*ИНФОРМАТИКА*

*Вестник Сыктывкарского университета.*
*Серия 1: Математика. Механика. Информатика.*
*Выпуск 4 (25). 2017*

**УДК 004.272.32**

# ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ ПОТОЧНОГО ШИФРА CHACHA20

***И. Ф. Королев***

Статья посвящена эффективной реализации алгоритма поточного шифрования ChaCha20 для архитектуры ARM. Данный алгоритм обладает возможностью параллельных вычислений. В статье описывается использование этой возможности для ускорения работы алгоритма шифрования с помощью технологии ARM NEON, векторные инструкции которой работают по принципу SIMD.
*Ключевые слова:* ChaCha20, ARM NEON, SIMD.

## Введение

Поточный шифр — симметричный шифр, который выполняет преобразование открытого входного сообщения по биту (или байту) за операцию, тем самым осуществляя операцию шифрования в реальном времени: скорость шифрования соизмерима со скоростью поступления входной информации. Наиболее часто используется в тех случаях, когда открытый текст поступает по частям, имеющим разную длину.

Примером является Salsa20 — семейство поточных шифров, разработанных в 2005 году Даниэлем Бернштейном (Daniel J. Bernstein). Алгоритм шифрования был представлен на конкурсе «eSTREAM», который проводился с целью поиска новых поточных шифров, пригодных для широкого применения, на основе критериев безопасности, простоты, гибкости и производительности (в отношении блочного шифра AES — утвержденного правительством США стандартом, а также других кандидатов), которые могли бы стать новым европейским стандартом для шифрования данных. Алгоритм Salsa20 прошёл все этапы конкурса и стал победителем.

Родственным к данному семейству поточных шифров является семейство ChaCha, опубликованное Даниэлем Бернштейном в 2008 году. Алгоритм ChaCha основан на тех же принципах, что и Salsa20. Изменения в алгоритме шифрования призваны улучшить перемешивание данных за один раунд, предположительно увеличивая устойчивость к криптоанализу, при той же или даже немного большей скорости [2]. Этот алгоритм помимо своего прямого назначения — симметричного шифрования — используется как основа для алгоритма аутентификации сообщений Poly1305, разработанного тем же автором. Алгоритмы ChaCha и Poly1305 обладают высокой производительностью в программных реализациях. Как отдельно друг от друга, так и в «комбинированном режиме» они используются, например, в наборе сетевых инструментов OpenSSH [6], в протоколе TLS [5], а также корпорацией Google в браузере Google Chrome [7].

Семейство Salsa20 включает в себя кроме основного алгоритма его сокращённые версии — Salsa20/12 и Salsa20/8, с двенадцатью и восемью раундами вместо двадцати оригинальных. Семейство ChaCha имеет аналогичные версии оригинального алгоритма. Сокращённые версии применяются в тех случаях, когда скорость важнее безопасности. Существуют атаки, использующие $2^{249}$ операций против Salsa20/8, а также атаки на такие версии алгоритмов с сокращённым количеством раундов, как Salsa20/7, ChaCha7 и некоторые другие. Единственная известная атака против алгоритмов Salsa20/12, Salsa20/20 [3], ChaCha8, ChaCha12 и ChaCha20 — это атака полным перебором.

Алгоритмы шифрования Salsa20 и ChaCha были сконструированы таким образом, чтобы преобразования над данными можно было осуществлять параллельно, т. е. одновременно. Это даёт существенный выигрыш в скорости для большинства современных платформ. Технология ARM NEON, включенная в большую часть новых планшетов и смартфонов с процессорами на основе архитектуры ARM, позволяет использовать возможности параллелизации обработки данных.

Целью данной статьи является описание эффективной реализации алгоритма поточного шифрования ChaCha20, которая подразумевает под собой использование технологии ARM NEON для параллелизации вычислений и ускорения работы алгоритма шифрования.

**Архитектура поточных шифров Salsa20 и ChaCha**

Существует несколько моделей построения поточных шифров. Salsa20 и ChaCha используют одну из наиболее распространённых моделей. Пусть $M$ — исходное сообщение (открытый текст) длиной $L$ байт, $L \in \{0, 1, ..., 2^{70}\}$, $K$ — ключ шифрования, $N$ — уникальный номер со-

общения (который служит так называемым вектором инициализации). Тогда зашифрованный текст $C$ получается путём применения к нему функции шифрования $E$, представляющей собой результат операции побитового исключающего ИЛИ (обозначается знаком $\oplus$) над открытым текстом $M$ и потоком, генерируемым хеш-функцией $H(K, N)$:

$$C = E(M) = M \oplus H(K, N).$$

Исходный текст $M$ получается путём применения к зашифрованному тексту $C$ функции дешифрования $D$, которая по своей сути является функцией шифрования зашифрованного текста $C$:

$$M = D(C) = E(C) = C \oplus H(K, N).$$

Открытый текст и зашифрованный текст не влияют на поток. Salsa20 следует этой модели: ядром алгоритма является хеш-функция, генерирующая поток, представляющий собой $L$-байтную последовательность, которая делится на части по 64 байта. Salsa20 зашифровывает часть открытого текста — блок длиной в 64 байта, выполняя операцию исключающее ИЛИ над этим блоком и значением хеш-функции от ключа, вектора инициализации и номера блока.

Функция шифрования Salsa20 представляет собой длинную цепочку из трех простых операций над 32-битными словами — элементами множества $\{0, 1, ..., 2^{32} - 1\}$:

- 32-битное сложение — $(a + b) \mod 2^{32}$;
- 32-битное исключающее ИЛИ — $a \oplus b$;
- 32-битный циклический сдвиг влево на постоянное значение — $a <<< const$.

Наличие только этих простых операций позволяет алгоритму шифрования достигать высоких скоростей на большинстве вычислительных устройств, при этом алгоритм шифрования не становится от этого менее безопасным [3].

Функция четвертьраунда (*quarterround*) алгоритма Salsa20, являющаяся основой хеш-функции, представляет собой следующую последовательность операций:

$$z1 = y1 \oplus ((y0 + y3) <<< 7),$$

$$z2 = y2 \oplus ((z1 + y0) <<< 9),$$

$$z3 = y3 \oplus ((z2 + z1) <<< 13),$$
$$z0 = y0 \oplus ((z3 + z2) <<< 18),$$

где $quarterround(y) = (z0, z1, z2, z3)$, $y = (y0, y1, y2, y3)$ — вектор из четырёх 32-битных слов. Результатом функции является новый вектор. Можно представить функцию $quarterround(y)$ как модификацию исходного вектора $y$, т. е. $y1$ изменяется на $z1, y2$ — на $z2, y3$ — на $z3$ и $y0$ — на $z0$.

Salsa20 помещает четыре входных слова (вектор инициализации и номер блока), восемь ключевых слов и четыре константы в матрицу состояния $4 \times 4$ следующим образом:

$$\begin{pmatrix} constant & key & key & key \\ key & constant & input & input \\ input & input & constant & key \\ key & key & key & constant \end{pmatrix}$$

Хеш-функция $H(K, N)$ представляет собой преобразование данной матрицы в десяти двойных раундах. В каждом двойном раунде происходят преобразования сначала столбцов, а затем строк с помощью функции четвертьраунда. Таким образом, преобразования ведутся только с четырьмя словами из 16 за раз. После десяти раундов результат преобразований складывается с исходной матрицей для получения выходного блока, состоящего из 16 слов (64 байт).

ChaCha, как и Salsa20, использует по четыре операции сложения, исключающего ИЛИ и циклического сдвига для обновления четырёх 32-битных слов. Однако ChaCha применяет операции в другом порядке и, в частности, обновляет каждое слово дважды, а не один раз:

$$a+ = b; \qquad d\oplus = a; \qquad d <<<= 16,$$
$$c+ = d; \qquad b\oplus = c; \qquad b <<<= 12,$$
$$a+ = b; \qquad d\oplus = a; \qquad d <<<= 8,$$
$$c+ = d; \qquad b\oplus = c; \qquad b <<<= 7.$$

Функция четвертьраунда ChaCha, в отличие от аналогичной функции Salsa20, дает возможность каждому входному слову влиять на каждое выходное слово. Ещё одно менее очевидное различие состоит в том, что теперь размытие битов в исходных данных происходит быстрее [2].

ChaCha подобно Salsa20 создает матрицу состояния $4 \times 4$, преобразует её с помощью десяти двойных раундов и добавляет результат к

исходной матрице, чтобы получить 64-байтовый выходной блок. Однако порядок слов в матрице другой:

$$\begin{pmatrix} constant & constant & constant & constant \\ key & key & key & key \\ key & key & key & key \\ input & input & input & input \end{pmatrix}$$

Ещё одно изменение в алгоритме состоит в том, что теперь в каждом двойном раунде происходят преобразования не столбцов и строк, а столбцов и диагоналей.

**Архитектура ARM**

Архитектура ARM (от англ. Advanced RISC Machine) — семейство лицензируемых микропроцессорных ядер, разрабатываемых компанией ARM Limited. Основным преимуществом данного семейства является низкое энергопотребление, благодаря чему процессоры на основе ARM чаще всего встречаются в мобильных устройствах. Архитектура ARM поддерживается множеством операционных систем. Наиболее широко используемые: Linux (в том числе Android), iOS, Windows Phone.

Технология ARM NEON включает в себя расширенный набор команд, предназначенный для увеличения производительности алгоритмов кодирования/декодирования аудио и видео, обработки изображений, 3D-графики, сигналов. Векторные инструкции ARM NEON работают по принципу SIMD — одной командой обрабатывается множество данных. Это позволяет уменьшить время работы алгоритма за счёт меньшего числа команд. Ещё одно преимущество использования ARM NEON заключается в том, что это расширение предоставляет гораздо больше места в регистрах, тем самым позволяя уменьшить количество операций загрузки и выгрузки данных [4].

**Эффективная реализация алгоритма ChaCha20**

Благодаря тому, что в алгоритме шифрования ChaCha преобразования каждого столбца и каждой диагонали, выполняемые функцией четвертьраунда, не зависят друг от друга, вычисления, необходимые для шифрования, можно выполнить параллельно. Если представить матрицу состояния в виде четырёх векторов, то четыре четвертьраунда можно выполнить одновременно с помощью операций над векторами. Таким образом, нижним уровнем алгоритма будет являться не функция четверьраунда, а преобразования столбцов и диагоналей в виде векторов.

Для проверки описанных теоретических основ алгоритм шифрования ChaCha20 был реализован на языке программирования Си. Данная

реализация стала опорной, и впоследствии её программный код был расширен ассемблерными вставками: преобразования столбцов и диагоналей матрицы состояния были реализованы с помощью векторных команд ARM NEON.

```
void columnround_asm(uint32_t y[16])
{
     asm(
           "vldm.32 %[y], {q0, q1, q2, q3}\n\t"
           /*
           *   y[0],   y[4],  y[8],   y[12]
           *   y[1],   y[5],  y[9],   y[13]
           *   y[2],   y[6],  y[10],  y[14]
           *   y[3],   y[7],  y[11],  y[15]
           *  q0 = a, q1 = b  q2 = c, q3 = d
           */
           "vadd.i32 q0, q1\n\t"              // a += b;
           "veor q4, q3, q0\n\t"              // e = d ^ a;
           "vshl.i32 q3, q4, #16\n\t"         // d = e <<< 16;
           "vsri.32 q3, q4, #16\n\t"
           "vadd.i32 q2, q3\n\t"              // c += d;
           "veor q4, q1, q2\n\t"              // e = b ^ c;
           "vshl.i32 q1, q4, #12\n\t"         // b = e <<< 12;
           "vsri.32 q1, q4, #20\n\t"
           "vadd.i32 q0, q1\n\t"              // a += b;
           "veor q4, q3, q0\n\t"              // e = d ^ a;
           "vshl.i32 q3, q4, #8\n\t"          // d = e <<< 8;
           "vsri.32 q3, q4, #24\n\t"
           "vadd.i32 q2, q3 \n\t"             // c += d;
           "veor q4, q1, q2\n\t"              // e = b ^ c;
           "vshl.i32 q1, q4, #7\n\t"          // b = e <<< 7;
           "vsri.32 q1, q4, #25\n\t"
           "vstm.32 %[y], {q0, q1, q2, q3}\n\t"
           :
           : [y] "r" (y)
           : "q0", "q1", "q2", "q3", "q4"
     );
}
```

Рис. Преобразования столбцов матрицы состояния

На рис. приведён код, реализующий преобразования столбцов матрицы состояния с помощью векторных команд ARM NEON.

С подробным описанием команд ARM NEON можно ознакомиться в [1].

Алгоритм, реализованный с использованием технологии ARM NEON, был протестирован. В качестве критерия эффективности принята скорость работы алгоритма. За эталонную версию программной реализации, с которой производилось сравнение, была взята реализация, представленная самим автором алгоритма и находящаяся в открытом доступе. Код программы был скомпилирован с помощью Linaro GCC 6.3.1 с ключом оптимизации -O3.

Тестирование производилось с помощью шифрования одного и того же файла разными реализациями алгоритма на каждом устройстве. Характеристики устройств, на которых производилось тестирование, приведены в табл. 1. По результатам шифрования каждого блока исходных данных проводилась сверка на совпадение результатов шифрования. Также была замерена скорость работы каждой реализации алгоритма. Все вычисления производились на одном ядре процессора.

Таблица 1

**Характеристики устройств**

| Тип устройства | Смартфон | Планшет |
|---|---|---|
| Процессор | Qualcomm Snapdragon 801, 1,0 ГГц | NVidia Tegra 3, 1,3 ГГц |
| Оперативная память | 2 ГБ, LPDDR3, 933 МГц | 1 ГБ, LPDDR2, 1066 МГц |
| Операционная система | Android 4.4.4 | Android 5.1 |

При создании эффективной реализации были обнаружены следующие проблемы, приводящие к низкой скорости выполнения (см. строку с начальной реализацией в табл. 2):

1. Специфичность выполнения команд в архитектуре ARM: если выполнение команды зависит от результата выполнения предыдущей команды, то это приводит к простоям обработки команд, увеличивая тем самым время работы алгоритма.

2. Несовершенство компилятора: код программы компилируется таким образом, что для выполнения каждого из десяти двойных

раундов данные из матриц, хранимые в регистрах, постоянно сохраняются в стек и загружаются из него. Это увеличивает дополнительную нагрузку на память и, соответственно, увеличивает время работы.

Первая проблема была решена благодаря выполнению преобразований сразу над несколькими матрицами: выполнение команд происходит последовательно для каждой матрицы, т. е. сначала выполняется команда преобразования данных первой матрицы, потом — второй матрицы, далее — третьей и так далее. Такой подход возможен благодаря независимости матриц состояний друг от друга. В эффективной реализации одновременно загружались три матрицы состояний, поскольку большее число матриц не помещалось в регистры ARM NEON, с учетом того, что последние также были необходимы для хранения результатов промежуточных вычислений.

Вторая проблема была решена с помощью написания цикла, выполняющего роль десяти двойных раундов напрямую с помощью ассемблерного кода. Это позволило обойтись без загрузки-выгрузки регистров в стек и увеличить скорость работы реализации алгоритма.

После внесения соответствующих корректировок в код программы скорость работы алгоритма значительно возросла (см. строку с конечной реализацией в табл. 2), а именно: на 40 % и 122 % для процессоров NVIDIA Tegra 3 и Qualcomm Snapdragon 801 соответственно. Данные результаты подтверждают состоятельность подхода, изложенного в статье. Полный код проекта, содержащий конечную и эталонную реализации алгоритма, расположен на сервисе GitHub[1].

Таблица 2

**Результаты замера скорости реализаций алгоритма**

| | Qualcomm 801 Snapdragon 801 | NVIDIA Tegra 3 |
|---|---|---|
| Эталонная реализация | 66,66 МБ/с | 25,5 МБ/с |
| Начальная реализация | 100 МБ/с | 24,4 МБ/с |
| Конечная реализация | 148,14 МБ/с | 35,9 МБ/с |

[1]https://github.com/ExceLLent404/ChaCha

**Заключение**

Результаты, приведённые выше, позволяют сделать заключение о том, что цель, поставленная в данной работе, была достигнута, а именно: использование технологии ARM NEON позволило эффективно реализовать алгоритм поточного шифрования ChaCha20.

На основе проделанной работы можно сделать следующий вывод: для достижения ожидаемых результатов при реализации алгоритма следует учитывать особенности архитектуры вычислительных устройств и используемого компилятора.

Представленное увеличение скорости работы программной реализации алгоритма (по сравнению с эталонной реализацией) не является исчерпывающим и его можно наращивать дальше, используя, например, подход, в котором для хранения и преобразования матриц состояния участвуют и регистры ARM NEON, и регистры ARM общего назначения. Помимо увеличения скорости работы алгоритма за счёт использования технологии ARM NEON, интерес вызывает вопрос об энергоэффективности данного подхода, изучение которого выходит за рамки данной работы.

# Список литературы

1. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. 2012. 2734 p.

2. **Bernstein D. J.** ChaCha, a variant of Salsa20. 2008. URL: https://cr.yp.to/chacha/chacha-20080128.pdf (дата обращения: 20.05.2017)

3. **Bernstein D. J.** The Salsa20 family of stream ciphers. 2007. URL: https://cr.yp.to/snuffle/salsafamily-20071225.pdf (дата обращения: 20.05.2017)

4. **Bernstein D. J., Schwabe P.** NEON crypto. 2012. URL: https://cryptojedi.org/papers/neoncrypto-20120320.pdf (дата обращения: 20.05.2017)

5. Internet Engineering Task Force (IETF), Google, Inc. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). 2016. URL: https://tools.ietf.org/html/rfc7905 (дата обращения: 20.05.2017)

6. OpenBSD: PROTOCOL.chacha20poly1305, v 1.3 2016/05/03. URL: http://bxr.su/OpenBSD/usr.bin/ssh/PROTOCOL.chacha20poly1305 (дата обращения: 20.05.2017)

7. Speeding up and strengthening HTTPS connections for Chrome on Android. URL: https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html (дата обращения: 20.05.2017)

**Summary**

**Korolev I. F.** Efficient implementation of ChaCha20 stream cipher

The article is about efficient implementation of ChaCha20 stream cipher for ARM architecture. This algorithm has the ability to parallel computations. The article describes the use of the ability to accelerate the operation of the encryption algorithm using ARM NEON which has SIMD vector instructions.

*Keywords: theory of plates, contact problem, antiphase.*

**References**

1. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. 2012. 2734 p.

2. **Bernstein D. J.** *ChaCha, a variant of Salsa20.* 2008. URL: https://cr.yp.to/chacha/chacha-20080128.pdf (date of the application: 20.05.2017)

3. **Bernstein D. J.** *The Salsa20 family of stream ciphers.* 2007. URL: https://cr.yp.to/snuffle/salsafamily-20071225.pdf (date of the application:
20.05.2017)

4. **Bernstein D. J., Schwabe P.** *NEON crypto.* 2012. URL: https://cryptojedi.org/papers/neoncrypto-20120320.pdf (date of the application: 20.05.2017)

5. Internet Engineering Task Force (IETF), Google, Inc. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). 2016. URL: https://tools.ietf.org/html/rfc7905 (date of the application: 20.05.2017)

6. OpenBSD: PROTOCOL.chacha20poly1305, v 1.3 2016/05/03. URL: http://bxr.su/OpenBSD/usr.bin/ssh/PROTOCOL.chacha20poly1305 (date of the application: 20.05.2017)

7. Speeding up and strengthening HTTPS connections for Chrome on Android. URL: https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html (date of the application: 20.05.2017)

**Для цитирования:** Королев И. Ф. Эффективная реализация поточного шифра CHACHA20 // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика. 2017. Вып. 4 (25). С. 33–43.*

**For citation:** Korolev I. F. Efficient implementation of ChaCha20 stream cipher, *Bulletin of Syktyvkar University, Series 1: Mathematics. Mechanics. Informatics*, 2017, №4 (25), pp. 33–43.

*СГУ им. Питирима Сорокина* *Поступила 20.12.2017*