Grøstl Implementation Guide

Krystian Matusiewicz[1], Martin Schläffer[2], and Søren S. Thomsen[3]

[1]Intel Technology Poland
[2]IAIK, Graz University of Technology
[3]DTU Mathematics, Technical University of Denmark

March 9, 2012

The current version can be downloaded from
http://www.groestl.info/groestl-implementation-guide.pdf

Abstract

The Grøstl hash function is a finalist in the SHA-3 competition organized by NIST. In this paper, we describe a number of implementation techniques and tricks for Grøstl. They allow us to develop a range of efficient implementations for various platforms, from 8-bit microcontrollers to modern desktop processors using 256-bit AVX instructions. These results demonstrate the implementation flexibility of the algorithm and we hope will inspire further optimizations and ports to other platforms. The information in this paper will also be useful for developers planning to implement Grøstl efficiently on the platform of their choice. Furthermore, we believe also hardware implementations may benefit from the optimization techniques presented in this paper.

1 Introduction

The hash function Grøstl was designed in 2008 as a candidate for the SHA-3 competition [18], organized by the National Institute of Standards and Technology (NIST). In 2010, Grøstl was selected as one of five finalists in the competition.

Grøstl borrows components from the AES block cipher, which became a United States federal government standard in 2001 [17]. The AES is known for its good performance on a wide variety of platforms, which is due to a large amount of flexibility in the choice of implementation methods. Recently, Intel introduced an instruction set extension for computing AES rounds [9], which makes encryption using the AES on CPUs implementing this instruction set very efficient.

Although several underlying components in Grøstl differ from the ones used in the AES, Grøstl still enjoys many of the same implementation benefits as the AES. Even the AES instruction set extension can be used to significantly speed up Grøstl. In this paper, we describe various software implementation techniques for Grøstl suitable for platforms ranging from 8-bit microcontrollers to processors with SIMD and AES instruction set extensions. All these implementations can be downloaded from http://www.groestl.info/.

2 Description of Grøstl

The Grøstl hash function iterates an underlying compression function in a variant of the Merkle-Damgård construction [7, 15], where the size of the state (or *chaining value*) passed on from one iteration to the next is at least twice as large as the final hash value. The final hash value is computed from the last chaining value using an output transformation. Hence, Grøstl is known as a wide pipe design.

The compression function and the output transformation are based on permutations using round transformations similar to those of the AES [16]. For the final round of the competition, Grøstl was tweaked in

Archived copy · cryp2nova.com

order to increase its security margin. The initial submission is called Grøstl-0. In the following, we describe the components of the (tweaked) Grøstl hash function in more detail.

2.1 The Hash Function

Grøstl comes in several variants with different output sizes. We denote by n the number of bits in the output, and the variant returning n bits is denoted Grøstl-n. Here, we focus on Grøstl-256 and Grøstl-512. Variants returning less than 256 bits differ from Grøstl-256 only in the initial value and in the final truncation to produce the hash value. Similarly, variants returning more than 256 bits differ from Grøstl-512 in the same two respects.

The input message M is padded and split into blocks $M_1, M_2, \ldots, M_t$ of ℓ bits with $\ell = 512$ for Grøstl-256, and $\ell = 1024$ for Grøstl-512. The initial value IV, the intermediate hash values H_i, and the permutations P and Q are of size ℓ bits as well. (The exact definition of the IV can be found in [8]). The message blocks are processed via the compression function $f(H_{i-1}, M_i)$, which accepts two ℓ-bit inputs and outputs an ℓ-bit value. After all t message blocks have been processed, an output transformation $\Omega(H_t)$ is applied which outputs the final n-bit hash value h:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, M_i) \quad \text{for } 1 \le i \le t \\ h &= \Omega(H_t). \end{aligned}$$

For all variants, ℓ is at least twice as large as n.

2.2 The Compression Function

The compression function f is based on two ℓ-bit permutations P and Q. The compression function is defined as follows:

$$f(H_{i-1}, M_i) = P(H_{i-1} \oplus M_i) \oplus Q(M_i) \oplus H_{i-1}.$$

The construction of the compression function of Grøstl is shown in Figure 1.

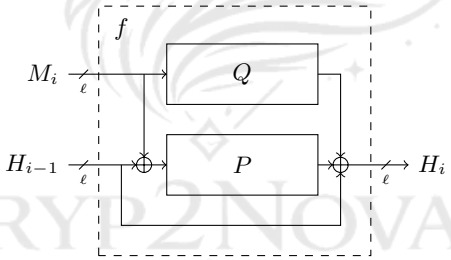


Figure 1: The compression function f of Grøstl. The permutations P and Q are of size $\ell \ge 2n$ bits.

2.3 The Output Transformation

After the last call to the compression function, an output transformation Ω is applied to H_t to give the final hash value of size n:

$$\Omega(H_t) = \text{trunc}_n(P(H_t) \oplus H_t),$$

where $\text{trunc}_n(x)$ discards all but the least significant n bits of x. The output transformation is also shown in Figure 2.

Archived copy · cryp2nova.com

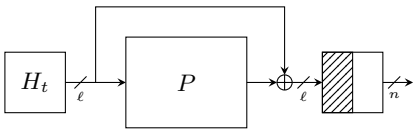
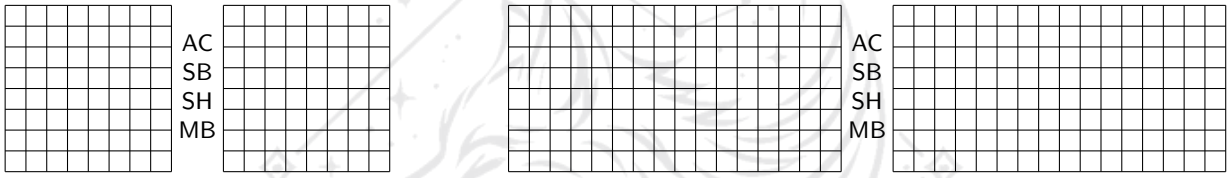


Figure 2: The output transformation Ω of Grøstl. The permutation P is of size $\ell \geq 2n$ bits and only the last n bits are returned.

2.4 The Permutations

Two permutations P and Q are defined for Grøstl. To distinguish between the permutations of Grøstl-256 and Grøstl-512 we sometimes write P_ℓ or Q_ℓ where ℓ is the size of the permutations. In each permutation, the four AES-like round transformations AddRoundConstant (AC), SubBytes (SB), ShiftBytes (SH), and MixBytes (MB) are applied to the state in the given order. The permutations differ only in the constants used in AddRoundConstant and ShiftBytes, and in their number of rounds.

Grøstl-256 has 10 rounds and the 512-bit state of permutation P_{512} and Q_{512} is viewed as an 8×8 matrix of bytes. One round of one permutation of Grøstl-256 is shown in Figure 3. For Grøstl-512, 14 rounds are used and the 1024-bit state of the two permutations P_{1024} and Q_{1024} is viewed as an 8×16 matrix of bytes.



(a) Grøstl-256 (b) Grøstl-512

Figure 3: One round of one permutation of the Grøstl-256 and Grøstl-512 hash function.

2.4.1 AddRoundConstant

The AddRoundConstant (AC) transformation XORs a round-dependent constant to one row of the state. The constant and the row is different for P and Q. Additionally, a round-independent constant ff is XORed to every byte in Q (we denote hexadecimal byte values by two-character values in sans serif font). The XOR constants for round i (where i is viewed as a hexadecimal digit and $\bar{i}$ denotes the bit-wise complement of i) are shown in Figure 4.

2.4.2 SubBytes

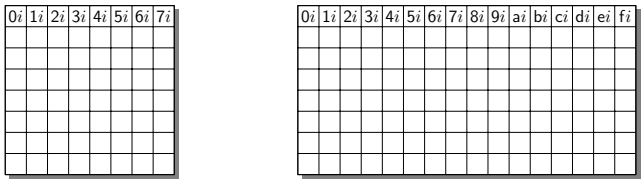
The SubBytes (SB) transformation applies the AES S-box to each byte of the state. The definition of this S-box can be found in [8].

2.4.3 ShiftBytes

ShiftBytes (SH) cyclically rotates the bytes of row r to the left by $\sigma[r]$ positions with different values for P and Q in Grøstl-256 and Grøstl-512. We have the following rotation values:

$$\begin{aligned}
\sigma &= \{0,1,2,3,4,5,6,7\} && \text{for } P \text{ in Grøstl-256} \\
\sigma &= \{1,3,5,7,0,2,4,6\} && \text{for } Q \text{ in Grøstl-256} \\
\sigma &= \{0,1,2,3,4,5,6,11\} && \text{for } P \text{ in Grøstl-512} \\
\sigma &= \{1,3,5,11,0,2,4,6\} && \text{for } Q \text{ in Grøstl-512}
\end{aligned}$$

Archived copy · cryp2nova.com



(a) P_{512} (b) P_{1024}

(c) Q_{512} (d) Q_{1024}

Figure 4: The XOR constants added by the AddRoundConstant transformation.

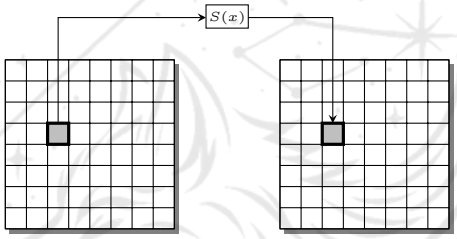


Figure 5: SubBytes substitutes each byte of the state using the AES S-box.

2.4.4 MixBytes

MixBytes (MB) is a linear diffusion layer, which multiplies each column A of the state with a constant, circulant 8×8 matrix B:

$$A \leftarrow B \cdot A$$

where

$$B = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}. \tag{1}$$

In this matrix B, constants are elements in the finite field $GF(2^8)$ defined by the polynomial $\theta^8 \oplus \theta^4 \oplus \theta^3 \oplus \theta \oplus 1$. An 8-bit value x with binary representation

$$x = x_7 \cdot 2^7 + x_6 \cdot 2^6 + x_5 \cdot 2^5 + x_4 \cdot 2^4 + x_3 \cdot 2^3 + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

is then represented by the following polynomial in the finite field $GF(2^8)$:

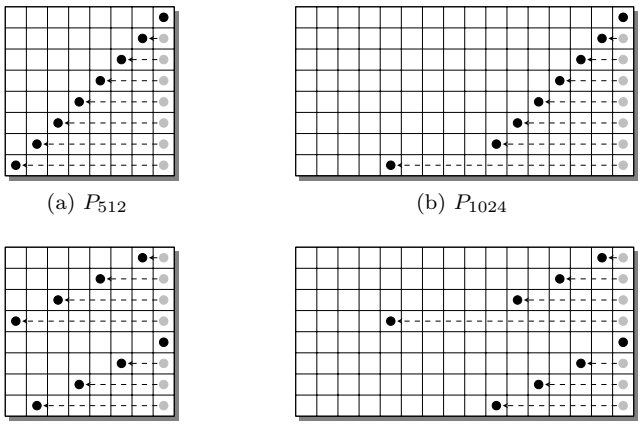
$$P(x) = x_7 \cdot \theta^7 + x_6 \cdot \theta^6 + x_5 \cdot \theta^5 + x_4 \cdot \theta^4 + x_3 \cdot \theta^3 + x_2 \cdot \theta^2 + x_1 \cdot \theta^1 + x_0$$

The multiplication $x \cdot y$ in the field $GF(2^8)$ is defined by a polynomial multiplication modulo the polynomial defining the field:

$$x \cdot y = P(x) \cdot P(y) \bmod (\theta^8 \oplus \theta^4 \oplus \theta^3 \oplus \theta \oplus 1)$$

Section 4.4 describes how the multiplications in this field are carried out efficiently in practice.

Archived copy · cryp2nova.com



(a) P_{512} (b) P_{1024}

(c) Q_{512} (d) Q_{1024}

Figure 6: The shift values used by the ShiftBytes transformation.

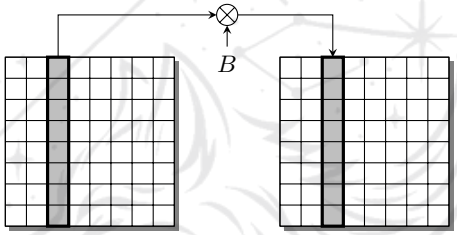


Figure 7: The MixBytes transformation multiplies each column of the state by a constant matrix B.

3 Introduction to Efficient Implementation Techniques

In this section we give a high-level overview on common efficient implementation techniques for `Grøstl`. Since `Grøstl` is an AES-based hash function, most implementation techniques developed for AES can be applied to `Grøstl` as well. The main implementation techniques for `Grøstl` are the T-table implementation, bit slicing, and byte slicing. In a parallel byte slice implementation, either the Intel AES-NI instruction or the vperm technique can be used to compute the AES S-box.

In Table 1 we list some benchmark results of `Grøstl` on current desktop processors. For details on more processors we refer to eBASH [3]. Additionally, the byte slice implementation technique has also been used to get efficient 8-bit implementations of `Grøstl`. Table 2 shows some time-memory trade-offs for 8-bit AVR implementations.

3.1 T-Table Implementation

Daemen and Rijmen have presented a table-based approach for AES in [6], which efficiently computes the combined SubBytes and MixColumns transformation. The same approach can be applied to `Grøstl`. Using this technique, at least one table lookup is needed for each S-box. The MixBytes transformation is computed in parallel for rows of the state and can be combined with the S-box lookup. This approach is most efficient if the column size matches the register size. This is the case on 32-bit platforms for AES and on 64-bits platforms for `Grøstl`. Since many current and future small-scale 32-bit processors also provide 64-bit instructions (MMX, NEON), `Grøstl` can be implemented very efficiently on these platforms using the T-table approach.

In T-table implementations, the state of `Grøstl` is stored in 64-bit registers in column ordering (see Figure 8). The AddRoundConstant transformation can be computed separately using 64-bit XORs. The computation of the SubBytes, ShiftBytes and MixBytes transformations are combined to efficiently compute

Archived copy · cryp2nova.com

Table 1: Grøstl software performance on current desktop processors sorted by their speed in cycles/byte (c/b). The byte slice implementations using AES-NI or vperm outperform table-based implementations on processors with 128-bit registers.

Hash function	Processor	Speed (c/b)	Technique
Grøstl-256	Intel Core i7-2600K	11.5	AES-NI
	AMD Phenom II X6	19.4	T-tables
	Intel Core2 Duo L9400	20.4 (22.5)	vperm (T-tables)
	Intel Core i7 620LM	23.3 (24.0)	vperm (T-tables)
	Intel Pentium M	38.8	T-tables
Grøstl-0-256	Intel Core2 Duo L9400	29.7	bit slicing
Grøstl-512	Intel Core i7-2600K	15.6	AES-NI
	Intel Core2 Duo L9400	28.9 (37.4)	vperm (T-tables)
	AMD Phenom II X6	31.7	T-tables
	Intel Core i7 620LM	33.4 (37.7)	vperm (T-tables)
	Intel Pentium M	76.1	T-tables

Table 2: Speed of three different Grøstl-256 8-bit AVR implementations in cycles/byte on ATMega163.

	HighSpeed	Balanced	LowMem
Grøstl	**469**	**530**	-
Grøstl-0	**456**	**517**	**738**
RAM	994	226	164

one 64-bit column (e.g., column 0) of Grøstl as follows:

$$\begin{bmatrix} b_{00} \\ b_{10} \\ b_{20} \\ b_{30} \\ b_{40} \\ b_{50} \\ b_{60} \\ b_{70} \end{bmatrix} = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix} \cdot \begin{bmatrix} S(a_{00}) \\ S(a_{11}) \\ S(a_{22}) \\ S(a_{33}) \\ S(a_{44}) \\ S(a_{55}) \\ S(a_{66}) \\ S(a_{77}) \end{bmatrix}$$

where $b_0 = [b_{00}, b_{10}, \cdots, b_{70}]^T$ is the resulting 64-bit value of the first column computation. The input bytes a_{ij} are extracted from the state according to the ShiftBytes transformation and the S-box $S(x)$ is applied to these bytes prior to the matrix multiplication of MixBytes. Expanding the matrix multiplication then gives:

$$\begin{bmatrix} b_{00} \\ b_{10} \\ b_{20} \\ b_{30} \\ b_{40} \\ b_{50} \\ b_{60} \\ b_{70} \end{bmatrix} = \begin{bmatrix} 02 \cdot S(a_{00}) \\ 07 \cdot S(a_{00}) \\ 05 \cdot S(a_{00}) \\ 03 \cdot S(a_{00}) \\ 05 \cdot S(a_{00}) \\ 04 \cdot S(a_{00}) \\ 03 \cdot S(a_{00}) \\ 02 \cdot S(a_{00}) \end{bmatrix} \oplus \begin{bmatrix} 02 \cdot S(a_{11}) \\ 02 \cdot S(a_{11}) \\ 07 \cdot S(a_{11}) \\ 05 \cdot S(a_{11}) \\ 03 \cdot S(a_{11}) \\ 05 \cdot S(a_{11}) \\ 04 \cdot S(a_{11}) \\ 03 \cdot S(a_{11}) \end{bmatrix} \oplus \begin{bmatrix} 03 \cdot S(a_{22}) \\ 02 \cdot S(a_{22}) \\ 02 \cdot S(a_{22}) \\ 07 \cdot S(a_{22}) \\ 05 \cdot S(a_{22}) \\ 03 \cdot S(a_{22}) \\ 05 \cdot S(a_{22}) \\ 04 \cdot S(a_{22}) \end{bmatrix} \oplus \begin{bmatrix} 04 \cdot S(a_{33}) \\ 03 \cdot S(a_{33}) \\ 02 \cdot S(a_{33}) \\ 02 \cdot S(a_{33}) \\ 07 \cdot S(a_{33}) \\ 05 \cdot S(a_{33}) \\ 03 \cdot S(a_{33}) \\ 05 \cdot S(a_{33}) \end{bmatrix} \oplus$$

$$\begin{bmatrix} 05 \cdot S(a_{44}) \\ 04 \cdot S(a_{44}) \\ 03 \cdot S(a_{44}) \\ 02 \cdot S(a_{44}) \\ 02 \cdot S(a_{44}) \\ 07 \cdot S(a_{44}) \\ 05 \cdot S(a_{44}) \\ 03 \cdot S(a_{44}) \end{bmatrix} \oplus \begin{bmatrix} 03 \cdot S(a_{55}) \\ 05 \cdot S(a_{55}) \\ 04 \cdot S(a_{55}) \\ 03 \cdot S(a_{55}) \\ 02 \cdot S(a_{55}) \\ 02 \cdot S(a_{55}) \\ 07 \cdot S(a_{55}) \\ 05 \cdot S(a_{55}) \end{bmatrix} \oplus \begin{bmatrix} 05 \cdot S(a_{66}) \\ 03 \cdot S(a_{66}) \\ 05 \cdot S(a_{66}) \\ 04 \cdot S(a_{66}) \\ 03 \cdot S(a_{66}) \\ 02 \cdot S(a_{66}) \\ 02 \cdot S(a_{66}) \\ 07 \cdot S(a_{66}) \end{bmatrix} \oplus \begin{bmatrix} 07 \cdot S(a_{77}) \\ 05 \cdot S(a_{77}) \\ 03 \cdot S(a_{77}) \\ 05 \cdot S(a_{77}) \\ 04 \cdot S(a_{77}) \\ 03 \cdot S(a_{77}) \\ 02 \cdot S(a_{77}) \\ 02 \cdot S(a_{77}) \end{bmatrix}$$

Archived copy · cryp2nova.com

which simplifies to

$$b_0 = T_0(a_{00}) \oplus T_1(a_{11}) \oplus T_2(a_{22}) \oplus T_3(a_{33}) \oplus T_4(a_{44}) \oplus T_5(a_{55}) \oplus T_6(a_{66}) \oplus T_7(a_{77})$$

where the tables $y = T_i(x)$ contain 8 to 64-bit lookups of the S-box together with the 8 multipliers of

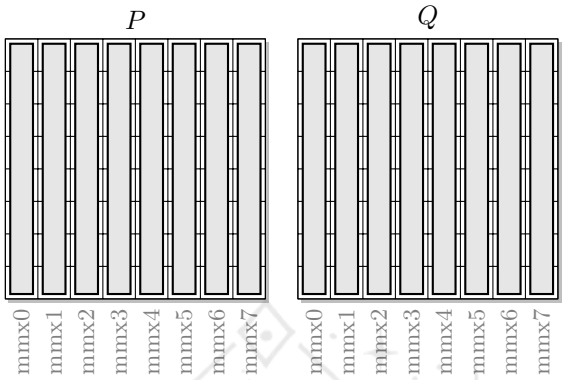


Figure 8: For the T-table approach, the Grøstl-256 state is stored column-wise in 64-bit registers.

MixBytes. For example, for the first table T_0 we get:

$$T_0(x) = \mathsf{02} \cdot S(x) \parallel \mathsf{07} \cdot S(x) \parallel \mathsf{05} \cdot S(x) \parallel \mathsf{03} \cdot S(x) \parallel \mathsf{05} \cdot S(x) \parallel \mathsf{04} \cdot S(x) \parallel \mathsf{03} \cdot S(x) \parallel \mathsf{02} \cdot S(x)$$

Extracting a single byte from a word can be implemented using a bit-shift and a masking (logical and) instruction. Then, the computation of one column consists of only 8 table lookups, 8 XOR (7 XOR for MB, 1 XOR for AC), 8 SHIFT and 8 AND instructions. On some platforms, single bytes a_{ij} can be extracted from 64-bit column words $a_j = [a_{00}, a_{10}, \ldots, a_{70}]^T$ at no cost. In this case, we can save (some of) the SHIFT and AND instructions.

The same T-table approach can also be used for efficient implementations on 32-bit processors. In this case, we split up the computation into an upper part and lower part. We need to split up the tables T_i into one table T'_i storing the upper 32 bits and one table T''_i storing the lower 32 bits. Due to the cyclic structure of the MixBytes transformation matrix, the tables T'_i can be reused to lookup also the lower 32 bits since we have $T''_i = T'_{(i+4) \bmod 8}$. Hence, we get

$$\begin{aligned} b'_0 &= T'_0(a_{00}) \oplus T'_1(a_{11}) \oplus T'_2(a_{22}) \oplus T'_3(a_{33}) \oplus T'_4(a_{44}) \oplus T'_5(a_{55}) \oplus T'_6(a_{66}) \oplus T'_7(a_{77}) \\ b''_0 &= T'_4(a_{00}) \oplus T'_5(a_{11}) \oplus T'_6(a_{22}) \oplus T'_7(a_{33}) \oplus T'_0(a_{44}) \oplus T'_1(a_{55}) \oplus T'_2(a_{66}) \oplus T'_3(a_{77}) \end{aligned}$$

with $b_0 = b'_0 \| b''_0$.

3.2 Byte Slice Implementation

Another option to implement Grøstl is a byte-wise parallel computation of columns. All round transformations except ShiftBytes and AddRoundConstant apply exactly the same computation to each column of the Grøstl state independently. Therefore, we can use a Single Instruction Multiple Data (SIMD) approach to compute these identical operations on more than one column at the same time. We call this a byte slice implementation [1] since the Grøstl state is cut into column slices of bytes. The state is stored in row ordering. Using w-bit registers, $w/8$ columns can be computed in parallel (see Figure 9). This approach is most efficient for small (8-bit) and large register sizes (128-bit and more).

A requirement for this approach to be efficient is that all round transformations of Grøstl can be parallelised using only a few w-bit SIMD instructions. AddRoundConstant and MixBytes can be computed in parallel simply using basic ALU instructions. For ShiftBytes we need a byte shuffling instruction or some mask and rotate instructions. The most difficult round transformation to parallelise is the 8-bit table lookup of SubBytes. However, using the Intel AES New Instructions extension (AES-NI) [12] or the vector-permute

Archived copy · cryp2nova.com

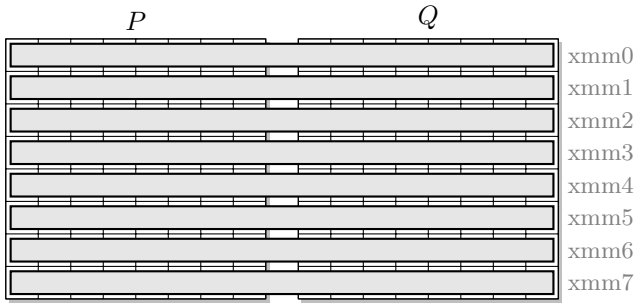


Figure 9: For the SIMD implementation, the Grøstl-256 state is stored row-wise in xmm registers to compute each column 16 times in parallel.

(vperm) approach of Hamburg [10], parallel AES S-box table lookups can be performed efficiently. Moreover, the fastest Grøstl implementation [3] is a byte slice implementation using AES-NI. For further details, on how the round transformations can be implemented, we refer to Section 4.

In a byte slice implementation, we need to use a row-ordering of the Grøstl state. However, the input bytes of the message are mapped to the Grøstl state in column-ordering. The column-ordering is a benefit for T-table based implementations but a drawback for byte slice implementations. To reduce the state transformation costs, the internal state is kept in row-ordering throughout the whole computation. Then, we only need to transform each input message block and the hash function output at the end (the IV can be stored already in row-ordering). Transforming the input message from column-ordering into row-ordering corresponds to transposing the state matrix of the input message block. Many algorithms for transposing a matrix are known and a square matrix can be transposed using only a few instructions.

3.3 Bit Slice Implementation

The fastest AES software implementations are bit slice implementations running at 7.6 cycles/byte on an Intel Core2 if multiple blocks are encrypted in parallel in counter mode [14]. Also the hash function Whirlpool which shares some similarities to Grøstl has been implemented efficiently using bit slicing techniques in [20]. Preliminary assembly implementations of Grøstl-0 show a speed of 29.7 cycles/byte on an Intel Core2 Duo processor for the computation of the hash of a single message [21]. Additionally, bit slice implementations of Grøstl-0 get even more efficient if two or more messages are hashed in parallel [4].

4 Implementing Grøstl Round Transformations

In this Section we list common techniques to efficiently implement the individual Grøstl round transformations. The listed techniques can be used on various platforms using different word sizes, as well as in hardware and in software. In most cases also special optimization techniques which combine round transformations may lead to better results on some platforms (also see Sections 5, 3.2 and 3.3).

4.1 AddRoundConstant

The AddRoundConstant transformation consists of XORs of bytes in the state with constants. In most cases, these constants will be stored using the same data structures and ordering conventions as the state itself, in which case the XORs are simply carried out word by word.

One may exploit the fact that the constants used in Q correspond to a complementation of each byte, followed by an XOR with the same constants as those used in P. Note, however, that in Q, these constants are XORed to the bottom row instead of the top row (as in P).

4.2 SubBytes

The SubBytes transformation is most simply implemented as an 8-bit table lookup.

Archived copy · cryp2nova.com

However, since the transformation corresponds to an inversion in the finite field $GF(2^8)$ followed by an affine transformation, in some scenarios it is more efficient to implement it via a computation. The most efficient known way to compute the AES S-box is using the formulas of Canright [5]. Originally developed for compact hardware implementations, Canright's formulas can also be used for the efficient computation of the AES S-box in software. The fastest known AES implementation uses these formulas to compute the S-box in bit slice mode [14].

A second efficient method to compute the AES S-box has been proposed by Hamburg in [10]. In this approach, the inverse in $GF(2^8)$ of the AES S-box is computed using small log tables of the finite field $GF(2^4)$. Small log tables can efficiently be implemented using byte shuffling instructions. Using registers containing 16 bytes, a 4-to-8 bit table lookup can be performed. For more details on this implementation we refer to the original publication [10].

The third possibility to compute SubBytes is using the Intel AES New Instructions extension (AES-NI) to the x86 instruction set [9]. This extension includes a number of instructions for computing AES rounds. The instruction `AESENCLAST`, as an example, computes the last round of AES (without any key additions), which means it computes the transformations SubBytes and ShiftRows. Hence, if this instruction is available, it can be used to compute parallel AES S-box lookups. To reduce the number of byte shuffling instructions, the computation of ShiftRows in `AESENCLAST` can be combined with the computation of ShiftBytes.

4.3 ShiftBytes

The ShiftBytes transformation simply moves bytes around within a row. Hence, this transformation can often be computed implicitly by simply changing the addressing of bytes. An implementation on a modern desktop processor might store rows of the state as a 64-bit or 128-bit word. In this case, the ShiftBytes transformation can be implemented via a simple byte shuffling instruction.

4.4 MixBytes

MixBytes consists of a multiplication of each column in the state by a constant 8×8 matrix B. All multiplications and additions needed to compute this transformation are done in the finite field $GF(2^8)$ defined by the polynomial $\theta^8 \oplus \theta^4 \oplus \theta^3 \oplus \theta \oplus 1$ ($\mathsf{11b}$ in hexadecimal notation). There are many ways to compute MixBytes and it depends on the hardware and CPU features which variant is most efficient. In the following, we explain the most important techniques.

4.4.1 Table-based Implementation

The most efficient way to implement MixBytes is using precomputed T-tables (also see Section 5). Especially since in this case, the table lookups for MixBytes can also be combined with those of the S-box. In the T-table approach, the effect of each state byte on one column (8 bytes) is precomputed and stored in a table of 256 64-bit entries. For each input byte of one column, we need a separate table. Then, e.g. the first column can be computed as follows:

$$b_0 = T_0(a_{00}) \oplus T_1(a_{11}) \oplus T_2(a_{22}) \oplus T_3(a_{33}) \oplus T_4(a_{44}) \oplus T_5(a_{55}) \oplus T_6(a_{66}) \oplus T_7(a_{77})$$

where the tables $y = T_i(x)$ contain 8 to 64-bit lookups of the S-box together with the 8 multipliers of MixBytes. Table T_0 is precomputed as follows:

$$T_0(x) = \mathsf{02} \cdot S(x) \parallel \mathsf{07} \cdot S(x) \parallel \mathsf{05} \cdot S(x) \parallel \mathsf{03} \cdot S(x) \parallel \mathsf{05} \cdot S(x) \parallel \mathsf{04} \cdot S(x) \parallel \mathsf{03} \cdot S(x) \parallel \mathsf{02} \cdot S(x)$$

Using this method, one column of MixBytes including one column of SubBytes can be computed using only 8 table-lookups and 7 64-bit XOR operations.

However, we need more instructions for ShiftBytes since byte values have to be extracted from 64-bit words. Furthermore, the T-table approach is not resistant against cache-timing and table lookups are still the bottleneck on most current processors.

Archived copy · cryp2nova.com

4.4.2 Using Double-and-Add

MixBytes can also be computed using repeated double-and-add operations. Then, we only need to XOR and efficiently multiply by 02 (see Section 4.4.3). For example, the multiplication by 05 can then be carried out by performing $02 \cdot (02 \cdot x) + x$. Using only double-and-add operations, we get the following formulas to compute each output byte b_i given the input bytes a_i of a single column:

$$\begin{aligned} b_i = 02 \cdot (02 \cdot (a_{i+3} \oplus a_{i+4} \oplus a_{i+6} \oplus a_{i+7}) \oplus a_i \oplus a_{i+1} \oplus a_{i+2} \oplus a_{i+5} \oplus a_{i+7}) \oplus \\ a_{i+2} \oplus a_{i+4} \oplus a_{i+5} \oplus a_{i+6} \oplus a_{i+7} \end{aligned} \tag{2}$$

where $i = 0, \ldots, 7$ and all the indices are taken modulo 8. Simply implementing this formula would require $8 \cdot 2 = 16$ multiplications by 02 and $8 \cdot 13 = 104$ XORs. However, the number of XORs can be significantly reduced to at least 48 (see Section 4.4.4 and Section 4.4.5).

4.4.3 The Multiplication by 02

In the finite field $GF(2^8)$, the doubling operation $02 \cdot x$ (where x is an 8-bit value) can be implemented using a left shift of x by one, followed by a conditional XOR using the irreducible polynomial 11b if an overflow occurs. When operating on bytes, the MSB is usually discarded by the shift. Hence, we first check whether the MSB is set and conditionally XOR the constant 1b after the shift. It is worth to note that in some cases the condition can also be checked efficiently by treating the byte as a *signed* value and comparing it to zero. If two's complement representation is used, the most significant bit is only set if the value x is negative.

4.4.4 Minimising the Number of XORs

By taking a look at Equation 2, we can observe that there are many terms of the form $a_i \oplus a_{i+1}$. By repeatedly computing temporary results in a tree-based form, we get an optimised way of computing MixBytes using the following set of formulas:

$$\begin{aligned} x_i &= a_i \oplus a_{i+1}, \\ y_i &= x_i \oplus x_{i+3}, \\ z_i &= x_i \oplus x_{i+2} \oplus a_{i+6}, \\ b_i &= 02 \cdot (02 \cdot y_{i+3} \oplus z_{i+7}) \oplus z_{i+4} \ . \end{aligned} \tag{3}$$

These formulas contain a minimum number of 16 multiplications by 02 and only $8 \cdot 6 = 48$ XOR operations. Furthermore, the computations are more independent, which allows a better parallelism on superscalar CPUs. For example, computing x_i is independent from any other x_j where $i \neq j$ and the same is true for the remaining temporary and final values.

4.4.5 Computing the Multiplication by 02 First

Sometimes (see Section 6.3), it can be more efficient to first compute the multiplication by 02 and 04 of all input values a_i:

$$\begin{aligned} b_i = a_{i+2} \oplus a_{i+4} \oplus a_{i+5} \oplus a_{i+6} \oplus a_{i+7} \oplus \\ 02 \cdot a_i \oplus 02 \cdot a_{i+1} \oplus 02 \cdot a_{i+2} \oplus 02 \cdot a_{i+5} \oplus 02 \cdot a_{i+7} \oplus \\ 04 \cdot a_{i+3} \oplus 04 \cdot a_{i+4} \oplus 04 \cdot a_{i+6} \oplus 04 \cdot a_{i+7} \end{aligned} \tag{4}$$

In this case, the previous optimization cannot be applied. However, it is still possible to minimize the number of XOR operations in this case as well. Since many terms $(a_i, 02 \cdot a_i, 04 \cdot a_i)$ in the computation are added to more than one result, we can save XORs by computing temporary results again. For example, the term

$$t = 02 \cdot a_0 \oplus 02 \cdot a_2 \oplus a_5 \oplus 04 \cdot a_7 \oplus a_7 \tag{5}$$

is added to b_0, b_1 and b_3. We can save many XOR operations by computing such temporary results first. In [1] the reuse of such temporary results has been optimized (see Table 3). These equations result in 16 multiplications by 02 and 58 XOR operations.

Archived copy · cryp2nova.com

Table 3: MixBytes computation with 58 XORs. A "•" denotes those inputs (a_i, $02 \cdot a_i$, $04 \cdot a_i$) which are added to get the results b_i. Superscripts denote the order in which the temporary results are computed (1 corresponds to the temporary results of Equation 5).

	a_0			a_1			a_2			a_3			a_4			a_5			a_6			a_7		
	04	02	01	04	02	01	04	02	01	04	02	01	04	02	01	04	02	01	04	02	01	04	02	01
b_0	–	$\bullet^1$	–	–	$\bullet^2$	–	–	$\bullet^1$	$\bullet^9$	$\bullet^d$	–	–	$\bullet^d$	–	$\bullet^2$	–	$\bullet^9$	$\bullet^1$	$\bullet^2$	–	$\bullet^2$	$\bullet^1$	$\bullet^2$	$\bullet^1$
b_1	$\bullet^5$	$\bullet^1$	$\bullet^5$	–	$\bullet^a$	–	–	$\bullet^1$	–	–	$\bullet^5$	$\bullet^b$	$\bullet^d$	–	–	$\bullet^5$	–	$\bullet^1$	–	$\bullet^b$	$\bullet^a$	$\bullet^1$	–	$\bullet^1$
b_2	$\bullet^5$	–	$\bullet^5$	$\bullet^7$	$\bullet^2$	$\bullet^7$	–	$\bullet^c$	–	–	$\bullet^5$	–	–	$\bullet^7$	$\bullet^2$	$\bullet^5$	–	–	$\bullet^2$	–	$\bullet^2$	–	$\bullet^2$	$\bullet^c$
b_3	–	$\bullet^1$	$\bullet^3$	$\bullet^7$	–	$\bullet^7$	$\bullet^3$	$\bullet^1$	$\bullet^3$	–	$\bullet^3$	–	–	$\bullet^7$	–	–	$\bullet^3$	$\bullet^1$	$\bullet^d$	–	–	$\bullet^1$	–	$\bullet^1$
b_4	$\bullet^d$	–	$\bullet^3$	–	$\bullet^a$	$\bullet^4$	$\bullet^3$	–	$\bullet^3$	$\bullet^4$	$\bullet^3$	$\bullet^4$	–	$\bullet^4$	–	–	$\bullet^3$	–	–	$\bullet^4$	$\bullet^a$	$\bullet^d$	–	–
b_5	$\bullet^d$	–	–	$\bullet^6$	–	$\bullet^4$	$\bullet^d$	$\bullet^c$	$\bullet^9$	$\bullet^4$	–	$\bullet^4$	$\bullet^6$	$\bullet^4$	$\bullet^6$	–	$\bullet^9$	–	–	$\bullet^4$	–	–	$\bullet^6$	$\bullet^c$
b_6	–	$\bullet^8$	$\bullet^3$	$\bullet^6$	–	–	$\bullet^3$	–	$\bullet^3$	–	$\bullet^3$	$\bullet^b$	$\bullet^6$	–	$\bullet^6$	$\bullet^8$	$\bullet^3$	$\bullet^8$	–	$\bullet^b$	–	–	$\bullet^6$	–
b_7	–	$\bullet^8$	–	–	$\bullet^2$	$\bullet^4$	–	–	–	$\bullet^4$	–	$\bullet^4$	–	$\bullet^4$	$\bullet^2$	$\bullet^8$	–	$\bullet^8$	$\bullet^2$	$\bullet^4$	$\bullet^2$	–	$\bullet^2$	–

4.4.6 Other Possible Optimisations

Another possibility to reduce the computation costs of MixBytes is to use a different basis of multipliers. Instead of $(01, 02, 04)$, Çalik used in his implementation the basis $(03, 05, 07)$ [4]. In this case, the Hamming weight of the multiplication constants reduces significantly. Unfortunately, this basis does not result in less than 58 XORs and needs 16 multiplications by 02 as well.

In platforms where many registers are available and multiplications by 02 are cheap, it can be of an advantage to compute the results of each multiplier separately [1]. In this case, the results $b_{i,1}$ of multiplier 01 can be reused to compute those for multiplier 02 since $b_{i,2} = 02 \cdot b_{i+3 \bmod 8,1}$ (see Table 5). For multipliers 01 and 04 ($b_{i,4}$) temporary results are used to further minimize the number of XORs. This approach results in 24 multiplications by 2 and 48 XORs.

5 T-Table Implementations

In this section we present some example implementations which use the T-table approach for Grøstl. This implementation technique is most efficient on 64-bit platforms. On 32-bit platforms, the number of necessary instructions double. In the following listing, we provide an unoptimized C code segment for the computation of one round of permutation P of Grøstl-256:

```
// AC
a0  = b0 ^ c0;
a1  = b1 ^ c1;
a2  = b2 ^ c2;
a3  = b3 ^ c3;
a4  = b4 ^ c4;
a5  = b5 ^ c5;
a6  = b6 ^ c6;
a7  = b7 ^ c7;
// SB+SH+MB (column 0 of P)
b0  = T0[(a0>>0 ) & 0xff];
b0 ^= T1[(a1>>8 ) & 0xff];
b0 ^= T2[(a2>>16) & 0xff];
b0 ^= T3[(a3>>24) & 0xff];
b0 ^= T4[(a4>>32) & 0xff];
b0 ^= T5[(a5>>40) & 0xff];
b0 ^= T6[(a6>>48) & 0xff];
b0 ^= T7[(a7>>56) & 0xff];
// SB+SH+MB (column 1 of P)
b1  = T0[(a0>>8 ) & 0xff];
b1 ^= T1[(a1>>16) & 0xff];
b1 ^= T2[(a2>>24) & 0xff];
b1 ^= T3[(a3>>32) & 0xff];
b1 ^= T4[(a4>>40) & 0xff];
b1 ^= T5[(a5>>48) & 0xff];
b1 ^= T6[(a6>>56) & 0xff];
b1 ^= T7[(a7>>0 ) & 0xff];
// SB+SH+MB (column 2 of P)
b2  = T0[(a0>>16) & 0xff];
b2 ^= T1[(a1>>24) & 0xff];
b2 ^= T2[(a2>>32) & 0xff];
b2 ^= T3[(a3>>40) & 0xff];
b2 ^= T4[(a4>>48) & 0xff];
b2 ^= T5[(a5>>56) & 0xff];
b2 ^= T6[(a6>>0 ) & 0xff];
b2 ^= T7[(a7>>8 ) & 0xff];
// SB+SH+MB (column 3 of P)
b3  = T0[(a0>>24) & 0xff];
b3 ^= T1[(a1>>32) & 0xff];
b3 ^= T2[(a2>>40) & 0xff];
b3 ^= T3[(a3>>48) & 0xff];
b3 ^= T4[(a4>>56) & 0xff];
b3 ^= T5[(a5>>0 ) & 0xff];
b3 ^= T6[(a6>>8 ) & 0xff];
b3 ^= T7[(a7>>16) & 0xff];
// SB+SH+MB (column 4 of P)
b4  = T0[(a0>>32) & 0xff];
b4 ^= T1[(a1>>40) & 0xff];
b4 ^= T2[(a2>>48) & 0xff];
b4 ^= T3[(a3>>56) & 0xff];
b4 ^= T4[(a4>>0 ) & 0xff];
b4 ^= T5[(a5>>8 ) & 0xff];
b4 ^= T6[(a6>>16) & 0xff];
b4 ^= T7[(a7>>24) & 0xff];
// SB+SH+MB (column 5 of P)
b5  = T0[(a0>>40) & 0xff];
b5 ^= T1[(a1>>48) & 0xff];
b5 ^= T2[(a2>>56) & 0xff];
b5 ^= T3[(a3>>0 ) & 0xff];
b5 ^= T4[(a4>>8 ) & 0xff];
```


Archived copy · cryp2nova.com

```
b5 ^= T5[(a5>>16) & 0xff];
b5 ^= T6[(a6>>24) & 0xff];
b5 ^= T7[(a7>>32) & 0xff];
// SB+SH+MB (column 6 of P)
b6  = T0[(a0>>48) & 0xff];
b6 ^= T1[(a1>>56) & 0xff];
b6 ^= T2[(a2>>0 ) & 0xff];
```

```
b6 ^= T3[(a3>>8 ) & 0xff];
b6 ^= T4[(a4>>16) & 0xff];
b6 ^= T5[(a5>>24) & 0xff];
b6 ^= T6[(a6>>32) & 0xff];
b6 ^= T7[(a7>>40) & 0xff];
// SB+SH+MB (column 7 of P)
b7  = T0[(a0>>56) & 0xff];
```

```
b7 ^= T1[(a1>>0 ) & 0xff];
b7 ^= T2[(a2>>8 ) & 0xff];
b7 ^= T3[(a3>>16) & 0xff];
b7 ^= T4[(a4>>24) & 0xff];
b7 ^= T5[(a5>>32) & 0xff];
b7 ^= T6[(a6>>40) & 0xff];
b7 ^= T7[(a7>>48) & 0xff];
```

A number of optimized C implementations have been published for Grøstl. The most important ones are the implementations submitted to NIST by the designers [8] and the crypto library sphlib3.0 [19]. Although sphlib is not fully optimized (e.g. the round constants are added byte-by-byte), it has a good performance on many constrained (32-bit) devices. In the following, we present optimized assembly implementations on a few example platforms which can serve as a reference for further T-table optimizations.

5.1 64-bit Processors

A T-table implementation of Grøstl on 64-bit processors needs 8 table lookups, 8 XOR, and at most 8 SHIFT and 8 AND instructions per column computation (see Section 5). However, on x86 CPUs, we can reduce the ALU instructions to 8 XOR, 8 MOV and 3 SHIFT instructions per column as follows.

Let rax contain column 0, where the least significant 8 bits correspond to the top byte. Now, the following two instructions each extract one byte out of rax:

```
movzbl edi, al   ; put least sig. byte (row 0) in edi
movzbl esi, ah   ; put second-least sig. byte (row 1) in esi
```

After this, edi is used as index to lookup table T_0, esi is used as index to lookup table T_7 (since ShiftBytes will move it to column 7). The results are stored in (or XORed to, for subsequent bytes) the new columns 0 and 7. Then, the register rax is shifted 16 bits to the right. Hence, next time we carry out the above two instructions again, edi will contain the byte in row 2, and esi will contain the byte in row 3. Note that we work on two columns at the same time in order to maximize instruction level parallelism.

Intel desktop processors prior to the Sandy Bridge architecture have one memory load and one store units and up to three arithmetic logic units (ALUs). This implies that the load instructions are dominant, and the maximal throughput is 1 cycle/byte for each round of Grøstl. This results in 20 cycles/byte for Grøstl-256 and 28 cycles/byte for Grøstl-512. The results given in Table 1 show that the speed of Grøstl is very close to this bound on the Intel Core2 Duo processor.

Since AMD Opteron and Intel Sandy Bridge processors have two memory load units, up to two parallel table lookups are possible within each CPU cycle. Assuming that single bytes can be extracted efficiently using one instruction, we get 0.5 cycles/byte for the loads and $(8 + 8 + 3)/8/3 = 0.79$ cycles/byte for the ALU instructions. Hence, the ALU instructions are dominant and we get a lower bound of 15.8 cycles/byte for Grøstl-256 and 22.1 cycles/byte for Grøstl-512. However, these ideal results are difficult to achieve in practice and our implementations are not approaching the theoretical lower bounds yet (see Table 1).

5.2 32-bit Processors

Since the number of table lookups and XORs double for the 32-bit T-table implementation, we get a lower bound of 40 cycles/byte for Grøstl-256 and 56 cycles/byte for Grøstl-512 if no parallel table lookups are possible. However, many current and future 32-bit processors have 64-bit instruction set extensions such as MMX for Intel/AMD processors [13] and NEON for ARM processors [2]. Using these extended instructions, we can get a speed close to 20 cycles/byte also on 32-bit x86 CPUs. A similar improvement can be expected from new NEON implementations.

6 SIMD-based Byte Slicing Implementations

In this section we describe a few concrete examples of the byte slicing implementation of Grøstl. We chose to present an implementation on an Intel-64 platform with SSSE3 instruction set as an example of popular,

Archived copy · cryp2nova.com

modern desktop-class CPU. We show an implementation taking advantage of the AES-NI instruction set present in Intel Core iX and Sandy Bridge processors. Byte sliced implementation with AES-NI instructions is currently the fastest Grøstl implementation [3] on Intel platforms.

We also discuss the vperm implementation, an alternative for processors not equipped with AES-NI instructions. In this case, the SubBytes transformation can still be implemented efficiently using only generic SSSE3 instructions.

6.1 Transposing the Input Message

Transforming the input message from column-ordering into row-ordering corresponds to transposing the input message block. Many algorithms for transposing a matrix are known and a square matrix can be transposed using only a few PUNPCK instructions [11]. If we store the whole Grøstl-256 state (P and Q) in 128-bit registers, we get an 8x16 rectangular matrix. Hence, additional byte shuffling (PSHUFB) and move (MOV) instructions are needed to transpose the input message [1].

6.2 Using AES-NI

In this section we describe the details of the fastest known Grøstl implementation using the Intel AES-NI extension. Together with Intel AVX instructions a speed of less than 10 cycles/byte can be reached.

```
// AC
pxor       xmm0, [CONST0]
pxor       xmm1, [CONST1]
pxor       xmm2, [CONST2]
pxor       xmm3, [CONST3]
pxor       xmm4, [CONST4]
pxor       xmm5, [CONST5]
pxor       xmm6, [CONST6]
pxor       xmm7, [CONST7]
// SH (with AES ShiftRowsInv)
pshufb     xmm0, [SIGMA0]
pshufb     xmm1, [SIGMA1]
pshufb     xmm2, [SIGMA2]
pshufb     xmm3, [SIGMA3]
pshufb     xmm4, [SIGMA4]
pshufb     xmm5, [SIGMA5]
pshufb     xmm6, [SIGMA6]
pshufb     xmm7, [SIGMA7]
// SB (with AES ShiftRows)
pxor       xmm8, xmm8
aesenclast xmm0, xmm8
aesenclast xmm1, xmm8
aesenclast xmm2, xmm8
aesenclast xmm3, xmm8
aesenclast xmm4, xmm8
aesenclast xmm5, xmm8
aesenclast xmm6, xmm8
aesenclast xmm7, xmm8
// MB (y_i = a_{i+6})
movdqa xmm14, xmm0");
movdqa xmm15, xmm1");
movdqa xmm8, xmm2");
movdqa xmm9, xmm3");
movdqa xmm10, xmm4");
movdqa xmm11, xmm5");
movdqa xmm12, xmm6");
movdqa xmm13, xmm7");
```

```
// MB (t_i = a_i + a_{i+1})
pxor xmm0, xmm1");
pxor xmm1, xmm2");
pxor xmm2, xmm3");
pxor xmm3, xmm4");
pxor xmm4, xmm5");
pxor xmm5, xmm6");
pxor xmm6, xmm7");
pxor xmm7, xmm14");
// MB (y_i = a_{i+6} + t_i)
pxor xmm8, xmm4");
pxor xmm9, xmm5");
pxor xmm10, xmm6");
pxor xmm11, xmm7");
pxor xmm12, xmm0");
pxor xmm13, xmm1");
pxor xmm14, xmm2");
pxor xmm15, xmm3");
// MB (y_i = y_i + t_{i+2})
pxor xmm14, xmm4");
pxor xmm15, xmm5");
pxor xmm8, xmm6");
pxor xmm9, xmm7");
pxor xmm10, xmm0");
pxor xmm11, xmm1");
pxor xmm12, xmm2");
pxor xmm13, xmm3");
movaps [TMP_Y4], xmm8");
movaps [TMP_Y5], xmm9");
// MB (x_i = t_i + t_{i+3})
movdqa xmm8, xmm0");
movdqa xmm9, xmm1");
movaps [TMP_T2], xmm2");
pxor xmm0, xmm3");
pxor xmm1, xmm4");
pxor xmm2, xmm5");
pxor xmm3, xmm6");
```

```
pxor xmm4, xmm7");
pxor xmm5, xmm8");
pxor xmm6, xmm9");
pxor xmm7, [TMP_T2]");
// MB (z_i = 02 * x_i)
movaps xmm9, [ALL_1B]");
MUL2(a0, b0, b1);
MUL2(a1, b0, b1);
MUL2(a2, b0, b1);
MUL2(a3, b0, b1);
MUL2(a4, b0, b1);
MUL2(a5, b0, b1);
MUL2(a6, b0, b1);
MUL2(a7, b0, b1);
// MB (w_i = z_i + y_{i+4})
pxor xmm0, [TMP_Y4]");
pxor xmm1, [TMP_Y5]");
pxor xmm2, xmm10");
pxor xmm3, xmm11");
pxor xmm4, xmm12");
pxor xmm5, xmm13");
pxor xmm6, xmm14");
pxor xmm7, xmm15");
// MB (v_i = 02 * w_i)
MUL2(a0, b0, b1);
MUL2(a1, b0, b1);
MUL2(a2, b0, b1);
MUL2(a5, b0, b1);
MUL2(a6, b0, b1);
MUL2(a7, b0, b1);
MUL2(a3, b0, b1);
MUL2(a4, b0, b1);
// MB (b_i = v_{i+3} + y_{i+4})
pxor xmm13, xmm0");
pxor xmm14, xmm1");
pxor xmm15, xmm2");
pxor xmm10, xmm5");
```


Archived copy · cryp2nova.com

```
pxor xmm11, xmm6");
pxor xmm12, xmm7");
```

```
movaps xmm8, [TMP_Y4]");
movaps xmm9, [TMP_Y5]");
```

```
pxor xmm8, xmm3");
pxor xmm9, xmm4");
```

6.2.1 AddRoundConstant

The AddRoundConstant transformation XORs a round-dependent row-wise constant to the first row in P and the last row in Q, and a round-independent constant to each row of Q. Since the `Grøstl` state is stored in row-ordering, these constants can be added efficiently in parallel to each column of the state. For example, the constants of `Grøstl`-256 are added as follows:

```
movaps xmm8, [0xffffffffffffffff0000000000000000]
pxor   xmm0, [ROUND_CONST_P0]
pxor   xmm1, xmm8
pxor   xmm2, xmm8
pxor   xmm3, xmm8
pxor   xmm4, xmm8
pxor   xmm5, xmm8
pxor   xmm6, xmm8
pxor   xmm7, [ROUND_CONST_Q7]
```

6.2.2 SubBytes

SubBytes is usually the most difficult transformation to implement efficiently in a byte slice implementation. As already mentioned, for w-bit registers we need an efficient method to compute $w/8$ parallel AES S-box lookups. This results in only one (parallel) table lookup in the case of 8-bit implementations ($w = 8$). Unfortunately, for larger register sizes, parallel table lookups are usually non-trivial.

Although `Grøstl` does not use the same MDS matrix as the AES, `Grøstl` can still take advantage of the Intel AES New Instructions extension (AES-NI). Since no MixColumns transformation is applied in the last round of the AES, Intel also provides an `AESENCLAST` instruction. This instruction is able to compute 16 AES S-boxes in parallel with a throughput of one cycle and a latency of 4 cycles. The byte shuffling of the `AESENCLAST` instruction can be combined with an extra byte shuffling to perform the ShiftBytes transformation of `Grøstl` (see Section 6.2.3).

6.2.3 ShiftBytes

Since ShiftBytes just moves bytes within one row of `Grøstl`, this transformation can be implemented using only byte shuffling instructions. If `AESENCLAST` is used to compute the S-box lookups, we need to take the ShiftRows transformation of the last round in AES into account. Note that any ShiftBytes rotation constants can be used for P and Q at no additional cost. The resulting instructions for the combined SubBytes and ShiftBytes transformation of `Grøstl`-256 is given below:

```
pxor       xmm8, xmm8
pshufb     xmm0, [03060a0d080205090c0f0104070b0e00]
aesenclast xmm0, xmm8
pshufb     xmm1, [04070c0f0a03060b0e090205000d0801]
aesenclast xmm1, xmm8
pshufb     xmm2, [05000e090c04070d080b0306010f0a02]
aesenclast xmm2, xmm8
pshufb     xmm3, [0601080b0e05000f0a0d040702090c03]
aesenclast xmm3, xmm8
pshufb     xmm4, [0702090c0f0601080b0e0500030a0d04]
aesenclast xmm4, xmm8
pshufb     xmm5, [00030b0e0907020a0d080601040c0f05]
aesenclast xmm5, xmm8
pshufb     xmm6, [01040d080b00030c0f0a0702050e0906]
aesenclast xmm6, xmm8
pshufb     xmm7, [02050f0a0d01040e090c000306080b07]
aesenclast xmm7, xmm8
```


Archived copy · cryp2nova.com

6.2.4 MixBytes

The MixBytes transformation is the most costly transformation in a byte sliced implementation of Grøstl. We need to combine the 8 rows of the Grøstl state according to the MixBytes matrix multiplication. For processors that can execute more than one SIMD instruction in parallel, a MixBytes computation with the minimum number of instructions does not necessarily result in the fastest implementation. For example, modern desktop CPUs can compute three independent SIMD XORs in parallel. However, when the MixBytes computation contains many long chains of dependencies, the ALU parallelism cannot be fully utilised.

To compute MixBytes using SIMD instructions, we use the formulas of Section 4.4.4. This variant contains the minimal possible number of 16 multiplications and 48 XORs, and the computations are also mostly independent. To illustrate this approach in details, we consider the implementation of MixBytes using equations (6) on Intel-64 architecture with SSE2 instructions. In order to closer reflect the constraints of the assembler code, we rewrite the last equation to contain only one type of operation in each pass. This yields the following sequential formulas:

$$\begin{aligned} t_i &= a_i \oplus a_{i+1}, \\ y_i &= a_{i+6} \oplus t_i, \\ y_i &= y_i \oplus t_{i+2}, \\ x_i &= t_i \oplus t_{i+3}, \\ z_i &= 02 \cdot x_i, \\ w_i &= z_i \oplus y_{i+4}, \\ v_i &= 02 \cdot w_i, \\ b_i &= v_{i+3} \oplus y_{i+4}. \end{aligned} \tag{6}$$

The main challenge is to minimise the number of register spills when performing the computation in 16 xmm registers and reorder instructions in a way ensuring maximal instruction throughput. The algorithm shown in Table 4 achieves this with only four spills that are not on a critical path and therefore can be masked by other operations. We start with $a_0, \ldots, a_7$ in registers xmm0,..., xmm7 and keep building $b_0, \ldots, b_7$ in xmm8,...,xmm15. The byte-wise multiplication by 02 of the content of xmm{i} is done by the sequence of five instructions

```
pxor    xmm{j}, xmm{j}   ; clear register
pcmpgtb xmm{j}, xmm{i}   ; comparing with 0 sets 0xff in bytes that correspond to MSB bits set
paddb   xmm{i}, xmm{i}   ; byte-wise shift left by one position
pand    xmm{j}, xmm{k}   ; pick only those 0x1b that correspond to MSB bit set in xmm{i}
pxor    xmm{i}, xmm{j}   ; and XOR reduction polynomial to the result where necessary
```

that requires an extra register xmm{j} as a scratch space and xmm{k} containing the constant reduction value of 1b1b...1b. To get those extra registers, we can temporarily spill xmm8, xmm9 since they hold the values y_4, y_5 which will not be used in a critical path of the computation.

When AVX instructions are available (starting from Intel Sandy Bridge) we can use three-operand instructions to reduce the number of instructions required by the multiplication to four, but the advantage is smaller than one instruction since Sandy Bridge cores recognize XORing to clear register and do not issue μops in that case anyway.

6.3 Using Vperm

Even if no AES-NI are available, we can still implement Grøstl efficiently using SIMD instructions. We just need a different method to parallelize the S-box lookups in SubBytes. One such method has been proposed by Hamburg [10] and uses vector permute (vperm) instructions to compute the inversion and affine transformation of the AES S-box.

In the following, we describe a Grøstl implementation using the vperm idea. The resulting implementation needs at least SSSE3 instructions and thus, also runs on the NIST reference platform. The resulting speed is comparable with the T-table implementation.

Archived copy · cryp2nova.com

Table 4: Optimised computation of MixBytes on a Intel-64 machine with SSE2 instructions. Each row describes eight operations for $i = 0, \ldots, 7$. Left columns show the content of two banks of registers with updated values shown in bold and the rightmost column describes the performed operation. MUL2(xmma,xmmb,xmmc) doubles the content of xmma using xmmb as scratch and assuming xmmc contains 1b..1b.

xmm0 ... xmm7	xmm8 ... xmm15	operation	equation
$a_0 a_1 \ldots a_7$	—		
$a_0 a_1 \ldots a_7$	$\mathbf{a_2\ a_3 \ldots a_1}$	movdqa xmm$\{i+8\}$, xmm$\{(i+2) \bmod 8\}$	
$\mathbf{t_0 t_1 \ldots t_7}$	$a_2\ a_3 \ldots a_1$	pxor xmm$\{i\}$, xmm$\{(i+1) \bmod 8\}$	$t_i = a_i \oplus a_{i+1}$
$t_0 t_1 \ldots t_7$	$\mathbf{y_4\ y_5 \ldots y_3}$	pxor xmm$\{i+8\}$, xmm$\{(i+4) \bmod 8\}$ pxor xmm$\{i+8\}$, xmm$\{(i+6) \bmod 8\}$	$y_i = a_{i+6} \oplus t_i$ $y_i = y_i \oplus t_{i+2}$
$\mathbf{x_0 x_1 \ldots x_7}$	$y_4\ y_5 \ldots y_3$	spill t_0, t_1, t_2 to memory in this computation: pxor xmm$\{i\}$, xmm$\{(i+3) \bmod 8\}$	$x_i = t_i \oplus t_{i+3}$
	y_4 1b..1b $y_6 \ldots y_3$	spill xmm8, xmm9 to memory, xmm9← 0x1b...1b	
$\mathbf{z_0 z_1 \ldots z_7}$	— 1b..1b $y_6 \ldots y_3$	MUL2(xmm$\{i\}$, xmm8, xmm9)	$z_i = 02 \cdot x_i$
$\mathbf{w_0 w_1 \ldots w_7}$	— 1b..1b $y_6 \ldots y_3$	pxor xmm$\{i\}$, xmm$\{(i+8)\}$, y_4, y_5 from memory	$w_i = z_i \oplus y_{i+4}$
$\mathbf{v_0 v_1 \ldots v_7}$	— 1b..1b $y_6 \ldots y_3$	MUL2(xmm$\{i\}$, xmm8, xmm9)	$v_i = 02 \cdot w_i$
$v_0 v_1 \ldots v_7$	$\mathbf{y_4\ y_5}\ y_6 \ldots y_3$	reload y_4, y_5 back to xmm8, xmm9	
$v_0 v_1 \ldots v_7$	$\mathbf{b_0\ b_1 \ldots b_7}$	pxor xmm$\{i+8\}$, xmm$\{(i+3) \bmod 8\}$	$b_i = v_{i+3} \oplus y_{i+4}$

The state is stored in row-ordering and hence, the input transformation of the message block can be performed as described in Section 6.1. In the following, the computation of SubBytes and MixBytes are somewhat merged. Therefore, we swap the order of AddRoundConstant and ShiftBytes for an easier description.

6.3.1 AddRoundConstant

The AddRoundConstant implementation can be implemented exactly using the same instructions as in the AES-NI implementation. However, since the vperm implementation uses a different basis, the constants need to be transformed to this basis as well. The resulting constants can be precomputed and stored in memory as well. For the specific constants, we refer to the actual vperm implementation of Grøstl.

6.3.2 ShiftBytes

ShiftBytes is computed using single byte shuffle instructions for each row of P and Q. For example, the used rotation constants for the PSHUFB instruction of Grøstl-256 are given in the following assembly code listing:

```
pshufb xmm0, [0x080f0e0d0c0b0a090706050403020100]
pshufb xmm1, [0x0a09080f0e0d0c0b0007060504030201]
pshufb xmm2, [0x0c0b0a09080f0e0d0100070605040302]
pshufb xmm3, [0x0e0d0c0b0a09080f0201000706050403]
pshufb xmm4, [0x0f0e0d0c0b0a09080302010007060504]
pshufb xmm5, [0x09080f0e0d0c0b0a0403020100070605]
pshufb xmm6, [0x0b0a09080f0e0d0c0504030201000706]
pshufb xmm7, [0x0d0c0b0a09080f0e0605040302010007]
```

6.3.3 SubBytes

In the vperm implementation, the inverse in $GF(2^8)$ of the AES S-box is computed using small log tables of the finite field $GF(2^4)$. To efficiently compute these log tables, the 128-bit PSHUFB instruction of SSSE3 is used as a 4-to-8 bit lookup table. For more details, we refer to the original paper [10]. The first vperm implementation has been published by Çalik [4] which served as a reference for our optimized implementation.

Using the vperm implementation, 16 AES S-boxes can be computed in parallel within less than 10 cycles. An additional advantage of this implementation is that we can multiply the resulting outputs by constants in

Archived copy · cryp2nova.com

$GF(2^8)$ almost without no additional cost. Hence, the vperm implementation of SubBytes actually returns the values $S(x_i)$, $02 \cdot S(x_i)$ and $04 \cdot S(x_i)$ for each of the 16 input bytes x_i.

6.3.4 MixBytes

Since the multiplication by 02 and 04 of the input bytes to MixBytes are already computed in SubBytes, the formulas resulting in only 48 XORs of Section 4.4.4 cannot be used. However, we can use the method of Section 4.4.5 which minimizes the number of XORs once the multiplications have already been performed. Note that this approach is still more efficient since the 16 multiplications by 02 are much more expensive if computed using 5 instructions each (see Section 6.2.4) than the additional 18 XORs.

7 Conclusions

In this paper, we have shown the details of the currently best known Grøstl implementations. Using AES-NI extensions and AVX instructions, we are able to implement Grøstl-256 with close to 10 cycles/byte. Furthermore, the design of Grøstl also provides many possibilities for efficient implementation techniques. We have presented the most important methods and hope that they will serve as an inspiration for further optimizations. Especially the vperm implementation has some room for improvements, on x86 CPUs as well as on new platforms. For example, NEON byte permute instructions can be used to speed-up Grøstl on new ARM platforms.

References

[1] K. Aoki, G. Roland, Y. Sasaki, and M. Schläffer. Byte Slicing Grøstl – Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Grøstl. In J. Lopez and P. Samarati, editors, *SECRYPT 2011, Proceedings*, pages 124–133. SciTePress, 2011.

[2] ARM Limited. NEON, March 2011. Available online: http://www.arm.com/products/processors/technologies/neon.php.

[3] D. J. Bernstein and T. Lange. eBASH: ECRYPT Benchmarking of All Submitted Hashes, January 2011. Available online: http://bench.cr.yp.to/ebash.html.

[4] Ç. Çalik. Multi-stream and Constant-time SHA-3 Implementations. NIST hash function mailing list, December 2010. Retrieved May 03, 2010, from http://www.metu.edu.tr/~ccalik/software.html#sha3.

[5] D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *CHES*, volume 3659 of *LNCS*, pages 441–455. Springer, 2005.

[6] J. Daemen and V. Rijmen. AES Proposal: Rijndael. NIST AES Algorithm Submission, September 1999. Available online: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[7] I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[8] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011. Available: http://www.groestl.info (2011/11/25).

[9] S. Gueron and Intel Corp. Intel® Advanced Encryption Standard (AES) Instructions Set, 2010. Retrieved December 21, 2010, from http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/.

[10] M. Hamburg. Accelerating AES with Vector Permute Instructions. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 18–32. Springer, 2009.

Archived copy · cryp2nova.com

[11] Intel Corporation. Using MMX Instructions to Transpose a Matrix, 1996. Available online: ftp://download.intel.com/ids/mmx/MMX_App_Transpose_Matrix.pdf.

[12] Intel Corporation. Intel Advanced Encryption Standard Instructions (AES-NI), March 2011. Available online: http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/.

[13] Intel Corporation. Pentium Processors with MMX Technology, March 2011. Available online: http://edc.intel.com/Platforms/Previous/Processors/Pentium-MMX/.

[14] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009.

[15] R. C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[16] National Institute of Standards and Technology. FIPS PUB 197: Advanced Encryption Standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce, November 2001. Available online: http://www.itl.nist.gov/fipspubs.

[17] National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce, November 2001.

[18] National Institute of Standards and Technology. Cryptographic Hash Project, 2007. Available online at http://www.nist.gov/hash-competition.

[19] T. Pornin. sphlib 3.0. Available: http://www.saphir2.com/sphlib/files/sphlib-3.0.zip (2011/11/25).

[20] K. Scheibelhofer. A Bit-Slice Implementation of the Whirlpool Hash Function. In M. Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 385–401. Springer, 2007.

[21] S. Tillich. Personal communication, 2008.

A Another MixBytes Computation Variant

CRYP2NOVA

Archived copy · cryp2nova.com

Table 5: The MixBytes computation separated for factors 01, 02 and 04. a_i denote the input bytes and $b_i = b_{i,1} \oplus b_{i,2} \oplus b_{i,4}$ are the output bytes. A "•" marks those inputs (a_i, $02 \cdot a_i$, $04 \cdot a_i$) which are added to get the intermediate results $b_{i,j}$. Superscripts denote the order in which temporary values are computed. The results for factor 02 are computed by multiplying the results of factor 01 by 02 where $b_{i,2} = 02 \cdot b_{i+3 \bmod 8,1}$.

	$01 \cdot a_0$	$01 \cdot a_1$	$01 \cdot a_2$	$01 \cdot a_3$	$01 \cdot a_4$	$01 \cdot a_5$	$01 \cdot a_6$	$01 \cdot a_7$
$b_{0,1}$	–	–	$\bullet^0$	–	$\bullet^2$	$\bullet^0$	$\bullet^2$	•
$b_{1,1}$	$\bullet^1$	–	–	$\bullet^1$	–	•	$\bullet^a$	•
$b_{2,1}$	$\bullet^b$	$\bullet^3$	–	–	$\bullet^2$	–	$\bullet^2$	$\bullet^3$
$b_{3,1}$	$\bullet^b$	$\bullet^3$	$\bullet^0$	–	–	$\bullet^0$	–	$\bullet^3$
$b_{4,1}$	$\bullet^1$	•	•	$\bullet^1$	–	–	$\bullet^a$	–
$b_{5,1}$	–	$\bullet^3$	•	•	•	–	–	$\bullet^3$
$b_{6,1}$	$\bullet^1$	–	$\bullet^0$	$\bullet^1$	•	$\bullet^0$	–	–
$b_{7,1}$	–	•	–	•	$\bullet^2$	•	$\bullet^2$	–

	$02 \cdot a_0$	$02 \cdot a_1$	$02 \cdot a_2$	$02 \cdot a_3$	$02 \cdot a_4$	$02 \cdot a_5$	$02 \cdot a_6$	$02 \cdot a_7$	$=$
$b_{0,2}$	•	•	•	–	–	•	–	•	$02 \cdot b_{3,1}$
$b_{1,2}$	•	•	•	•	–	–	•	–	$02 \cdot b_{4,1}$
$b_{2,2}$	–	•	•	•	•	–	–	•	$02 \cdot b_{5,1}$
$b_{3,2}$	•	–	•	•	•	•	–	–	$02 \cdot b_{6,1}$
$b_{4,2}$	–	•	–	•	•	•	•	–	$02 \cdot b_{7,1}$
$b_{5,2}$	–	–	•	–	•	•	•	•	$02 \cdot b_{0,1}$
$b_{6,2}$	•	–	–	•	–	•	•	•	$02 \cdot b_{1,1}$
$b_{7,2}$	•	•	–	–	•	–	•	•	$02 \cdot b_{2,1}$

	$04 \cdot a_0$	$04 \cdot a_1$	$04 \cdot a_2$	$04 \cdot a_3$	$04 \cdot a_4$	$04 \cdot a_5$	$04 \cdot a_6$	$04 \cdot a_7$
$b_{0,4}$	–	–	–	$\bullet^0$	$\bullet^1$	–	$\bullet^0$	$\bullet^1$
$b_{1,4}$	$\bullet^2$	–	–	–	$\bullet^1$	$\bullet^2$	–	$\bullet^1$
$b_{2,4}$	$\bullet^2$	$\bullet^3$	–	–	–	$\bullet^2$	$\bullet^3$	–
$b_{3,4}$	–	$\bullet^3$	$\bullet^4$	–	–	–	$\bullet^3$	$\bullet^4$
$b_{4,4}$	$\bullet^5$	–	$\bullet^4$	$\bullet^5$	–	–	–	$\bullet^4$
$b_{5,4}$	$\bullet^5$	$\bullet^6$	–	$\bullet^5$	$\bullet^6$	–	–	–
$b_{6,4}$	–	$\bullet^6$	$\bullet^7$	–	$\bullet^6$	$\bullet^7$	–	–
$b_{7,4}$	–	–	$\bullet^7$	$\bullet^0$	–	$\bullet^7$	$\bullet^0$	–

Archived copy · cryp2nova.com