Math-Net.Ru

Общероссийский математический портал

A. S. Rybkin, О программной реализации алгоритма Кузнечик на процессорах Intel, *Матем. вопр. криптогр.*, 2018, том 9, выпуск 2, 117–127

DOI: <https://doi.org/10.4213/mvk255>

Использование Общероссийского математического портала Math-Net.Ru подразумевает, что вы прочитали и согласны с пользовательским соглашением

<http://www.mathnet.ru/rus/agreement>

Параметры загрузки:

IP: 34.227.88.159

26 апреля 2023 г., 11:41:58

On software implementation of Kuznyechik on Intel CPUs

A. S. Rybkin

JSC “InfoTeCS”, Moscow

Получено 05.II.2017

Abstract. In this paper we investigate high speed software performance issues of the Kuznyechik block cipher on Intel CPUs. We consider general block cipher implementation methods, including byte slicing technique, available speed-up possibilities on Intel architecture, and evaluate their efficiency when applied to Kuznyechik. The contributions of each element of the algorithm into its overall complexity are investigated in dependence on the optimization method used. Practical implementation results are given, and potential speed-ups are discussed.

Key words: block cipher, Kuznyechik, fast software implementation, byte slicing, high speed, performance

О программной реализации алгоритма Кузнечик на процессорах Intel

А. С. Рыбкин

ОАО «ИнфоТеКС», Москва

Аннотация. Исследуются вопросы высокоскоростной реализации блочного шифра Кузнечик на процессорах Intel различных серий. Рассматриваются несколько методов реализации, включая технику «битслайс» и доступные на архитектуре Intel способы оптимизации вычислений; оценивается эффективность их применения к алгоритму Кузнечик. Изучается вклад каждого из структурных элементов алгоритма в результирующую сложность его реализации в зависимости от используемого способа оптимизации. Приводятся результаты практической реализации для различных процессоров и возможные способы дальнейшей оптимизации.

Ключевые слова: блочный шифр, Кузнечик, быстрая программная реализация, битслайс, быстродействие

Citation: *Mathematical Aspects of Cryptography*, 2018, v. 9, № 2, pp. 117–127 (Russian)

© Академия криптографии Российской Федерации, 2018 г.

1. Introduction

Kuznyechik is one of the block ciphers specified in Russian national standard GOST R 34.12-2015 [1]. There are a number of papers dealing with its performance issues [2–5]. In this paper we focus on high-speed software implementation of Kuznyechik on Intel CPUs.

We consider several techniques of implementing block ciphers such as look-up tables and slicing, and describe the basic ways to use data and instruction parallelism on modern Intel CPUs. Applying these techniques, we have constructed a number of Kuznyechik implementations on CPUs of various generations.

We present performance results and discuss possible improvements and perspectives.

2. Block ciphers in software

There are several general ways to implement block ciphers in software.

One of the most efficient methods is to use precomputed look-up tables (LUTs for short). LUTs may significantly simplify any operation by replacing it by easy table look-ups (see, for example, [6]).

In block ciphers this method is often used to implement linear transformation together with non-linear transformation when the latter precedes the former. Actually any operation may be implemented by using LUT. However, it is necessary to watch out for the size of the resulting tables in a view of different performance of various memory levels. Therefore the LUTs size may have a strong effect on the overall performance and it is very important to analyze the cipher structure when choosing the operations to implement via LUTs.

Another popular method is slicing (see, for example, [7]). It is usually based on bitwise (bit slicing) or byte-wise (byte slicing) manipulation with input values for calculating the output ones.

Slicing techniques do not require a lot of precomputed data as LUTs do. In some cases it is possible to provide constant working time for slicing implementation [7], and thereby to safeguard the cipher implementation against cache-timing attacks [8]. The major condition to apply a slicing method is a bit- or byte-oriented cipher structure. Another important prerequisite is a possibility of parallel block processing. In case of both, slicing will be very effective technique.

Slicing and LUTs, of course, do not exclude each other. For example, slicing implementations may use small look-up tables.

3. Look-up tables for Kuznyechik

3.1. Linear transformation

The linear transformation L in Kuznyechik may be represented as multiplication of a 128-bit vector by a fixed 128×128 binary matrix. There is a well-known folkloristic implementation method for such multiplication using LUTs. It generalizes the approach presented in [6], and involves a parameter s which specifies the number of matrix rows defining one table. If s is a divisor of 128, then the total size of the LUTs is $\frac{128}{s} \cdot 2^s \cdot 128$ bits.

A significant feature of Kuznyechik is a recursive structure of L : $L = R^{16}$. Therefore to implement L it suffices to implement any of the transformations R^i , $i \in \{1, 2, 4, 8, 16\}$. In turn, transformation R^i may be represented as multiplication of a 128-bit vector by a fixed $128 \times n$ binary matrix, where $n = 8 \cdot i$. The remaining $(128 - n)$ bits of the output vector are calculated from the input vector by bit shifting.

Thus the size of LUTs to implement L depends on the choice of the particular transformation R , R^2 , R^4 , R^8 , or R^{16} and on the value of the parameter s . The first one defines the length of the table elements, while the second fixes the number of the tables and the number of elements in one table. Altogether, the total size of LUTs to implement L via R^i with parameter s , which is a divisor of 128, is

$$\frac{128}{s} \cdot 2^s \cdot n = \frac{128}{s} \cdot 2^s \cdot 8 \cdot i \quad \text{bits.}$$

3.2. Non-linear transformation

The non-linear transformation S in Kuznyechik is defined via the bitwise non-linear substitution $\pi : V_8 \rightarrow V_8$. Hence the size of the trivial LUT for implementing S is $2^8 \cdot 8 = 2048$ bits. In [9] an alternative representation of the substitution π is given in which all non-linear functions are of the form $V_4 \rightarrow V_4$. This representation significantly increases flexibility when implementing S .

The non-linear transformation S may also be implemented jointly with the linear transformation R^i for any $i \in \{1, 2, 4, 8, 16\}$ if the latter is implemented via LUTs with $s = 8k$ for some $k = 1, 2, \dots, 16$. In this case, the LUTs for the composition $R^i S$ may be easily obtained from the LUTs for R^i by a simple reordering of elements of tables. It should be noted that despite the easy transition between respective tables they are different, and each of them is needed to be stored if we plan to use both transformations: R^i and $R^i S$.

4. Features of Intel CPUs

The target computational platform abilities are also very important concerning implementation. For example registers length, instruction latency and throughput, possibility of parallel data processing have a direct impact on performance and variety of implementations.

4.1. Long registers, special instructions and parallel processing

One of the data-level parallelism methods on modern Intel CPUs is a SIMD technology. This technology is based on the usage of long registers for Multiple Data and special instruction sets to perform computations using a Single Instruction. There are several SIMD instruction sets on Intel CPUs suitable for registers with different lengths.

Let us consider what benefits can be gained using SIMD when the linear transformation L is implemented. For this purpose we consider two L implementations: via R^i , $i \in \{1, 2, 4, 8\}$, and via R^{2i} . We have seen that when L is implemented via R^i the size of LUTs twice as small as in the implementation of L via R^{2i} with the same parameter s . So from this point of view the implementation via R^i looks more preferable.

Let the maximal length of registers be $m = 2^t$ bits. For implementing L via LUTs the LOAD and XOR operations are only needed. Assume that both operations are defined only on values of length less or equal to m bit, and the execution time of a LOAD (XOR) operation is the same for all these values. If values longer than m bits must be processed by the LOAD (XOR) operation, then these values are should be cut into parts of length less than or equal to m and be processed by parts.

At first assume that each register is used to store only one table value or part of this value.

If $8 \cdot 2i \leq m$, then in both implementations any table value fits in an m -bit register. Therefore the implementations of R^i and R^{2i} require the same number of LOAD (XOR) operations. Since $R^{2i} = R^i R^i$, then the implementation via R^i involves twice as many LOAD (XOR) operations as the implementation via R^{2i} . Note that the implementation via R^i uses at most a half length of each m -bit register.

If $2m \geq 8 \cdot 2i > m$, then any table value in the implementation via R^{2i} requires twice more registers compared with any table value in the implementation via R^i . Therefore the implementation of R^i requires twice as less the number of LOAD (XOR) operations as the implementation of R^{2i} . Hence the implementations via R^i and via R^{2i} involve the same number of LOAD (XOR) operations.

Assume now that each register may be used to store several table values. In case of independent blocks processing it allows us to group similar parts from different blocks to fill in long registers before XOR operation is performed. It does not change the number of XOR operations needed to implement R^i , but it allows to process several blocks in parallel.

In this case for any fixed m the total number of XOR operations per one block remains the same regardless of the value of i . In other words SIMD usage allows us to implement L transformation via any R^i , $i \in \{1, 2, 4, 8, 16\}$, with the identical number of XOR operations. However the number of LOAD operations is changing in the same way as in the previous case.

In order to reduce the number of LOAD operations, it is necessary to load data from a LUT in a parallel way. There are several Intel instructions for this purpose. The first of them is *pshufb* (or *vpshufb*). This instruction is designed to implement $V_4 \rightarrow V_8$ LUTs. And it allows performing 16, 32 or 64 simultaneous LOAD operations from tables depending on register length. The other instruction is *vpgatherdd* (or *vpgatherdq*, *vpgatherqd*, *vpgatherqq*). It is designed for LUTs implementation up to

$$V_{32} \rightarrow V_{32}, \quad V_{32} \rightarrow V_{64}, \quad V_{64} \rightarrow V_{32}, \quad V_{64} \rightarrow V_{64}.$$

In this case the number of simultaneous LOAD operation is equal to 2, 4, 8 or 16 depending on register length and particular *vpgather*** instruction form.

Based on this instructions structures we may conclude that *pshufb* and *vpshufb* are useful to implement R with parameter $s = 4$, and *vpgather*** is useful to implement R^4 or R^8 with parameter s up to 64 (it should be taken into account that large s may lead to extremely large LUTs).

4.2. Execution units and ports

The execution core of most modern Intel CPU consists of various execution units. To provide instruction-level parallelism within a CPU core one can use several execution units simultaneously. In general, precise instruction distribution among execution ports is a very hard job. However, there are few simple ways to do this rather effectively. For example, if several instructions operate on independent data parts, then one can try to parallelize its execution by its reordering and renaming variables. If parallel processing of blocks is possible, then several blocks may be considered as an independent data parts. Smart interleaving of instructions to process different blocks may lead to a significant speed-up.

5. Implementations

5.1. Description

We have applied the above considerations in practice and have obtained a variety of Kuznyechik implementations. Let us consider some of them.

Table 1 contains a description of the implementations. Each implementation uses LUTs in one way or another. Transformations implemented via LUTs and the values of s are given in the 3rd and 4th columns. The parallelism degree is the number of blocks processed simultaneously. The “Data”(-level parallelism) column gives the number of blocks which parts are simultaneously loaded in one long register (see section). The “Instruction”(-level parallelism) column gives the number of groups of blocks where each group is processed independently for a more efficient usage of execution units (see section). Thus, the total number of blocks processed simultaneously by a certain implementation is the product of the values from “Data” and “Instruction” columns. For simplicity we append this number to the end of the implementation name.

Table 1. Description of the implementations

Name	SIMD	Transformations by LUTs	s	LUTs size (KBytes)	Parallelism degree	
					Data	Instr.
GPR-LS-1	—	LS	8	64	1	1
GPR-R8S-S-1	—	R^8S, S^{-1}	8	$32 + 0.25$	1	1
GPR-R8S-S-2	—	R^8S, S^{-1}	8	$32 + 0.25$	1	2
GPR-R4-R4S-1	—	R^4, R^4S	8	$16 + 16$	1	1
SSE-LS-1	SSE2	LS	8	64	1	1
SSE-LS-4	SSE2	LS	8	64	1	4
SSE-R8S-S-1	SSE2	R^8S, S^{-1}	8	$32 + 0.25$	1	1
AVX-LS-2	AVX2	LS	8	64	2	1
AVX-LS-8	AVX2	LS	8	64	2	4
AVX-R4-R4S-8	AVX2	R^4, R^4S	8	$16 + 16$	8	1
AVX-R-S-32	AVX2	R, S	4	$0.5 + 0.25$	32	1
AVX-R-S*-32	AVX2	R, S^*	4	$0.5 + 0.12$	32	1

AVX-R4-R4S-8 and AVX-R-S-32 (AVX-R-S*-32) implementations use *vpgatherdd* and *vpshufb* instructions for loading data from LUTs respectively. AVX-R-S-32 and AVX-R-S*-32 involve only bitwise operations, so we regard them as byte slicing implementations. They perform all look-up operations in constant time independently of input values. Therefore AVX-R-S-32 and AVX-R-S*-32 appears to be cache-timing attack resistant. The only difference between AVX-R-S-32 and AVX-R-S*-32 is that the latter uses the representation of the substitution π given in [9].

5.2. Speed results

For computations we have used Intel CPUs of various generations:

- Core i7-2600 (3.40 GHz),
- Core i7-4770 (3.40 GHz),
- Core i7-6700 (3.40 GHz),
- Xeon E5-1650 v4 (3.60 GHz),
- Xeon E5-2650 v4 (2.20 GHz).

All these CPUs have the same L1 and L2 cache size per core: L1i cache size is 32 KB, L1d cache size is 32 KB, and L2 cache is 256 KB.

The source code was written in C language, and compiled using the following compilers:

- Visual C++ 2015 (Windows),
- Intel C++ 17.0 (Windows),
- GCC 4.7.2 (Debian GNU/Linux).

For testing purposes we have fixed the cipher key, but have generated it randomly for every start of the code. For measurements we did not take into account the key schedule of Kuznyechik. So the round keys were computed before measurements. The number of input blocks for an implementation was as specified in the end of its name. For measurements all input blocks were re-encrypted in ECB encryption mode [10] up to 10^8 times.

Besides key change and key schedule restrictions there are three main constraints of such measurement method. The first one is a parallel processing of several blocks. However this is inherent to ECB, CTR (both encryption/decryption), CFB, CBC (both decryption) modes of operations and to OFB, CFB, CBC (all encryption/decryption) extended modes of operations [10]. The second one is an only encryption algorithm usage. This is inherent to CTR, OFB, CFB and OMAC1 [10]. The third one is a re-encryption of short data parts. This may lead to some decreasing of speed in the real systems because of the additional time required to transfer data from and to memory.

It should be noted that in the case of non-parallelizable modes parallel processing of several blocks may be done by parallel processing of several messages. Constraints related to an only encryption algorithm usage may be easily lifted by addition of analogous implementations based on decryption algorithm. Such implementations will presumably be on 10-20% slower because of larger complexity of the decryption algorithm (except AVX-R-S-32 and AVX-R-S * -32 — their speed will not be changed).

5.2.1. Using one thread on one core

Table 2 shows the speed values obtained using one thread on one core.

Table 2. Performance using one thread on one core (MBytes/s)

CPU → Compiler ↘ Name ↓	Core i7-2600			Core i7-4770			Core i7-6700	Xeon E5-1650 v4	Xeon E5-2650 v4
	VC	Intel	GCC	VC	Intel	GCC	VC	GCC	GCC
GPR-LS-1	110	125	100	135	145	125	145	120	75
GPR-R8S-S-1	130	125	120	160	160	140	170	140	85
GPR-R8S-S-2	160	155	125	210	210	155	235	150	95
GPR-R4-R4S-1	65	70	55	95	95	65	105	65	40
SSE-LS-1	145	150	140	160	165	150	170	150	90
SSE-LS-4	230	225	210	300	290	270	360	260	160
SSE-R8S-S-1	95	100	105	120	125	115	125	110	70
AVX-LS-2	—	—	—	200	170	195	210	185	110
AVX-LS-8	—	—	—	255	270	275	290	265	160
AVX-R4-R4S-8	—	—	—	65	65	45	115	65	40
AVX-R-S-32	—	—	—	235	250	220	255	205	130
AVX-R-S*-32	—	—	—	300	—	—	—	—	—

The analysis of Table 2 leads us to the following conclusions.

— Implementations

- The transition from LS tables to R^8S and S^{-1} tables having only general purpose registers ("GPR-...") turns out to be very effective. The number of operations remains about the same but the total size of LUTs is reduced twice, and it seems that almost all LUTs fit in L1d cache.
- The transition from R^8S and S^{-1} tables to R^4 and R^4S tables with only general purpose registers dramatically decreases the speed because the number of load operations increases too much.
- The transition from LS tables to R^8S and S^{-1} tables in the case of SSE registers ("SSE-...") is not effective either by the same reason.
- The *vpgatherdd* instruction (AVX-R4-R4S-8) is not good for high-speed implementations on up to the 6th generation Intel CPUs due to its large latency and reciprocal throughput. The same is true of any other *vpgather*** instruction.
- A significant speed-up is achieved if processing of independent blocks is possible. The best of our results is 360 MB/s (SSE-LS-4). In case of using non-parallelizable modes of operation the attained speed is 170 MB/s (GPR-R8S-S-1).

- Byte slicing technique and *vpshufb* instructions (AVX-R-S*-32) with 256-bit registers gave the speed of about 300 MB/s. It is not the highest among the others. However, it is mainly defined by the *S* transformation. We tried a simplified Kuznyechik version without *S*, and attained the speed of more than 500 MB/s which is higher than speed of any other implementation of this simplified cipher. This completely downgrades the complains of some practitioners that the *L* transformation of Kuznyechik is too complicated for implementation. So a potential speed-up of the *S* transformation may make byte slicing implementations to be the fastest. Moreover, using longer 512-bit registers will presumably double the speed of byte slicing implementation. It is very unlikely that any other software implementation on Intel CPUs will give such an increase in speed.

– Platforms

- The transition from 2nd to 4th (from 4th to 6th) generation of CPU leads to 20-25% (5-10%) speed-ups on average. Furthermore there is a proportionality between implementations speed-ups and instruction-level parallelism degrees.
- The *vpgatherdd* instruction (AVX-R4-R4S-8) speeds up by 75% in the transition from 4th to 6th generation of CPU. So, this instruction can become rather effective if this tendency will continue.

– Compilers

- In case of using (not using) instruction-level parallelism, Visual C++ compiler is slightly more (less) efficient than Intel C++ compiler.
- GCC is almost always less efficient than Visual C++ and Intel C++ compilers (except AVX-LS-8).

In Table 3 we compare performance of our implementations with previous ones. Each cycles per byte value is given as an interval where the lower (upper) bound of the interval is calculated from processor base (maximal) frequency.

Table 3. Performance comparison

Paper	Processor	MBytes/s	Cycles/Byte	Notes
[2]	Core i7-2600 @ (3.4 GHz)	138	23.5 – 26.3	non-parallelizable mode
[3]	Core i5-2500K @ (3.3 GHz)	135	23.3 – 26.1	ECB mode
		129	24.4 – 27.4	CTR mode
[5]	Core i5-6500 @ (3.2 GHz)	335	9.1 – 10.2	CTR mode
This paper	Core i7-6700 @ (3.4 GHz)	360	9 – 10.6	parallelizable mode
		170	19.1 – 22.4	non-parallelizable mode
	Core i7-4770 @ (3.4 GHz)	300	10.8 – 12.4	byte slicing implementation

5.2.2. Using several cores

We also performed the measurements on multiple cores simultaneously. Obtained results in MBytes/s per core are given in Table 4. The values of the columns “1” correspond to the measurements using one thread on one core (see Table 2). The values of the column “P” (“L”) correspond to the measurements using all physical (logical) cores simultaneously.

Table 4. Performance using several cores (MBytes/s per core)

CPU → Cores ↘ Name ↓	Core i7-2600			Core i7-4770			Core i7-6700	
	1	P	L	1	P	L	1	P
GPR-R8S-S-1	130	120	60	160	155	75	—	—
GPR-R8S-S-2	160	150	60	210	200	75	—	—
SSE-LS-1	145	135	80	160	155	85	—	—
SSE-LS-4	230	220	90	300	290	105	360	335
AVX-LS-2	—	—	—	200	195	100	—	—
AVX-LS-8	—	—	—	255	240	100	—	—

Table 4 (continued). Performance using several cores (MBytes/s per core)

CPU → Cores ↘ Name ↓	Xeon E5-1650 v4			Xeon E5-2650 v4		
	1	P	L	1	P	L
GPR-R8S-S-1	140	140	70	85	85	40
GPR-R8S-S-2	150	150	60	95	95	35
SSE-LS-1	150	150	85	90	90	50
SSE-LS-4	260	260	100	160	160	60
AVX-LS-2	185	185	90	110	110	55
AVX-LS-8	265	265	100	160	160	60

Several conclusions based on results in Table 4 are given below.

– Turbo Boost

- There is 3-8% slow-down on Intel Core CPUs when all physical cores are used. It may be explained by smaller Turbo Boost effectiveness in case of all physical cores usage.
- There is no slow-down on Intel Xeon CPUs if all physical cores are used. It is because Turbo Boost was being turned off during all computations on this CPUs.

– Hyper-threading

- If instruction-level parallelism is not used then passing from physical cores to logical cores leads to about 2 times slow-down. There is one exception: hyper-threading is effective for SSE-LS-1 implementation.
- If instruction-level parallelism is used then passing from physical cores to logical cores leads to about 2.5-3 times slow-down.

6. Conclusion

In this paper, performance issues of the Kuznyechik block cipher when implemented in software were considered. For implementations using look-up tables we described trade-offs for the size of LUTs and the number of operations needed. We gave particular insights and recommendations on how to use the resources of Intel CPUs to speed up Kuznyechik. Finally we presented the speed values for practical implementations on CPUs of various generations, using different compilers.

The best of our speed results using one thread on one CPU core were 360 MB/s and 170 MB/s for parallelizable and non-parallelizable cipher modes of operations respectively. We also presented the fairly effective byte slicing implementation of the Kuznyechik and attained the speed of 300 MB/s using one thread on one core.

References

- [1] “GOST R 34.12-2015 – National standard of the Russian Federation – Information technology – Cryptographic data security – Block ciphers” (2015), 21 pp.
- [2] Borodin M. A., Rybkin A. S., “High-speed software implementation of the Kuznyechik block cipher”, *Information Security Problems. Computer Systems*, **3** (2014), 67–73.
- [3] 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*, **6:2** (2015), 7–17. (In Russian.)
- [4] Fomin D. B., “Implementation of an XSL block cipher with MDS-matrix linear transformation on NVIDIA CUDA”, *Mathematical Aspects of Cryptography*, **6:2** (2015), 99–108. (In Russian.)
- [5] Ahmetzyanova L., Alekseev E., Oshkin I., Smyshlyaev S., Sonina L., “On the properties of the CTR encryption mode of the Magma and Kuznyechik block ciphers with re-keying method based on CryptoPro Key Meshing”. In: *Pre-proceedings of 5th Workshop on Current Trends in Cryptology*, CTCrypt 2016. (June 6-8, 2016, Yaroslavl, Russia), 42–54.
- [6] Daemen J., Rijmen V., *The Design of Rijndael*, Berlin, Heidelberg: Springer-Verlag, 2002, xvii+238 pp.
- [7] Kasper E., Schwabe P., “Faster and timing-attack resistant AES-GCM”, *Lect. Notes Comput. Sci.*, **5747** (2009), 1–17.
- [8] Bernstein D. J., *Cache-timing attacks on AES*, 2005. <https://cr.yp.to/antiforgery/cachetiming-20050414.pdf>
- [9] Biryukov A., Perrin L., Udovenko A., “Reverse-Engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1”, *Lect. Notes Comput. Sci.*, **9665** (2016), 372–402.
- [10] “GOST R 34.13-2015 – National standard of the Russian Federation – Information technology – Cryptographic data security – Modes of operation for block ciphers” (2015), 38 pp.