BYTE SLICING GRØSTL
Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Grøstl

Kazumaro Aoki[1*], Günther Roland[2], Yu Sasaki[1] and Martin Schläffer[2]
[1]*NTT Corporation, Tokyo, Japan*
[2]*IAIK, Graz University of Technology, Graz, Austria*

Keywords: Hash function, SHA-3 competition, Grøstl, Software implementation, Byte slicing, Intel AES new instructions, 8-bit AVR.

Abstract: Grøstl is an AES-based hash function and one of the 5 finalists of the SHA-3 competition. In this work we present high-speed implementations of Grøstl for small 8-bit CPUs and large 64-bit CPUs with the recently introduced AES instructions set. Since Grøstl does not use the same MDS mixing layer as the AES, a direct application of the AES instructions seems difficult. In contrast to previous findings, our Grøstl implementations using the AES instructions are currently by far the fastest known. To achieve optimal performance we parallelize each round of Grøstl by taking advantage of the whole bit width of the used processor. This results in implementations running at 12.2 cylces/byte for Grøstl-256 and 18.6 cylces/byte for Grøstl-512.

1 INTRODUCTION

In 2007, NIST has initiated the SHA-3 competition (National Institute of Standards and Technology, 2007) to find a new cryptographic hash function standard. 51 interesting hash functions with different design strategies have been accepted for the first round. Many of these SHA-3 candidates are AES-based and might benefit from the Intel AES new instructions set (AES-NI) (Gueron and Intel Corp., 2010) to speed up their implementations. In (Benadjila et al., 2009) those candidates which use the AES round transformation as a main building block have been analyzed and implemented using AES-NI. In that work, the authors claim that algorithms which use a very different MDS mixing matrix (than AES) are too distant from AES and there is no easy way to benefit from AES-NI.

Since December 2010, Grøstl(Gauravaram et al., 2011) is one of 5 finalists of the SHA-3 competition and uses the same S-box as AES but a very different MDS mixing matrix. In this work we show that it is still possible to efficiently implement Grøstl using AES-NI. Moreover, our AES-NI implementation of Grøstl is the fastest known implementation of Grøstl so far. Furthermore, we present a self-byte sliced implementation strategy which allows to implement Grøstl very efficiently on both 8-bit and 128-bit platforms. We achieve very good performance for larger bit widths by optimizing the MDS mixing matrix computation of Grøstl and by computing multiple columns in parallel. The parallel computation of the whole Grøstl round is possible and if parallel AES S-box table lookups (using AES-NI or the vpaes implementation of (Hamburg, 2009)) are available.

The paper is organized as follows. In Section 2, we give a short description of Grøstl. In Section 3, we describe requirements and general optimization techniques of our byte sliced implementations. In Section 4, we show how to minimize the computational requirements for MixBytes, the MDS mixing layer of Grøstl. In Section 5, we present the specific details of the 8-bit and 128-bit implementations. Finally, we conclude in Section 6.

2 DESCRIPTION OF GRØSTL

The hash function Grøstl was designed by Gauravaram et al. as a candidate for the SHA-3 competition (Gauravaram et al., 2011). In January 2011, Grøstl has been tweaked for the final round of the competition and we only consider this variant here. It is an iterated hash function with a compression function built from two distinct permutations P and Q, which are based on the same principles as the AES

[*] Parts of this work were done while the author stayed at TU Graz.

124 Aoki K., Roland G., Sasaki Y. and Schläffer M..
BYTE SLICING GRØSTL - Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Grøstl.
DOI: 10.5220/0003515701240133
In *Proceedings of the International Conference on Security and Cryptography* (SECRYPT-2011), pages 124-133
ISBN: 978-989-8425-71-3
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

round transformation (National Institute of Standards and Technology, 2001). Grøstl is a wide pipe design with security proofs for the collision and preimage resistance of the compression function (Fouque et al., 2009). In the following, we describe the Grøstl hash function and the permutations of Grøstl-256 and Grøstl-512 in more detail.

2.1 The Grøstl Hash Function

The input message M is padded and split into blocks $M_1, M_2, \dots, M_t$ of ℓ bits with $\ell = 512$ for Grøstl-256 and $\ell = 1024$ for Grøstl-512. The initial value H_0, the intermediate hash values H_i, and the permutations P and Q are of size ℓ as well. The message blocks are processed via the compression function f, which accepts two inputs of size ℓ bits and outputs an ℓ-bit value. The compression function f is defined via the permutations P and Q as follows:

$$f(H, M) = P(H \oplus M) \oplus Q(M) \oplus H.$$

The compression function is iterated with $H_0 = IV$ and $H_i \leftarrow f(H_{i-1}, M_i)$ for $1 \le i \le t$. The output H_t of the last call of the compression function is processed by an output transformation g defined as $g(x) = \mathrm{trunc}_n(P(x) \oplus x)$, where n is the output size of the hash function and $\mathrm{trunc}_n(x)$ discards all but the least significant n bits of x. Hence, the digest of the message M is defined as $h(M) = g(H_t)$.

2.2 The Grøstl-256 Permutations

As mentioned above, two permutations P and Q are defined for Grøstl-256. Both permutations operate on a 512-bit state, which can be viewed as an 8×8 matrix of bytes. Each permutation of Grøstl-256 consists of 10 rounds, where the following four AES-like round transformations are applied to the state in the given order:

- AddRoundConstant (AC) XORs a constant to one row of the state for P and to the whole state for Q. The constant changes for every round.
- SubBytes (SB) applies the AES S-box to each byte of the state.
- ShiftBytes (SH) cyclically rotates the bytes of rows to the left by $\{0,1,2,3,4,5,6,7\}$ positions in P and by $\{1,3,5,7,0,2,4,6\}$ positions in Q.
- MixBytes (MB) is a linear diffusion layer, which multiplies each column with a constant 8×8 circulant MDS matrix.

2.2.1 MixBytes

As the MixBytes transformation is the most run-time intensive part of Grøstl in our case, we will describe this transformation in more detail here. The MixBytes transformation is a matrix multiplication performed on the state matrix as follows:

$$A \leftarrow B \times A,$$

where A is the state matrix and B is a circulant MDS matrix specified as $B = circ(02,02,03,04,05,03,05,07)$ or by the following matrix:

$$B = \begin{pmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{pmatrix}.$$

The multiplication is performed in a finite field $\mathbb{F}_{256}$ defined by the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ (0x11B). As the multiplication by 2 only consists of a shift and a conditional XOR in binary arithmetic, we will calculate all multiplications by combining multiplications by 2 and additions (XOR), *e.g.* $7 \cdot x = (2 \cdot (2 \cdot x)) \oplus (2 \cdot x) \oplus x$.

For more details on the round transformations we refer to the Grøstl specification (Gauravaram et al., 2011).

2.3 The Grøstl-512 Permutations

The permutations used in Grøstl-512 are of size $\ell = 1024$ bits and the state is viewed as an 8×16 matrix of bytes. The permutations use the same round transformations as in Grøstl-256 except for ShiftBytes: Since the permutations are larger, the rows are shifted by $\{0,1,2,3,4,5,6,11\}$ positions to the left in P. In Q the rows are shifted by $\{1,3,5,11,0,2,4,6\}$ positions to the left. The number of rounds is increased to 14.

3 BYTE SLICED IMPLEMENTATIONS OF GRØSTL

In this section, we describe some requirements for the efficient parallel computation of the Grøstl round transformations. Due to the fact that MixBytes applies the same algorithm to every column of the state we can 'byte slice' Grøstl. In other words, we apply the same computations for every byte-wise column of the Grøstl state. On platforms with register sizes larger than 8-bit we can parallelize every transformation by placing several bytes of one row (of the state) inside one register. One column of the state is then distributed over 8 different registers (see Figure 1).

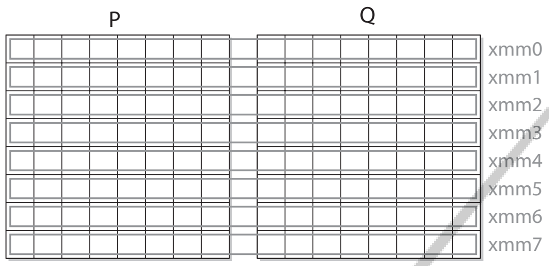

Figure 1: For the AES-NI implementation, the Grøstl-256 state is stored row-wise in xmm registers to compute each column 16 times in parallel.

3.1 Transposing the State

Unfortunately, the byte mapping in Grøstl is exactly the opposite of this requirement. The state is mapped to a byte sequence column-wise. Therefore we have to transpose each input state to get bytes of the same row into one register.

Once this realignment is done we can apply the same operations on each column (or byte) stored in the row registers at once. Even SubBytes which only reorders the bytes of one row, is easier to implement this way, because no data has to be moved between registers.

3.2 AddRoundConstant

In AddRoundConstant a constant is XORed to the state. This constant is different for P and Q and changes every round. When using large registers, these constants can be precomputed and XORed row-by-row and in parallel to each column of the state.

3.3 SubBytes

In order to improve the performance of the SubBytes layer, we need to compute as many parallel S-box lookups as possible.

In general, there is no easy way to lookup and replace each byte of a register using generic instructions on large platforms. For this reason the T-table based implementations are currently still the fastest on most bigger platforms. However, the AES new instructions set gives us the possibility of 16 parallel S-box lookups within only one instruction (see Section 5).

Another approach for parallel AES S-box table lookups is to use small Log tables to efficiently compute the inverse of the AES S-box using the vpaes implementation presented in (Hamburg, 2009).

3.4 ShiftBytes

ShiftBytes is generally simple to implement on any platform if the state is stored in row ordering. Only byte shufflings, bitshifts and XORs, or addressing different state bytes (or words) is necessary.

3.5 MixBytes

As stated above, MixBytes is the transformation that benefits most from byte slicing. MixBytes is computed using a large number of XORs and multiplications by two in $\mathbb{F}_{256}$. The multiplication in the finite field $\mathbb{F}_{256}$ will be simplified to simple multiplications by two and additions in $\mathbb{F}_{256}$ (XORs).

For the multiplication by two we only need to shift each byte to the left by one bit. To keep the result in $\mathbb{F}_{256}$ we have to observe the carry bit (MSB before the shift operation). If the carry bit is zero the result is already correct (still in $\mathbb{F}_{256}$), if the carry bit is one we have to reduce by the irreducible polynomial (*i.e.* XOR 0x11B).

There are many strategies to reduce the number of XOR computations for MixBytes and we discuss two optimization strategies in detail in Section 4.

4 OPTIMIZING THE MIXBYTES COMPUTATION

The MDS matrix multiplication is the most complex operation of Grøstl. Without optimizations all bytes of a column have to be multiplied by $2,3,4,5$ and 7 and then summed up according to the following matrix multiplication:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 3 & 4 & 5 & 3 & 5 & 7 \\ 7 & 2 & 2 & 3 & 4 & 5 & 3 & 5 \\ 5 & 7 & 2 & 2 & 3 & 4 & 5 & 3 \\ 3 & 5 & 7 & 2 & 2 & 3 & 4 & 5 \\ 5 & 3 & 5 & 7 & 2 & 2 & 3 & 4 \\ 4 & 5 & 3 & 5 & 7 & 2 & 2 & 3 \\ 3 & 4 & 5 & 3 & 5 & 7 & 2 & 2 \\ 2 & 3 & 4 & 5 & 3 & 5 & 7 & 2 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

If we use only multiplications by 2 as described above we can rewrite the same equations with factors of only 2 and 4. See Listing 1. Without optimization, the total number of XORs is $13 \cdot 8 = 104$ and we need 16 multiplications by 2 (if we can store the results).

Note that a multiplication by 2 is usually about 3-5 times more expensive than an XOR operation.

4.1 Using Temporary Results

In this section, we show a MixBytes computation which tries to minimize the number of XORs and the used registers while keeping the minimum number of 16 multiplications by 2. This strategy is used for the Intel AES-NI implementation in Section 5.1.

Since many terms $(a_i, 2 \cdot a_i, 4 \cdot a_i)$ in the computation are added to more than one result, we can save XORs by computing temporary results (see Table 1). For example, the term

$$t = 2 \cdot a_0 + 2 \cdot a_2 + 1 \cdot a_5 + 4 \cdot a_7 + 1 \cdot a_7 \qquad (1)$$

needs to be added to b_0, b_1 and b_3. This has a total cost of $3 \cdot 5 = 15$ XORs using the naive approach. If we first compute the temporary result t and then add t to each of b_0, b_1 and b_3, we can save $15 - (4+3) = 8$ XORs.

There are many possibilities to compute temporary results and we used a greedy approach to find a good sequence. In each step of this approach, we try out all possible temporary results and compute the number of XORs we can save. In the first step, the maximum number of XORs we can save is 8. After we remove the already added terms, we continue with the greedy approach until only single terms are left. Using this approach we found a sequence of computing MixBytes which requires 58 XORs and 16 multiplications by two. This sequence is shown by Table 1 and we use superscript numbers to denote the order of computing temporary results.

$$\begin{aligned}
b_0 &= a_2 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus 2a_0 \oplus 2a_1 \oplus \\
&\quad 2a_2 \oplus 2a_5 \oplus 2a_7 \oplus 4a_3 \oplus 4a_4 \oplus 4a_6 \oplus 4a_7 \\
b_1 &= a_0 \oplus a_3 \oplus a_5 \oplus a_6 \oplus a_7 \oplus 2a_0 \oplus 2a_1 \oplus \\
&\quad 2a_2 \oplus 2a_3 \oplus 2a_6 \oplus 4a_0 \oplus 4a_4 \oplus 4a_5 \oplus 4a_7 \\
b_2 &= a_0 \oplus a_1 \oplus a_4 \oplus a_6 \oplus a_7 \oplus 2a_1 \oplus 2a_2 \oplus \\
&\quad 2a_3 \oplus 2a_4 \oplus 2a_7 \oplus 4a_0 \oplus 4a_1 \oplus 4a_5 \oplus 4a_6 \\
b_3 &= a_0 \oplus a_1 \oplus a_2 \oplus a_5 \oplus a_7 \oplus 2a_0 \oplus 2a_2 \oplus \\
&\quad 2a_3 \oplus 2a_4 \oplus 2a_5 \oplus 4a_1 \oplus 4a_2 \oplus 4a_6 \oplus 4a_7 \\
b_4 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_6 \oplus 2a_1 \oplus 2a_3 \oplus \\
&\quad 2a_4 \oplus 2a_5 \oplus 2a_6 \oplus 4a_0 \oplus 4a_2 \oplus 4a_3 \oplus 4a_7 \\
b_5 &= a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_7 \oplus 2a_2 \oplus 2a_4 \oplus \\
&\quad 2a_5 \oplus 2a_6 \oplus 2a_7 \oplus 4a_0 \oplus 4a_1 \oplus 4a_3 \oplus 4a_4 \\
b_6 &= a_0 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus 2a_0 \oplus 2a_3 \oplus \\
&\quad 2a_5 \oplus 2a_6 \oplus 2a_7 \oplus 4a_1 \oplus 4a_2 \oplus 4a_4 \oplus 4a_5 \\
b_7 &= a_1 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus 2a_0 \oplus 2a_1 \oplus \\
&\quad 2a_4 \oplus 2a_6 \oplus 2a_7 \oplus 4a_2 \oplus 4a_3 \oplus 4a_5 \oplus 4a_6
\end{aligned}$$

Listing 1: MixBytes computation for one column with factors 1, 2 and 4. a_i are the input bytes and b_i are the output bytes.

4.2 Reusing Results of $\cdot 1$

In this section, we show a different MixBytes optimization technique which might be faster if more registers are available. This technique has been used for the 8-bit AVR implementation (see Section 5.2).

In Table 2 we have separated the MixBytes computation for each factor a_i, $2 \cdot a_i$ and $4 \cdot a_i$. We use superscript numbers to denote the order in which we compute temporary results again. The values marked with letters are added to the temporary results after computing the first (intermediate) results to further optimize the computation, *e.g.*:

$$\begin{aligned}
b_{1,1} &= a_0 \oplus a_3 \\
b_{6,1} &= b_{1,1} \qquad (2) \\
b_{1,1} &= b_{1,1} \oplus a_6 \\
b_{4,1} &= b_{1,1}
\end{aligned}$$

In this version the values that are multiplied by 2 are not calculated from the original inputs a_i but from the results of the first part of the calculation $b_{i,1}$. While this significantly reduces the number of XORs the number of multiplications increases from 16 to 24: instead of multiplying every byte of the column first by 2 and then again by 2 to get the values multiplied by 4, we need to multiply the intermediate values too. Although the number of multiplications increases to 24 we only need 47 XOR operations. This variant has been used for the 8-bit implementation of Grøstl in Section 5.2.

4.3 Minimizing the Number of XORs

The previous technique leads to another method to minimize the number of instructions in MixBytes. Using Table 2, we can observe that for each result b_i, many $a_j \oplus a_{j+1}$ terms are needed for each factor of 1, 2 and 4. By computing temporary results t_i, x_i and y_i we get the following optimized MixBytes computation formulas with $i = \{0, \ldots, 7\}$:

$$\begin{aligned}
t_i &= a_i + a_{i+1} \\
x_i &= t_i + t_{i+3} \qquad (3) \\
y_i &= t_i + t_{i+2} + a_{i+6} \\
b_i &= 2 \cdot (2 \cdot x_{i+3} + y_{i+7}) + y_{i+4}
\end{aligned}$$

Table 1: MixBytes computation with 58 XORs. A “•” denotes those inputs (a_i, $2 \cdot a_i$, $4 \cdot a_i$) 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 1).

	a_0			a_1			a_2			a_3			a_4			a_5			a_6			a_7		
	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1
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	–	$\bullet^{8}$	–	–	$\bullet^{2}$	$\bullet^{4}$	–	–	–	$\bullet^{4}$	–	$\bullet^{4}$	–	$\bullet^{4}$	$\bullet^{2}$	$\bullet^{8}$	–	$\bullet^{8}$	$\bullet^{2}$	$\bullet^{4}$	$\bullet^{2}$	–	$\bullet^{2}$	–

These formulas contain a minimum number of $8 \cdot 2 = 16$ multiplications by 2 and in total, only $8 \cdot 6 = 48$ XOR operations. However, it is still an open problem to find the smallest number of XORs needed to compute MixBytes of Grøstl. This strategy is also used for the Intel AES-NI implementation in Section 5.1. Note that this variant can also be used to improve Grøstl on the 8-bit platform.

5 IMPLEMENTATIONS

In the following we will describe specific implementation details for both the Intel AES-NI and 8-bit AVR platforms.

5.1 Intel AES-NI

Intel Processors based on the microarchitecture codename Westmere come with a new AES instructions set (AES-NI) (Gueron and Intel Corp., 2010). This set consists of six new instructions used for AES encryption and decryption. Next to improving the performance of AES they also provide more security due to their constant-time execution by avoiding cache-based table lookups. Furthermore, all processors with AES-NI come with different versions of SSE which we will also use to improve our implementations. For more information about the instructions used in this document we refer to the Intel Manual (Intel Corp., 2010).

Since Grøstl uses the same S-box as AES we can use AES-NI to improve the performance of Grøstl significantly. The implementation requires the processor to run in 64-bit mode to have access to the 16 128-bit XMM registers. This helps to avoid unnecessary memory accesses that would significantly reduce the performance. These 16 128-bit XMM registers provide enough space for the whole Grøstl state. Critical parts of the AES-NI implementation of Grøstl are written in assembler. In the following, we will discuss the main principles of the implementation and important observations.

5.1.1 State Alignment in Registers

For optimal performance, the alignment of the state inside the XMM registers is crucial. We have found that the best solution for Grøstl-256 is to compute P and Q simultaneously and put one row (64-bit) of each state side by side in one 128-bit XMM register. We then need 8 XMM registers to store both states. Thanks to MixBytes having the same MDS matrix for P and Q we can apply an optimized MixBytes algorithm to the whole XMM register and thus, to 16 columns of the state in parallel.

In Grøstl-512 we have 16 columns for each permutation which perfectly fit into 16 XMM registers. Hence, P and Q are computed separately and after each other but the same MixBytes algorithm can still be used 16 times in parallel again.

5.1.2 Transposing the State

To align the column-ordered message to fit the required row-ordering, the message has to be transposed after being loaded. For this purpose we use the PUNPCK instructions. The same has to be done with the IV for the initialization and in reverse order for the last chaining value before truncation. All the intermediate chaining values are kept in the transposed form.

In more detail, the PUNPCK instruction merges two XMM registers into one XMM register by interleaving the high or low bytes/words/doublewords or quadwords of the two source registers (Intel Corp., 2010). A simple square matrix can be transposed using only PUNPCK instructions (Intel Corp., 1996). As we initially have two 64-bit columns in each register, and therefore an 8x16 matrix, we also need PSHUFB and MOV instructions to reorder the data correctly.

Table 2: MixBytes computation separated for factor 1, 2 and 4. a_i are the input and b_i the output bytes. A "$\bullet$" denotes those inputs (a_i, $2 \cdot a_i$, $4 \cdot a_i$) which are added to get the intermediate results $b_{i,j}$. Superscripts denote the order in which temporary values are computed. Note that the results for factor 2 can be computed only by multiplying the results of factor 1 by 2 (*e.g.* $b_{0,2} = 2b_{3,1}$).

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

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

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

The details of the transpositions for each step are shown in table form in the Appendix.

5.1.3 AddRoundConstant

AC adds a constant to the state matrix. For P a constant is only added to the first row, for Q the constant is added to all rows. Therefore we need 8 128-bit XORs for Grøstl-256 since P and Q share registers. For Grøstl-512 we need 1 XOR for P and 8 for Q.

5.1.4 SubBytes

The Intel AES instructions provide exactly the functionality required for SubBytes of Grøstl since the same AES S-box is used. The last round of the AES encryption applies ShiftRows, SubBytes and AddRoundKey to the state. This functionality is available by the AESENCLAST instruction. To isolate SubBytes we can invert the ShiftRows transformation (using PSHUFB with the mask 0x0306090c0f0205080b0e0104070a0d00) and use an empty RoundKey. The following assembler code shows the S-box implementation using AES-NI: (Gueron and Intel Corp., 2010)

```
pshufb     xmm0, 0x0306090c0f...70a0d00
aesenclast xmm0, 0x0000000000...0000000
```

These instructions combined will take less than 3 cycles to compute with a latency of about 6 cycles (Fog, 2010).

5.1.5 ShiftBytes

We can use the PSHUFB instruction (SSSE3) to quickly reorder the bytes in the XMM registers for ShiftBytes. This instruction is even faster than a simple shift instruction. Furthermore the PSHUFB instruction of ShiftBytes can be combined with the PSHUFB instruction to correct ShiftRows for AESENCLAST.

Two PSHUFB instructions with constant masks can be merged by shuffling the first mask using the second mask:

```
pshufb xmm0, mask1
pshufb xmm0, mask2
```

is equal to:

```
(pshufb mask1, mask2)
pshufb xmm0, mask1
```

where the shuffled mask is again a constant. The new mask (mask1) can be precomputed. This way we can save one PSHUFB instruction and only need to store one constant.

5.1.6 MixBytes

With the new row ordering we can compute 16 columns in parallel in one pass. We have implemented both variants which need a minimum number of 16 multiplications by 2. Apart from the multiplications by 2, the first variant of Section 4.1 needs 8 MOV and 32 XOR operations without memory access, and 25 MOV and 26 XOR operations with memory access (33 MOV and 58 XOR operations in total). For the second variant given in Section 4.3, we need 11 MOV and 32 XOR operations without memory access, and 8 MOV and 16 XOR operations with memory access (19 MOV and 48 XOR operations in total).

In the following, we show different implementation variants of the multiplication by 2 which can be used to implement MixBytes.

Multiplication by 2 in $\mathbb{F}_{256}$ with SSE or Similar. If SSE is available, the code shown below can be used to calculate the multiplication in parallel. In this example *xmm*1 is multiplied by 2 and *xmm*0 will be lost (paddb is used instead of psllq because of the shorter opcode):

```
movdqa xmm0, xmm1
psrlw  xmm1, 7
pand   xmm1, 0x0101...01
pmullw xmm1, 0x1b1b...1b
paddb  xmm0, xmm0
pxor   xmm1, xmm0
```

Multiplication by 2 in $\mathbb{F}_{256}$ with PBLENDVB. If SSE 4.1 is available we can use the PBLENDVB instruction to slightly speed up the algorithm described above. PBLENDVB merges two XMM registers into one. The source register is selected by the MSB of each byte in a third register. The MSB is also the bit that decides whether or not it is necessary to reduce the byte after shifting. Therefore we can use this instruction to generate a mask to XOR 0x1B where necessary. The multiplication as implemented is shown below where *xmm*2 is multiplied by 2, *xmm*0 and *xmm*1 are lost:

```
movdqa   xmm0, xmm2
pand     xmm2, 0x1b1b...1b
pxor     xmm1, xmm1
paddb    xmm2, xmm2
pblendvb xmm1, 0x7f7f...7f
pxor     xmm2, xmm1
```

Multiplication by 2 in $\mathbb{F}_{256}$ with PCMPGTB. We get the fastest implementation using the PCMPGTB instruction. PCMPGTB compares signed bytes. If the MSB is set, the comparison with zero results in 0xFF and in 0x00 otherwise. The multiplication is shown below where xmm1 will be multiplied by 2, xmm0 will be lost and xmm2 has to be all 0x1B.

```
pxor     xmm0, xmm0
pcmpgtb  xmm0, xmm1
paddb    xmm1, xmm1
pand     xmm0, xmm2
pxor     xmm1, xmm0
```

If ALU instructions are the bottleneck in the MixBytes implementation, we can also replace some instructions by their memory variant and get for example:

```
movaps   xmm0, 0x0000...00
pcmpgtb  xmm0, xmm1
paddb    xmm1, xmm1
pand     xmm0, 0x1b1b...1b
pxor     xmm1, xmm0
```

5.1.7 Further Optimizations

For even higher performance we tried different local optimization techniques. We:

- tried different variants of the MixBytes computation;
- unrolled loops;
- used precomputed constants for AddRoundConstant;
- analyzed different instruction orders to improve parallel executions in different ALUs (micro-ops);
- used equivalent instructions with smaller opcode where possible;
- tried different variants for the multiplication by 2;
- used different variants of equivalent LOAD/ STORE or ALU; instructions to keep all units busy.

While unrolling loops works perfectly for Grøstl-256, we found that unrolling all loops in Grøstl-512 increases the code size to exceed the cache size of the used CPU. This causes an immense drop in performance. Therefore using loops is necessary in this implementation.

5.1.8 Log Tables (vpaes)

As presented in (Hamburg, 2009), there is another way to compute the S-box relatively fast and cache-timing resistant without the use of AES instructions. By using Log tables to calculate inverses in $\mathbb{F}_{2^4}$ it is possible to compute the inverse in $\mathbb{F}_{2^8}$. This way the S-box can be realized with only 4-bit table lookups. These lookups can be implemented with PSHUFB instructions. Recently, Grøstl has been implemented this way in (Çalik, 2010). We have implemented the AES S-box computation and improved the previous results using our optimized MixBytes computations. Note that the resulting vpaes implementation is now as fast as the T-table based implementation.

5.1.9 Intel AVX

The next generation of Intel processors feature a 256-bit extension to SSE called AVX. Using these new registers the possible bandwidth for parallel computations can be doubled. Even though PSHUFB and AESENCLAST will not be available for 256-bit registers, Grøstl-512 might run up to 50% faster because MixBytes and AddRoundConstant can be applied to P and Q at the same time.

Table 3: Speed of the Grøstl AES-NI and vpaes implementations in cylces/byte on an Intel Core i7-M620 and Intel Core2 Duo L9400 processor (v1: using MixBytes computation of Section 4.1; v3: using MixBytes computation of Section 4.3).

CPU	Version	Grøstl-256	Grøstl-512
Core i7	aesni v3	12.2	18.6
	aesni v1	13.0	18.6
	vpaes v1	23.2	32.6
	(T-tables)	24.0	35.9
Core2 Duo	vpaes	21.2	29.2
	(T-tables)	20.4	30.3

5.1.10 Benchmarks

The final round version of Grøstl has been benchmarked on an Intel Core i7-620LM and Intel Core2 Duo L9400. For comparison, we also show benchmarks of the T-table implementation and the vpaes implementation. The results are shown in Table 3.

5.2 8-bit AVR (ATmega163)

The ATmega163 is an 8-bit microcontroller with 32 8-bit multi-purpose registers, 1024 Bytes of SRAM and 16K of flash memory. The multi-purpose registers can be used to manipulate data. The controller needs 2 cycles to read from and write to the SRAM and 3 cycles to read from flash memory. Six of the 8-bit registers are used as 16-bit address registers X, Y and Z, thus they can usually not be used for computations. For a list of instructions see (Atmel, 2003).

Because of the limited bit width of the architecture we can only compute one column at once. Therefore the most important part of the 8-bit optimization is minimizing the number of XORs in MixBytes as described above. With 26 available general purpose registers we have just enough space to keep the intermediate values loaded at all times during the computation of one column. All the other columns have to be written back to RAM.

The multiplication in $\mathbb{F}_{256}$ can be implemented with the code shown in Listing 2, where r0 is multiplied by 2, r1 has to be pre-set to 0x1B and r2 will be lost. These instructions will take 4 cycles to process

```
LSL  r0       # r0 = r0 << 1
IN   r2, 0x3F # r2 = status register
SBRC r2, 0    # skip next if no carry
EOR  r0, r1   # r0 = r0 + 0x1B
```

Listing 2: Multiplication by 2 for 8-bit version.

on the selected CPU. As they are executed for every byte of the state it is faster to have a lookup table for the multiplication if enough memory is available.

AddRoundConstant and SubBytes are computed for each byte separately using XORs and table lookups. ShiftBytes is achieved at no cost by simply loading from shifted positions in RAM. For more details on the 8-bit implementation we refer to the full description of the implementation (Roland, 2009). Note that this implementation can probably be further improved using the MixBytes computation of Section 4.3.

Table 4: Speed of three different Grøstl-256 8-bit AVR implementations in cycles/byte on an ATMega163. The last line shows the RAM usage in bytes (using MixBytes computation of Section 4.2).

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

5.2.1 Benchmarks

We have implemented two different versions of the 8-bit implementation with the final round tweak (Grøstl) and three versions without the final round tweak (Grøstl-0) using different amounts of RAM. The versions are compared in Table 4.

6 CONCLUSIONS

In this work we have proposed two optimized algorithms for MixBytes, the MDS mixing layer of Grøstl, which allow to speed up Grøstl on various platforms. Furthermore, byte slicing provides the possibility to parallelize the Grøstl computation if the registers are large enough and parallel AES S-box table lookups are available. This is the case for Intel processors including the new AES instructions set or in general, using the vpaes implementation.

Both implementations show that Grøstl can be implemented efficiently on very different platforms. The 8-bit implementation will run at 469 cycles per byte on this very limited target hardware. The AES-NI implementation shows that even though Grøstl is very different from AES it still can take advantage of these new instructions. More specifically, our Intel AES-NI implementation of Grøstl is the fastest known implementation so far. Grøstl-256 runs at about 12.2 cycles per byte on an Intel Core i7-M620, which is about 50% faster than the table-based version on the same CPU.

We have reduced the number of operation needed to compute MixBytes to only 48 XORs with 16 multiplications by 2. Future work includes the optimization of the MixBytes computation to take more advantage of the 3 available ALUs in current Intel processors by minimizing the dependency chains. Also future CPU features like AVX will provide another opportunity to increase the performance, especially for the larger variant Grøstl-512.

ACKNOWLEDGEMENTS

The authors thank Krystian Matusiewicz for useful discussions and for fine-tuning the AES-NI implementations. This work was supported in part by the European Commission through the ICT Programme under Contract ICT-2007-216646 ECRYPT II, by the Austrian Science Fund (FWF), project P21936 and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

REFERENCES

Atmel (2003). 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash. ATmega163. Retrieved December 21, 2010, from http://www.atmel.com/dyn/resources/prod_documents/doc1142.pdf.

Benadjila, R., Billet, O., Gueron, S., and Robshaw, M. (2009). The Intel AES Instructions Set and the SHA-3 Candidates. Retrieved December 22, 2010, from http://crypto.rd.francetelecom.com/ECHO/sha3/AES/.

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

Fog, A. (2010). Instruction tables - Lists of instruction latencies, throughputs and microoperation breakdowns for Intel, AMD and VIA CPUs. Retrieved December 22, 2010, from http://www.agner.org/optimize/.

Fouque, P.-A., Stern, J., and Zimmer, S. (2009). Cryptanalysis of Tweaked Versions of SMASH and Reparation. In Avanzi, R., Keliher, L., and Sica, F., editors, *Selected Areas in Cryptography 2008, Proceedings*, volume 5381 of *LNCS*, pages 136–150. Springer.

Gauravaram, P., Knudsen, L. R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., and Thomsen, S. S. (2011). Grøstl – a SHA-3 candidate. Submission to NIST (Round 3). Retrieved May 03, 2010, from http://www.groestl.info.

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

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

Intel Corp. (1996). Using MMX™Instructions to Transpose a Matrix. Retrieved July 12, 2011, from ftp://download.intel.com/ids/mmx/MMX_App_Transpose_Matrix.pdf.

Intel Corp. (2010). Intel®64 and IA-32 Architectures Software Developers Manual. Retrieved December 21, 2010, from http://www.intel.com/products/processor/manuals/.

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

National Institute of Standards and Technology (2007). Cryptographic Hash Project. Available online at http://www.nist.gov/hash-competition.

Roland, G. A. (2009). Efficient Implementation of the Grøstl-256 Hash Function on an ATmega163 Microcontroller. Retrieved May 03, 2010, from http://groestl.info.

APPENDIX

The following tables show how the message is loaded, transposed, XORed to the chaining value and stored in XMM registers for the byte slice implementation of Grøstl-256. We use a sequence of PUNPCK and PSHUFB instructions to get the required formats.

First, the message block bytes M_{ij} are loaded into 4 XMM registers (see Table 5). Note that in Grøstl the message is loaded in column ordering format. Hence, the message needs to get transposed to get two rows of the M_{ij} in one XMM register (see Table 6). The chaining value is kept in the same format. Then, the initial XOR is computed to get $P_{ij} = H_{ij} \oplus M_{ij}$ (see Table 7).

To get one row of P and Q in one XMM register, we need to reorder and transpose both, P_{ij} and M_{ij} again (see Table 8 and Table 9). This format is used throughout all 10 rounds of Grøstl-256 and we transpose back to the chaining value format to compute the final XOR of P and Q and the feed-forward.

Table 5: Loading the message block into XMM0-XMM3.

XMM3	XMM2	XMM1	XMM0
M48	M32	M16	M0
M49	M33	M17	M1
M50	M34	M18	M2
M51	M35	M19	M3
M52	M36	M20	M4
M53	M37	M21	M5
M54	M38	M22	M6
M55	M39	M23	M7
M56	M40	M24	M8
M57	M41	M25	M9
M58	M42	M26	M10
M59	M43	M27	M11
M60	M44	M28	M12
M61	M45	M29	M13
M62	M46	M30	M14
M63	M47	M31	M15

Table 6: After transposing the message block two rows are stored in one XMM register. Note that the chaining value is stored in the same format in registers XMM4-XMM7.

XMM3	XMM2	XMM1	XMM0
M6	M4	M2	M0
M14	M12	M10	M8
M22	M20	M18	M16
M30	M28	M26	M24
M38	M36	M34	M32
M46	M44	M42	M40
M54	M52	M50	M48
M62	M60	M58	M56
M7	M5	M3	M1
M15	M13	M11	M9
M23	M21	M19	M17
M31	M29	M27	M25
M39	M37	M35	M33
M47	M45	M43	M41
M55	M53	M51	M49
M63	M61	M59	M57

Table 7: After computing the initial XOR: $P_{ij} = H_{ij} \oplus M_{ij}$.

XMM7	XMM6	XMM5	XMM4
P6	P4	P2	P0
P14	P12	P10	P8
P22	P20	P18	P16
P30	P28	P26	P24
P38	P36	P34	P32
P46	P44	P42	P40
P54	P52	P50	P48
P62	P60	P58	P56
P7	P5	P3	P1
P15	P13	P11	P9
P23	P21	P19	P17
P31	P29	P27	P25
P39	P37	P35	P33
P47	P45	P43	P41
P55	P53	P51	P49
P63	P61	P59	P57

Table 8: Reordering and transposing again to get one row of P_{ij} and one row of M_{ij} in one XMM register. XMM0-XMM3 contain row 0-3 of P and Q.

XMM3	XMM2	XMM1	XMM0
P3	P2	P1	P0
P11	P10	P9	P8
P19	P18	P17	P16
P27	P26	P25	P24
P35	P34	P33	P32
P43	P42	P41	P40
P51	P50	P49	P48
P59	P58	P57	P56
M3	M2	M1	M0
M11	M10	M9	M8
M19	M18	M17	M16
M27	M26	M25	M24
M35	M34	M33	M32
M43	M42	M41	M40
M51	M50	M49	M48
M59	M58	M57	M56

Table 9: And XMM4-XMM7 contain row 4-7 of P and Q.

XMM7	XMM6	XMM5	XMM4
P7	P6	P5	P4
P15	P14	P13	P12
P23	P22	P21	P20
P31	P30	P29	P28
P39	P38	P37	P36
P47	P46	P45	P44
P55	P54	P53	P52
P63	P62	P61	P60
M7	M6	M5	M4
M15	M14	M13	M12
M23	M22	M21	M20
M31	M30	M29	M28
M39	M38	M37	M36
M47	M46	M45	M44
M55	M54	M53	M52
M63	M62	M61	M60