Efficient Vector Implementations of AES-based Designs: A Case Study and New Implemenations for Grøstl

Severin Holzer-Graf, Thomas Krinninger, Martin Pernull, Martin Schläffer[1], Peter Schwabe[2], David Seywald, and Wolfgang Wieser

[1] IAIK, Graz University of Technology, Austria
[2] Research Center for Information Technology Innovation, Academia Sinica, Taiwan

Abstract. In this paper we evaluate and improve different vector implementation techniques of AES-based designs. We analyze how well the T-table, bitsliced and bytesliced implementation techniques apply to the SHA-3 finalist Grøstl. We present a number of new Grøstl implementations which improve upon many previous results. For example, our fastest ARM NEON implementation of Grøstl is 40% faster than the previously fastest ARM implementation. We present the first Intel AVX2 implementations of Grøstl, which require 40% less instructions than previous implementations. Furthermore, we present ARM Cortex-M0 implementations of Grøstl which improve the speed by 55% or the memory requirements by 15%.

Keywords: efficient software implementations, vector implementation, AVX2, NEON, AES-based designs, Grøstl

1 Introduction

Since the Advanced Encryption Standard (AES) was chosen by NIST in October 2000 [25], it has been used in innumerable applications. Apart from applications, the components of AES or its design principles are also used as the basis for many new cryptographic algorithms. Especially the announcement of Intel to add an AES instruction (AES-NI) to its future processors [22] has caused an increasing amount of new AES-based designs. As a consequence, many AES-based designs and a few more AES-inspired designs have been submitted to the SHA-3 competition [26] initiated by NIST.

Building an AES-based design has several advantages. From a security point of few, AES-based designs can benefit from proofs against a large class of attacks. Additionally, the design and security analysis of AES is kept particularly simple to provide security assurance within a short amount of time. As a consequence, the first single-key attack on 7 rounds of AES-128 [14] has been found before the AES competition was finished and the number of rounds (of non-marginal attacks) did not improve since then [13].

Unfortunately, the ease in analyzing the security of AES-based designs comes at some cost. Developing efficient implementations takes some time. For example, the fastest AES implementation (without AES-NI) running at 7.6 cycles/byte on Intel Core 2 [24] has been found 10 years after the design of AES was published. However, it is probably better to find better implementation techniques than finding better attack strategies after a design has been standardized.

Furthermore, the implementation characteristics of AES-based designs may be quite distorted. If AES-NI is available, the design may be remarkably fast, while without AES-NI at can be quite slow. This is especially true for AES-based hash functions which consist of a large state with additional operations for mixing more than one AES state. This effect can be observed for many AES-based designs submitted to the SHA-3 competition.

In this work we focus on the three main software implementation techniques of AES-based designs: T-tables [12, Sect. 5.2], bitslicing [7] and byteslicing [1], which are discussed in detail in Section 3. Which technique results in the fastest implementation largely depends on the cryptographic algorithm and the target platform. We apply all techniques to the AES-based SHA-3 finalist

Grøstl [16] (see Section 2) and provide a number of new and improved results. We focus on implementations using vector-instruction sets.

In Section 4 we propose the first 256-bit vector implementation of Grøstl using the Intel AVX2 instructions [11]. Since no processor using AVX2 is available, we have to be content with comparing the number of instructions instead of performing a proper benchmark. The first AVX2 implementation is a bytesliced implementation of Grøstl-512 which improves the number of instructions by 40% compared to the AVX implementation. The second implementation uses the new AVX2 vpgatherqq instructions which allows to perform parallel table lookups.

In Section 5, we present the first ARM NEON [3, Chapter A7] implementations of Grøstl by applying all three techniques of Section 3. We show that the T-table and bitslicing approach result in equally fast implementations. Byteslicing is slower since the vperm technique [20] is needed to compute the AES S-box. Hoever, using future AES instructions of ARMv8, byteslicing will most likely be faster than the other implementations, similar to the Intel AES-NI implementation.

Finally, in Section 6 we show that vector implementations using byteslicing can even be used efficiently in low-memory environments. We present 32-bit bytesliced implementations of Grøstl which consume much less memory than T-table implementations at almost the same speed.

2 Description of Grøstl

The hash function Grøstl [15] was designed as a candidate for the SHA-3 competition [27]. For the final round of the competition, Grøstl was tweaked in order to increase its security margin. 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 round transformation. In the following, we describe the components of the Grøstl hash function in more detail.

2.1 The Hash Function

Grøstl comes in two main variants, Grøstl-256 and Grøstl-512 which are used for different hash value sizes of $n = 256$ and $n = 512$ bits. The hash function first pads the input message M and splits the message 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 [16]). 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:

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

The compression function f is based on two ℓ-bit permutations P and Q and 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 output transformation Ω is applied to H_t to give the final hash value of size n by computing:

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

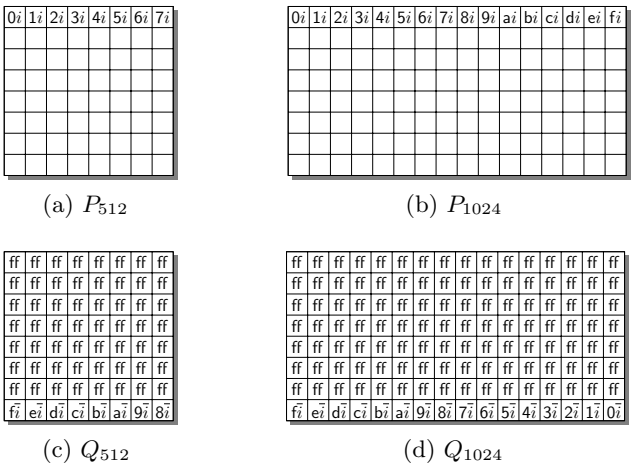
where $\text{trunc}_n(x)$ discards all but the least significant n bits of x.

2.2 The Permutations

As mentioned above, two permutations P and Q are defined for Grøstl. To distinguish between the permutations of Grøstl-256 ($\ell = 512$) and Grøstl-512 ($\ell = 1024$) 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 their size, the constants used in AC and SH, 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. 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.

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 Fig. 1.



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

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

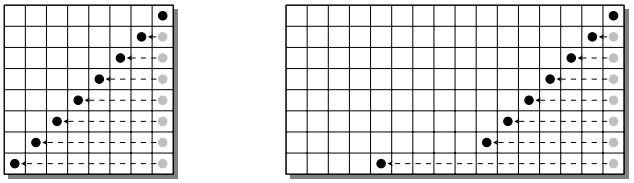
Fig. 1: The XOR constants added by the AddRoundConstant transformation.

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 [16].

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. The rotation values are:

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

This is illustrated in .



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

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

Fig. 2: The shift values used by the ShiftBytes transformation.

MixBytes (MB) is a linear diffusion layer, which multiplies each column A of the state with a constant, circulant 8×8 matrix M by computing $A \leftarrow M \cdot A$. The multiplication is performed in the finite field $GF(2^8)$ using the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ (0x11B). Since the multiplication by 2 can be carried out very efficiently using a single shift operation and a conditional XOR, we will calculate all multiplications by combining multiplications by 2 and additions (XOR).

Moreover, optimized formulas for computing MixBytes have been published in [1]. Using these formulas, only 48 XORs and 16 multiplications by 2 are needed to compute MixBytes. These formulas are shown below, where each line contains only a single XOR or MUL2 computation. Note that the formulas only need a single temporary state variable. The values a_i and b_i correspond to the i-th row of the input and output matrix A and a temporary matrix B and indices are computed modulo 8:

$$M = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}$$

$$\begin{aligned} b_i &= a_i + a_{i+1}, \\ a_i &= b_i + a_{i+6}, \\ a_i &= a_i + b_{i+2}, \\ b_i &= b_i + b_{i+3}, \\ b_i &= 02 \cdot b_i, \\ b_i &= b_i + a_{i+4}, \\ b_i &= 02 \cdot b_i, \\ a_i &= b_{i+3} + a_{i+4}. \end{aligned}$$

3 Implementation Methods for AES-based Designs

In this section we give a high-level overview on common implementation techniques for AES-based designs. The main implementation for AES-based designs are the T-table aproach [12, Sect. 5.2], bitslicing [7], and byteslicing [1]. Which technique gives the best result largely depends on the cryptographic algorithm and the target (micro-)architecture.

In the following, we give an overview of these techniques using the `Grøstl`-256 hash function as an example. While the T-table implementation is the straight-forward technique, the parallel bytesliced implementation is much faster if the Intel AES-NI instructions [18] are available. Without AES-NI, byteslicing can still be as fast as the T-table implementation by implementing the AES

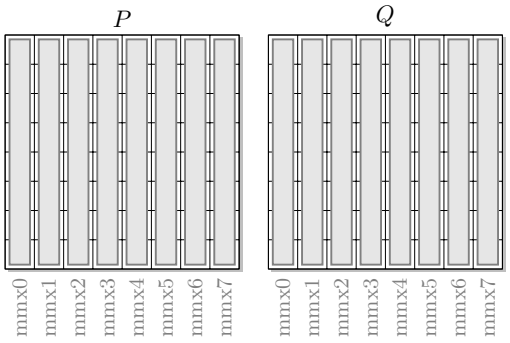


Fig. 3: For the T-table approach, the Grøstl-256 state is stored column-wise in 64-bit registers.

S-box using the vperm technique [20]. In Section 5 we show that the T-table approach results in the fastest available ARM NEON implementation of Grøstl.

3.1 T-Table Approach

Daemen and Rijmen have presented a table-based approach for AES in [12, Sect. 5.2], which efficiently computes the combined SubBytes and MixColumns transformation. Using this technique, at least one table lookup is needed for each S-box. The MixColumns transformation of AES 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 also be implemented efficiently on these platforms using the T-table approach. Even the 32-bit ARMv6 instruction set supports 64-bit loads which can be used for a T-table based implementation of Grøstl [28].

In T-table implementations, the state of Grøstl is stored in 64-bit registers in column ordering (see Fig. 3). 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 one 64-bit column (e.g., column 0) of Grøstl as follows, where a_{ij} are individual bytes of the state:

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

where the tables $y = T_i(x)$ contain 8-to-64-bit lookups of the S-box together with the 8 multipliers of MixBytes. 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 $T_i'' = T'_{(i+4) \bmod 8}$.

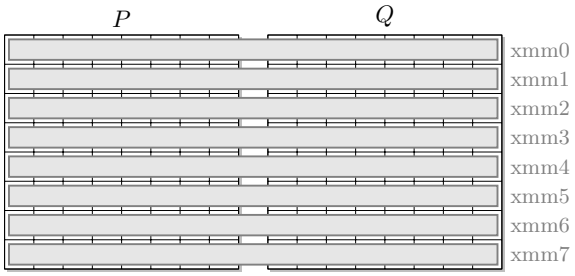


Fig. 4: For the SIMD implementation, the Grøstl-256 state is stored row-wise in xmm registers to compute each column 16 times in parallel.

3.2 Bytesliced Implementation

Another option to implement AES-based designs is a byte-wise parallel computation of columns [1]. This works especially well if we have a larger state and a platform, where we can compute many columns in parallel. In Grøstl, 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. The state is stored in row ordering. Using w-bit registers, $w/8$ columns can be computed in parallel (see Fig. 4). This approach is most efficient for small (8-bit) and large register sizes (128-bit and more). Additionally, a bytesliced implementation may also result in a very small memory footprint implementation on constraint 32-bit platforms (see Section 6).

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) [22] or the vector-permute (vperm) approach by Hamburg [20], parallel AES S-box table lookups can be performed efficiently. Moreover, the fastest Grøstl implementation [4] is a bytesliced implementation using AES-NI.

In a bytesliced 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 bytesliced 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.

3.3 Bitsliced Implementation

Bitslicing is an implementation technique proposed by Biham to improve the performance of DES [7]. Currently, the fastest software implementation of AES (without AES-NI) uses bitslicing [24]. Therefore, bitslicing is also a promising approach for other AES-based designs. Bitslicing works especially well if the same operations can be performed many times in parallel. In AES, this is the case if multiple blocks are encrypted in parallel using counter mode. Since the hash function Grøstl has a large state with many independent columns, bitslicing can be applied efficiently as well.

In general, bitslicing mimics hardware implementations in software. The data is transposed and for example, a 32-bit value is stored in 32 registers, one bit per register. With this bitsliced repre-

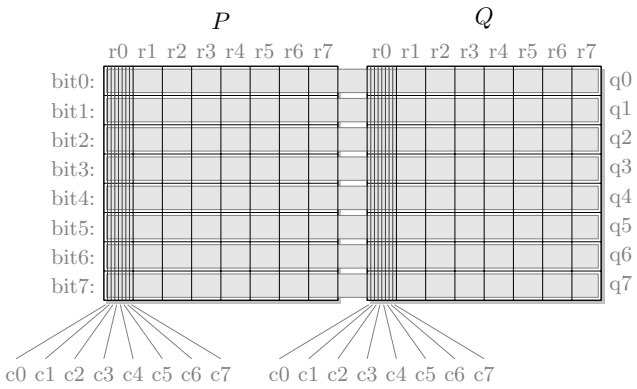


Fig. 5: In the bitsliced implementation of Grøstl-256, one bit of an 8-byte row is stored within one bytes of the NEON q registers. This way, ShiftBytes operates on bytes instead of rows.

sentation of data we can simulate hardware gates with the corresponding bit-logical instructions. To use all m bits of a register the same stream of operations is computed on m independent data streams in parallel. Registers of width m are used as vector registers with m 1-bit entries.

Then, the AES S-boxes are computed using their algebraic structure (inversion in $\mathbb{F}_2$) as it is also done in efficient hardware implementations [8, 10]. With minor modifications, the formulas underlying these hardware implementations are also used for bitslicing. More specifically, Käsper and Schwabe use 128 XOR/AND/OR instructions and 35 MOV (register to register) instructions in [24]. The MOV instructions are required because in the SSE instruction set the output of an instruction has to overwrite one of the inputs. With 3-operand instructions (as provided by AVX and NEON) and 16 registers, the AES S-box can be implemented using only 128 Boolean instructions. Although this is much slower than a table lookup for a single AES S-box computation, the high degree of parallelism (128 independent computations) lets bitsliced implementations achieve higher speeds than table lookups.

The AES implementation by Käsper and Schwabe needs to process 8 blocks in parallel to achieve the required level of parallelism. In Grøstl-256 we can compute all 128 AES S-boxes of P and Q in parallel without the need for multiple blocks. However, ShiftBytes is more difficult to implement in this case. Note that for the S-box, it does not matter in which order the bytes are stored in registers. Therefore, we can choose a bitsliced state which fits the linear operations ShiftBytes and MixBytes best. By storing the Grøstl-256 bitsliced state as shown in Fig.5, we get an efficient implementation using 128-bit ARM NEON instructions (see Sect. 5.2).

4 Implementing Grøstl using AVX2

The Intel AVX2 instruction set is an extension of the AVX instruction set and will be released by Intel for new processors in 2013 [23]. AVX2 provides a number of additions which can improve the efficiency of AES-based designs. AVX2 extends the functionality of integer instructions to 256 bits. Furthermore, new gather instructions have been added, which provide new possibilities to implement parallel T-table lookups in AES-based designs.

Since no processors supporting AVX2 are available yet, all our AVX2 implementations been tested using the Intel Software Development Emulator [21]. Because benchmarking of those implementations is not possible, we have to be content with comparing the number of instructions. Using AVX2, we show how to reduce the number of instructions for Grøstl by up to 40%, compared to previous AVX or AES-NI implementations [1]. Note that a similar comparison has been made by Gueron and Krasnov for their new AVX2 SHA-2 implementations using parallelized message schedules [19].

4.1 Byteslicing Grøstl-512 using AVX2 and AES-NI Instructions

Using 256-bit registers of AVX2, P and Q of Grøstl-512 can be computed completely in parallel, except for the `aesenclast` instruction. Note that using AES-NI with SSE, P and Q had to be computed after each other. AVX2 also brings a major improvement compared to AVX. Many AVX instructions used by Grøstl-512 were only working on 128-bits (`vaesenclast`, `vpshufb`, `vpcmpgtb`, `vpaddb`). Especially `vpcmpgtb` and `vpaddb` are used very often in the multiplication by 2 of MixBytes. Hence, also many insertion and extraction instructions were needed to process the upper 128 bits of a 256-bit register separately.

Additionally, we have replaced the floating point AVX instructions (`vxorps`, `vxorpd`) by their integer AVX2 instructions (`vpxor`). This avoids possible penalties caused by switching between integer and floating point domains [11]. Furthermore, we hope that integer AVX2 instructions will have a higher throughput than floating point AVX instructions, which is also the case for 128-bit SSE instructions.

To summarize our implementation, AddRoundConstant and ShiftBytes both can be fully parallelized and need only 8 instructions each. Note that `vpshufb` treats both 128-bit lanes separately. However, by storing P and Q in separate 128-bit lanes, we avoid all lane switching penalties. In SubBytes, we need to use two 128-bit `vaesenclast` instructions for each row of the state. Together with the necessary `vinserti128` and `vextracti128` instructions, SubBytes of Grøstl-512 needs 32 instructions per round.

The most expensive round transformation is MixBytes. As shown in [1], MixBytes can be implemented using 48 XORs and 16 multiplications by 2 (MUL2). Using the 256-bit `vpblendvb` instruction of AVX2, a single MUL2 computation can be implemented using only three 256-bit instructions. Together with 16 MOV/XOR instructions to load/store/copy/clear temporary values, we get $48 + 3 \cdot 16 + 16 = 112$ instructions for MixBytes. Note that other variants to create the reduction mask in MUL2 are possible. For example, we may get a better throughput using `vpcmpgtb` with `vpand` instead of `vpblendvb` once AVX2 is available:

```
// ymm0 will be multiplied by 2
// ymm1 has to be all 0x1b
// ymm2 has to be all zero
// ymm3 will be lost
vpblendvb ymm3, ymm2, ymm1, ymm0
vpaddb ymm0, ymm0, ymm0
vpxor ymm0, ymm0, ymm3
```

```
// ymm0 will be multiplied by 2
// ymm1 has to be all 0x1b
// ymm2 has to be all zero
// ymm3 will be lost
vpcmpgtb ymm3, ymm2, ymm0
vpaddb ymm0, ymm0, ymm0
vandpd ymm3, ymm3, ymm1
vxorpd ymm0, ymm0, ymm3
```

Together with 5 instructions overhead, we get $8+8+32+112 = 165$ instructions for one round of Grøstl-512. Note that previously published AVX implementations need 271 instructions per round and the 128-bit AES-NI implementation needs 338 instructions [1]. Hence, using our new AVX2 implementation of Grøstl-512 we are able to save 40% of the instructions. Furthermore, using AVX2 instructions, we were also able to reduce the number of instructions to transpose the input message block into bytesliced representation.

4.2 Parallel T-Table Lookups for Grøstl-256 using VPGATHERQQ

The new AVX2 instruction `vpgatherqq` allows to load four independent 64-bit values from memory into one 256-bit register. Using this instruction, we have implemented a fourfold parallel T-table implementation of Grøstl-256. We store the Grøstl-256 state column-wise and need two 256-bit registers for each of P and Q.

To perform the i-th T-table lookup for SubBytes and MixBytes, we first need a `vpshufb` instruction to extract the i-th byte of each 64-bit word. Note that we also use `vpshufb` to clear the unused bytes. To perform the actual lookups, `vpgatherqq` scales the extracted byte by a factor of 8 and adds the

table address. The scaling takes into account that we actually perform 8-to-64 bit lookups. The table addresses are stored in the general purpose registers r0-r7.

The `vpgatherqq` instruction uses a mask to determine for which 64-bit words the lookup is performed. If the MSB of the corresponding 64-bit word is not set, this word is left unchanged. However, `vpgatherqq` clears the mask after each invocation and we have to restore the mask each time, e.g. using a `vpcmpeqq` instruction. Additionally, all registers used by `vpgatherqq` have to be distinct. Hence, we need 8 instructions for each of the 8 T-table lookups together with xoring the results. Since we can save two initial xors, we get 62 instructions for SubBytes and MixBytes per permutation and round. The code to compute the lookup of table i of one round is given below:

```
// SubBytes+MixBytes (Table i)
// byte extraction
vpshufb tmp0, ymm0, [EXTR+i*256]
vpshufb tmp1, ymm1, [EXTR+i*256]
// restore gather mask
vpcmpeqq mask, mask, mask
vpcmpeqq mask, mask, mask
```

```
// 4 parallel T-table lookups
// address of table i is stored in ri
vpgatherqq tmp2, [8*tmp0+ri], mask
vpgatherqq tmp3, [8*tmp1+ri], mask
// xor table lookup results
vpxor ymm2, ymm2, tmp2
vpxor ymm3, ymm3, tmp3
```

If table lookups can be performed in parallel, ShiftBytes together with the byte extractions can get the most costly operations in T-table implementations. Since most processors do not offer byte extraction instructions, a couple of ALU instructions are needed. In the case of AVX2, we can us a number of byte shuffles to compute ShiftBytes and to extract the bytes needed for the lookup. Since the `vpshufb` instruction can not move bytes across 128-bit lanes, we need additional `vpermq` instructions to cross lanes. To swap bytes between the two 256-bit registers storing the state, we use `vpblendd` which merges two vectors at 32-bit word granularity. To compute ShiftBytes, we need 8 instructions per permutation and round. The instructions for ShiftBytes are given below:

```
// pre-shuffle
vpshufb ymm0, ymm0, [SHIFT_P0]
vpshufb ymm1, ymm1, [SHIFT_P0]
// cross lanes
vpermq ymm2, ymm0, 0xd8
vpermq ymm3, ymm1, 0xd8
```

```
// combine registers
vpblendd ymm0, ymm2, ymm3, 0xaa
vpblendd ymm1, ymm3, ymm2, 0xaa
// final shuffle
vpshufb ymm0, ymm0, [SHIFT_P1]
vpshufb ymm1, ymm1, [SHIFT_P1]
```

Together with two instructions for AddRoundConstant we get in total, $(2 + 8 + 62) \cdot 2 = 144$ instructions per round of `Grøstl`-256. The currently fastest `Grøstl`-256 implementation uses AES-NI and needs 169 instructions per round. However, since it is still unknown how many cycles the `vpgatherqq` instruction will need to compute 4 lookups, we cannot conjecture any speed improvement.

5 ARM NEON Implementations of `Grøstl`

In this section, we present three new `Grøstl` implementations using ARM NEON instructions. We are focusing on the ARM Cortex A8 processor. The NEON vector instruction set is available also on other processors and the implementations presented here will work on them as well. However, the performance may be different from what we describe here. Each implementation corresponds to one of the implementation methods given in Section 3. With the T-tables and the bitslicing approach, we get almost equally fast implementations running at around 46 cycles/byte. The bytesliced implementation is slower since we need to use the vperm approach to compute the AES S-box. However, once ARMv8 instructions with AES extensions are available [17], the bytesliced implementation will most likely be the fastest again. Detailed benchmarking results are given in Table 1.

The ARM NEON unit is a general-purpose SIMD (Single Instruction, Multiple Data) engine, which has its own registers and instruction set. It has 16 128-bit quadword registers (q0-q15) which can also be viewed (aliased) as 32 64-bit doubleword registers (d0-d31). The registers are considered

Table 1: Benchmark results of our NEON `Grøstl` implementations in cycles/byte for long messages. We used the SUPERCOP benchmarking suite [5] and performed the measurements using an ARM Cortex-A8 (Hercules eCafe).

hash function	T-table 5.1	bitsliced 5.2	vperm 5.3	arm32 [30]	arm11 [28]
`Grøstl`-256	45.8	48.5	92.0	76.9	99.4
`Grøstl`-512	67.0	-	-	103.2	-

as vectors of elements of the same data type and NEON instructions perform the same operation on all elements.

NEON on the Cortex A8 has limited dual issue capabilities. Instructions are divided between load/store/permute instructions and data processing (ALU) instructions. A data processing instruction can be dual issued with a load/store/permute instruction. For multi-cycle instructions dual issue is only performed at the first and last cycle (see [2, Sect. 16.5.3]).

The ARM processor has 16 user accessible general-purpose registers r0-r15, and one register which holds the current program status (CPSR). Register r15 contains the program counter, r14 the link register and r13 the stack pointer. In ARM mode, the link register and stack pointer can also be used as a general purpose register. One important property of the ARM processor is the built-in barrel shifter, which can shift and rotate the last operand of an ALU instruction at no cost. The ARM processor consists of two ALU units and one load/store unit.

Since ARM and NEON have separate instruction queues, ARM instructions also can be dual issued with NEON instructions. However, several minimal restrictions apply. First, at most 2 instructions can be executed per cycle. Second, at most one load/store/permute can be performed per cycle. Third, moving data from NEON to ARM cases a penalty of at least 20 cycles, since the NEON unit lags behind the ARM unit.

5.1 T-Table Implementation of `Grøstl` using NEON

Using NEON, one column of the `Grøstl` state can be stored in a 64-bit doubleword register. This reduces the number of xors compared to a 32-bit ARM implementation. Unfortunately, the indices used for the table lookups need to be stored in ARM registers. Hence, we compute one `Grøstl` round as follows: We load bytes of the state from memory into ARM registers and compute the table lookup address using ARM instructions. The table lookup itself and the xors are performed using NEON instructions. Finally, we store the result in memory using NEON stores.

Note that the 20 cycle penalty also occurs when transferring data from NEON to ARM via memory. We avoid this penalty by interleaving the computation of one round of P with a round of Q, since no data dependency between the two permutations exist. Hence, the ARM unit can continue to work on Q until the NEON unit is finished with computing and storing the result of one P round. Furthermore, to reduce the data dependency of the instructions, we interleave the computation of 8 different columns of one permutation.

To avoid expensive byte extractions, we load single bytes of the state into the ARM registers using `ldrb`. We load bytes and compute the lookups row-by-row. This has the additional advantage, that we can use the same table address for 8 consecutive lookups. The address for the lookup is computed using `add` including a barrel shift to account for 8-to-64 bit table lookups. The actual T-table lookup is performed using `vld1.64`. We reduce the number of xors by using 128-bit `veor` instructions. The computation of one example row is given in Listing 1.

Equivalent code blocks are repeated 8 times for each row and round of P and Q. Additionally, we need four 128-bit stores at the end of each round. For AddRoundConstant we need four 128-bit loads and four `veor` instructions. To summarize, the load/store instructions will be the bottleneck and we

```
/* ROW 1 (SH+SB+MB) */          /* increase T-table address */
/* load state bytes */          /* compute lookup address */
/* T-table lookups  */          /* xor results */
ldrb r0, [%[P], #9 ];           add %[T], %[T], #2048;
ldrb r1, [%[P], #17];
ldrb r2, [%[P], #25];
ldrb r3, [%[P], #33];           add r0, %[T], r0, asl #3;
ldrb r4, [%[P], #41];           add r1, %[T], r1, asl #3;
ldrb r5, [%[P], #49];           add r2, %[T], r2, asl #3;
ldrb r6, [%[P], #57];           add r3, %[T], r3, asl #3;
ldrb r7, [%[P], #1 ];           add r4, %[T], r4, asl #3;
vld1.64 d8,  [r0, :64];         add r5, %[T], r5, asl #3;
vld1.64 d9,  [r1, :64];         add r6, %[T], r6, asl #3;
vld1.64 d10, [r2, :64];         add r7, %[T], r7, asl #3;
vld1.64 d11, [r3, :64];
vld1.64 d12, [r4, :64];         veor q0, q0, q4;
vld1.64 d13, [r5, :64];         veor q1, q1, q5;
vld1.64 d14, [r6, :64];         veor q2, q2, q6;
vld1.64 d15, [r7, :64];         veor q3, q3, q7;
```

Listing 1: T-table implementation of Grøstl-256 using ARM NEON.

get a lower bound of $(16 \cdot 8 + 4 + 4) \cdot 10 \cdot 2/64 = 42.5$ cycles/byte. Using our new implementation, we get 45.9 cycles/byte on a Cortex-A8 processor.

5.2 Bitsliced Implementation of Grøstl-256 using NEON

As mentioned in Section 3.3, the representation of the bitsliced state has a large influence in the performance of a bitsliced implementations. It is important to evaluate the different implementation costs resulting for AddRoundConstant, SubBytes, ShiftBytes and MixBytes. If we represent the state as shown in Figure 5, we store bit0-bit7 in the 128-bit registers q0-q7. Then, AddRoundConstant and SubBytes always have the same implementation costs, no matter how the bits are arranged within the 128-bit registers. For AddRoundConstant we need 8 loads and 8 xors, while for SubBytes we need 128 ALU instructions [6].

In our bitsliced representation of Grøstl-256, ShiftBytes rotates octets of bits (of a row) by a different amount of positions. To avoid expensive masking operations, it is most efficient to store these 8 bits within one byte. To rotate bits within each byte, we make use of the variable shift instruction `vshl.u8` (left side of listing below). Note that the shift constants for shifting bits in bitsliced representation are the same as for bytes in standard representation.

The multiplication by 2 of MixBytes is rather cheap in bitsliced implementations and consists of only 3 xors [24]. What remains is to xor different rows of the non-bitsliced state to each other. Since we store bits of rows within bytes, we need to shuffle bytes of q-registers such that the corresponding bytes overlap and can get xored. Since crossing 64-bit lanes causes additional penalties, we store P and Q in the lower, respectively upper half of the 128-bit registers. Furthermore, we store the rows such that we can overlap corresponding bytes by rotating 8-byte blocks using the `vext.8` instruction. For example, we compute $b_i = a_i + a_{i+1}$ of bit 0 as shown in the right side of the listing below:

```
// SH: byte-wise rotate bit6
vshl.u8  q6, q14, q4;
vshl.u8 q14, q14, q5;
vorr q6, q6, q14;
```

```
// MB: b_i = a_i + a_{i+1}
vext.8 d24, d4, d4, #1;
vext.8 d25, d5, d5, #1;
veor q10, q2, q12;
```

Note that we can dual issue `vext.8` instructions with ALU instructions. In our implementation, we are able to interleave all `vext.8` instructions with the `veor` instructions of MixBytes, as well as

the `vshl.u8` and `vorr` instructions of ShiftBytes. A sample excerpt of the implementation is given in Listing 2.

```
vext.8 d24, d4, d4,#1;
vext.8 d25, d5, d5,#1;
vext.8 d26, d6, d6,#1;   vshl.u8  q6, q14,  q4;   # bit6: shift left
vext.8 d27, d7, d7,#1;   veor    q10,  q2, q12;   # b2_i = a2_i + a2_{i+1}
vext.8 d24, d4, d4,#6;
vext.8 d25, d5, d5,#6;   veor    q11,  q3, q13;   # b3_i = a3_i + a3_{i+1}
vext.8 d26, d6, d6,#6;   vshl.u8  q1,  q9,  q4;   # bit1: shift left
vext.8 d27, d7, d7,#6;   veor     q2, q10, q12;   # a2_i = b2_i + a2_{i+6}
vext.8 d24,d20,d20,#2;
vext.8 d25,d21,d21,#2;   veor     q3, q11, q13;   # a3_i = b3_i + a3_{i+6}
vext.8 d26,d22,d22,#2;   vshl.u8 q14, q14,  q5;   # bit6: shift right
vext.8 d27,d23,d23,#2;   veor     q2,  q2, q12;   # a2_i = a2_i + b2_{i+2}
vext.8 d24,d20,d20,#3;
vext.8 d25,d21,d21,#3;   veor     q3,  q3, q13;   # a3_i = a3_i + b3_{i+2}
vext.8 d26,d22,d22,#3;   vshl.u8  q9,  q9,  q5;   # bit1: shift right
vext.8 d27,d23,d23,#3;   veor    q10, q10, q12;   # b2_i = b2_i + b2_{i+3}
vext.8  d4, d4, d4,#4;
vext.8  d5, d5, d5,#4;   veor    q11, q11, q13;   # b3_i = b3_i + b3_{i+3}
vext.8  d6, d6, d6,#4;   vorr     q6,  q6, q14;   # bit6: combine SHL+SHR
vext.8  d7, d7, d7,#4;   vorr     q1,  q1,  q9;   # bit1: combine SHL+SHR
```

Listing 2: Bitsliced implementation of `Grøstl`-256 using ARM NEON.

In the bitsliced representation of `Grøstl`-256, we have 128 ALU instructions for SubBytes, followed by 96 `vext.8` instructions which are interleaved with the ALU instructions of ShiftBytes and MixBytes. Hence, in the first part of one round, ALU instructions are the bottleneck, in the second part, it is load/store/permute instructions. Together with 8 loads and 8 `veor` for AddRoundConstant (interleaved), we get a lower bound of $(8 + 128 + 96) \cdot 10/64 = 36.25$ cycles/byte. In reality, our best benchmark resulted in 48.5 cycles/byte, which is still about the same speed as the T-table implementation. We are continuing to investigate the reasons for the difference between the lower bound and our actual performance.

5.3 Bytesliced Vperm Implementation of `Grøstl`-256

The third option to implement `Grøstl` using NEON is a bytesliced implementation using vperm to compute the SubBytes transformation. On x86, the vperm implementation has a similar speed as the T-table implementation. Unfortunately, vector-permute or byte-shuffle instructions are more expensive using NEON.

In vperm implementations, each byte is split into nibbles which are then used as 4-bit indices to several 16-byte lookup tables. Four lookup tables are needed to compute the SubBytes transformation. Using the vperm approach, the S-box result can be multiplied by any factor without additional costs. This has been used by all previous vperm implementations of `Grøstl` [1, 9]. However, if all multipliers are computed in advance, many temporary results are needed and also the optimized MixBytes formulas cannot be used.

In our implementation, we only compute the plain SubBytes transformation and separately multiply by 2. The resulting NEON implementation is slightly faster than using the previous approach. The computation of one row of SubBytes is shown in the listing below:

Table 2: Benchmark results of the low-memory 32-bit vector implementation of Grøstl-256 on an ARM Cortex-M0 processor. We have measured the speed in cycles/byte for long messages and the memory requirements in bytes. The evaluation using 4·RAM+ROM has been proposed by XBX [29].

	speed [cycles/byte]	RAM [Bytes]	ROM [Bytes]	$4 \cdot \text{RAM} + \text{ROM}$ [Bytes]
bytesliced (fast)	469	344	1948	3324
bytesliced (small)	801	304	1464	2680
T-table (2kB)	406	704	6952	9768
T-table (8kB)	383	508	12630	14662
sphlib	856	792	15184	18352
8bit-c	1443	632	2796	5324
armcryptolib	17496	400	1260	2860

```
// SubBytes
vand  q2, q0, q8
vshr.u8 q1, q0, #4
veor  q0, q2, q1
vtbl.8  d6, {d24-d25}, d2
vtbl.8  d7, {d24-d25}, d3
vtbl.8  d8, {d26-d27}, d4
vtbl.8  d9, {d26-d27}, d5
veor  q3, q3, q4
vtbl.8  d4, {d24-d25}, d0
vtbl.8  d5, {d24-d25}, d1
veor  q2, q2, q4
vtbl.8  d6, {d24-d25}, d4
vtbl.8  d7, {d24-d25}, d5
veor  q3, q3, q1
vtbl.8  d8, {d24-d25}, d6
vtbl.8  d9, {d24-d25}, d7
veor  q4, q4, q0
vtbl.8  d0, {d28-d29}, d6
vtbl.8  d1, {d28-d29}, d7
vtbl.8  d2, {d30-d31}, d8
vtbl.8  d3, {d30-d31}, d9
veor  q0, q1, q0
```

Note that we need two `vtbl.8` to shuffle 16 bytes and each instruction costs 2 cycles since we shuffle across 64-bit lanes. Hence, 16 AES S-box lookups need 22 instructions and we get a lower bound of 28 cycles (14 `vtbl.8` instructions with 2 cycles each). For MUL2 we get 7 instructions and a lower bound of 8 cycles. The listing for one MUL2 is given in the following:

```
// MUL2
vand  q1, q0, q8
vshr.u8 q0, q0, #4
vtbl.8  d2, {d20-d21}, d2
vtbl.8  d3, {d20-d21}, d3
vtbl.8  d0, {d22-d23}, d0
vtbl.8  d1, {d22-d23}, d1
veor  q0, q0, q1
```

AddRoundConstant needs 8 `veor` instructions and for ShiftBytes we can use 14 `vext` instructions to rotate bytes within 64-bit lanes. Additionally we have 19 load/stores of constants and temporary values. Using the optimized MixBytes formulas with 48 `veor` and 16 MUL2, we get a vperm NEON implementation for Grøstl-256 running at 92 cycles/byte. Since the `vtbl.8` instructions are clearly the bottleneck, we get a lower bound of $((14 \cdot 8 + 4 \cdot 16) \cdot 2 + 14 + 19) \cdot 10/64 = 60.2$ cycles/byte.

6 Low-Memory Vector Implementation of Grøstl

On 32-bit platforms, the straight-forward way to implement Grøstl or other AES-based designs is the T-table approach. However, this method is not very suitable in low-memory environments since tables of a few kilobytes are needed. In this case, a bytesliced implementation can be the better choice. If the cache is small, it may even be faster than a T-table implementation. In this section, we give two short examples of bytesliced implementations using very small vectors.

6.1 32-bit Bytesliced Implementation of Grøstl-256 for Cortex-M0

Since the ARM Cortex-M0 processor has only a small cache, memory accesses are rather expensive. Therefore, it turned out to be more efficient to compute MixBytes using a bytesliced implementation

instead of using precomputed T-tables. In a 32-bit bytesliced implementation, we can compute 4 columns in parallel. Only for the SubBytes layer we need to extract bytes and perform single S-box lookups using a small table. Since the Cortex-M0 has only 8 registers we need to store the state in memory and process only a small fraction of the state at once.

However, load and store instructions on the Cortex-M0 are more expensive than ALU instructions. Therefore, we try to keep values in the registers and perform as many computations on them as possible. The constants for AddRoundConstant are computed instead of storing them in memory. To compute the SubBytes layer, we load 32-bit values of the state into registers and extract single bytes using ALU instructions to perform the AES S-box lookup. For ShiftBytes we load two 32-bit values containing one row of the state and rotate and swap the values inside registers.

For MixBytes we use the optimized formulas with a minimal amount of 48 xor operations. Due to the small number of registers, we need a rather high number of temporary variables, in-register `mov` instructions and memory loads. Note that on the ARM Cortex-M0 platform, `push` and `pop` need only $N+1$ cycles to push or pop N registers to or from the stack, compared to $2 \cdot N$ instructions for loads and stores. By computing blocks of 8 32-bit values and using `push` and `pop`, we can significantly reduce the number of cycles needed to store temporary values.

Furthermore, we have implemented the multiplication by 2 completely within memory. We use an MSB mask `0x80808080` to generate the value which is conditionally xored to the bits determined by the irreducible polynomial `0x11b`. This method is similar to the multiplication by 2 used in the bitsliced implementation. The following listing shows the Thumb assembly code used:

```
// MUL2
// r5: input, output
// r6: msbmask
// r1,r2: temporary
movs r1, r0
ands r1, r6
mvns r2, r6
ands r0, r2
lsls r0, #1
lsrs r1, #7
lsls r2, r1, #1
orrs r1, r2
lsls r2, r1, #3
orrs r1, r2
eors r0, r1
```

We have implemented a fast and a small Thumb 32-bit bytesliced implementation for the Cortex-M0. The main difference is the use of macros and loop unrolling to speed up the computation at the cost of more memory. The results are given in Table 2. Additionally, we have implemented improved T-table implementations using 2kB or 8kB tables. We compare our results with previously published T-table implementations of `Grøstl`-256. The results show, that in low-memory environments, the bytesliced implementation consumes much less memory by causing only a minor decrease in speed.

7 Conclusions

In this work we have analyzed three different implementation techniques for AES-based designs and presented various new and improved vector implementations of the SHA-3 finalist `Grøstl`. Depending on the target platform and the available instructions, a different implementation technique may be the fastest. For example, in the case of ARM NEON implementations we currently get the best result using the T-table approach, while the lower bound for the bitsliced implementation is better. Furthermore, once AES instructions of ARMv 8 will b available, the bytesliced implementation technique will most likely outperform the others. The case is similar for many other platforms. We hope that our work will help implementers, but also designers of new AES-based cryptographic primitives to find the right balance of implementation characteristics.

References

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

2. ARM Limited: Cortex-a8 technical reference manual, revision r3p2 (2010), http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html
3. ARM Limited: NEON (March 2011), available online: http://www.arm.com/products/processors/technologies/neon.php
4. Bernstein, D.J., Lange, T.: eBASH: ECRYPT Benchmarking of All Submitted Hashes (January 2011), available online: http://bench.cr.yp.to/ebash.html
5. Bernstein, D.J., Lange, T.: Supercop (2012), http://bench.cr.yp.to/supercop.html, accessed September 9, 2012
6. Bernstein, D.J., Schwabe, P.: NEON crypto (2012), document ID: 9b53e3cd38944dcc8baf4753eeb1c5e7, http://cryptojedi.org/papers/#neoncrypto
7. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) Fast Software Encryption. Lecture Notes in Computer Science, vol. 1267, pp. 260–272. Springer-Verlag Berlin Heidelberg (1997), http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1997/CS/CS0891.pdf
8. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Festa, P. (ed.) Experimental Algorithms. LNCS, vol. 6049, pp. 178–189. Springer (2010)
9. Çalik, Ç.: Multi-stream and Constant-time SHA-3 Implementations. NIST hash function mailing list (December 2010), available online: http://www.metu.edu.tr/~ccalik/software.html#sha3
10. Canright, D.: A very compact S-box for AES. In: Sunar, B., Rao, J.R. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer (2005)
11. Corp., I.: Intel advanced vector extensions programming reference (2011), http://software.intel.com/file/36945
12. Daemen, J., Rijmen, V.: AES Proposal: Rijndael. NIST AES Algorithm Submission (September 1999), available online: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf
13. Derbez, P., Fouque, P.A., Jean, J.: Improved Key Recovery Attacks on Reduced-Round AES. CRYPTO rump session (2012)
14. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.: Improved Cryptanalysis of Rijndael. In: FSE. LNCS, vol. 1978, pp. 213–230. Springer (2000)
15. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (2008), retrieved July 04, 2010, from http://www.groestl.info.
16. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (Round 3) (2011), available: http://www.groestl.info (2011/11/25).
17. Grisenthwaite, R.: Armv8 technology preview (2011), http://www.arm.com/files/downloads/ARMv8_Architecture.pdf
18. Gueron, S.: Advanced Encryption Standard (AES) instructions set. Tech. rep., Intel Corporation (2008), http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf
19. Gueron, S., Krasnov, V.: Simultaneous hashing of multiple messages. Cryptology ePrint Archive, Report 2012/371 (2012), http://eprint.iacr.org/2012/371
20. Hamburg, M.: Accelerating AES with Vector Permute Instructions. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2009. LNCS, vol. 5747, pp. 18–32. Springer (2009), http://mikehamburg.com/papers/vector_aes/vector_aes.pdf
21. Intel: Intel software development emulator (2012), http://software.intel.com/en-us/articles/intel-software-development-emulator/
22. 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/
23. Intel (Mark Buxton): Haswell New Instruction Descriptions Now Available! Available online: http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/ (June 2011)
24. Käsper, E., Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer (2009)
25. 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)

26. National Institute of Standards and Technology: Cryptographic Hash Project (2007), available online at http://www.nist.gov/hash-competition.
27. NIST: Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Federal Register 72(212), 62212–62220 (2007), http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf
28. Schwabe, P., Yang, B.Y., Yang, S.Y.: SHA-3 on ARM11 processors. In: Mitrokotsa, A., Vaudenay, S. (eds.) Progress in Cryptology – AFRICACRYPT 2012. LNCS, vol. 7374, pp. 324–341. Springer (2012), http://cryptojedi.org/papers/#sha3arm
29. Wenzel-Benner, C., Gräf, J.: XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework (2012), available online at https://xbx.das-labor.org/.
30. Wieser, W.: Optimization of Grøstl for 32-bit ARM Processors. Bachelor's thesis, Graz University of Technology, Austria (2011)