Math-Net.Ru

Общероссийский математический портал

Е. К. Alekseev, V. O. Popov, A. S. Prokhorov, S. V. Smyshlyaev, L. A. Sonina, Об эксплуатационных качествах одного перспективного блочного шифра типа LSX, *Матем. вопр. криптогр.*, 2015, том 6, выпуск 2, 7–17

DOI: <https://doi.org/10.4213/mvk140>

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: 54.224.135.184

26 апреля 2023 г., 10:09:11

УДК: 519.719.2

On the performance of one perspective LSX-based block cipher

E. K. Alekseev, V. O. Popov, A. S. Prokhorov,
S. V. Smyshlyaev, L. A. Sonina

LLC «CRYPTO-PRO», Moscow

Получено 16.IX.2014

We consider certain performance issues related to one block cipher scheme that is based on LSX transformations. A number of software implementation tweaks that manage to improve total speed characteristics are presented, helping to understand general performance properties of the proposed scheme.

Key words: block ciphers, LSX ciphers, fast implementations of cryptographic algorithms

Об эксплуатационных качествах одного перспективного блочного шифра типа LSX

Е. К. Алексеев, В. О. Попов, А. С. Прохоров, С. В. Смышляев, Л. А. Сони́на

ООО "КРИПТО-ПРО", Москва

Аннотация. Рассматриваются вопросы, связанные с практической реализацией одной схемы блочного шифрования, основанной на преобразованиях типа LSX. Описан ряд приемов, позволяющих улучшить временные характеристики программной реализации, что позволяет оценить общие эксплуатационные качества рассматриваемой схемы.

Ключевые слова: блочные шифры, LSX-схемы, быстрые реализации криптографических алгоритмов

Citation: *Mathematical Aspects of Cryptography*, 2015, vol. 6, no. 2, pp. 7–17 (Russian).

© 2015 E.K. Alekseev, V.O. Popov, A.S. Prokhorov, S.V. Smyshlyaev, L.A. Sonina

1. Introduction

Due to some recent successful theoretical attacks on GOST 28147-89 block cipher [6, 7, 9], an introduction of a new block cipher [4] (see also [5]) with block size equal to 128 bits seems to be very timely. The usage of a large block size is necessary for ensuring the security of modern cryptographic protocols using symmetric encryption of large payload data without frequent key transformations. The description of this block cipher (denoted as K-128) provided in Appendix A is identical to the description proposed in [4].

2. Related work

A lot of theoretical and experimental investigations aimed to study certain block cipher properties that may be used for efficient implementations was carried out. A large amount of developments are dedicated to implementations (on different hardware) of parallel data processing, e.g. CPUs with SIMD instruction extensions or GPUs. Papers concerned with optimization of block cipher algorithms based on LSX-construction such as AES (Advanced Encryption Standard) are of principal interest.

A wide variety of software AES implementations concerned on speed maximization have appeared. The most efficient software implementations make N encryptions in parallel on CPUs with N -bit registers so 128 encryptions may be performed simultaneously on CPUs with 128-bit XMM-registers [10]. These implementations exploit certain properties of substitution S that are related to particular issues of calculating multiplicative inverse in $GF(2^8)$ [11], [12]. There are also a lot of GPU implementations of AES algorithm that use different optimizations for performance boost such as using pre-computed look-up tables [13], [14], memory optimization [14], [15], reducing register usage [15].

From 2010 all Intel Core processor family based on the 32nm Intel microarchitecture codenamed Westmere provides special instructions for AES algorithm computation. Such a hardware support provides a significant advantage in AES algorithm processing speed over other block ciphers.

A comparison of K-128 and GOST 28147-89 properties related to the possibility of efficient implementations allows to emphasize certain important features. GOST 28147-89 algorithm is based on Feistel network that differs greatly from LSX-scheme in the context of algorithm implementation. The block size and the method of combining state with round keys in GOST 28147-89 algorithm are also different from K-128 and AES algorithms. From the standpoint of CPU implementations with sufficient usage of SIMD instructions

the most advantageous characteristics of GOST 28147-89 construction are half-byte non-linear transformation (S-box) and simplicity of linear transformation (left cyclic shift). The paper [2] shows the possibility of using the properties listed above to represent the GOST 28147-89 algorithm as a composition of byte-wise transformation. Due to this representation several blocks may be processed simultaneously which provides the increase in encryption speed by more than 5 times in comparison with the implementations not making use of the mentioned representation. Approaches to GOST 28147-89 implementations using SIMD instructions are also provided in [3].

Structure of compression function of hash algorithm GOST 34.11-2012 [17] is similar to K-128 round transformation in many ways. So ideas of GOST 34.11-2012 optimization such as pre-computing of look-up tables and bytes reordering [16] may be transferred to the implementation of K-128.

3. CPU implementations

In the current section we consider certain properties of K-128 construction that may be exploited in efficient implementations based on instruction sets of present-day CPU generation. The possibilities of significant usage of SIMD instructions are discussed more thoroughly in the next section. In the current section we use a small set of SIMD instructions that turns out to be very suitable for implementing block ciphers with the same block size as in K-128 and the same method of combining state with round keys.

Usage of precomputed matrix. Block transformation is realized as multiplication of the block row by the 16×16 matrix A over $GF(2^8)$. The resulting row is a linear combination of certain rows of the matrix A , defined by the components of the block row. It is possible to precompute, for example, all possible values of each row of A multiplied by all possible coefficients, obtaining 16 precomputed 256×16 tables (64 KB). Calculation of the linear combination mentioned above may be effectively performed via 128-bit XMM-registers. Thus we are able not to implement finite field operations and use table element access and bitwise modulo 2 addition operations instead.

Special matrix representation. One important idea is to exploit special matrix representation, decreasing the number of operations needed to extract specific element of matrix A . The opportunity of this approach is caused by the size of linear transformation matrix L and particular properties of current CPUs instruction sets.

Calculations on registers. The cipher construction makes possible the efficient usage of CPUs register set. K-128 allows implementations that minimize

the number of memory-reference instructions by involving all general purpose registers and XMM-registers.

Issues connected with task scheduler. K-128 allows the implementation where the operations of selecting required data from precomputed tables may be grouped to optimize reading from memory. The speedup is achieved due to reading different memory blocks into different registers.

The important issue for CPU implementations is the total size of cache memory used for command instructions and related data. Below we provide an estimate of a memory capacity necessary for storing miscellaneous data during K-128 transforms computations. Memory space needed for the precomputed table of linear transformation L depends on the sizes of bit tuples that the input block is divided into. If we split each block into parts of k bits, then there are 2^k possible values for each part and each value corresponds to 16 bytes in the table. Thus the approximate size of the table is equal to $S_k = 2^k \cdot \lfloor \frac{128}{k} \rfloor \cdot 16$ bytes. For example, $S_4 = 8\text{ KB}$, $S_6 \approx 22\text{ KB}$, $S_8 = 64\text{ KB}$, $S_{16} = 8192\text{ KB} = 8\text{ MB}$. Storing table for substitution S (256 B) and extended keys ($10 \cdot 16 = 160\text{ B}$) is also required. From the above considerations we can draw a conclusion that if L1 cache size equals 32 KB and L2 cache size equals 256 KB, then to fit in L1 cache the inequality $k \leq 6$ should be satisfied. If $k = 8$ then data is stored in L2 cache, if $k = 16$ data can't fit in cache memory. Experimental trials showed up the best results with $k = 8$.

It is evident that the memory capacity needed for storing command instructions depends on the size of the precomputed table for the transformation L , but our experimental observations lead us to conclude that 8 KB is a reliable upper bound for implementations with no loops in block processing. We would like to emphasize that the overwhelming majority of current CPUs provide L1 cache of size that is several times larger than this boundary.

We have compared the performance of K-128 algorithm in ECB and CNT modes with keys stored with and without masks. Here CNT is taken to mean encrypting message P consisting of s blocks $(P_1, \dots, P_s)$ with parameter $IV \in V_{128}$ into $(C_1, \dots, C_s)$, where $C_i = E_K^{ECB}(IV \boxplus i) \oplus P_i$, $\boxplus$ denotes an addition modulo 2^{128} operation. In our experiments with implementations based on the properties listed above we have obtained the following results.

The characteristics of the computer where measurements were taken are specified below: Intel Core i5-2500K CPU @ 3.30GHz, Core Speed 3368.4 MHz, Cache L1 Data 4x32 KB, L1 Inst. 4x32 KB, L2 4x256 KB, L3 6 MB.

The experimental results show that the encryption performance increases linearly with the number of utilized threads in all specified cases.

Table 1. Performance on CPU

	ECB no mask	ECB mask	CNT no mask	CNT mask
1 thread	135 MB/s	132 MB/s	129 MB/s	127 MB/s
2 threads	270 MB/s	265 MB/s	257 MB/s	253 MB/s
3 threads	402 MB/s	393 MB/s	382 MB/s	374 MB/s
4 threads	525 MB/s	514 MB/s	498 MB/s	493 MB/s

On the basis of the above considerations, it's reasonable to conclude that the possibility of using all above-mentioned optimal decisions is caused by the following properties of K-128 algorithm: the block size being equal to 128 bits, round keys being combined with cipher states via bitwise modulo 2 addition, the linear transformation matrix size being 16×16 . All mentioned properties may be used in implementations due to methods of processing of 128-bit operations in present-day CPUs.

4. SIMD CPU implementations

The current generation of CPUs allows to implement effectively mappings $V_n \rightarrow V_m$ for $n = 4$ and any m (most effective for $m \leq 8$) with SIMD instructions, particularly PSHUFB.

PSHUFB instruction. Let $a, b, mask$ be byte arrays of the length 16. Then $b = \text{PSHUFB}(a, mask)$ if $b[i] = a[mask[i] \& 0x0F]$ when $mask[i] < 0x80$ and $b[i] = 0$ otherwise. Element-wise transformation $S : V_4 \rightarrow V_4$ on nibble (four bits) array of the length 16 (so, input is $x = (x[0], x[1], \dots, x[15])$ where $0 \leq x[i] \leq 15$, output is $y = (y[0], \dots, y[15])$) using PSHUFB instruction is performed in the following way: let $a[i] = S(i)$, $mask[i] = x[i]$, then $y = \text{PSHUFB}(a, mask)$. 128-bit XMM-registers are used as function arguments. It should be particularly noted that in the substitution S implementation a mask and an input vector actually change places in instruction argument list. PSHUFB instruction corresponds to intrinsic function `_mm_shuffle_epi8` which behavior is described in MSDN [1].

The substitution that is used in K-128 cipher is a mapping $V_8 \rightarrow V_8$. Such a mapping $V_8 \rightarrow V_8$ may be implemented efficiently using SIMD instructions PSHUFB if it may can be represented in the form $g(h_1, h_2, \dots, h_k)$ for a small k , where $g : V_{8k} \rightarrow V_8$, all coordinate functions of mapping g ($V_k \rightarrow V_1$) are equal and $h_i : V_4 \rightarrow V_8$. For $k = 4$ the total number of mappings f that allow such a decomposition is equal to $2^{24} \cdot 2^{8 \cdot 2^4} \cdot \binom{8}{4}^4 = 2^{16} \cdot 2^{512} \cdot 24010000 < 2^{553}$.

Since the total number of mappings $V_8 \rightarrow V_8$ is equal to 2^{2048} , the probability of obtaining (with uniformly random choice) a mapping f that may be represented in the described form is negligible.

Another major problem with implementing K-128 using PSHUFB is the construction of the linear transformation L . Efficient usage of PSHUFB instruction is possible only when each nibble of an input vector x affects a small number of bytes of a vector $y = L(x)$. In this case L transformation may be performed byte-wise. Such a property is satisfied for the linear transformation used in GOST 28147-89 algorithm (left cyclic 11-bit shift). An example of efficient implementation of GOST 28147-89 algorithm exploiting this property may be found in [2].

Thus obtaining an effective SIMD implementation of K-128 cipher seems to be impossible using instruction sets of the current generations of CPUs. Optimization of the single block encryption, for which speed of 130 MB/s appears to be maximum possible, seems to be the reachable maximum. This is corroborated by the fact that this realization have all general purpose registers and XMM-registers involved.

Next we consider several classes of toy modifications of K-128 algorithm to illustrate our reasoning listed above. The exploration of these constructions is aimed to emphasize the influence of scheme elements on overall performance of implementations. Some of the considered modifications allow efficient implementations based on SIMD instructions. It has to be stressed that the estimation of cryptographic qualities of these toy constructions is beyond the scope of this work.

Suppose that mapping $S : V_8 \rightarrow V_8$ may be represented in the following form:

$$S(x_1, x_2, \dots, x_8) = S_5(S_1(x_1, x_2, x_3, x_4) \oplus S_2(x_5, x_6, x_7, x_8)) \parallel S_6(S_3(x_1, x_3, x_5, x_7) \oplus S_4(x_2, x_4, x_6, x_8)),$$

where $S_i : V_4 \rightarrow V_4$, $i = \overline{1, 6}$.

We have examined two alternatives to K-128 linear transformation: left 11-bit cyclic shift (this variation is denoted as K-128-A) or application of left 11-bit cyclic shift followed by a bitwise multiplication by constants from $GF(2^8)$ (denoted as K-128-B).

Specified alternative versions may be effectively implemented with the constructions listed above. The usage of PSHUFB operation makes it possible to achieve a sufficient performance improvement: 215 MB/s for K-128-A and 196 MB/s for K-128-B.

Imposing a condition that each output bit should depend on at least a half of input bits makes it impossible to efficiently use PSHUFB operation (alongside with bit mapping and bitwise modulo 2 addition), simultaneously denying the opportunity to make effective implementations based on SIMD instructions usage. As a specific illustration we give the following example (K-128-B*): if the multiplication in $GF(2^8)$ field in K-128-B modification is replaced with some substitution that generally can't be decomposed into small number of $V_4 \rightarrow V_4$ mappings (which is one natural way of making the algorithm non-implementable with PSHUFB, shifts and xors), we observe loss of efficiency to the speed of 96 MB/s that is considerably worse than speed of existing implementations that don't make use of SIMD instructions.

Table 2. Performances of various modifications of K-128

K-128	K-128-A	K-128-B	K-128-B*
130 MB/s	215 MB/s	196 MB/s	96 MB/s

Therefore to allow efficient implementation based on SIMD instruction usage it is necessary for all the elements of cipher construction to fulfill the requirement of decomposition into small number of $V_4 \rightarrow V_8$ mappings.

It is important to note that the usage of AVX extensions does not provide significant advantage compared to the K-128 implementations based on SSSE3. It may be explained by the fact that AVX extensions do not have fundamentally novel instructions for data processing and the usage of the same operations with 256-bit registers does not provide an essential impact on implementation effectiveness.

5. GPU implementations

Modern GPU architectures provide to a developer a lot of opportunities for optimization such as, for example, the choice of the dimension of the grid, the data memory spaces, the data shared between threads [8].

The GPU is designed to execute high number of threads in parallel, so K-128 should be adapted to run most part of the algorithm in parallel. Such adaptation is possible only for ECB and CNT modes.

The GPU implementation of K-128 cipher consists of the following steps:

1. Transferring input data and round keys to GPU global memory.
2. Execution of the GPU kernel code.
3. Returning result from GPU to CPU.

Steps 1 and 3 are performed on CPU, step 2 is performed completely parallel on GPU. The input data is divided in chunks of 4096 bytes and each block of threads encrypts one chunk. The block is composed of 256 threads so each thread processes 16 bytes of data that corresponds to the block size of K-128 cipher. Each GPU thread computes all LSX-transformations on its data and put the output to GPU global memory, then the CPU reads the result.

One of the main factors limiting the number of threads running in parallel on GPU is the number of registers used by one thread. To reduce this number the implementation uses the build-in vector data types. Structure of K-128 algorithm allows effectively apply this strategy and use only 11 registers. Our experiments with AMD Radeon HD 6970, 880Mhz, 2048Mb RAM (5500Mhz), have shown that it manages to use 95% GPU computing power. For example, if the kernel code used 16 registers instead of 11, GPU occupancy would be 76%.

K-128 cipher's block size is 16 bytes. It allows using key meshing procedure much less frequently than in GOST-28147-89 [4]. This property of K-128 algorithm gives significant performance boost because the encryption runs completely parallel on GPU and is not interrupted by sequential CPU code.

L-transformation is very slow on GPU. If one replaces this transformation with a lighter one, for example, a cyclic shift to the left by 11 bits, then the performance doubles (this modification is denoted as K-128-C).

Table 3. Performance on GPU, AMD Radeon HD 6970

Algorithm	Mode	GPU Speed
GOST 28147-89	ECB/CNT	920 MB/s
GOST 28147-89	ECB/CNT (key meshing)	346 MB/s
K-128	ECB/CNT	558 MB/s
K-128-C	ECB/CNT	1150 MB/s

6. Conclusion

This paper presents the results of theoretical and experimental research of K-128 block cipher suggested in [4] in the context of making efficient implementation based on current generation CPUs with support of SIMD instructions and GPU.

After considering the results of the research it's reasonable to conclude that the following property of K-128 algorithm serves to making effective implementations on GPU and CPU with SIMD extenstions: *the block size equals to 128 bits*. This property may be naturally supported in CPU implementations

with SIMD instructions via 128-bits XMM-registers. In parallel encryption of huge amount of data on GPU this property makes it possible not to perform key meshing frequently that produces a remarkable increase in encryption speed in comparison with block ciphers with smaller block size. It is necessary to notice that data structures used in K-128 allow implementations using a small number of registers that leads up to the efficient usage of GPU computational resources.

Properties that undoubtedly create difficulties for effective implementations with significant usage of SIMD instruction set are *the fact that nonlinear transformation S can't be decomposed into $V_4 \rightarrow V_8$ mappings* and *influence of each input bit of linear transformation L on approximately half of output bits*. Note that the efficient implementation can be made only for K-128 modifications without both mentioned properties. Mentioned properties preclude also the possibility of efficient usage of GPU analogue of SIMD instructions.

From the positive and negative properties listed above the following conclusions may be drawn:

- K-128 makes the software implementation developer able to use only a limited part of CPU SIMD instructions. Nevertheless, usage of a small set of SIMD instructions allows to make implementations with higher performance than GOST 28147-89 implementations without usage of SIMD instructions.

Fig. 1.

- The effective usage of the full range of opportunities provided by CPU SIMD extensions seems impossible in K-128 implementations. Hence the efficiency of K-128 implementation mentioned above is significantly lower than GOST 28147-89 implementation based on SIMD instructions;
- K-128 allows making more efficient GPU implementations than GOST 28147-89. The speed gain is mostly achieved by the possibility of processing large data without key transformation.

References

- [1] <http://msdn.microsoft.com/en-us/library/bb531427%28v=vs.90%29.aspx>.
- [2] Kazimirov A.V., Smyshlyaev S.V., “On creating effective software implementations of national cryptographic standards”, Presentation at RusCrypto’2013 (In Russian), <http://www.ruscrypto.ru/resource/summary/rc2013>.
- [3] “Hardcore way to productivity” (in Russian), <http://www.xakep.ru/post/60145/>.
- [4] Shishkin V.A., “Design principles of a prospective block cipher with 128 bit block length”, Presentation at RusCrypto’2013 (in Russian <http://www.ruscrypto.ru/resource/summary/rc2013>).
- [5] Dygin D., Lavrikov I., Marshalko G., Rudskoy V., Trifonov D., Shishkin V., “Low-weight and hi-end: draft Russian Encryption Standard”, *Математические вопросы криптографии*, **6:2** (2015), 29–34.
- [6] Isobe T., “A single-key attack on the full GOST block cipher”, FSE 2011, Lect. Notes Comput. Sci., **6733**, Springer, 2011, 290–305.
- [7] Dinur I., Dunkelman O., Shamir A., “Improved attacks on full GOST”, FSE 2012, Lect. Notes Comput. Sci., **7549**, 2012, 9–28.
- [8] Accelerated parallel processing OpenCL programming guide. AMD, May 2012..
- [9] “GOST 28147-89. Information Processing Systems. Cryptographic Protection. Cryptographic Transformation Algorithm” (In Russian).
- [10] Rebeiro C., Selvakumar D., Devi A.S.L., “Bitslice implementation of AES”, CANS 2006,, Lect. Notes Comput. Sci., **4301**, 2006, 203–212.
- [11] Satoh A., Morioka S., Takano K., Munetoh S., “A compact Rijndael hardware architecture with S-box optimization”, ASIACRYPT 2001, Lect. Notes Comput. Sci., **2248**, 2001, 239–254.
- [12] Canright D., “A very compact Rijndael S-box”, CHES 2005, Lect. Notes Comput. Sci., **3659**, 2005, 441–455.
- [13] Manavski S. A., “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography”, ICSPC 2007, 65–68.
- [14] Daniel T.R., Mircea S., “AES algorithm adapted on GPU using CUDA for small data and large data volume encryption”, *Int. J. Appl. Math. and Informatics*, **5:2** (2011), 71–81.
- [15] Gilger J., Barnickel J., Meyer U., “GPU-acceleration of block ciphers in the OpenSSL cryptographic library”, ISC 2012, Lect. Notes Comput. Sci., **7483**, 2012, 338–353.

- [16] Lebedev P.A., “Implementation of Stribog hash function family on GPUs using NVIDIA CUDA Platform in comparison with CPU implementations and old GOST R 34.11-94 standard”, Presentation at CTCrypt 2012.
- [17] “Information technology. Cryptographic Data Security. Hash-function, GOST R 34.11-2012”.
- [18] <http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set>.

7. Appendix A: Description of the proposed algorithm

We consider the following model block cipher, denoted as K-128 cipher. Block size is 128 bit, key size is 256 bits, main transformation is of the form $L(S(X[K](in)))$, where

- $X[K]$ – bitwise addition modulo 2 with the tuple K ($X[K](z) = z \oplus K$),
- S – substitution of each byte in a block: $\pi : V_8 \rightarrow V_8$ – bijective mapping and $S(x_{15}x_{14} \dots x_1x_0) = \pi(x_{15})\pi(x_{14}) \dots \pi(x_1)\pi(x_0)$,
- L – multiplication by the 16×16 matrix A with elements from $GF(2^8)$.

Encryption: $E_{K_1, \dots, K_{10}} = X[K_{10}]LSX[K_9] \dots LSX[K_2]LSX[K_1]$.

Decryption: $D_{K_1, \dots, K_{10}} = X[K_1]S^{-1}L^{-1} \dots X[K_9]S^{-1}L^{-1}X[K_{10}]$.

The round keys $K_i \in V_{128}$ are computed from the key $K \in V_{256}$ by the following scheme:

$$K \in V_{256}, (K_1, K_2) = K,$$

$$(K_{2i+1}, K_{2i+2}) = F[C_{8(i-1)+8}] \dots F[C_{8(i-1)+1}](K_{2i-1}, K_{2i}), i = 1, \dots, 4,$$

$$F[C] : V_{128}^2 \rightarrow V_{128}^2, F[C](a_1, a_0) = (LSX[C](a_1) \oplus a_0, a_1),$$

$$C_i \in V_{128}, C_i = L([i]_2), i = 1, \dots, 32.$$