Yehuda Lindell. AES-GCM-SIV Nonce Misuse-Resistant Authenticated Encryption. CFRG Meeting EUROCRYPT 2016, May, 2016.

18. Daemen J. and Rijmen V. The design of Rijndael. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2002.

19. Conrado P. L. Gouvea, Julio Lopez. High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller. Progress in Cryptology LATINCRYPT 2012. LNCS, Vol. 7533, pp. 288-304. Springer, Heidelberg (2012).

20. "The Cifra Project. A collection of cryptographic primitives targeted at embedded use." https://github.com/ctz/cifra, Feb., 2017.

21. F. De Santis, A. Schauer and G. Sigl. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 692–697.

22. Atmel Corporation. 8-bit AVR Microcontroller with 8/16K Bytes of ISP Flash and USB Controller. Technical Reference Manual, 2008.

23. Texas Instruments. User's Guide. MSP430FR58xx/59xx/68xx, and MSP430FR69xx Family, 2015.

24. ARM, "ARM and Thumb-2 Instruction Set", 2016.

25. McGrew D. A. and Viega J. The Galois/Counter Mode of Operation (GCM). Submission to NIST, 2005.

26. Loup Vaillant. The design of Poly1305, 2017. http://loup-vaillant.fr/tutorials/poly1305-design.

27. https://github.com/floodyberry/poly1305-donna/blob/master/poly1305-donna-32.h.

COMPARISON OF AEAD-ALGORITHMS FOR EMBEDDED SYSTEMS INTERNET OF THINGS

Y. Sovyn, V. Khoma, V. Otenko

Lviv Polytechnic National University,
Information Security Department

The article compares the performance and memory requirements of AES-GCM and ChaCha20-Poly1305 AED encryption solutions for typical 8/16/32-bit embedded low-end processors in the Internet of Things device with different approaches to providing tolerance to Timing Attacks and Simple Power Analysis Attacks. Particular attention is given to the low-level multiplication implementation in GF(2^{128}) with constant execution time as a key GCM operation, since low-end processors do not have ready instructions for carry-less multiplication. For each AVR/MSP430/ARM Cortex-M3 processor core, a carry-less multiplication with a constant execution time, which is similar in efficiency to algorithms with a non-constant execution time, is proposed.

Key words: AEAD, AES-GCM, ChaCha20-Poly1305, Timing Analysis, Side Channel Attacks, IoT, polynomial multiplication, microcontrollers.