Implementation of «Kuznyechik» cipher using vector instructions

Article · January 2018

CITATION
1

READS
890

3 authors:

Semyon Dorokhin
Moscow Institute of Physics and Technology
16 PUBLICATIONS **28** CITATIONS
SEE PROFILE

Sergei Kachkov
Moscow Institute of Physics and Technology
1 PUBLICATION **1** CITATION
SEE PROFILE

Anton Sidorenko
Moscow Institute of Physics and Technology
1 PUBLICATION **1** CITATION
SEE PROFILE

УДК 519.719.2

С. В. Дорохин[1,2], *С. С. Качков*[1,3], *А. А. Сидоренко*[1,3]

[1]Московский физико-технический институт (государственный университет)
[2]АО «ПКК Миландр»
[3]АО «Интел»

Реализация блочного шифра «Кузнечик» с использованием векторных инструкций

Целью данной работы является создание оптимизированной программной реализации блочного шифра ГОСТ Р 34.12 2015, известного как «Кузнечик». В ходе исследования был проведён анализ возможных средств улучшения скорости работы шифра. Основное внимание уделено использованию SIMD (Single Instruction Multiple Data) инструкций и учёту строения Execution Engine процессоров Intel®Core™. Отличительной особенностью статьи является то, что в ней представлены измерения скорости зашифрования и расшифрования в режимах ECB, CBC, CFB, OFB на процессорах четырёх различных поколений, в открытом доступе выложен исходный код высокоскоростной реализации. Предлагается использование 256-битных регистров ymm для ускорения зашифрования и расшифрования в режиме ECB, расшифрования в режиме CFB.

Ключевые слова: ГОСТ Р 34.12 2015, высокоскоростная реализация, LSX-преобразование, блочный шифр, SSE, AVX.

S. V. Dorokhin[1,2], *S. S. Kachkov*[1,3], *A. A. Sidorenko*[1,3]

[1]Moscow Institute of Physics and Technology (State University)
[2]JSC «Milandr»
[3]JSC «Intel»

Implementation of «Kuznyechik» cipher using vector instructions

This paper is concentrated on the highly optimized implementation of block cipher GOST R 34.12 2015 also known as Kuznyechik. A comparative analysis of possible improvements is presented, with the SIMD (Single Instruction Multiple Data) instructions in focus. The publicly available information of Intel®Core™ Execution Engine is taken into consideration. The key feature of the paper is that the cutting edge open source implementation is presented. The above implementation allows us to compare four modes of operation such as ECB, CBC, CFB and OFB. The results are given for four modern generations of Intel®Core™ processor line. We also suggest using 256-bit ymm registers and AVX2 instruction set to boost ECB mode encryption & decryption and CFB mode decryption.

Key words: GOST R 34.12 2015, high-speed implementation, LSX-transform, block cipher, SSE, AVX.

1. Введение

«Кузнечик» представляет собой блочный шифр с размером блока 128 бит. Обозначим множество всевозможных двоичных строк длины s как V_s. Алгоритм зашифрования сводится к последовательному применению следующих операций:

- Побитовая операция сложения по модулю 2: $X[k](a) = k \oplus a$, где $k, a \in V_{128}$.
- Биективное нелинейное преобразование: $S(a) = S(a_{15}||...||a_0) = \pi(a_{15})||...||\pi(a_0)$, где $a_i \in V_8$, символом $||$ обозначена операция конкатенации, а $\pi : V_8 \to V_8$ — некоторая известная подстановка.
- Линейное преобразование: $L : V_{128} \to V_{128}$.

С учётом этих обозначений функции зашифрования $E(a)$ и расшифрования $D(a)$, $a \in V_{128}$, могут быть представлены в следующем виде:

$$E_{k_1,...,k_{10}}(a) = X[k_{10}]LSX[k_9]...LSX[k_2]LSX[k_1](a); \tag{1}$$

$$D_{k_1,...,k_{10}}(a) = X[k_1]S^{-1}L^{-1}X[k_2]...S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}](a), \tag{2}$$

где $k_1, ... k_{10}$ — раундовые ключи. Эти ключи вырабатываются один раз в начале алгоритма, вычисляются также с помощью X-, S-, и L-преобразований и не оказывают существенного влияния на скорость работы шифра, поэтому в рамках статьи алгоритм развёртывания ключа не обсуждается. Здесь и далее раундовые ключи предполагаются уже вычисленными. Линейное преобразование L может быть осуществленно с помощью 16 циклов работы РСЛОС (регистра сдвига с линейной обратной связью), показанного на рис. 1. Под умножением подразумевается умножение над полем Галуа по модулю неприводимого многочлена $p(x) = x^8 + x^7 + x^6 + x + 1$, под a_i подразумеваются байты одного блока.

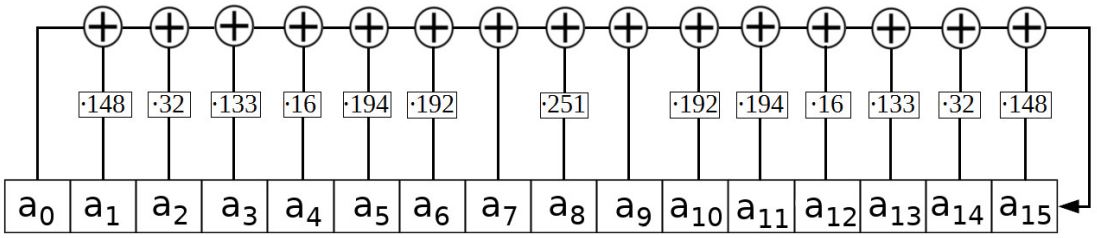

Рис. 1. РСЛОС, реализующий L-преобразование

Такое решение является предпочтительным при аппаратной реализации шифра, однако при создании программной реализации удобнее представить L-преобразование в матричной форме. Результат работы одного такта РСЛОС можно записать в виде

$$\begin{pmatrix} a^*_{n-1} \\ a^*_{n-2} \\ a^*_{n-3} \\ \vdots \\ a^*_1 \\ a^*_0 \end{pmatrix} = \begin{pmatrix} c_{n-1} & c_{n-2} & c_{n-3} & \dots & c_2 & c_1 & c_0 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ a_{n-3} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix},$$

где a^*_i — новые компоненты вектора. Результат работы после k тактов:

$$\begin{pmatrix} a^*_{n-1} \\ a^*_{n-2} \\ a^*_{n-3} \\ \vdots \\ a^*_1 \\ a^*_0 \end{pmatrix} = \begin{pmatrix} c_{n-1} & c_{n-2} & c_{n-3} & \dots & c_2 & c_1 & c_0 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}^k \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ a_{n-3} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}. \tag{3}$$

Применение L-преобразования в такой форме существенно быстрее, чем при моделировании РСЛОС. Однако профилирование программы при помощи утилиты callgrind показало, что на L-преобразование приходится приблизительно 75% времени исполнения программы.

2. Обзор существующих методов

Так как большую часть работы программы занимает L-преобразование, оптимизация операций зашифрования и расшифрования сводится к оптимизации умножения вектора на матрицу в конечном поле. Для ускорения этой операции общепринятым является использование таблиц предвычислений (LUT, Lookup Table).

2.1. Построение LUT

Особенностям построения таких таблиц уделено должное внимание в [2]. Для полноты изложения кратко приведём рассуждения из указанной статьи. Вычисляется матрица уравнения (3) для случая $k = 16$. Определим для i-го столбца полученной матрицы отображение $L_i(b) : V_8 \to V_{128}$ такое, что

$$L_i(b) = c_{i,15} \cdot b \,||\, c_{i,14} \cdot b \,||\, \ldots \,||\, c_{i,0} \cdot b,\ b \in V_8.$$

Тогда L-преобразование может быть представлено в виде

$$L(a) = L_{15}(a_{15}) \oplus L_{14}(a_{14}) \oplus \ldots \oplus L_1(a_1) \oplus L_0(a_0).$$

LUT имеет размер 16x256, каждый элемент таблицы является блоком данных из 16 байт и строится по следующему принципу: LUT$[i][j]$ есть результат покомпонентого умножения j-го байта исходного вектора на i-й столбец матрицы L-преобразования. Таким образом, для осуществления LS-преобразования над блоком необходимо произвести сложения по модулю 2 всех 16 блоков из LUT.

2.2. Объединение S- и L-преобразования

Определим теперь преобразование $L_i' : V_8 \to V_{128}, \quad i = 0, ..., 15$ как

$$L_i'(b) = c_{i,15} \cdot \pi(b)||c_{i,14} \cdot \pi(b)|| \ldots ||c_{i,0} \cdot \pi(b), b \in V_8.$$

Тогда композицию L- и S-преобразований можно представить в виде

$$LS(a) = L_{15}'(a_{15}) \oplus L_{14}'(a_{14}) \oplus \ldots \oplus L_0'(a_0),$$

то есть строится таблица предвычислений, по структуре аналогичная предыдущему пункту. Отличие заключается в том, что L- и S-преобразования объединены в одно. Теперь не составляет труда реализовать все три базовые преобразования из (1) как одно LSX-преобразование (enc_ls_table – это LUT):

Листинг 5.1. LSX-преобразование с использованием LUT

```
void Grasshopper::ApplyLSX(Block& data, const Block& key)
{
    ApplyX(data, key);
    Block tmp{};
    for (size_t i = 0; i < block_size; i++)
        ApplyX(tmp, enc_ls_table[i][data[i]]);
    data = tmp;
}
```

2.3. LUT для $L^{-1}S^{-1}$

Особенности построения LUT для обратного преобразования связаны с тем, что в уравнении (2) преобразование L применяется перед S. Решение этой проблемы может быть найдено в [2]. А именно, преобразуем это уравнение:

$$D_{k_1,\dots,k_{10}}(a) = X[k_1]S^{-1}L^{-1}X[k_2]\dots S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}](a) = \\ = X[k_1]S^{-1}L^{-1}X[k_2]\dots S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}]S^{-1}S(a). \quad (4)$$

С учётом линейности L-преобразования

$$L^{-1}X[k_i]S^{-1}(a) = L^{-1}(S^{-1}(a) \oplus k_i) = L^{-1}(S^{-1}(a)) \oplus L^{-1}(k_i) = L^{-1}S^{-1}(a) \oplus L^{-1}(k_i) = \\ = X[L^{-1}(k_i)]L^{-1}S^{-1}(a). \quad (5)$$

Применяя (5) к (4), получим

$$D_{k_1,\dots,k_{10}}(a) = X[k_1]S^{-1}L^{-1}X[k_2]\dots S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}]S^{-1}S(a) = \\ = X[k_1]S^{-1}(L^{-1}X[k_2]S^{-1})\dots(L^{-1}X[k_{10}]S^{-1})S(a) = \\ = X[k_1]S^{-1}(X[L^{-1}(k_2]L^{-1}S^{-1})\dots(X[L^{-1}(k_{10}]L^{-1}S^{-1})S(a). \quad (6)$$

Таким образом, операция зашифрования также может быть сведена к объединённому $L^{-1}S^{-1}X$-преобразованию, реализованному через LUT (в предположении, что значения $L^{-1}(k_i)$ заданы). От операции зашифрования операция расшифрования отличается применением дополнительно подстановок S и S^{-1}, поэтому расшифрование в режимах ECB и CBC будет заведомо медленнее.

2.4. Использование векторных инструкций

Для дальнейшей оптимизации можно использовать 128-битные xmm регистры, размер которых совпадает с размером блока шифра. Однако дизассемблирование модифицированной таким образом программы показывает, что цикл, отвечающий за применение склеенного LSX-преобразования, содержит дополнительную инструкцию битового сдвига влево:

Листинг 5.2. Часть результата дизассемблирования цикла LSX-преобразования

```
movzbl  0x3(%rsi),          %eax
shl     $0x4,               %rax
xorps   0x3000(%rdx,%rax,1), %xmm0
```

Адрес A в памяти вычисляется по формуле $A = B + I * S + D$, где B — базовое смещение, I — индекс, S — масштаб (scale), D — смещение (displacement). В приведённом коде rdx содержит базовое смещение B, rax используется для вычисления смещения D. Так как в LUT 16-байтовые векторы хранятся друг за другом, приходится домножать смещение на 16, используя битовый сдвиг, и передавать результат как D. Это смещение нельзя задать, используя S, так как $S \in \{2, 4, 8\}$ (см. [5]). Ещё одна оптимизация заключается в предвычислении смещений в начале цикла с тем, чтобы не исполнять в дальнейшем операцию битового сдвига, а передавать в качестве memory operand готовое D. Для предвычисления смещений также используются регистры xmm. Из блока данных посредством операций И и И НЕ с помощью маски выделяются отдельно чётные и нечётные байты. К полученным после применения И и И НЕ блокам применяются операции битового сдвига на 4 вправо (srli) и влево (slli) соответственно (см. рис. 2) с тем, чтобы считывать двубайтные блоки с отступом в 4 байта от начала (то есть кратные шестнадцати значения).

В [1] приведено описание этой модификации. В частности, указано, что следует учитывать особенности планировщика. В [4] упоминается, что, к примеру, микроархитектура Intel Sandy Bridge имеет в своём распоряжении 2 исполнительных устройства для вычисления

адреса (AGU – address generation unit), а также 3 ALU (arithmetic and logic unit), способные производить операции над векторными регистрами. Чтобы помочь планировщику выполнять загрузки блока из таблицы и суммирование по модулю 2 с результатом параллельно, можно использовать чередование регистров в этих командах (подробное описание использованных функций можно найти в [6]):

Листинг 5.3. LS-преобразование с использованием чередования регистров

```
__m128i vec1 = _mm_load_si128(reinterpret_cast<const __m128i*>
(table+_mm_extract_epi16(tmp2, 0)+0x0000));
__m128i vec2 = _mm_load_si128(reinterpret_cast<const__m128i*>
(table+_mm_extract_epi16(tmp1, 0)+0x1000));

vec1 = _mm_xor_si128(vec1, CastBlock(table+_mm_extract_epi16(tmp2, 1)+0x2000));
vec2 = _mm_xor_si128(vec2, CastBlock(table+_mm_extract_epi16(tmp1, 1)+0x3000));

vec1 = _mm_xor_si128(vec1, CastBlock(table+_mm_extract_epi16(tmp2, 2)+0x4000));
vec2 = _mm_xor_si128(vec2, CastBlock(table+_mm_extract_epi16(tmp1, 2)+0x5000));
...
vec1 = _mm_xor_si128(vec1, CastBlock(table+_mm_extract_epi16(tmp2, 7)+0xE000));
vec2 = _mm_xor_si128(vec2, CastBlock(table+_mm_extract_epi16(tmp1, 7)+0xF000));
data = _mm_xor_si128(vec1, vec2);
```

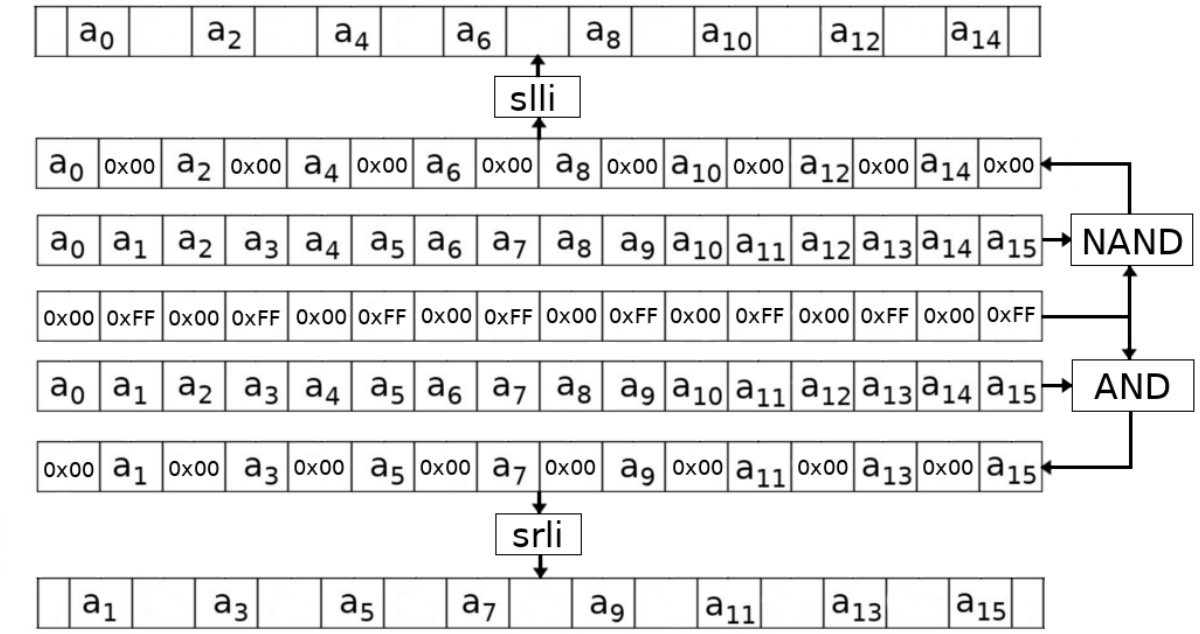

Рис. 2. Предвычисление смещений

Предвычисление инструкций, которое было описано на рис. 2:

Листинг 5.4. Результат дизассемблирования предвычисления смещений

```
movdqa (%rsi),%xmm1
movdqa %xmm1,%xmm0
psrlq  $0x4,%xmm0
psllq  $0x4,%xmm1
movdqa 0x1876(%rip),%xmm2
pand   %xmm2,%xmm0
pextrw $0x0,%xmm0,%eax
pand   %xmm2,%xmm1
0x1000(%rdx,%rax,1),%xmm2
```

Эти предварительные вычисления позволяют вычислять xor за две операции вместо трёх, что особенно важно, учитывая, что каждое применение LSX-преобразования представляет собой 16 итераций:

Листинг 5.5. Операция xor для случая с предвычислением

```
pextrw  $0x0,%xmm1,%eax
xorps   (%rdx,%rax,1),%xmm2
pextrw  $0x1,%xmm1,%eax
xorps   0x2000(%rdx,%rax,1),%xmm2
pextrw  $0x1,%xmm0,%eax
xorps   0x3000(%rdx,%rax,1),%xmm2
...
```

В статье [1] авторы пришли к заключению, что скорость работы «Кузнечика» в реализации с использованием xmm регистров достигла своего предела. Хороший сравнительный анализ многочисленных вариантов реализации шифра с использованием LUT различных размеров дан в [3]. Из этого анализа следует, что наибольшая скорость работы достигается именно с использованием xmm регистров и LUT размером 64 кБ, описанной в этой статье. Примечательно, что реализация с использованием 256-битных регистров ymm и тем же LUT в 64 кБ по результатам этого анализа существенно проигрывает реализации на xmm регистрах.

3. Использование AVX2 в режимах ECB и CFB

И всё-таки в отдельных случаях можно получить ускорение за счёт использования бóльших по размеру регистров. Мы предлагаем обрабатывать сразу два блока данных в некоторых режимах шифрования, используя расширенные до 256 бит векторные регистры (расширение набора инструкций AVX2, поддерживаемое с Intel Haswell и AMD Excavator). Это возможно в тех случаях, когда шифрование (или расшифрование) текущего блока возможно производить независимо от предыдущего. В этой работе были реализованы следующие режимы работы шифра:

- ECB (Electronic Codebook)
- CBC (Cipher Block Chaining)
- CFB (Cipher Feedback)
- OFB (Output Feedback)

Данная оптимизация была осуществлена для шифрования и расшифрования в режиме ECB и для расшифрования в режиме CFB. Попытка использовать ymm регистры непосредственно внутри LSX-преобразования не приводит к желаемому ускорению, так как загрузка 128-битных блоков в половинки регистра ymm занимает слишком много времени. Поэтому успешно использовать AVX2 для всех режимов не удаётся. Предлагается следующая модификация для режима ECB: в половины 256-битного ymm регистра загружается по блоку открытого текста, применяется объединённое LSX-преобразование, аналогичное описанному в разделе 2.4. Различие заключается в том, что одним применением LSX-преобразования обрабатывается сразу два блока. Используются аналогичные предвычисления смещений с 256-битной маской (см. рис. 3), основанные на операциях И и НЕ И и битовых сдвигах на четыре бита.

Синим цветом показаны байты, относящиеся к смещению второго блока, srli и slli обозначают побитовый сдвиг на 4 бита вправо и влево соответственно. Два ymm регистра инициализируются в начале цикла и накапливают результат исполнения xor один для чётных, другой для нечётных байт. Ещё два ymm регистра необходимы, чтобы загружать

в них данные на каждой итерации, выполнять xor с накапливающими регистрами и сохранять в эти регистры промежуточный результат:

Листинг 5.6. Использование AVX2 в режиме ECB

```
    ...
    data = _mm256_inserti128
_si256(data, table+_mm256_extract_epi16(tmp2, 0)+0x0000, 0);
    data = _mm256_inserti128
_si256(data, table+_mm256_extract_epi16(tmp2, 8)+0x0000, 1);

    vec2 = _mm256_inserti128
_si256(vec2, table+_mm256_extract_epi16(tmp2, 1)+0x2000, 0);
    vec2 = _mm256_inserti128
_si256(vec2, table+_mm256_extract_epi16(tmp2, 9)+0x2000, 1);

    vec1 = _mm256_inserti128
_si256(vec1, table+_mm256_extract_epi16(tmp1, 0)+0x1000, 0);
    vec1 = _mm256_inserti128
_si256(vec1, table+_mm256_extract_epi16(tmp1, 8)+0x1000, 1);

    vec3 = _mm256_inserti128
_si256(vec3, table+_mm256_extract_epi16(tmp1, 1)+0x3000, 0);
    vec3 = _mm256_inserti128
_si256(vec3, table+_mm256_extract_epi16(tmp1, 9)+0x3000, 1);

    data = _mm256_xor_si256(data, vec2);
    vec1 = _mm256_xor_si256(vec1, vec3);
    ...
```

В данном случае data и vec1 выступают в качестве накапливающих регистров для xor чётных и нечётных байт соответсвенно. Такая схема позволяет учитывать особенности планировщика так же, как и в пункте 2.4. Однако LSX-преобразование ускорится заведомо меньше, чем в два раза, из-за расходов на загрузку блоков в ymm регистры и из-за ограниченного количества AGU (не более 3 штук). Аналогичным образом ускоряется расшифрование. Такую же идею можно применить для расшифрования в режиме CFB.

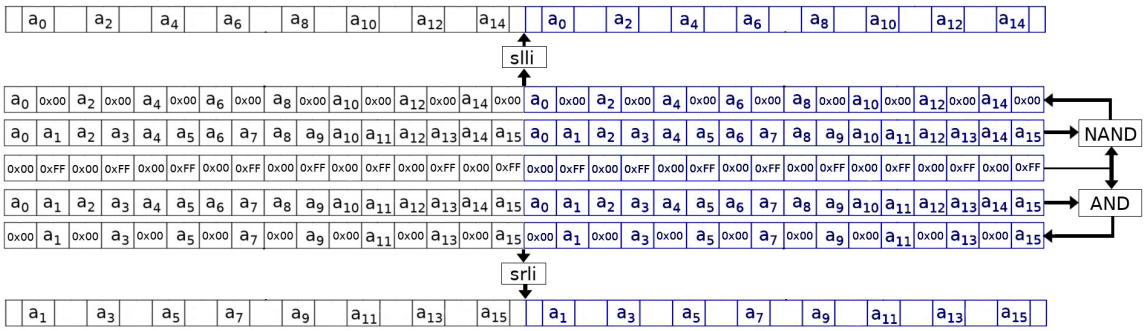

Рис. 3. Предвычисление смещений для модификации с AVX2

4. Сравнительный анализ

В заключение приведём результаты для различных степеней оптимизации и различных процессоров (табл. 1, 2). Под baseline подразумевается реализация с L-преобразованием через сдвиговый регистр; with LUT — реализация, в которой L-преобразование выполнено

при помощи LUT, но без использования xmm-регистров; LUT, xor — аналогично, но предвычисляется уже LS-преобразование, LSX объединено в одну функцию, для xor используется векторная инструкция; offset — использование xmm регистров и предвычисления смещений; AVX2 — описанная в пункте 3 модификация.

Таблица 1

Скорость шифрования в режиме ECB, Мб/с

	i3-4030u	i5-5200u	i5-7200u	i5-8250u
Baseline	6	8,8	10,2	11,4
With LUT	53	77	99	107
LUT, xor	69	99	131	139
Offset	79	113	135	142
AVX2	84	121	150	159

Таблица 2

Скорость расшифрования в режиме CFB, Мб/с

	i3-4030u	i5-5200u	i5-7200u	i5-8250u
Baseline	5,2	8,7	10,2	9,4
With LUT	50	73	94	103
LUT, xor	73	105	131	145
Offset	80	118	135	150
AVX2	82	120	145	159

Более того, выбор компилятора тоже немаловажен. В работе [3] сравнивались результаты, полученные при комплияции с помощью Visual C++, Intel C++ и gcc. Нами было установлено, что наибольшая скорость достигается при компиляции clang (см. рис. 4).

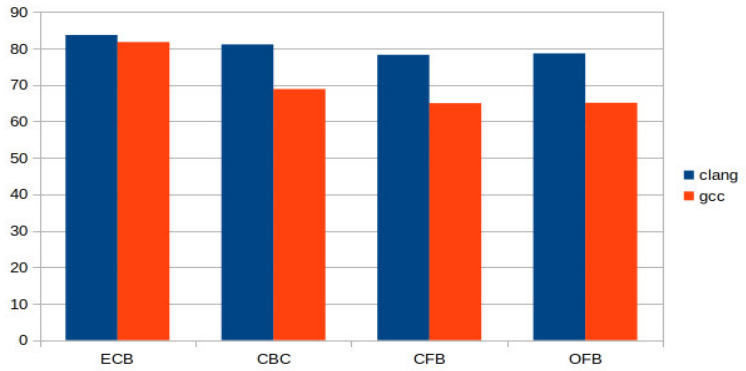

Рис. 4. Сравнение скорости работы (МБ/с) при компиляции clang и gcc (i3-4030)

5. Заключение

В данной статье подробно описаны известные на текущий момент методы оптимизации программной реализации блочного шифра «Кузнечик». Предложена модификация реализации для режимов ECB и OFB, использующая расширение набора инструкций AVX2. Данная модификация даёт заметный прирост. Дальнейшее ускорение скорости работы шифра напрямую связано только с увеличением производительности процессоров и не может быть достигнуто программными методами (за исключением распараллеливания). Методы, описанные в этой статье и в [3], представляют собой программные модификации,

ведущие к максимальному приросту скорости зашифрования и расшифрования. Исходный код реализации доступен по ссылке [7]

Литература

1. *Алексеев Е.К., Попов В.О., Прохоров А.С., Смышляев С.В., Сонина Л.А.* Об эксплуатационных качествах одного перспективного блочного шифра типа LSX // Математические вопросы криптографии. 2015. Т. 6, вып. 2. С. 6–17.
2. *Бородин М.А., Рыбкин А.С.* Высокоскоростные программные реализации блочного шифра Кузнечик // Проблемы информационной безопасности. Компьютерные системы. 2014. Вып. 3. С. 67–73.
3. *Рыбкин А.С.* О программной реализации алгоритма Кузнечик на процессорах Intel // Математические вопросы криптографии. 2018. Т. 9, вып. 2. С. 117–127.
4. Intel®64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation, 2016.
5. Intel®64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, 2016.
6. Intel®Intrinsics Guide. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/
7. Авторская реализация. URL: https://github.com/svdprima/Grasshopper

References

1. *Alekseev E.K., Popov V.O., Prokhorov A.S., Smyshlyaev S.V., Sonina L.A.* On the performance of one perspective LSX-based block cipher. Mathematical Aspects of Cryptography. 2015. V. 6, I. 2. P. 6–17.
2. *Borodin M. A., Rybkin A. S.* High-Speed Software Implementation of Kuznyetchik block cipher. — Information Security Problems. Computer Systems. 2014. N 3. P. 67-73.
3. *Rybkin A. S.* On software implementation of Kuznyechik on Intel CPUs. Mathematical Aspects of Cryptography. 2018. V. 9, I. 1. P. 117–127.
4. Intel®64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation, 2016.
5. Intel®64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, 2016.
6. Intel®Intrinsics Guide. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/
7. Authors' code repository. URL: https://github.com/svdprima/Grasshopper

Поступила в редакцию 06.12.2018