

Bitsliced implementations of the Streebog and Kuznechik algorithms

Anatoly Solovyev and Denis Fomin

Advantages of bitslice implementation

Bitsliced implementation has transformed the approach to cryptography. It provides significant improvements in speed and efficiency.



High Speed

Bitwise operations greatly accelerate cryptographic computations.



Memory Efficiency

This method eliminates the need to store lookup tables.



Wide Applicability

The approach is applicable to algorithms ranging from DES to post-quantum Kyber.

Advantages of bitsliced implementation



Reduced Dependency on Execution Time

Resistance to timing attacks.



Increased Resistance to Side-Channel Analysis

Protection against power consumption and electromagnetic radiation attacks.



Data Masking Capabilities

Additional resistance to side-channel analysis.



Adaptability Across Architectures

Support for x86, ARM, FPGAs, ASICs, and GPUs.

Streebog and Kuznechik algorithms

Block Cipher Kuznyechik

128-bit block cipher, 10 rounds.

- Nonlinear transformation (S-function)
- Linear transformation (L-function)
- Round key addition (XOR)

Hash Function Streebog

Uses block cipher E , 12 rounds.

- Nonlinear transformation (S-function)
- Linear transformation (P-function)
- Linear transformation (L-function)
- Round key addition (XOR)

Common Properties

Both based on SPN (Substitution-Permutation Network) structure.

Use similar mathematical principles and primitives.

Streebog and Kuznechik algorithms

Block Cipher Kuznyechik

128-bit block cipher, 10 rounds.

- Nonlinear transformation (S-function)
- Linear transformation (L-function)
- Round key addition (XOR)

Hash Function Streebog

Uses block cipher E , 12 rounds.

- Nonlinear transformation (S-function)
- Linear transformation (P-function)
- Linear transformation (L-function)
- Round key addition (XOR)

Common Properties

Both based on SPN (Substitution-Permutation Network) structure.

Use similar mathematical principles and *primitives*.

Implementation of Operations for Bitsliced Representation



Initial/Final Transformation

Divide-and-conquer method for transposing bit matrices.



Addition in the Ring $\mathbb{Z}_{2^N}$

Efficient "column-wise" implementation with carry optimization.



Nonlinear Transformation (S-function)

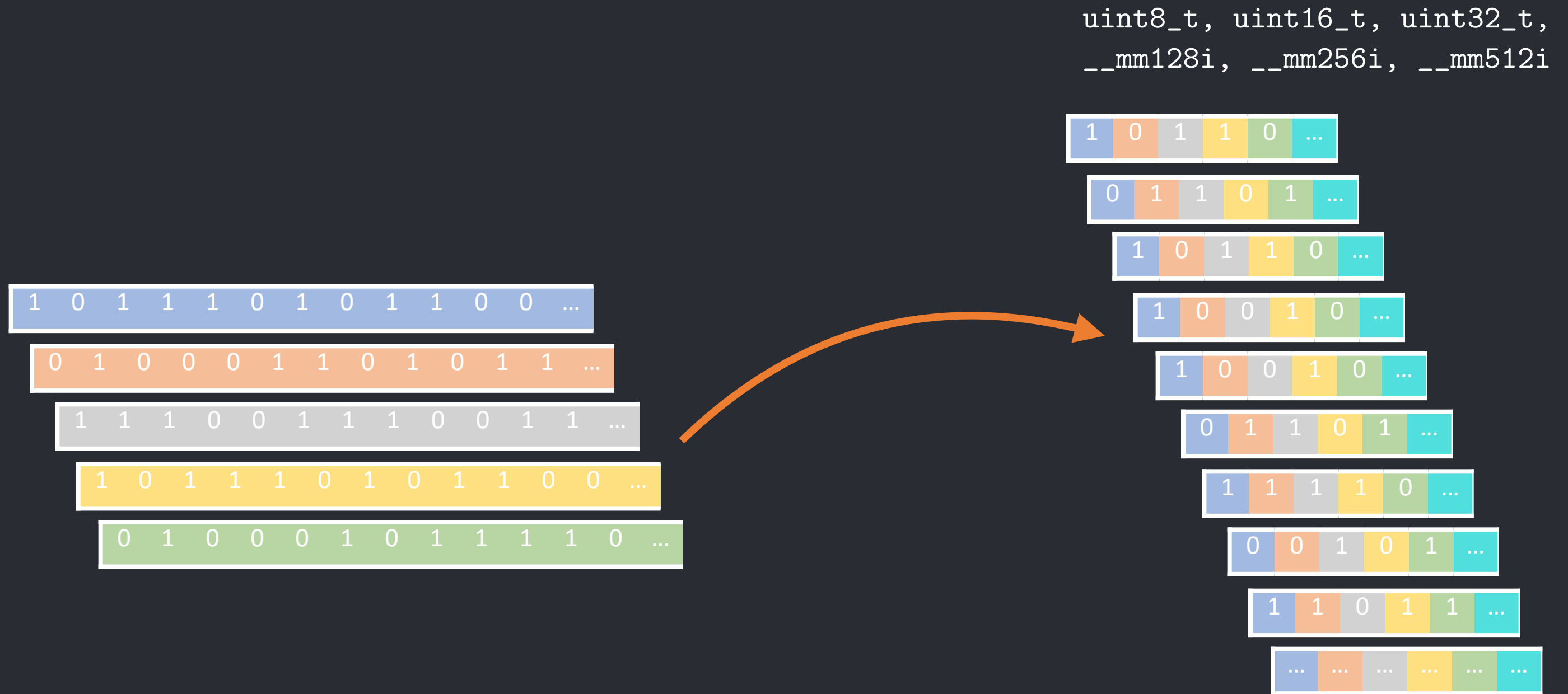
Substitution π used in both Streebog and Kuznyechik.



L-functions

Original greedy algorithm to minimize XOR operations.

Initial/Final Transformation



Initial/Final Transformation

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \end{bmatrix}$$

$$A_1 = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{0,2} & a_{1,2} & a_{0,4} & a_{1,4} & a_{0,6} & a_{1,6} \\ a_{0,1} & a_{1,1} & a_{0,3} & a_{1,3} & a_{0,5} & a_{1,5} & a_{0,7} & a_{1,7} \\ a_{2,0} & a_{3,0} & a_{2,2} & a_{3,2} & a_{2,4} & a_{3,4} & a_{2,6} & a_{3,6} \\ a_{2,1} & a_{3,1} & a_{2,3} & a_{3,3} & a_{2,5} & a_{3,5} & a_{2,7} & a_{3,7} \\ a_{4,0} & a_{5,0} & a_{4,2} & a_{5,2} & a_{4,4} & a_{5,4} & a_{4,6} & a_{5,6} \\ a_{4,1} & a_{5,1} & a_{4,3} & a_{5,3} & a_{4,5} & a_{5,5} & a_{4,7} & a_{5,7} \\ a_{6,0} & a_{7,0} & a_{6,2} & a_{7,2} & a_{6,4} & a_{7,4} & a_{6,6} & a_{7,6} \\ a_{6,1} & a_{7,1} & a_{6,3} & a_{7,3} & a_{6,5} & a_{7,5} & a_{6,7} & a_{7,7} \end{bmatrix}$$

Initial/Final Transformation

$$A_1 = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{0,2} & a_{1,2} & a_{0,4} & a_{1,4} & a_{0,6} & a_{1,6} \\ a_{0,1} & a_{1,1} & a_{0,3} & a_{1,3} & a_{0,5} & a_{1,5} & a_{0,7} & a_{1,7} \\ a_{2,0} & a_{3,0} & a_{2,2} & a_{3,2} & a_{2,4} & a_{3,4} & a_{2,6} & a_{3,6} \\ a_{2,1} & a_{3,1} & a_{2,3} & a_{3,3} & a_{2,5} & a_{3,5} & a_{2,7} & a_{3,7} \\ a_{4,0} & a_{5,0} & a_{4,2} & a_{5,2} & a_{4,4} & a_{5,4} & a_{4,6} & a_{5,6} \\ a_{4,1} & a_{5,1} & a_{4,3} & a_{5,3} & a_{4,5} & a_{5,5} & a_{4,7} & a_{5,7} \\ a_{6,0} & a_{7,0} & a_{6,2} & a_{7,2} & a_{6,4} & a_{7,4} & a_{6,6} & a_{7,6} \\ a_{6,1} & a_{7,1} & a_{6,3} & a_{7,3} & a_{6,5} & a_{7,5} & a_{6,7} & a_{7,7} \end{bmatrix}$$

$$A_2 = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} & a_{0,5} & a_{1,5} & a_{2,5} & a_{3,5} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} & a_{0,6} & a_{1,6} & a_{2,6} & a_{3,6} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{0,7} & a_{1,7} & a_{2,7} & a_{3,7} \\ a_{4,0} & a_{5,0} & a_{6,0} & a_{7,0} & a_{4,4} & a_{5,4} & a_{6,4} & a_{7,4} \\ a_{4,1} & a_{5,1} & a_{6,1} & a_{7,1} & a_{4,5} & a_{5,5} & a_{6,5} & a_{7,5} \\ a_{4,2} & a_{5,2} & a_{6,2} & a_{7,2} & a_{4,6} & a_{5,6} & a_{6,6} & a_{7,6} \\ a_{4,3} & a_{5,3} & a_{6,3} & a_{7,3} & a_{4,7} & a_{5,7} & a_{6,7} & a_{7,7} \end{bmatrix}$$

Initial/Final Transformation

$$A_2 = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} & a_{0,5} & a_{1,5} & a_{2,5} & a_{3,5} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} & a_{0,6} & a_{1,6} & a_{2,6} & a_{3,6} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{0,7} & a_{1,7} & a_{2,7} & a_{3,7} \\ a_{4,0} & a_{5,0} & a_{6,0} & a_{7,0} & a_{4,4} & a_{5,4} & a_{6,4} & a_{7,4} \\ a_{4,1} & a_{5,1} & a_{6,1} & a_{7,1} & a_{4,5} & a_{5,5} & a_{6,5} & a_{7,5} \\ a_{4,2} & a_{5,2} & a_{6,2} & a_{7,2} & a_{4,6} & a_{5,6} & a_{6,6} & a_{7,6} \\ a_{4,3} & a_{5,3} & a_{6,3} & a_{7,3} & a_{4,7} & a_{5,7} & a_{6,7} & a_{7,7} \end{bmatrix}$$

$$A_3 = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{4,0} & a_{5,0} & a_{6,0} & a_{7,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} & a_{5,1} & a_{6,1} & a_{7,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} & a_{4,2} & a_{5,2} & a_{6,2} & a_{7,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{4,3} & a_{5,3} & a_{6,3} & a_{7,3} \\ a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} & a_{5,4} & a_{6,4} & a_{7,4} \\ a_{0,5} & a_{1,5} & a_{2,5} & a_{3,5} & a_{4,5} & a_{5,5} & a_{6,5} & a_{7,5} \\ a_{0,6} & a_{1,6} & a_{2,6} & a_{3,6} & a_{4,6} & a_{5,6} & a_{6,6} & a_{7,6} \\ a_{0,7} & a_{1,7} & a_{2,7} & a_{3,7} & a_{4,7} & a_{5,7} & a_{6,7} & a_{7,7} \end{bmatrix}$$

Initial/Final Transformation

21-bit operation

$$A_3 = A^T$$

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \end{bmatrix}$$

$$A_3 = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{4,0} & a_{5,0} & a_{6,0} & a_{7,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} & a_{5,1} & a_{6,1} & a_{7,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} & a_{4,2} & a_{5,2} & a_{6,2} & a_{7,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{4,3} & a_{5,3} & a_{6,3} & a_{7,3} \\ a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} & a_{5,4} & a_{6,4} & a_{7,4} \\ a_{0,5} & a_{1,5} & a_{2,5} & a_{3,5} & a_{4,5} & a_{5,5} & a_{6,5} & a_{7,5} \\ a_{0,6} & a_{1,6} & a_{2,6} & a_{3,6} & a_{4,6} & a_{5,6} & a_{6,6} & a_{7,6} \\ a_{0,7} & a_{1,7} & a_{2,7} & a_{3,7} & a_{4,7} & a_{5,7} & a_{6,7} & a_{7,7} \end{bmatrix}$$

Addition in $\mathbb{Z}_{2^N}$

Input:

A[N-1:0] — first N-bit operand (LSB at index 0)

B[N-1:0] — second N-bit operand (LSB at index 0)

Output:

result[N-1:0] — N-bit sum of A and B in $\mathbb{Z}_{2^N}$

Variables:

carry = 0

Procedure:

for i **from** 0 **to** N - 1:

 s_i = A[i] XOR B[i]

 c_i = A[i] AND B[i]

 result[i] = s_i XOR carry

 carry = c_i OR (s_i AND carry)

return result

Addition in $\mathbb{Z}_{2^N}$ is used, for example, in the Streebog hash function and in the encryption modes CTR and CTR-ACPKM.

Addition of the Constant 2^t in $\mathbb{Z}_{2^N}$

Input:

$A[0..N-1]$ — an N-bit vector representing a number in $\mathbb{Z}_{2^N}$ (LSB at index 0)

Output:

result[0..N-1] — N-bit vector representing $A + 2^t$ in $\mathbb{Z}_{2^N}$

Variables:

carry = 0

Procedure:

result = [0] * N

for i **from** 0 **to** t - 1:

 result[i] = A[i]

▶ First t bits remain unchanged

for i **from** t **to** N - 1:

 s_i = NOT A[i]

▶ Invert current bit

 result[i] = s_i XOR carry

 carry = A[i] OR carry

return result

Addition of the Constant 2^t in $\mathbb{Z}_{2^N}$ is used, for example, in the Streebog hash function and in the encryption modes CTR and CTR-ACPKM.

S-box implementation

This work adopts the π -substitution representation proposed in [1]

 **169 bit-operations**
for π implementation

[1] D.B. Fomin, D.I. Trifonov. Computational work for some TU-based permutation. (Submitted for publication)

Linear expression optimization

1 Greedy Algorithm for Optimization

Minimize XOR operations via variable substitution.

Create temporary variables for subfunctions.

2 Reduction of Temporary Variables

Remove unnecessary temporary variables.

Optimize the substitution list.

3 Results

Significant reduction in XOR operations.

High practical efficiency.



727 XORs

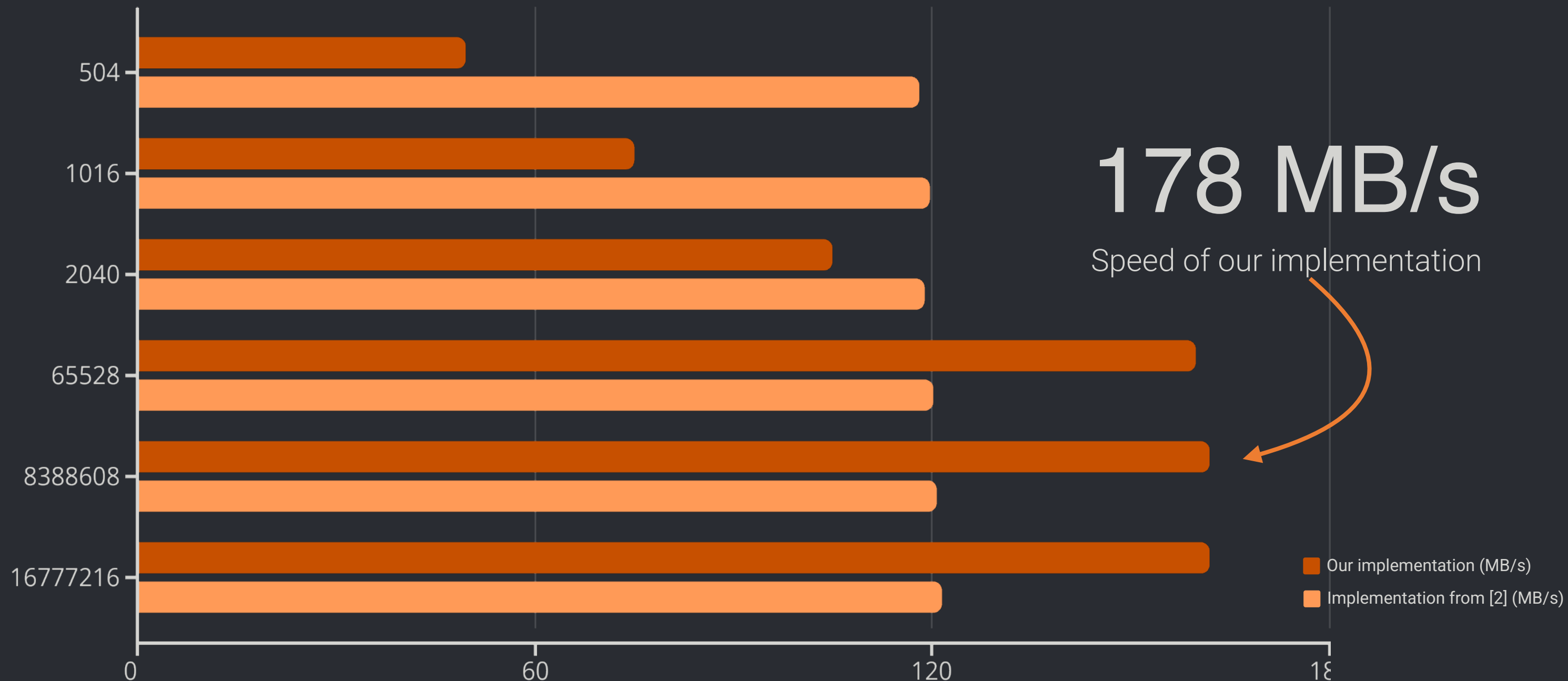
for matrix multiplication in Streebog
(**x2.3** reduction)



173 XORs

for LFSR implementation in Kuznechik
(**x2.4** reduction)

Experimental Results: Streebog



Implementation from this work, Intel Core i5-8400 CPU @ 2.80GHz

[2] Aleksey Degtyarev's implementation. https://github.com/be1ay/MyHash_Stribog/tree/master

Profiling: Streebog

Nº	Function	% Time	Description
1	LPS_bitslice	37.09% (73.90%)	Includes Pi_8; performs a cascade of τ , π , and L
2	Pi_8	36.81% (36.81%)	Applies an $8 \rightarrow 8$ S-box substitution 8 times
3	functionE_bitslice	9.37% (80.31%)	Standard function E; includes calls to LPS_bitslice_SSE, Pi_8.
4	ConvertMsgToBitSlice_Stage3	3.37% (4.19%)	Prepares the next 512 bits of message data for bitsliced processing.
5	Streebog_bitslice_ControlValue	5.52% (100.00%)	Generates messages and computes hash values
6	ConvertHashFromBitSlice	1.98% (1.97%)	Converts bitsliced results back to standard format
7	ConvertToBitSlice_32x32	0.82% (0.82%)	Transposes four 256-bit vectors (i.e., a 32×32 bit matrix)
8	Streebog_bitslice	4.52% (94.48%)	Computes hash values for 256 messages
9	functionG_Zero_bitslice	0.27% (56.99%)	Function g_0 ; includes calls to functionE_bitslice and LPS_bitslice_SSE.
10	functionG_bitslice	0.25% (28.00%)	Function g_n ; includes calls to functionE_bitslice и LPS_bitslice_SSE.

Experimental Results: Kuznechik

395 MB/s

182 MB/s

360 MB/s

Speed of our implementation*

Speed of implementation [3]*

Speed of implementations [4,5]

*Implementation from this work, Intel Core i5-8400 CPU @ 2.80GHz

*Implementation from the author, Intel Core i7-4700 CPU @ 3.40GHz

[3] Дорохин, С. В. Реализация блочного шифра "Кузнечик" с использованием векторных инструкций. ТРУДЫ МФТИ. Труды Московского физико-технического института (национального исследовательского университета), 10:4 (2018), с. 45-53.

[4] Rybkin A. S. On software implementation of Kuznyechik on Intel CPUs. Matem. Vopr. Kriptogr., 9:2 (2018), p. 117-127.

[5] Patent, 2016, RU 2710669 C1, <https://patents.google.com/patent/RU2710669C1/>.

Profiling: Kuznechik

Nº	Function	% Time	Description
1	L_transform_bitslice	40.31%	Bitslice L-transformation.
2	functionPiStroke	27.35%	Nonlinear substitution (S-box, Pi_8)
3	ConvertFromBitSlice	6.29%	Conversion from bitsliced format
4	ConvertToBitSlice	6.27%	Conversion into bitslice format
5	xor_bitslice	6.07%	Bitwise XOR operation
6	Kuznechik_bitslice_encrypt_test	13.70%	Main encryption and performance measurement function

Future research directions

NVIDIA CUDA

Leverage thousands of parallel threads.

The `lop3` instruction allows single-instruction ternary boolean operations.

This is great for high-performance cryptographic tasks.

Intel AVX-512

The `VPTERNLOGD/Q` instruction implements 3-input LUTs.

This performs arbitrary logical operations across 512-bit registers.

It offers flexible SIMD-level bit manipulation.

Future research directions

The use of the `_mm256_movemask_epi8` instruction (AVX2) made it possible to implement a fast bit matrix transpose, significantly outperforming classical bit-wise transposition in terms of speed.

This allowed achieving a performance of **402 MB/s** in ECB encryption mode, measured on a single CPU core without explicit parallelization.



Algorithmic Optimizations

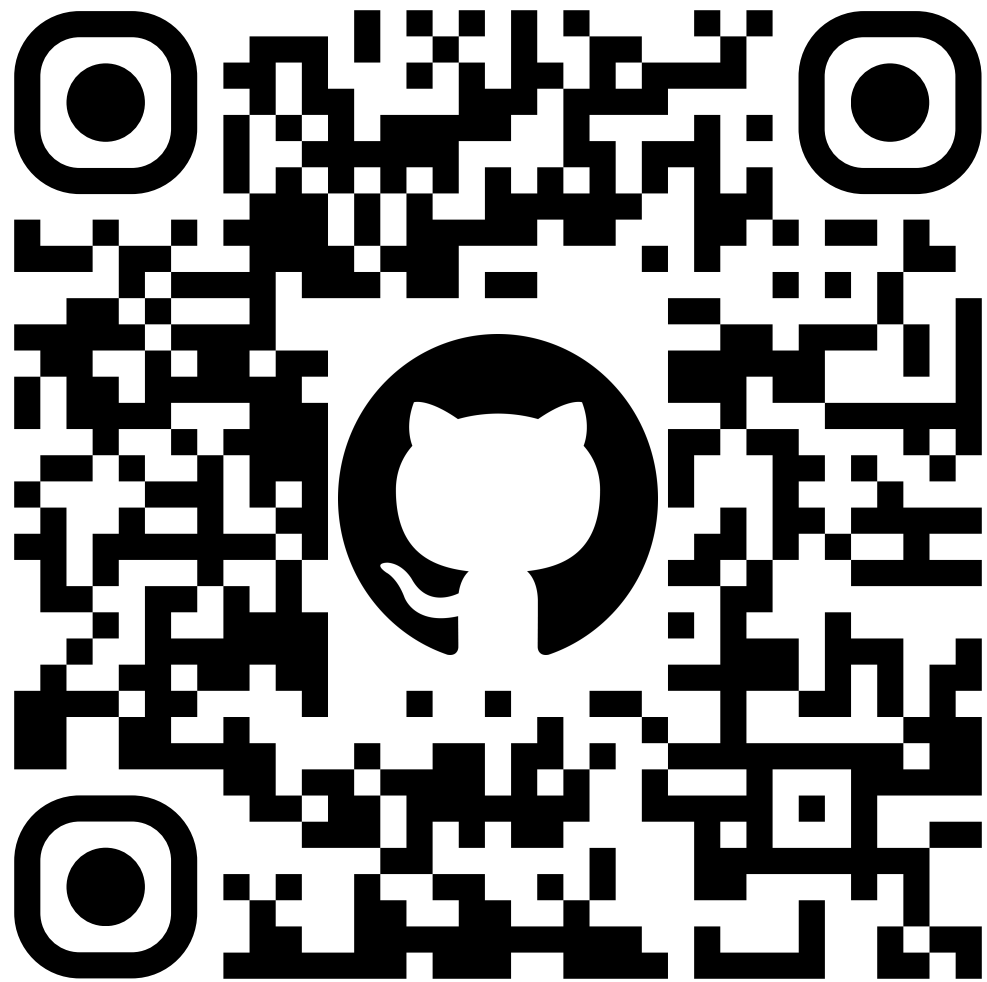
Further optimize linear and non-linear transformations for Streebog and Kuznechik algorithms.

10110
10110
11101

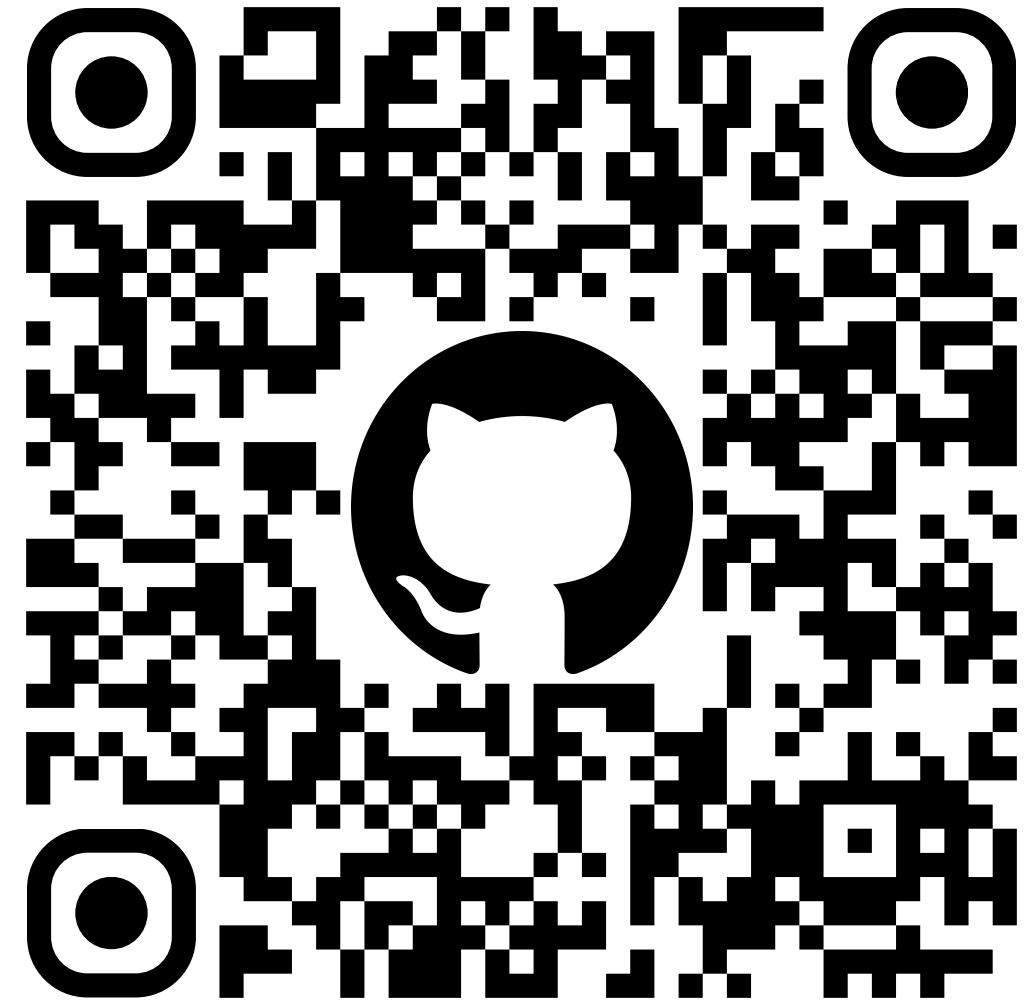
Bitsliced Modes

Implement bitsliced modes, including CTR-ACPKM, MGM and etc.

Thank you for your attention!



<https://github.com/ASSol82/Streebog-bitslice>



<https://github.com/ASSol82/Kuznechik-bitslice>

For educational, scientific research, and non-commercial purposes only