Grøstl – a SHA-3 candidate *

http://www.groestl.info

Praveen Gauravaram[1], Lars R. Knudsen[1], Krystian Matusiewicz[2], Florian Mendel[3], Christian Rechberger[4], Martin Schläffer[3], and Søren S. Thomsen[1]

[1]Department of Mathematics, Technical University of Denmark, Matematiktorvet 303S, DK-2800 Kgs. Lyngby, Denmark
[2]Intel Technology Poland, Juliusza Słowackiego 173, 80-298 Gdansk, Poland
[3]Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
[4]Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium

January 16, 2011

Summary

Grøstl is a SHA-3 candidate proposal. Grøstl is an iterated hash function with a compression function built from two fixed, large, distinct permutations. The design of Grøstl is transparent and based on principles very different from those used in the SHA-family.

The two permutations are constructed using the wide trail design strategy, which makes it possible to give strong statements about the resistance of Grøstl against large classes of cryptanalytic attacks. Moreover, if these permutations are assumed to be ideal, there is a proof for the security of the hash function.

Grøstl is a byte-oriented SP-network which borrows components from the AES. The S-box used is identical to the one used in the block cipher AES and the diffusion layers are constructed in a similar manner to those of the AES. As a consequence there is a very strong confusion and diffusion in Grøstl.

Grøstl is a so-called wide-pipe construction where the size of the internal state is significantly larger than the size of the output. This has the effect that all known, generic attacks on the hash function are made much more difficult.

Grøstl has good performance on a wide range of platforms, and counter-measures against side-channel attacks are well-understood from similar work on the AES.

*Document version no. 2.0 (updated January 16, 2011). A few simple changes have been made to the constants in Grøstl in order to increase its security margin without penalizing its speed. This document describes the changed algorithm. We refer to the previous version as Grøstl-0.

Contents

1 Introduction

In this proposal we present the cryptographic hash function `Grøstl`, candidate for the SHA-3 competition initiated by the National Institute of Standards and Technology (NIST).

The paper is organised as follows. In Section 2, we give a high-level summary of the `Grøstl` proposal, and state the design goals. In Section 3, we present the details of the proposal and in Section 4, we describe the features specific to `Grøstl` and motivate our design choices. Section 5 introduces some alternative descriptions of `Grøstl` and Section 6 describes some modes of operation of `Grøstl` for the use as message authentication codes. In Section 7, we present our preliminary cryptanalysis results on `Grøstl`. Section 8 deals with implementation aspects of `Grøstl`, including benchmarks results and performance estimates. Finally, we conclude in Section 9.

The name "`Grøstl`" may cause some problems in terms of pronunciation, and also due to the character '`ø`', which has different encodings around the world. Whenever problems with character encodings may arise, we recommend the spelling `Groestl`. With respect to pronunciation and other information on the name, see Appendix A.

2 Design goals

In this section, we give a brief motivation of the `Grøstl` proposal. Elegance of the design and simplicity of analysis, as well as proofs of desirable properties are the overall goals. The fact that it iteratively applies a compression function is among the few similarities with commonly used hash functions. Additionally, we aim to have security margins at several layers of abstraction in the design.

2.1 Overall goals for the hash

Here we state overall design goals for `Grøstl`.

- Simplicity of analysis, hence, `Grøstl` is based on a small number of permutations instead of a block cipher (with many permutations).
- Provably secure construction (assuming ideal permutations).
- Well-known design principles underlying the permutations (again, allowing simple analysis, provable properties).
- No special preference for a particular platform or word size, and good performance on a very wide range of platforms.
- Side-channel resistance at little additional cost.
- Defining reduced variants for cryptanalysis is made straightforward.
- Prevention of length-extension attacks.
- Allow implementers to exploit parallelisation within the compression function[1].

[1]Using `Grøstl` in a tree-mode, as any other cryptographic hash function for that matter, will also allow to exploit parallelisation at a higher level, but we consider this outside the scope of our submission.

2.2 Failure-tolerant design

Non-random behaviour of the employed permutations do not necessarily lead to non-ideal properties of the compression function. Attacks on the compression function, in turn, may not lead to attacks on the hash function.

- The internal state is significantly larger than the final output – hence, all known generic attacks are thwarted.
- Known techniques that exhibit non-ideal behaviour of the permutations work only for reduced variants.
- Attacks on the compression function do not necessarily translate to attacks on the hash function.
- There are no known attacks on the compression function meeting the proven lower bounds.

2.3 Design considerations for the compression function

Traditional design approaches of hash functions are based on block ciphers, e.g., MD5, SHA-1, SHA-256 [73], Whirlpool [7], Tiger [1], etc. This may seem sound since block cipher designs are well understood. However, the key schedule of the block cipher becomes more important in a setting where the attacker has control over every input and there is little consensus in the community what constitutes a good key schedule. The recent attacks [27, 49, 67, 70, 91] on SHA-1 and Tiger illustrate this issue. For this reason we base our proposal on a few individual permutations rather than a large family of permutations indexed by a key. The advantages of such a design methodology is as follows:

- No threat of attacks via the key schedule (e.g., weak keys).
- Since the key schedule of a block cipher is often rather slow, performance may be improved.
- Simplicity.

3 Specification of `Grøstl`

`Grøstl` is a collection of hash functions, capable of returning message digests of any number of bytes from 1 to 64, i.e., from 8 to 512 bits in 8-bit steps. The variant returning n bits is called `Grøstl`-n. We explicitly state here that this includes the message digest sizes 224, 256, 384, and 512 bits. We now specify the `Grøstl` hash functions.

3.1 The hash function construction

The `Grøstl` hash functions iterate the compression function f as follows. The message M is padded and split into ℓ-bit message blocks $m_1, \ldots, m_t$, and each message block is processed sequentially. An initial ℓ-bit value $h_0 = \mathrm{iv}$ is defined, and subsequently the message blocks m_i are processed as

$$h_i \leftarrow f(h_{i-1}, m_i) \quad \text{for } i = 1, \ldots, t.$$

Hence, f maps two inputs of ℓ bits each to an output of ℓ bits. The first input is called the *chaining input*, and the second input is called the *message block*. For `Grøstl` variants returning up to 256 bits, ℓ is defined to be 512. For larger variants, ℓ is 1024.

After the last message block has been processed, the output $H(M)$ of the hash function is computed as

$$H(M) = \Omega(h_t),$$

where Ω is an *output transformation* which is defined in Section 3.3. The output size of Ω is n bits, and we note that $n \leq 2 \cdot \ell$. See Figure 1.

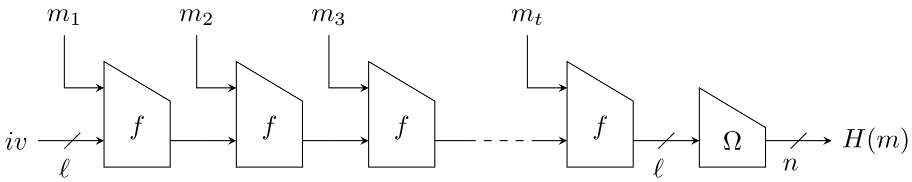

Figure 1: The `Grøstl` hash function.

3.2 The compression function construction

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

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h. \tag{1}$$

The construction of f is illustrated in Figure 2. In Section 3.4, we describe how P and Q are defined.

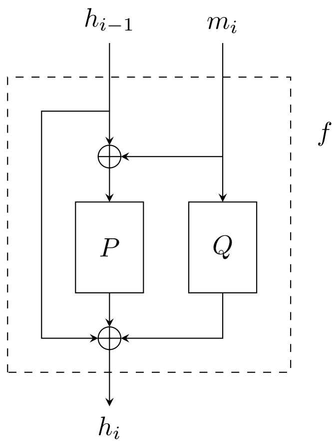

Figure 2: The compression function f. P and Q are ℓ-bit permutations.

3.3 The output transformation

Let $\mathrm{trunc}_n(x)$ be the operation that discards all but the trailing n bits of x. The output transformation Ω illustrated in Figure 3 is then defined by

$$\Omega(x) = \mathrm{trunc}_n(P(x) \oplus x).$$

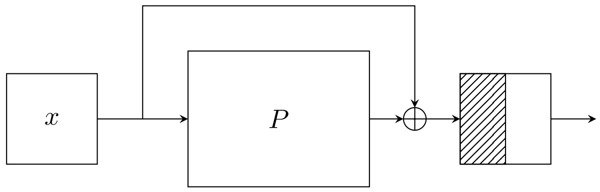

Figure 3: The output transformation Ω computes $P(x) \oplus x$ and then truncates the output by returning only the last n bits.

3.4 The design of P and Q

As mentioned, the compression function f comes in two variants; one is used for short message digests, and one is used for long message digests. Each variant uses its own pair of permutations P and Q. Hence, we define four permutations in total. The permutations will be assigned with subscripts 512 or 1024, whenever it is necessary to distinguish them.

The design of P and Q was inspired by the Rijndael block cipher algorithm [23, 24]. This means that their design consist of a number of *rounds* R, which consists of a number of *round transformations*. Since P and Q are much larger than the 128-bit state size of Rijndael, most round transformations have been redefined. In `Grøstl`, a total of four round transformations are defined for each permutation. These are

- AddRoundConstant
- SubBytes
- ShiftBytes
- MixBytes.

When a distinction is necessary, the third transformation ShiftBytes will be called ShiftBytesWide when used in the large permutations P_{1024} and Q_{1024}. While AddRoundConstant and ShiftBytes are different for each permutation, SubBytes and MixBytes are identical in all four permutations.

A round R consists of these four round transformations applied in the above order as illustrated in Figure 4. Hence,

$$R = \mathsf{MixBytes} \circ \mathsf{ShiftBytes} \circ \mathsf{SubBytes} \circ \mathsf{AddRoundConstant}.$$

We note that all rounds follow this definition. We denote by r the number of rounds. Concrete recommendations for r will be given in Section 3.4.6.

The transformations operate on a state, which is represented as a matrix A of bytes (of 8 bits each). For the short variants the matrix has 8 rows and 8 columns, and for the large variants, the matrix has 8 rows and 16 columns. In the following, we denote by v the number of columns, and we write constant byte values in sans serif font, e.g., c3. In the following, we describe how to map a byte sequence to a state matrix and back, and then we describe each round transformation.

3.4.1 Mapping from a byte sequence to a state matrix and vice versa

Since `Grøstl` operates on bytes, it is generally endianness neutral. However, we need to specify how a byte sequence is mapped to the matrix A, and vice versa. This mapping is done in a

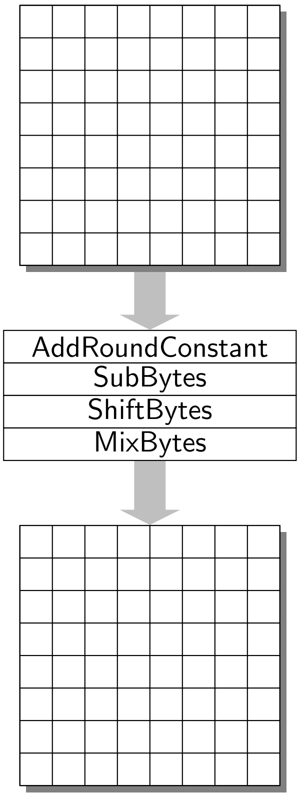

Figure 4: One round of the `Grøstl` permutations P and Q is a composition of four basic transformations.

similar way as in Rijndael. Hence, the 64-byte sequence 00 01 02 ... 3f is mapped to an 8×8 matrix as

$$\begin{bmatrix} \mathsf{00} & \mathsf{08} & \mathsf{10} & \mathsf{18} & \mathsf{20} & \mathsf{28} & \mathsf{30} & \mathsf{38} \\ \mathsf{01} & \mathsf{09} & \mathsf{11} & \mathsf{19} & \mathsf{21} & \mathsf{29} & \mathsf{31} & \mathsf{39} \\ \mathsf{02} & \mathsf{0a} & \mathsf{12} & \mathsf{1a} & \mathsf{22} & \mathsf{2a} & \mathsf{32} & \mathsf{3a} \\ \mathsf{03} & \mathsf{0b} & \mathsf{13} & \mathsf{1b} & \mathsf{23} & \mathsf{2b} & \mathsf{33} & \mathsf{3b} \\ \mathsf{04} & \mathsf{0c} & \mathsf{14} & \mathsf{1c} & \mathsf{24} & \mathsf{2c} & \mathsf{34} & \mathsf{3c} \\ \mathsf{05} & \mathsf{0d} & \mathsf{15} & \mathsf{1d} & \mathsf{25} & \mathsf{2d} & \mathsf{35} & \mathsf{3d} \\ \mathsf{06} & \mathsf{0e} & \mathsf{16} & \mathsf{1e} & \mathsf{26} & \mathsf{2e} & \mathsf{36} & \mathsf{3e} \\ \mathsf{07} & \mathsf{0f} & \mathsf{17} & \mathsf{1f} & \mathsf{27} & \mathsf{2f} & \mathsf{37} & \mathsf{3f} \end{bmatrix}.$$

For an 8×16 matrix, this method is extended in the natural way. Mapping from a matrix to a byte sequence is simply the reverse operation. From now on, we do not explicitly mention this mapping.

3.4.2 AddRoundConstant

The AddRoundConstant transformation adds a round-dependent constant to the state matrix A. By addition we mean exclusive-or (XOR). To be precise, the AddRoundConstant transformation in round i (starting from zero) updates the state A as

$$A \leftarrow A \oplus C[i],$$

where $C[i]$ is the round constant used in round i. P and Q have different round constants.

The round constants for P_{512} and Q_{512} are

$$P_{512}:\ C[i] = \begin{bmatrix} \mathsf{00}\oplus i & \mathsf{10}\oplus i & \mathsf{20}\oplus i & \mathsf{30}\oplus i & \mathsf{40}\oplus i & \mathsf{50}\oplus i & \mathsf{60}\oplus i & \mathsf{70}\oplus i \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \end{bmatrix}$$

and

$$Q_{512}:\ C[i] = \begin{bmatrix} \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} \\ \mathsf{ff}\oplus i & \mathsf{ef}\oplus i & \mathsf{df}\oplus i & \mathsf{cf}\oplus i & \mathsf{bf}\oplus i & \mathsf{af}\oplus i & \mathsf{9f}\oplus i & \mathsf{8f}\oplus i \end{bmatrix}$$

where i is the round number viewed as an 8-bit value, and all other values are written in hexadecimal notation.

Similarly, the round constants for P_{1024} and Q_{1024} are

$$P_{1024}:\ C[i] = \begin{bmatrix} \mathsf{00}\oplus i & \mathsf{10}\oplus i & \mathsf{20}\oplus i & \mathsf{30}\oplus i & \mathsf{40}\oplus i & \mathsf{50}\oplus i & \mathsf{60}\oplus i & \mathsf{70}\oplus i & \cdots & \mathsf{f0}\oplus i \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \cdots & \mathsf{00} \end{bmatrix}$$

and

$$Q_{1024}:\ C[i] = \begin{bmatrix} \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \mathsf{ff} & \cdots & \mathsf{ff} \\ \mathsf{ff}\oplus i & \mathsf{ef}\oplus i & \mathsf{df}\oplus i & \mathsf{cf}\oplus i & \mathsf{bf}\oplus i & \mathsf{af}\oplus i & \mathsf{9f}\oplus i & \mathsf{8f}\oplus i & \cdots & \mathsf{0f}\oplus i \end{bmatrix}$$

where i is again the round number viewed as an 8-bit value.

3.4.3 SubBytes

The SubBytes transformation substitutes each byte in the state matrix by another value, taken from the s-box S. This s-box is the same as the one used in Rijndael and its specification can be found in Appendix B. Hence, if $a_{i,j}$ is the element in row i and column j of A, then SubBytes performs the following transformation:

$$a_{i,j} \leftarrow S(a_{i,j}), \quad 0 \leq i < 8, \ 0 \leq j < v.$$

See Figure 5.

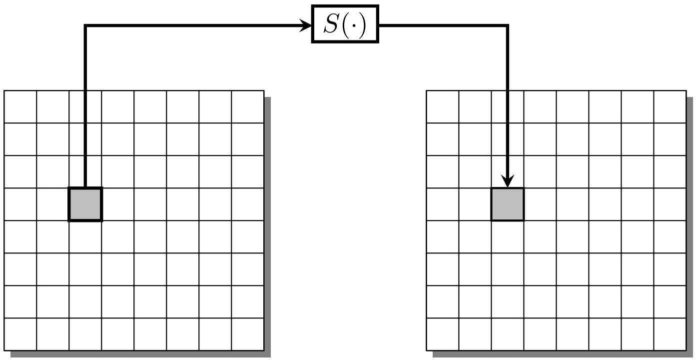

Figure 5: SubBytes substitutes each byte of the state by its image under the s-box S.

3.4.4 ShiftBytes and ShiftBytesWide

ShiftBytes and ShiftBytesWide cyclically shift the bytes within a row to the left by a number of positions. Let $\sigma = [\sigma_0, \sigma_1, \ldots, \sigma_7]$ be a list of distinct integers in the range from 0 to $v-1$. Then, ShiftBytes moves all bytes in row i of the state matrix σ_i positions to the left, wrapping around as necessary. The vector σ in ShiftBytes respectively ShiftBytesWide is different for P and Q. For ShiftBytes in P, we use $\sigma = [0,1,2,3,4,5,6,7]$ and for ShiftBytes in Q, we use $\sigma = [1,3,5,7,0,2,4,6]$. Similarly, for ShiftBytesWide in P and Q, we use $\sigma = [0,1,2,3,4,5,6,11]$ and $\sigma = [1,3,5,11,0,2,4,6]$ respectively. The transformations ShiftBytes and ShiftBytesWide for P and Q are illustrated in Figure 6 and Figure 7.

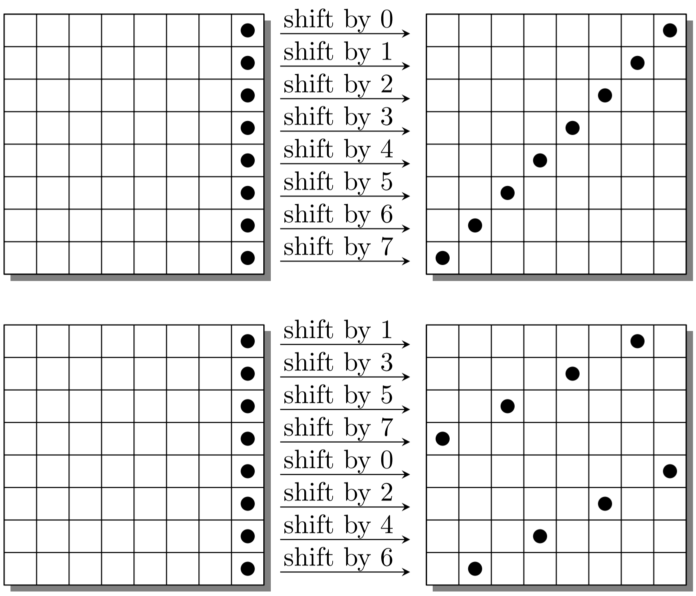

Figure 6: The ShiftBytes transformation of permutation P_{512} (top) and Q_{512} (bottom).

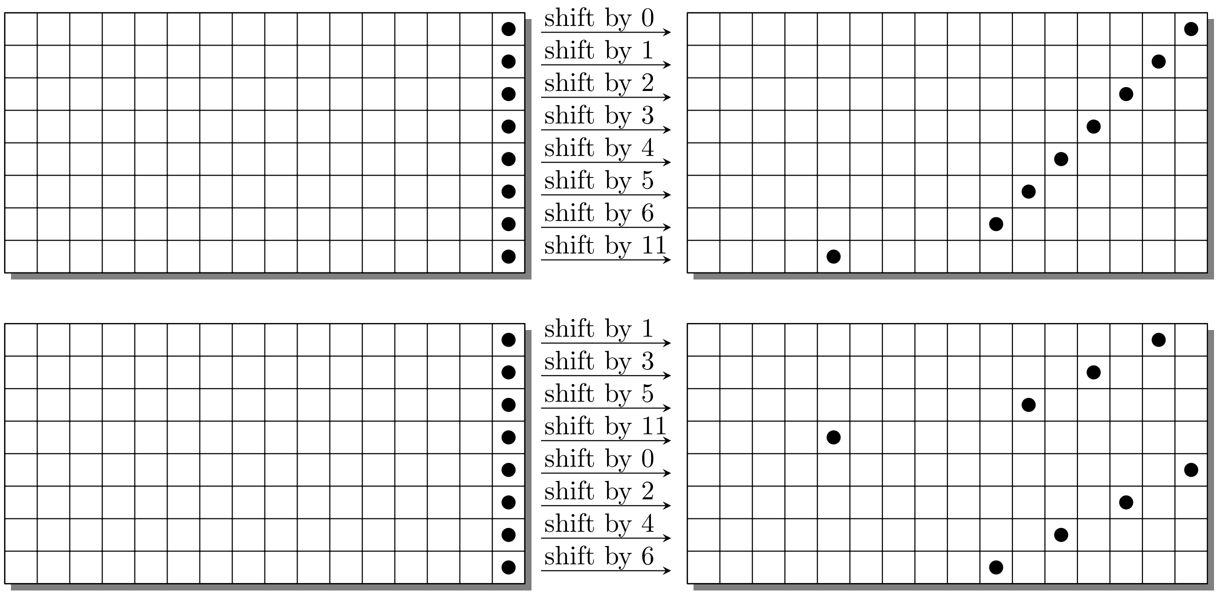

Figure 7: The ShiftBytesWide transformation of permutation P_{1024} (top) and Q_{1024} (bottom).

3.4.5 MixBytes

In the MixBytes transformation, each column in the matrix is transformed independently. To describe this transformation we first need to introduce the finite field $\mathbb{F}_{256}$. This finite field is defined in the same way as in Rijndael via the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ over $\mathbb{F}_2$. The bytes of the state matrix A can be seen as elements of $\mathbb{F}_{256}$, i.e., as polynomials of degree at most 7 with coefficients in $\{0,1\}$. The least significant bit of each byte determines the coefficient of x^0, etc.

MixBytes multiplies each column of A by a constant 8×8 matrix B in $\mathbb{F}_{256}$. Hence, the transformation on the whole matrix A can be written as the matrix multiplication

$$A \leftarrow B \times A.$$

The matrix B is specified as

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

This matrix is *circulant*, which means that each row is equal to the row above rotated right by one position. In short, we may write $B = \text{circ}(02, 02, 03, 04, 05, 03, 05, 07)$ instead. See also Figure 8.

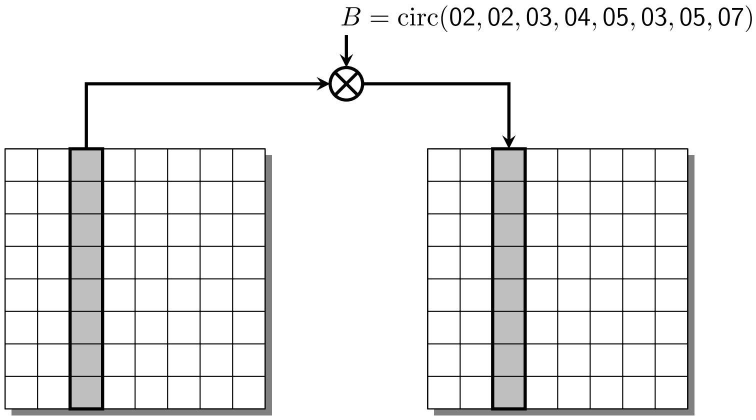

Figure 8: The MixBytes transformation left-multiplies each column of the state matrix treated as a column vector over $\mathbb{F}_{256}$ by a circulant matrix B.

3.4.6 Number of rounds

The number r of rounds is a tunable security parameter. We recommend the following values of r for the four permutations.

Permutations	Digest sizes	Recommended value of r
P_{512} and Q_{512}	8–256	10
P_{1024} and Q_{1024}	264–512	14

3.5 Initial values

The initial value iv_n of Grøstl-n is the ℓ-bit representation of n. The table below shows the initial values of the required output sizes of 224, 256, 384, and 512 bits.

n	iv_n
224	00 ... 00 00 e0
256	00 ... 00 01 00
384	00 ... 00 01 80
512	00 ... 00 02 00

3.6 Padding

As mentioned, the length of each message block is ℓ. To be able to operate on inputs of varying length, a padding function pad is defined. This padding function takes a string x of length N bits and returns a padded string $x^* = \mathsf{pad}(x)$ of a length which is a multiple of ℓ.

The padding function does the following. First, it appends the bit '1' to x. Then, it appends $w = -N - 65 \bmod \ell$ '0' bits, and finally, it appends a 64-bit representation of $(N + w + 65)/\ell$. This number is an integer due to the choice of w, and it represents the number of message blocks in the final, padded message.

Since it must be possible to encode the number of message blocks in the padded message within 64 bits, the maximum message length is 65 bits short of $2^{64} - 1$ message blocks. For the short variants, the maximum message length in bits is therefore $512 \cdot (2^{64} - 1) - 65 = 2^{73} - 577$, and for the longer variants it is $1024 \cdot (2^{64} - 1) - 65 = 2^{74} - 1089$.

3.7 Summary

First, a message which is to be digested by `Grøstl` is padded using the padding function `pad`. The hash function then iterates a compression function $f : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}^\ell$, which is based on two permutations P and Q. If the output size n of the hash function is at most 256 bits, we set $\ell = 512$. For the longer variants, we set $\ell = 1024$. Hence, we ensure that $\ell \geq 2n$ for all cases. The initial value of `Grøstl`-n is the ℓ-bit representation of n. At the end, the output of the last call to f is processed by the output transformation Ω, which reduces the output size from ℓ to n bits.

4 Design decisions and design features

In this section, we explain the design decisions made for `Grøstl` and some features of the `Grøstl` design. First, we list a number of advantages of `Grøstl` compared to many other hash functions proposed in the past.

- **Security proof of the construction.** The compression function construction used in `Grøstl` is provably collision resistant and preimage resistant assuming that the permutations P and Q are ideal. The `Grøstl` construction was proved to be indifferentiable from a random oracle assuming that the permutations P and Q are ideal and independent from each other. See Section 4.1.
- **Flexibility.** The algorithm can be efficiently implemented on many platforms. The security parameter r, the number of rounds, can be easily changed.
- **Simplicity.** Both the construction and the design of the permutations are simple and easy to understand and remember.
- **Familiarity.** Being based on the well known Rijndael design, most cryptographers and cryptographic software implementors will quickly feel acquainted with `Grøstl`. Moreover, the design principles behind Rijndael have already proven themselves advantageous.

4.1 The security of the construction

In general, security reduction proofs for hash functions address how secure they are when they are instantiated with ideal components. The proofs aim to estimate the security levels of the hash functions with respect to the standard properties such as collision resistance and (second) preimage resistance as well as for the strong security notions such as indifferentiability from the random oracle. In general, security reduction proofs may be given for the compression functions, hash functions or both.

In this direction, the compression function f was proved to be secure assuming that the two permutations P and Q are ideal [32]. The security proof states that at least $2^{\ell/4}$ evaluations of P and/or Q are required to find a collision for the hash function that iterates f, and that at least $2^{\ell/2}$ evaluations are required to find a preimage. Note that these levels are the square root of the security levels for an ideal compression function. However, since $\ell \geq 2n$, internal collision and preimage attacks on the hash functions have complexities of at least $2^{n/2}$ and 2^n. This analysis assumes that the ℓ output bits of the last call to f are the final output of the hash function. However, in `Grøstl`, an output transformation is applied. We discuss this output transformation in Section 4.6.

The `Grøstl` construction was also proved to be indifferentiable from a random oracle [3] upto the birthday bound. This result states that when the permutations P and Q are assumed ideal and independent from each other, `Grøstl` behaves like a random oracle for up to $O(2^{n/2})$ queries.

We remark that although these security proofs assume that the permutations P and Q are ideal, we do not claim their ideality. We only use these security proofs to show that the compression function and hash function constructions are sound. This is similar to using the security proof [16] of one of the PGV constructions [81] to show that this construction is sound, without claiming that the underlying block cipher is ideal. On the other hand, an attack that demonstrates non-ideality of the permutations does not necessarily extend to an attack on the hash function.

4.2 AddRoundConstant

The purpose of adding round constants is to make each round different and at the same time this provides a natural opportunity to make P and Q independent from each other. If the rounds are all the same, then fixed points x such that $R(x) = x$ for the round function R extend to the entire permutation. For example, if $P = R^{10}$, then fixed points for R^2 and R^5 would also extend to P. Therefore, one can expect several fixed points for P, whereas for an ideal permutation only a single fixed point is expected. By choosing round-dependent constants for AddRoundConstant, we expect the number of fixed points of P and Q to be 1.

In addition, by having different round constants for AddRoundConstant in P and Q, the internal differential attack [80], which considers differences between the permutations P and Q, can be made infeasible. Hence, the consequences of this differential attack, such as the distinguisher for the compression function [80] and collisions for the reduced round `Grøstl` function [43], can be thwarted.

4.3 SubBytes

The SubBytes transformation is the only non-linear transformation in `Grøstl`. It uses the same s-box as used in Rijndael. For a walk-through of its properties, we refer to one of [23, 24].

The choice for this particular transformation was driven by the following reasoning:

- Size: 8-bit s-boxes are a convenient trade-off between implementation aspects (smallest word size on popular platforms) and cryptanalytic considerations. On the other hand, there are $2^8!$ different permutations to choose from.
- Single s-box rather than many different s-boxes: this is again a trade-off between implementation and cryptanalytic considerations.
- No random s-box: A structured s-box allows for significantly more efficient hardware implementation than a random s-box.
- The particular structure of the chosen s-box was already proposed in 1993 [76] and has therefore undergone a long period of study.
- Since the s-box is inherited from the AES, implementation aspects (especially in hardware) are well studied.

4.4 ShiftBytes and ShiftBytesWide

We had two design criteria for ShiftBytes and ShiftBytesWide. First, we needed shift values which result in optimal diffusion. Let $\nu_{t,c}(a_{i,j})$ be the number of times that a state byte $a_{i,j}$ affects every state byte of column c after t rounds. In detail, $\nu_{t,c}(a_{i,j})$ defines how often (or in how many ways) every state byte of column c depends on $a_{i,j}$. Hence, we have full diffusion after t rounds if $\nu_{t,c}(a_{i,j}) \geq 1$ for all columns c and state bytes $a_{i,j}$. In other words, each state byte is affected by every state byte $a_{i,j}$ at least once. Let t^* be the value of t for which this happens. Then we get optimal diffusion, if $\min(\nu_{t^*,c}(a_{i,j}))$ is maximal for a specific geometry.

Second, to make P and Q more independent form each other, we use different shift values in P and Q. In more detail, we use shift values in Q such that no row is shifted by the same amount as in P, and such that the resulting states in P and Q are not simply shifted versions of each other. This way, it becomes much more difficult to ensure that differences or any other pattern in P and Q may line-up or cancel each other. We achieve this property using shift values in Q with a different (halved or doubled) slope than in P.

The shift values used for P_{512} are the most obvious ones. For Q_{512} we used the same shift values in a different order to get a halved slope (see Figure 6). Both cause optimal diffusion after two rounds. For P_{1024} and Q_{1024} (ShiftBytesWide) we have searched for shift values with optimal diffusion after three rounds (two rounds is not possible) and get optimal diffusion if $\min(\nu_{3,c}(a_{i,j})) = 2$. For P_{1024}, we have chosen the first set of such values when sorted in lexicographical order. Again for Q_{1024}, we used the same shift values as in P_{1024} in a different order to get optimal independence (see Figure 7).

4.5 MixBytes

The main design goal of the MixBytes transformation is to follow the wide trail strategy. Hence, the MixBytes transformation is based on an error-correcting code with the MDS (maximum distance separable) property. This ensures that both the differential and linear branch number is 9. In other words, a difference in $k > 0$ bytes of a column will result in a difference of at least $9 - k$ bytes after one MixBytes application.

Since there exist many MDS codes, we have chosen a code which can be implemented efficiently in many settings. The MixBytes transformation multiplies each column of A with the MDS matrix $B = \mathrm{circ}(02, 02, 03, 04, 05, 03, 05, 07)$ (see Section 3.4.5) over the finite field $\mathbb{F}_{256}$. In most environments, the multiplication with a constant of this matrix is the most expensive part. The implementation costs can be reduced by using constants of low degree. The minimum degree of the constants for an MDS code of size 8 is 2. However, this comes at a higher cost for the additions due to a slightly higher Hamming weight of the elements. Therefore, we have chosen a set of values where we can compensate these costs by the possibility of combining more intermediate results during the matrix multiplication. Especially on 8-bit platforms, this results in more efficient implementations.

4.6 Output transformation

Since the size of the chaining variables is larger than the required output size, an output transformation is needed. Simple truncation would be a possibility. However, since the compression function is not ideal (see Section 7.2), we chose to apply a function which is believed to be one-way and collision resistant, but does not compress before the truncation.

Let $\omega(x) = P(x) \oplus x$. The Matyas-Meyer-Oseas construction [64] for hash functions based on block ciphers provides a compression function g based on the encryption function E_K (with K being the key) as follows:

$$g(h, m) = E_h(m) \oplus m.$$

This function g has been proved to provide a collision resistant and one-way hash function when iterated in the Merkle-Damgård mode [16], under the assumption that E is an ideal block cipher. This implies that g is collision resistant and one-way if h is fixed, since this corresponds to hashing a one-block message. Hence, $\tilde{g}(m) = E_{h^*}(m) \oplus m$, where h^* is a constant, is one-way and collision resistant as well. Since $\tilde{g} = \omega$ with $P = E_{h^*}$, we believe that ω is one-way and collision resistant. This seems to make it difficult to attack `Grøstl` via the output transformation.

4.7 Number of rounds

The choice of the (recommended) number of rounds is primarily based on the cryptanalysis results described in Section 7. The square/integral attack indicates that the permutations might be distinguishable from ideal if the number of rounds is 7 or less in the short variants, and 9 or less in the long variants. The classic rebound attack and its developments show that finding collisions for the compression function of short and long variants is difficult beyond 7 rounds due to insufficient degrees of the freedom. For a similar reason, collision attacks on the hash function are currently limited to 3 rounds. Moreover, having two distinct permutations in the compression function avoids the internal differential attacks that aim to distinguisher the compression function or attack the hash function by analyzing differences between the two permutations. To summarise, we believe that the final choice of the number of rounds provides a reasonably large security margin for the `Grøstl` hash functions.

4.8 Absence of trap-doors

It should be clear that all constants used in `Grøstl`, including the s-box, have been selected in a way that does not leave enough freedom to deliberately insert trap-doors in the hash function. In general, we faithfully declare that we have not inserted any hidden weaknesses in `Grøstl`.

5 Alternative descriptions of `Grøstl`

The alternative descriptions of functions serve several purposes. They potentially bring greater insights into its security, and may also lead to more efficient implementations. In the standard description of `Grøstl`, the hash function iterates a permutation-based compression function, and then applies an output transformation to form the final hash of a message. However, as we shall see in this section, there are other ways of describing `Grøstl`.

5.1 The output transformation as a compression function call

The output transformation is defined as $\omega(x) = \mathrm{trunc}_n(P(x) \oplus x)$. Notice that $\omega(x) = \mathrm{trunc}_n(f(x, 0^\ell) \oplus Q(0^\ell))$. Hence, if H is the `Grøstl` hash function, $\tilde{H}$ is `Grøstl` *without the output transformation* and M is the already padded message, then $H(M) = \mathrm{trunc}_n(\tilde{H}(M\|0^\ell) \oplus Q(0^\ell))$, which is also illustrated by Figure 9. Since the XOR with $Q(0^\ell)$ has no cryptographic significance, we may ignore it and consider the description $\mathrm{trunc}_n(\tilde{H}(M\|0^\ell))$. The suffix 0^ℓ can be seen as an additional padding block.

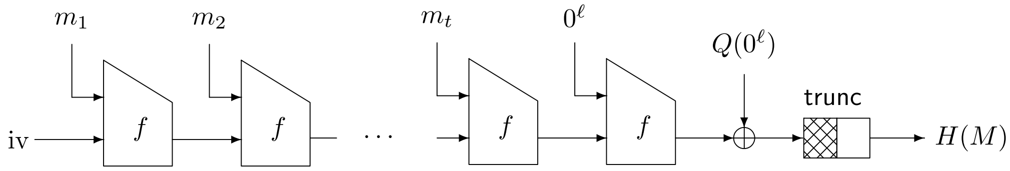

Figure 9: An alternative description of the `Grøstl` hash function.

This description more clearly shows the validity of Kelsey's observation that `Grøstl` without truncation does not protect against length extension attacks as described in Section 7.5. What precludes an attack based on this observation is the truncation from ℓ to n bits. Since at least n bits are dropped in this truncation, the probability of correctly guessing those bits is about 2^{-n}. The alternative description can also be seen as an indication that `Grøstl` is in fact an instance of the chop-MD construction, which prevents length extension attacks [20].

Some implementations of `Grøstl` might benefit from this alternative description. It shows that one does not have to specifically implement an output transformation function; the compression function can be used instead. Although this is not likely to improve the speed of implementations, it might reduce code size or area.

Finally, the alternative description shows that the "$P(x) \oplus x$" part of the output transformation does not have unexpected negative side effects. Hence, it does not lead to attacks that would not be possible with mere truncation. Since, as mentioned in Section 7.5, mere truncation leads to attacks that are not possible with the true definition of the `Grøstl` output transformation, we can conclude that the "$P(x) \oplus x$" part strictly improves the security of the hash function.

5.2 Tessaro's observation

Similar to the above description of `Grøstl`, Tessaro [84] observed that

$$H(M) = \mathrm{trunc}_n(\hat{H}(M \| Q^{-1}(0^\ell)) \oplus Q^{-1}(0^\ell)) \ ,$$

where $\hat{H}$ is the Merkle-Damgård with Permutation (MDP) iteration [41] of `Grøstl`'s compression function, with permutation $\pi(x) = x \oplus Q^{-1}(0^\ell)$.

5.3 Barreto's observation

Barreto [6] observed that the `Grøstl` compression function can be seen as an Even-Mansour cipher [29] in Davies-Meyer mode, which is defined as $f(h, m) = E_m(h) \oplus h$ for a block cipher E keyed via m. In the case of `Grøstl`, the block cipher is defined as $E_k(x) = P(k \oplus x) \oplus Q(k)$, where Q can be seen as a key schedule. In other words, the key is XORed with the plaintext (pre-whitening), the resulting value is permuted, and the output is XORed with a permuted version of the key (post-whitening).

6 Modes of use for `Grøstl`

`Grøstl` can be used in a "randomisation mode", e.g., as a message authentication code. Such modes include an additional input, which can be a key, a salt, a randomisation value, etc. We believe that `Grøstl` is secure when used in existing randomisation modes making use of a hash function, but we also propose a dedicated MAC mode for `Grøstl`.

6.1 Message authentication

HMAC [9, 74] is a method of constructing a message authentication code (MAC) from a hash function. Given a message M, a key K and a hash function H, the HMAC construction is defined as follows.

$$\mathrm{HMAC}(K, M) = H(\overline{K} \oplus \mathsf{opad} \| H(\overline{K} \oplus \mathsf{ipad} \| M)),$$

where $\overline{K}$ is K padded to a length equal to the block length of the hash function, and ipad and opad are two different constants as defined in [9]. HMAC has been proven to be secure if the compression function of the underlying hash function is a “dual” PRF [8]. A compression function is a dual PRF if it is a PRF when keyed via either the message block or the chaining input. We believe HMAC based on `Grøstl` is a secure MAC.

The HMAC construction requires two calls to the hash function, which in the case of `Grøstl` means that the output transformation must be evaluated twice. A more efficient method is the *envelope construction* [88]:

$$\mathrm{MAC}(K, M) = H(\overline{K}\|\overline{M}\|K), \tag{2}$$

where $\overline{M}$ is M padded to a multiple of ℓ bits, and $\overline{K}$ is K padded to ℓ bits. We propose this envelope construction as a dedicated MAC mode using `Grøstl`. This construction has been proved to be a secure MAC under similar assumptions as HMAC [95]. For the security proof to hold, the key must be processed in blocks that are separate from the blocks of the message M, which explains the additional padding required.

6.2 Randomised hashing

In order to free the security of digital signatures from relying on the collision resistance of a hash function, the input message to the hash function can be randomised using a fresh random value z for every signature following the technique outlined in [26, 38]. The randomised message is then processed using the hash function. This procedure is called randomised hashing. Let the message be M, padded to a multiple of the message block length, and split into message blocks $m_1, \ldots, m_t$. The randomised variant $\tilde{H}$ of the hash function H given randomisation value z is then (roughly) defined as

$$\tilde{H}(z, M) = H(z\|(m_1 \oplus z)\|(m_2 \oplus z)\| \ldots \|(m_t \oplus z)).$$

We believe `Grøstl` to be suitable for use in this randomisation mode. When `Grøstl` is used in the mentioned randomisation mode, we restrict the length of the randomisation value to at most n bits.

Being suitable for randomised hashing requires that the following attack [75] has complexity at least 2^{n-k}. The attacker chooses a message M of length at most 2^k bits. The attacker then receives a randomly chosen randomisation value z (not under the control of the attacker). The value $y = \tilde{H}(z, M)$ is computed, and the attacker's task is now to find a pair $(z^*, M^*) \neq (z, M)$ such that $\tilde{H}(z^*, M^*) = y$. In addition, the wide-pipe mode of operation in `Grøstl` with an internal state size at least twice of the digest size avoids some online birthday forgery attacks [34] on the digital signatures based on randomised mode of `Grøstl`. Note that these online birthday forgery attacks apply [34] to the digital signatures based on the randomised hash function modes proposed in [26, 38].

6.3 Security claims for the mentioned modes of operation

We claim the following security levels for the applications where `Grøstl`-n is deployed. The claimed complexity of the “randomised hashing attack” assumes a first message of at most 2^k blocks.

Attack type	Claimed complexity	Best known attack
Forgery on n-bit HMAC	$2^{n/2}$	2^n
Key recovery on n-bit HMAC	$2^{\|K\|}$	$2^{\|K\|}$
Forgery on n-bit envelope MAC	$2^{n/2}$	2^n
Key recovery on n-bit envelope MAC	$2^{\|K\|}$	$2^{\|K\|}$
Randomised hashing	2^{n-k}	2^n

7 Cryptanalytic results

In this section, we describe some preliminary cryptanalysis results on `Grøstl`, and we state our security claims

7.1 Attacks exploiting properties of the permutations

We first consider well known attack methods that aim to exploit potential weaknesses in the permutations P and Q.

7.1.1 Differential cryptanalysis

The permutations P and Q have diffusion properties according to the wide trail design strategy. Since the MixBytes transformation has branch number 9, and ShiftBytes is diffusion optimal (moves the bytes in each column to eight different columns), it is guaranteed that for `Grøstl` there are at least $9^2 = 81$ active s-boxes in any four-round differential trail [24, Theorem 9.5.1]. Note that this holds for `Grøstl`-256 as well as for `Grøstl`-512. Hence, there are at least $2 \cdot 81 = 162$ and $3 \cdot 81 = 243$ active s-boxes in any eight-round, respectively twelve-round differential trail. This, combined with the maximum difference propagation probability of the s-box of 2^{-6}, means that the probabilities of any differential trail (assuming independent rounds) over eight and twelve rounds (for either P or Q) are expected to be at most $2^{-6 \cdot 162} = 2^{-972}$, respectively 2^{-1458}. Therefore, in a classical differential attack where one specifies a differential trail for every round for both P and Q, there is only a very small chance that this would lead to a successful attack for `Grøstl`-256 and `Grøstl`-512.

In the collision attack [79] on Grindahl-256 [53], the low probability of any difference propagation through the s-box is circumvented by ignoring the actual values of differences, and instead only considering whether a byte is active or not. Since in Grindahl, a message block overwrites part of the state, the actual values of any differences in this part of the state are irrelevant. This approach means that the probabilistic behaviour of the hash function is now related to the MixColumns/MixBytes transformation, since without knowing the value of an input difference, one cannot predict the output difference. On the other hand, the number of degrees of freedom is essentially doubled, since one does not need to consider a fixed input/output difference. The relatively slow diffusion of Grindahl-256 combined with the continuous ability to influence the state led to the collision attack. In the `Grøstl` permutations, this approach will result in a complexity well above that of a birthday attack because diffusion is more effective (requiring only two rounds compared to four), and the attacker does not have continuous control over parts of the state. Moreover, since no part of the state is discarded (until the output transformation in the end), the actual value of a difference is significant and therefore, it seems that any input or output difference will have to (probabilistically) match a given difference.

7.1.2 Rebound attacks

The rebound attack [59, 69] is a new attack method for the cryptanalysis of hash functions. It gives the best known results for a number of AES-based hash functions and many SHA-3 candidates [51, 52, 58, 63, 66, 68, 82, 83, 93, 94].

In general, the rebound attack works with any differential or truncated differential. However, the diffusion properties of AES based hash functions allow a very simple construction of good truncated differential paths, which facilitates the analysis.

The rebound attack is most successful if a high number of degrees of freedom is available. Therefore, attacks on hash functions with a key schedule to the underlying block cipher or other sources of freedom are more likely to succeed (see the attacks on ECHO [83], LANE [63] or Whirlpool [58]). However, Grøstl has been designed to limit the degrees of freedom available to an attacker. Moreover, in attacks on the *hash* function, much fewer degrees of freedom are available (compared to an attack on the compression function or permutation). The best attacks on the hash function for Grøstl-256 and Grøstl-512 are for 3 rounds (out of 10 and 14), respectively.

On the other hand, the best (collision) attacks on the compression functions are for 6 rounds of Grøstl-256 and Grøstl-512. An extension of these attacks to 7 rounds might be possible. However, more rounds seems unlikely, due to the limited degrees of freedom in the attacks. Therefore, Grøstl still enjoys a comfortable security margin.

7.1.3 Internal differential attack

Another type of differential cryptanalysis traces differences between P and Q, instead of the more traditional tracing of differences between pairs of inputs [80]. This method has in particular been used in collision attacks on reduced-round versions of the Grøstl-0 hash function [43], and in the distinguishing attacks on the compression function [80] of Grøstl-0. However, the two permutations P and Q in Grøstl are more distinct and hence the internal differential attack gets infeasible, even for a small number of rounds.

7.1.4 Linear cryptanalysis

Linear and differential trails propagate in a very similar way. Since the MixBytes transformation has linear branch number 9, it is guaranteed that for Grøstl there are at least $9^2 = 81$ active s-boxes in any four-round linear trail [24, Theorem 9.5.1]. Hence, there are at least $2 \cdot 81 = 162$ and $3 \cdot 81 = 243$ active s-boxes in any eight-round, respectively twelve-round linear trail. Since the s-box has maximum correlation of 2^{-3}, the maximum correlation for any four-round linear trial is $2^{-3 \cdot 81} = 2^{-243}$. This means that the correlation of any linear trail over eight and twelve rounds (for either P or Q) are expected to be at most $2^{-3 \cdot 162} = 2^{-486}$, respectively 2^{-729}.

7.1.5 Integrals

Some of the best known attacks on AES are based on so-called integrals [22, 54]. Integrals can be specified also for Grøstl, and although it has not been shown how to utilise integrals in attacks on a hash function, they might say something about the used structure.

Integrals for Grøstl-256 are very similar to integrals for AES. We have identified an integral with 2^{120} texts over 6 rounds of Grøstl-256. The texts in this collection are balanced in every byte of the input and output. Also, we identified an integral with 2^{120} texts over 7 rounds of Grøstl-256. The texts in this collection are balanced in every byte of the input and balanced in every bit of the output. These are similar to the integrals for AES reported in [54]. Note that for AES reduced to 7 rounds, the last round is special. This is not the case for Grøstl.

For Grøstl-512 we have identified integrals for 8 and 9 rounds. For an 8-round variant the texts are balanced in every byte of the input and output; for an 9-round variant the texts are balanced in every byte of the input and in every bit of the output. For both these integrals, the number of texts is 2^{704}.

With the chosen number of rounds in the Grøstl permutations, 10 respectively 14, we believe it is safe to conclude that integrals cannot be used to show any non-random behaviour of Grøstl.

7.1.6 Algebraic cryptanalysis

It is well-known [21] that one can establish 39 quadratic equations (equations of degree two) over $\mathbb{F}_2$ in the input and output bits of the AES s-box, and there is one additional quadratic equation of probability $\frac{255}{256}$ for the AES s-box. Hence, this is also the case for the s-box in `Grøstl`. There is a total of 200 s-box applications for one encryption of the AES. Using these 40 equations for AES, it has been shown that from a single AES encryption, one can establish a set of 8000 quadratic equations in 1600 variables (unknowns). The solution of these equations can be used to derive the value of the secret key used in the encryption. The time complexity to solve the above mentioned system of equations for AES is unknown; to the best of our knowledge, it has not been shown that this can lead to an attack faster than an exhaustive search for the key.

For comparison, there is a total of 1280 s-box applications in the compression function of `Grøstl`-256 and a total of 3584 s-box applications in the compression function of `Grøstl`-512. It is clear that there are some advantages in an algebraic attack on a hash function compared to a similar attack on a block cipher, since there are no secret keys in the former. However, given that the number of s-box applications is much larger for `Grøstl` than for AES, we think it is safe to conclude that if an efficient algebraic attack method should be found which exploits the quadratic s-box equations in `Grøstl`, then a similar attack would be able to break the AES.

7.1.7 Zero-sum partitions

Zero-sum properties have been investigated in [4, 17, 19, 54] with cryptanalytic results on, among others, the full Keccak permutation [13]. A zero-sum partition for an n-bit permutation P of size 2^k is a set of 2^{n-k} zero-sums. Let R_{256} be the round transformation of `Grøstl`-256 and R_{512} be the round transformation of `Grøstl`-512 (details do not matter for our treatment here), then we observe that the degree of the algebraic normal form (ANF) of R_{256} and R_{512} is 7. We can bound the degree of the ANF for 4 rounds of R_{256} with 508, and for 5 rounds of R_{512} with 1022. This results in a zero-sum partition of size 2^{508} for the 8-round `Grøstl`-256 permutation and compression function and in a zero-sum partition of size 2^{1022} for the 10-round `Grøstl`-512 permutation and compression function.

7.2 Generic collision attacks

This section deals with collision attacks that do not depend on weaknesses in P and Q. We distinguish between collision attacks on the compression function, and collision attacks on the hash function. Collision attacks on the compression function, where the chaining input is determined by the attack (and is not under the direct control of the attacker), cannot be directly extended to cover the full hash function. The security proof of the construction (1) relates to collision attacks on the hash function. Hence, we cannot rule out the possibility of generic attacks on the compression function below the $2^{\ell/4}$ bound. However, there are good reasons to believe that the bound holds also for the compression function as will be shown next.

7.2.1 Collision attacks on the compression function

Wagner's generalised birthday attack [90] applies to the compression function f: form four lists via the two functions $f_P(x) = P(x) \oplus x$ and $f_Q(x) = Q(x) \oplus x$. Note that $f(h,m) = f_P(h \oplus m) \oplus f_Q(m)$. Find a quadruple (x, x^*, y, y^*) such that $f_P(x) \oplus f_P(x^*) \oplus f_Q(y) \oplus f_Q(y^*) = 0$. Then the two pairs $(x \oplus y, y)$ and $(x^* \oplus y^*, y^*)$ collide.

This attack has complexity $2^{\ell/3}$, and hence is faster than a birthday attack on the compression function. Note that this is still above the proven bound of $2^{\ell/4}$ and above the complexity of a birthday attack on the hash function, since $n \leq \ell/2$. The attack does not provide the attacker with much control over the chaining input, and hence we do not see any methods to extend the attack to the full hash function.

Wagner notes that if f_P and f_Q are considered random functions, then finding a quadruple (x, x^*, y, y^*) such that $f_P(x) \oplus f_P(x^*) \oplus f_Q(y) \oplus f_Q(y^*) = 0$ has complexity at least $2^{\ell/4}$. Assuming this is correct, the complexity extends to the full hash function (where the output transformation is omitted) via the same proof as that of the Merkle-Damgård construction [25, 71].

Wagner's generalised birthday attack is the best attack on the compression function we are aware of. We note that in a Merkle-Damgård hash function, a collision attack on the compression function always extends to a pseudo- or free-start collision attack on the hash function. Hence, Wagner's generalised birthday attack can be used to carry out a free-start collision attack on `Grøstl` in time $2^{\ell/3}$. Again, we remind the reader that this complexity is above the complexity of a birthday attack on `Grøstl`.

7.2.2 Collision attacks on the hash function

The construction (1) is provably collision resistant up to the level of $2^{\ell/4}$ permutation calls. Still, no collision attack of this complexity is known when the permutations are assumed to be ideal. The best known collision attack requires $2^{3\ell/8}$ permutation calls [32], but the true complexity in terms of compression function call equivalents is higher than $2^{\ell/2}$. Hence, a large security margin remains.

7.3 Generic attacks on the iteration

The internal state being *at least* twice the size of the hash value for all versions of `Grøstl`, generic attacks applying to the Merkle-Damgård construction cannot be applied to `Grøstl` directly via brute force or birthday attacks. However, since the construction used for `Grøstl` does not achieve security comparable to an ideal iterated hash function with the same internal state size, we do not claim that generic attacks do not apply using some other methods than the standard brute force and birthday attacks.

7.3.1 Multicollision attack

Recall that a d-collision is a set of d messages that all collide pairwise. The multicollision attack of Joux [46] on iterated hash functions applies also to `Grøstl`; the complexity to find a d-collision is roughly $\log_2(d)2^{\ell/2} \geq \log_2(d)2^n$. This should be compared to a brute-force multicollision attack on the hash function for which the complexity is around $(d!)^{1/d} \cdot 2^{n(d-1)/d}$. For values of d and n of cryptographic relevance, the brute-force attack is always faster than Joux's approach.

7.3.2 Second preimage attack

The second preimage attack of Kelsey and Schneier [50] on the Merkle-Damgård construction also seems to be complicated by the large internal state size. For an n-bit iterated hash function based on an n-bit compression function, given a first preimage of length 2^k message blocks this attack finds a second preimage of the same length in 2^{n-k} evaluations of the compression function. A variant of this attack was published in [2]. Using the techniques of [2, 50], the complexity of carrying out the second preimage attack on `Grøstl` given a 2^{64}-block first preimage is about $2^{\ell-64}$. For all the message digest sizes of `Grøstl`, this complexity is well above 2^{n-k}. Hence, our claimed security level for the second preimage resistance is at least 2^{n-k} for any first message of at most 2^k blocks. However, we do not know of an attack with complexity below 2^n.

7.3.3 Length extension attack

The length extension attack on Merkle-Damgård hash functions works as follows. Let (M, M^*) be a collision for the hash function H, with $|M| = |M^*|$. H pads M and M^* to $\overline{M}$ and $\overline{M}^*$ before hashing, and by choosing any message suffix y, we have that $B = \overline{M}\|y$ and $B^* = \overline{M}^*\|y$ also collide. Hence, a single collision gives rise to many new collisions that "come for free".

The length extension method is not trivial to carry out in `Grøstl`, unless the messages collide before the output transformation. Finding a collision before the output transformation takes time $2^{\ell/2} \geq 2^n$ by the birthday attack. As mentioned several times, there may be collision attacks on the hash function with the output transformation omitted, that have complexity below the birthday attack, but we do not know of any such attack.

A related weakness of the Merkle-Damgård transformation is the following. Assume the two values $H(M)$ and $|M|$ are known, but M itself is not. Knowing $|M|$, one also knows how M was padded, and hence for any suffix y, one may compute $H(\overline{M}\|y)$, where $\overline{M}$ is the padded version of M, without knowing M. This weakness leads to attacks when a Merkle-Damgård hash function underlies a secret prefix MAC. In `Grøstl`, this attack does not seem possible due to the output transformation.

7.4 Non-random properties of the compression function

Non-random properties of the `Grøstl` hash function are not known. Here we consider non-random properties of the compression function of `Grøstl`. Although attacks on the compression function do not necessarily translate to the attacks on the hash function, we claim that the `Grøstl` compression function is collision and (second) preimage resistant up to the level needed for the hash function. The collision and (second) preimage resistance properties of the compression function serve as a reassurance of the collision and preimage resistance properties of the `Grøstl` hash function. On the other hand, the `Grøstl` compression function is known to be non-random. Hence, the wide pipe and the strong output transformation are essential parts of the design. Nevertheless, here we give an incomplete list of known non-random properties of the compression function.

7.4.1 Fixed points

Most existing hash functions, for instance SHA-1 and SHA-2, are based on the Davies-Meyer construction [64], and hence *fixed points* can be easily found for these hash functions [72]. Some applications where this property can be used to attack hash functions have been identified, for instance, in finding an expandable message to carry out the second preimage attack [28, 50]. However, finding an expandable message is only one part of the second preimage attack, and in most cases it is not the most time-consuming task of the attack.

Fixed points can also be efficiently found for the compression function f of `Grøstl`: Choose m arbitrarily, and let $h = P^{-1}(Q(m)) \oplus m$. Then $f(h, m) = h$. Hence, h is computed as a (claimed) one-way function of m, and therefore is not under the direct control of the attacker.

In the case of `Grøstl`, we note that the internal state is at least twice the size of the hash value, and hence the cost of constructing, e.g., an expandable message using fixed points is expected to be about $2^{\ell/2} \geq 2^n$.

7.4.2 k-sums and differential q-multicollisions

Distinguishers based on k-sums (of value zero) and differential q-multicollisions [15] are easy to find for the compression function of `Grøstl`. We give one example for a 4-sum here. Let

$H_1 + H_2 + H_3 + H_4 = 0$ and $H_1 + H_2 = M_1 + M_2$, then $f(H_1, M_1) + f(H_2, M_2) + f(H_3, M_1) + f(H_4, M_2) = 0$, which is a 4-sum of value zero. Note that this also implies $H_1 + H_2 = H_3 + H_4 = \Delta_1$ and $f(H_1, M_1) + f(H_2, M_2) = f(H_3, M_1) + f(H_4, M_2) = \Delta_2$ which is a differential 2-multicollision.

7.4.3 Generalised birthday collisions

As observed in section 7.2, generalised birthday collision attack is applicable to the ℓ-bit compression function of `Grøstl` with a complexity of $2^{\ell/3}$.

7.4.4 Memoryless preimage attack

Memoryless preimage attack is applicable to the compression function of `Grøstl` in time $2^{\ell/2}$, where $\ell \geq 2n$ is the output size of the compression function. Note that for a given target T, one can compute M, X using cycle finding algorithms such that $T = H + P(H + M) + Q(M) = X + P(X) + M + Q(M)$ with $H = X + M$.

7.5 Kelsey's observations

Kelsey [48] noted that without truncation, the `Grøstl` hash function does not protect against length extension attacks, and he argues that the "$P(x) \oplus x$" part of the output transformation therefore accomplishes little security. However, as explained in Section 5.1, the "$P(x) \oplus x$" part in the output transformation still serves an important purpose: If the output of the last iteration of the compression function is merely truncated to form the output of the hash function, then Wagner's generalized birthday attack [90] on the compression function would extend to the hash function, and it would have a complexity of $2^{n/3}$ since it can be applied to the truncated (n-bit) hash value. With the "$P(x) \oplus x$" part, Wagner's generalized birthday attack has to be applied on an internal ℓ-bit value, and since $\ell \geq 2n$, the attack has complexity above the birthday attack on the hash function.

7.6 Security claims and summary of known attacks

With the number of rounds proposed in Section 3.4.6, we claim the following security levels for the `Grøstl`-n hash function. In the second preimage attack, the first preimage is assumed to be of length at most 2^k blocks.

Attack type	Claimed complexity	Best known attack
Collision	$2^{n/2}$	$2^{n/2}$
d-collision	$\lg(d) \cdot 2^{n/2}$	$(d!)^{1/d} \cdot 2^{n(d-1)/d}$
Preimage	2^n	2^n
Second preimage	2^{n-k}	2^n

Even though compression function attacks do not necessarily translate into attacks on the hash function, we claim the following properties for the compression function:

Attack type	Claimed complexity	Best known attack
Collision	$2^{\ell/4}$	$2^{\ell/3}$
Preimage	$2^{\ell/2}$	$2^{\ell/2}$

8 Implementation aspects

Like Rijndael, `Grøstl` can be efficiently implemented on a wide variety of processors and allows many trade-offs between resource requirements (memory, registers) and speed. In this section, we describe and estimate performance and resource requirements of implementations on 128–, 64–, 32–, and 8–bit architectures, as well as on ASICs and FPGA hardware. As `Grøstl` is designed to prevent preference for a particular word size, this will also allow efficient implementation of future architectures (like Intel's AVX with 256-bit registers [45]).

8.1 Software implementations

In software, `Grøstl` is targeted 64-bit processors, but performance is nearly as good on 32-bit processors offering MMX instructions, and we currently get the best performance on processors using the AES-NI instruction and 128-bit XMM registers.

8.1.1 128-bit processors

`Grøstl` can be implemented efficiently on processors featuring wide register sizes by exploiting parallelism of the round transformations. For example, the SSE instructions of modern processors provide additional 128-bit XMM registers which can be used by various types of `Grøstl` implementations. There are at least 3 different (cache-timing resistant) implementation strategies which make use of these 128-bit registers.

AES-NI implementation. The first implementation strategy is to use the Intel AES-NI instructions [44] to speed up the computation of `Grøstl`. Although `Grøstl` does not use the MixColumns transformation of the AES, `Grøstl` can still benefit from the AES-NI instructions. For example, we can use the `AESENCLAST` instruction (which computes only the last round of AES) to compute 16 S-box lookups in parallel. If the `Grøstl` state is stored in row ordering, the ShiftBytes computation of P and Q is a simple reordering of bytes and can be performed efficiently by the `PSHUFB` instruction of SSSE 3. Also the MixBytes transformation of `Grøstl` can be computed 16 times in parallel using XMM registers, 128-bit wide XORs and multiplications by 2. Table 1 shows the benchmark results of an assembly implementation running on an Intel Core i7 M620 processor with AES-NI instructions. Note that in this implementation, MixBytes consumes about 70% of the total computation time and we expect some speed improvements by developing a more efficient MixBytes computation.

Table 1: `Grøstl` performance using the Intel AES-NI instructions and 128-bit XMM registers.

Processor	Hash function	Speed (cycles/byte)
Intel Core i7 M620	`Grøstl`-224/256	12.45
	`Grøstl`-384/512	17.85

vperm implementation. Almost the same implementation can also be used on processors without the Intel AES-NI instructions by using the vperm (vector-permute) implementation of Mike Hamburg [39] for the SubBytes layer. The vperm approach allows to compute 16 S-box lookups in parallel within only a few cycles. By replacing the `AESENCLAST` instruction of the AES-NI implementation with the vperm implementation, we expect a speed close to those given in Table 1, even on processors without AES-NI instruction.

Bitsliced implementation. The third implementation which benefits from 128-bit XMM registers and SSE instructions are bitsliced implementations of `Grøstl`. Preliminary assembly implementations of `Grøstl`-0 show a speed of less than 30 cycles/byte on Intel Core2 Duo processors for the computation of a single message [85]. Additionally, bitsliced implementations get even more efficient if two or more messages are hashed in parallel [18].

8.1.2 64-bit processors

`Grøstl` can be efficiently implemented on 64-bit processors following a technique very similar to the efficient 32-bit implementation of Rijndael [23]. Consider an implementation of the round function of P_{512} focusing on the effect on column 0. Assume that the AddRoundConstant transformation adds the byte C to $a_{0,0}$. Note that the new column 0 after the round function has been applied depends solely on the 8 bytes $a_{i,i}$, $0 \leq i < 8$, because the ShiftBytes transformation moves these bytes into column 0.

As an example, the round function has the following effect on $a_{0,0}$, the new value of which we denote by $a'_{0,0}$.

$$\begin{aligned} a'_{0,0} \quad \leftarrow \quad & \mathsf{02} \times S(a_{0,0} \oplus C) \oplus \mathsf{02} \times S(a_{1,1}) \oplus \mathsf{03} \times S(a_{2,2}) \oplus \mathsf{04} \times S(a_{3,3}) \oplus \\ & \mathsf{05} \times S(a_{4,4}) \oplus \mathsf{03} \times S(a_{5,5}) \oplus \mathsf{05} \times S(a_{6,6}) \oplus \mathsf{07} \times S(a_{7,7}). \end{aligned}$$

Similarly, the effect on $a_{1,0}$ is

$$\begin{aligned} a'_{1,0} \quad \leftarrow \quad & \mathsf{07} \times S(a_{0,0} \oplus C) \oplus \mathsf{02} \times S(a_{1,1}) \oplus \mathsf{02} \times S(a_{2,2}) \oplus \mathsf{03} \times S(a_{3,3}) \oplus \\ & \mathsf{04} \times S(a_{4,4}) \oplus \mathsf{05} \times S(a_{5,5}) \oplus \mathsf{03} \times S(a_{6,6}) \oplus \mathsf{05} \times S(a_{7,7}). \end{aligned}$$

If we continue, we see that, e.g., $a_{0,0}$ affects every byte of the column by the addition of $b \times S(a_{0,0} \oplus C)$, where b is a value from the first column of the matrix B (defined in Section 3.4.5). Hence, when the column is represented by a 64-bit word in an implementation, we may compute the effect of $a_{0,0}$ on *all* bytes in the new column 0 by a single table lookup, the output of which is exactly 8 concatenations of $b \times S(a_{0,0} \oplus C)$, with b varying as defined by the matrix B. Let the table be T_0 containing 256 64-bit words. The value $T_0[i]$ at index i (ignore the addition of C for a moment) will then be

$$\mathsf{02} \times S(i) \parallel \mathsf{07} \times S(i) \parallel \mathsf{05} \times S(i) \parallel \mathsf{03} \times S(i) \parallel \mathsf{05} \times S(i) \parallel \mathsf{04} \times S(i) \parallel \mathsf{03} \times S(i) \parallel \mathsf{02} \times S(i),$$

interpreted as an 8-byte (64-bit) word. Here, we define the first byte of the word to mean the byte of row 0 in A. In practice, the most convenient ordering depends on the endianness of the processor (the ordering used above is more convenient on big-endian processors, whereas on little-endian processors the byte ordering should be reversed).

A byte in a different row affects the column in a different way, and hence we must define 8 different tables $T_0, \ldots, T_7$. The only difference between them is the ordering of the bytes; they are rotated versions of each other, since the matrix B is circulant. To save space, a single table can be used, and the rotations can be done afterwards.

To sum up, column 0 can be computed as

$$T_0[a_{0,0} \oplus C] \oplus T_1[a_{1,1}] \oplus T_2[a_{2,2}] \oplus T_3[a_{3,3}] \oplus T_4[a_{4,4}] \oplus T_5[a_{5,5}] \oplus T_6[a_{6,6}] \oplus T_7[a_{7,7}],$$

hence using 8 table lookups and 8 XORs (7 for all other columns, since adding C is only needed in column 0).

When the columns are internally represented as 64-bit words, in most programming languages we don't have direct access to the bytes $a_{i,i}$, and hence we must access them by a right-shift and a logical *and*. However, many processors provide instructions for accessing a particular byte of a word.

We note that this technique requires storing 8 tables of 256 64-bit words, taking up 16 kilobytes of memory. As mentioned, a single table of 2 kilobytes can be used instead, but then a rotation is needed for every 7 out of 8 table lookups. This can be generalised; with k tables, $8 - k$ rotations are needed for every 8 table lookups. A crude estimate on the performance loss with $0 < k \leq 8$ tables compared to 8 tables is a factor about $\frac{23-k}{15}$. This is based on the estimate that a rotation, a table lookup, and an XOR take about the same time to carry out.

8.1.3 32-bit processors

On a 32-bit processor the above technique cannot be applied directly, but there is a (slower) variant operating with 32-bit words. This method requires half the amount of memory compared to the 64-bit implementation described above, and (roughly) twice the amount of computation. For more details we refer to the Whirlpool specification [7]. The same time/memory trade-offs as mentioned above are possible.

On 32-bit microprocessors with SIMD instruction sets such as MMX, SSE, or SSE2, an implementation like the one described for 64-bit processors is possible. Some overhead is introduced compared to the implementation on a native 64-bit processor, but nevertheless, performance on such 32-bit processors is almost as good as on a 64-bit processor. Most modern 32-bit processors used in personal computers provide these instruction sets. These include virtually all Intel and AMD processors since 1997.

8.1.4 8-bit processors

On 8-bit processors, the round transformations can be applied individually on a byte-by-byte basis. Both the SubBytes and the MixBytes operation can be efficiently realised with lookups in small tables or computed without lookup tables. Various implementation techniques that allow a trade-off between memory usage and performance are possible. Especially in the computation of the MixBytes operation, many intermediate results can be reused depending on the memory requirements. Note that there is no setup time needed for the 8-bit implementation of `Grøstl`.

As an example for possible trade-offs, preliminary implementation results suggest that `Grøstl`-256 and `Grøstl`-224 can be implemented with a performance of (roughly) between 450 and 550 cycles/byte on an 8-bit AVR micro-controller (ATmega163)[2] using between 200 and 1000 bytes of RAM, and a code size of less than about 4KB. The code includes a 256-byte lookup table for SubBytes and up to two 256-byte lookup tables for the multiplication with the constant 02 and/or 04 in the finite field $\mathbb{F}_{256}$ for MixBytes. Depending on the desired Time/Memory trade-off, more RAM can be used to speed up the computation of the S-box lookups. On the other hand `Grøstl` can also be implemented very compact with a maximum of 164 bytes of RAM usage. Table 2 gives a brief overview of 3 Time/Memory trade-offs for 8-bit implementations of `Grøstl`-256.

8.2 Benchmarks on PC platforms

The performance in software of the submitted optimised `Grøstl` implementations in C has been tested and benchmarked on four different systems. The benchmarks presented below refer to the hash computation of long messages. For more detailed and continuously up-to-date benchmarks we refer to eBASH [12].

[2] Running at e.g., 8MHz with no operating system

Table 2: `Grøstl` performance of 8-bit implementations on the ATmega163 micro-controller. The performance is given for different Time/Memory trade-offs long messages. Results indicated by * are estimates based on an implementation for `Grøstl`-0.

Hash function	Processor	Version (state)	Memory (bytes)	Speed (cycles/byte)
`Grøstl`-256	ATmega163	high-speed (192)	994	469
	ATmega163	balanced (192)	226	531
	ATmega163	low-mem (128)	164	760*

The following implementations were benchmarked (`Grøstl`-384/512 has not yet been implemented in all of these versions).

Name	Short name	Language
Optimized 64-bit	`o64`	C
Optimized 32-bit	`o32`	C
`sphlib`[3]adapted to tweaked `Grøstl`	`sphlib`	C
MMX intrinsics	`mmx`	C with MMX intrinsics
Optimized for Core 2 Duo	`c2d`	C with inline assembly
Optimized for Opteron	`opt`	C with inline assembly
Optimized for Opteron, rounds unrolled	`optunr`	C with inline assembly
Optimized for AES-NI	`aesni`	C with inline assembly

Additional implementations are under development; examples are bitsliced and vperm implementations using the SSE3 and SSE4.1 instruction sets. We expect that these implementations have similar performances as the implementations of `Grøstl`-0.

8.2.1 System I

This system consists of an Intel Core 2 Duo E4600 processor running at 2.40 GHz. The operating system is Ubuntu 10.10. The installed compilers are `gcc` v.4.4.5 and Intel's C compiler (`icc`) v.12.0.0.084. The optimisation flags used varied, but included `-O3`, `-fno-regmove`, `-fmodulo-sched` for `gcc`, and `-fast` and `-O1 -xHost` for `icc`.

On this system, the benchmarked implementations are `opt64`, `c2d`, `opt` (since no `c2d` impl. was developed for `Grøstl`-384/512 yet), and `sphlib`. The benchmarks are sorted by their speed presented below.

[3]Based on the `Grøstl`-0 implementations from sphlib v.2.1, `http://www.saphir2.com/sphlib/`

Hash function	Compiler	Implementation	Speed (cycles/byte)
Grøstl-224/256	icc	c2d	22.5
	gcc	c2d	22.5
	gcc	opt64	25.7
	icc	opt64	26.4
	gcc	sphlib	28.5
	icc	sphlib	29.0
Grøstl-384/512	gcc	opt	37.4
	icc	opt	37.4
	icc	sphlib	42.8
	icc	opt64	43.1
	gcc	opt64	45.6
	gcc	sphlib	47.4

8.2.2 System II

This system consists of an Intel Core i7 M620 processor running at 2.67 GHz. The operating system is Debian 5.0. The installed compiler is `gcc` v.4.4.5. The optimisation flags used varied, but included `-O3`, `-fno-regmove`, and `-fmodulo-sched`.

On this system, the benchmarked implementations are `aesni`, `opt64`, `mmx`, `c2d`, `opt` `optunr`, and `sphlib`. The benchmarks are sorted by their speed and presented below.

Hash function	Compiler	Implementation	Speed (cycles/byte)
Grøstl-224/256	gcc	aesni	12.8
	gcc	optunr	19.5
	gcc	opt	20.9
	gcc	c2d	20.9
	gcc	opt64	22.6
	gcc	sphlib	25.1
Grøstl-384/512	gcc	aesni	18.2
	gcc	opt	32.8
	gcc	sphlib	41.3
	gcc	opt64	57.3
	gcc	c2d	-
	gcc	optunr	-

8.2.3 System III

This system consists of an AMD Opteron 6168 processor running at 1.9 GHz. The operating system is Red Hat Linux v.4.1.2-48. The installed compiler is `gcc` v.4.1.2. The optimisation flags used were always a combination of `-O3`, `-fno-regmove`, `-fmodulo-sched`, and `-funroll-loops`.

The benchmarked implementations are `opt64`, `opt`, `optunr` (not implemented for Grøstl-384/512), and `sphlib`. The benchmarks are presented below.

Hash function	Compiler	Implementation	Speed (cycles/byte)
Grøstl-224/256	gcc	optunr	19.5
	gcc	opt	20.7
	gcc	sphlib	31.1
	gcc	opt64	31.6
Grøstl-384/512	gcc	opt	32.1
	gcc	sphlib	45.7
	gcc	opt64	48.6

8.2.4 System IV

This system consists of an Intel Pentium M 760 processor running at 2.0 GHz. The operating system is Ubuntu 10.04. The installed compilers are `gcc` v.4.4.3 and `icc` v.12.0.1.107. The optimisation flags used were `-O3`, `-fno-regmove`, `-fmodulo-sched`, and `-msse` for `gcc`, and `-O3 -xHost` and `-O3 -msse2` for `icc`.

The implementations that were benchmarked are `opt32`, `sphlib`, and `mmx`. The benchmarks are presented below.

Hash function	Compiler	Implementation	Speed (cycles/byte)
Grøstl-224/256	icc	mmx	40.0
	gcc	mmx	43.9
	icc	sphlib	66.5
	gcc	sphlib	80.8
	icc	opt32	90.3
	gcc	opt32	92.7
Grøstl-384/512	icc	mmx	79.0
	icc	sphlib	94.1
	gcc	sphlib	112.8
	icc	opt32	124.8
	gcc	opt32	126.6

We expect that Grøstl-384/512 implementations can be improved significantly on this processor. Theoretically, one would expect Grøstl-384/512 to be running about 40% slower than Grøstl-224/256, although factors such as program code size etc. may be a cause for deviations from this estimate.

8.3 Hardware implementations

Potential settings and scenarios for hardware implementations can be at least as diverse as for software implementations. The many different ways to implement Grøstl allows for a wide range of trade-offs between throughput, latency, gate count, power consumption, etc[5, 10, 33, 40, 42, 47, 55, 86, 87]. Grøstl can be implemented efficiently on architectures with data paths starting from 8-bit up to 1024-bit. In the following, we briefly discuss estimates in various settings.

8.3.1 Low-gate count implementations

To illustrate the multitude of different implementation trade-off possibilities the design offers, we consider implementations where very small area requirements and very low-power requirements are important.

We estimate that `Grøstl`-256 and `Grøstl`-224 can be implemented on an ASIC with standard-cell libraries requiring an area of less than 15000 gate equivalents (GE). The dominating factor here is the memory. We use 12390 GE for register-based RAM in our estimate instead of RAM hard-macros. Since low gate count implementations are usually also low-power implementations, the register-based RAM can be used to minimise power consumption by clock gating. We base our estimates on numbers obtained from actual implementations of the AES and other algorithms [30, 31]. This results in 354 GE for the SubBytes and 800 GE for the MixBytes transformation of `Grøstl`-256. Table 3 gives an overview for all `Grøstl` variants.

Table 3: Estimates for a low-power architecture with an 8-bit data path implementation of `Grøstl`, that also has a low gate count.

Part	GE	
	`Grøstl`-224/256	`Grøstl`-384/512
RAM	12390	24780
SubBytes	354	354
MixBytes	800	800
Others (conservative)	1400	2000
Sum	< 15000	< 28000

8.3.2 High-throughput implementations

High-throughput implementations of `Grøstl` can be developed using data paths up to 512 or 1024 bit. Further, the execution of the two permutations can be implemented in parallel or pipelined and interleaved. This results in 1 cycle per round and 10 or 14 cycles per compression function computation. As an example, looking at implementation results of `Grøstl`-0, even on old and cheap manufacturing processes with 180nm structure size more than 6 Gbit/sec with a modest area requirement of less than 60kGE can be achieved.

8.4 Implementation attacks

Whenever a key is handled by a machine that implements a cryptographic mechanism in addition to inputs and outputs, various side-channel information may be available to an attacker. Sources for such side-channels can be (but are not limited to) timing information, power consumption and electromagnetic emanations, error messages, etc.

8.4.1 Cache based timing attacks

Cache based timing attacks have been mentioned, discussed and investigated in [11, 14, 56, 78, 89]. Bitslicing or vector-permute [39] techniques applied to `Grøstl`, as well as the Intel AES-NI instructions allow for implementations that are resistant against cache based timing attacks (also see Section 8.1.1). Note that bitslicing implementations are actually the fastest AES implementations on many modern platforms. Also for bitsliced implementations of `Grøstl`, the expected overhead is only about 50% compared to the table-based approach (estimate based on bitsliced implementations of `Grøstl`-0). Furthermore, the Intel AES-NI implementation of `Grøstl` is the fastest `Grøstl` implementation so far and vector-permute based implementations are expected to have very little performance overhead as well.

8.4.2 Power- and EM side-channel attacks

Published in 1999 [57], side-channel attacks that exploit information from the power consumption in the form of differential power analysis (DPA) attacks and electromagnetic emanations turn out to be a real threat for many implementations. Many generic (e.g., dual-rail logic) countermeasures and countermeasures specialised for particular algorithms have been proposed since then. Again, the similarity of our proposal to the AES allows to reuse many ideas from previous work.

Popular MAC implementations such as HMAC-SHA-1 and HMAC-SHA-2 have been exposed to DPA attacks [60, 65]. MACs constructed using block cipher based hash functions can be analysed against side channel attacks by assuming that the block cipher or the compression function is side channel resistant. Under this assumption, DPA attacks on several hash function based MACs including HMAC instantiated with the provably secure block cipher based hash functions were demonstrated in [35, 36, 77]. For the MAC modes of `Grøstl`, we note that these observations do not seem to be directly applicable.

8.4.3 On countermeasures

The in Section 8.4.1 mentioned work on constant-time implementations of AES, and the huge body of work on countermeasures against power- and EM side-channel attacks (see e.g., [61] for a good overview) which is also primarily applied to AES, give a sound basis to counter implementation attacks. On top of that, instruction set extensions that are frequently proposed by CPU manufacturers may be used as well. Preliminary implementation results (see also Section 8.1.1) suggest that the new crypto-related instructions, which Intel introduced in their CPUs [44], can efficiently be used to implement `Grøstl` in a constant-time manner and hence resistant against timing attacks. This yet again serves as a powerful illustration for the many ways `Grøstl` can be implemented.

9 Conclusion

The SHA-3 candidate `Grøstl` has been proposed. `Grøstl` is a permutation-based hash function, based on a construction which is provably collision resistant when the permutations are assumed to be ideal. The particular permutations used in `Grøstl` are based on components of the Rijndael block cipher. As an effect of this, `Grøstl` has excellent diffusion and confusion properties. The design of `Grøstl` is very simple and easy to understand. Therefore, it is relatively easy to identify possible attacks and thereby easy to gain confidence in the strength of the construction. We believe that `Grøstl` is a very strong hash function, yet it can be efficiently implemented on a wide range of platforms. Reference and optimised implementations, test vectors, this document and other information on `Grøstl` is available at [37].

Acknowledgements

A number of people contributed or influenced this proposal in some way. Here we want to thank them (in alphabetical order).

Zoran Milinkovic, for discussions on his implementations of earlier versions of the design. Thomas Peyrin, for discussions on early design considerations. Vincent Rijmen, for insightful comments on various aspects of the design. Günther Roland, for the AES-NI and 8-bit implementations of `Grøstl`. Stefan Tillich, for discussion on many implementation aspects, and having an influential role in the selection of the MixBytes and ShiftBytesWide transformations.

Jürgen Windhaber, for discussions on his implementations of earlier versions of the design.
Sébastien Zimmer, for providing us an early version of [32].

References

[1] R. J. Anderson and E. Biham. TIGER: A Fast New Hash Function. In D. Gollmann, editor, *Fast Software Encryption 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 1996.

[2] E. Andreeva, C. Bouillaguet, P.-A. Fouque, J. J. Hoch, J. Kelsey, A. Shamir, and S. Zimmer. Second Preimage Attacks on Dithered Hash Functions. In N. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008, Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[3] E. Andreeva, B. Mennink, and B. Preneel. On the Indifferentiability of the Grøstl Hash Function. In J. A. Garay and R. D. Prisco, editors, *Security and Cryptography for Networks, 7th International Conference, SCN*, volume 6280 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2010.

[4] J.-P. Aumasson and W. Meier. Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi. NIST mailing list, 2009.

[5] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan, and W. P. Marnane. FPGA Implementations of SHA-3 Candidates:CubeHash, Grøstl, Lane, Shabal and Spectral Hash. Cryptology ePrint Archive, Report 2009/342, 2009. `http://eprint.iacr.org/2009/342` (2011/01/15).

[6] P. S. L. M. Barreto. An observation on Grøstl. Comment submitted to the NIST hash function mailing list, `hash-forum@nist.gov`. Available: `http://www.larc.usp.br/~pbarreto/Grizzly.pdf` (2011/01/15), 2008.

[7] P. S. L. M. Barreto and V. Rijmen. The Whirlpool Hashing Function. Submitted to NESSIE, September 2000 and Adopted in the ISO/IEC 10118-3:2004. Revised May 2003. Available: `http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html` (2011/01/15).

[8] M. Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In C. Dwork, editor, *Advances in Cryptology – CRYPTO 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, 2006.

[9] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[10] J. A. Bernhard Jungk, Steffen Reith. On Optimized FPGA Implementations of the SHA-3 Candidate Grostl. Cryptology ePrint Archive, Report 2009/206, 2009. `http://eprint.iacr.org/2009/206`(2011/01/15).

[11] D. J. Bernstein. Cache-timing attacks on AES. Available: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (2010/01/15).

[12] D. J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to` (accessed 16 January, 2011).

[13] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak sponge function family, June 2010. SHA-3 candidate. Available: `http://keccak.noekeon.org/Keccak-main-2.1.pdf`(2011/01/15).

[14] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *International Conference on Information Technology: Coding and Computing (ITCC 2005), Proceedings*, volume 1, pages 586–591. IEEE Computer Society, April 2005.

[15] A. Biryukov, D. Khovratovich, and I. Nikolić. Distinguisher and Related-Key Attack on the Full AES-256. In S. Halevi, editor, *Advances in Cryptology – CRYPTO 2009, Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 231–249. Springer, 2009.

[16] J. Black, P. Rogaway, and T. Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2002.

[17] C. Boura, A. Canteaut, and C. D. Cannière. Higher-order differential properties of Keccak and Luffa. Cryptology ePrint Archive, Report 2010/589, 2010. `http://eprint.iacr.org/2010/589/`(2011/01/15).

[18] Çaŏdaş Çalik. Multi-stream and Constant-time SHA-3 Implementations, 2010. Available online: `http://www.metu.edu.tr/~ccalik/software.html#sha3` (2010/01/12).

[19] A. C. Christina Boura. Zero-Sum Distinguishers for Iterated Permutations and Application to Keccak-f and Hamsi-256. In *SAC*, LNCS. Springer, 2010. To appear.

[20] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[21] N. Courtois and J. Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In Y. Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[22] J. Daemen, L. R. Knudsen, and V. Rijmen. The Block Cipher Square. In E. Biham, editor, *Fast Software Encryption 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[23] J. Daemen and V. Rijmen. AES Proposal: Rijndael. AES Algorithm Submission, September 1999. Available: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf` (2011/01/15).

[24] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer, 2002.

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

[26] Q. Dang. Randomized Hashing for Digital Signatures, 2009. Available:`http://csrc.nist.gov/publications/PubsSPs.html` (2011/01/13).

[27] C. De Cannière and C. Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[28] R. D. Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, January 1999.

[29] S. Even and Y. Mansour. A Construction of a Cipher from a Single Pseudorandom Permutation. *Journal of Cryptology*, 10(3):151–162, 1997.

[30] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong Authentication for RFID Systems Using the AES Algorithm. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 357–370. Springer, 2004.

[31] M. Feldhofer and C. Rechberger. A Case Against Currently Used Hash Functions in RFID Protocols. In R. Meersman, Z. Tari, and P. Herrero, editors, *OTM Workshops (1)*, volume 4277 of *Lecture Notes in Computer Science*, pages 372–381. Springer, 2006.

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

[33] K. Gaj, E. Homsirikamol, and M. Rogawski. Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. In Mangard and Standaert [62], pages 264–278.

[34] P. Gauravaram and L. R. Knudsen. On randomizing hash functions to strengthen the security of digital signatures. In A. Joux, editor, *Advances in Cryptology – EUROCRYPT 2009, Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2009.

[35] P. Gauravaram and K. Okeya. An Update on the Side Channel Cryptanalysis of MACs Based on Cryptographic Hash Functions. In *Progress in Cryptology – INDOCRYPT 2007, Proceedings*, volume 4859 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 2007.

[36] P. Gauravaram and K. Okeya. Side Channel Analysis of Some Hash Function Based MACs: A Response to SHA-3 Requirements. In L. Chen, M. D. Ryan, and G. Wang, editors, *International Conference on Information and Communications Security – ICICS 2008, Proceedings*, volume 5308 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2008.

[37] The `Grøstl` web page. `http://www.groestl.info`.

[38] S. Halevi and H. Krawczyk. Strengthening Digital Signatures Via Randomized Hashing. In C. Dwork, editor, *Advances in Cryptology – CRYPTO 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2006.

[39] M. Hamburg. Accelerating AES with Vector Permute Instructions. In *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2009.

[40] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. K. Gürkaynak. Developing a Hardware Evaluation Method for SHA-3 Candidates. In Mangard and Standaert [62], pages 248–263.

[41] S. Hirose, J. H. Park, and A. Yun. A Simple Variant of the Merkle-Damgård Scheme with a Permutation. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 113–129. Springer, 2007.

[42] E. Homsirikamol, M. Rogawski, and K. Gaj. Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. Cryptology ePrint Archive, Report 2010/445, 2010. `http://eprint.iacr.org/2010/445`(2011/01/15).

[43] K. Ideguchi, E. Tischhauser, and B. Preneel. Improved collision attacks on the reduced-round Grøstl hash function. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, editors, *Information Security Conference (ISC) 2010, Proceedings*, volume 6531 of *Lecture Notes in Computer Science*. Springer, 2011. To appear.

[44] Intel Corporation. Advanced Encryption Standard (AES) Instructions Set. Available: `http://softwarecommunity.intel.com/articles/eng/3788.htm` (2008/10/30).

[45] Intel Corporation. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. Available: `http://softwarecommunity.intel.com/articles/eng/3775.htm` (2008/10/30).

[46] A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[47] B. Jungk and S. Reith. On FPGA-based implementations of Grøstl. Cryptology ePrint Archive, Report 2010/260, 2010. `http://eprint.iacr.org/2010/260`(2011/01/15).

[48] J. Kelsey. Some notes on Grøstl. Comment submitted to the NIST hash function mailing list, `hash-forum@nist.gov`. Available: `http://ehash.iaik.tugraz.at/uploads/d/d0/Grostl-comment-april28.pdf` (2011/01/15), 2009.

[49] J. Kelsey and S. Lucks. Collisions and Near-Collisions for Reduced-Round Tiger. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.

[50] J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[51] D. Khovratovich, I. Nikolic, and C. Rechberger. Rotational Rebound Attacks on Reduced Skein. In M. Abe, editor, *Advances in Cryptology – ASIACRYPT 2010, Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2010.

[52] D. Khovratovich, I. Nikolic, and C. Rechberger. Rotational Rebound Attacks on Reduced Skein. In M. Abe, editor, *Advances in Cryptology – ASIACRYPT 2010, Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2010.

[53] L. R. Knudsen, C. Rechberger, and S. S. Thomsen. The Grindahl Hash Functions. In A. Biryukov, editor, *Fast Software Encryption 2007, Proceedings*, volume 4593 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2007.

[54] L. R. Knudsen and V. Rijmen. Known-Key Distinguishers for Some Block Ciphers. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2007.

[55] K. Kobayashi, J. Ikegami, S. Matsuo, K. Sakiyama, and K. Ohta. Evaluation of hardware performance for the sha-3 candidates using sasebo-gii. Cryptology ePrint Archive, Report 2010/010, 2010. `http://eprint.iacr.org/2010/010`(2011/01/15).

[56] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96, Proceedings*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[57] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO '99, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[58] M. Lamberger, F. Mendel, C. Rechberger, V. Rijmen, and M. Schläffer. Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009, Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2009.

[59] M. Lamberger, F. Mendel, C. Rechberger, V. Rijmen, and M. Schlffer. The Rebound Attack and Subspace Distinguishers: Application to Whirlpool. Cryptology ePrint Archive, Report 2010/198, 2010. `http://eprint.iacr.org/2010/198`(2011/01/15).

[60] K. Lemke, K. Schramm, and C. Paar. DPA on n-bit sized boolean and arithmetic operations and its application to IDEA, RC6, and the HMAC-construction. In *Cryptographic Hardware and Embedded Systems – CHES 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2004.

[61] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007.

[62] S. Mangard and F.-X. Standaert, editors. *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*. Springer, 2010.

[63] K. Matusiewicz, M. Naya-Plasencia, I. Nikolić, Y. Sasaki, and M. Schläffer. Rebound Attack on the Full LANE Compression Function. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009, Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 106–125. Springer, 2009.

[64] S. M. Matyas, C. H. Meyer, and J. Oseas. Generating strong one-way functions with crypographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

[65] R. P. McEvoy, M. Tunstall, C. C. Murphy, and W. P. Marnane. Differential Power Analysis of HMAC Based on SHA-2, and Countermeasures. In *Workshop on Information Security Applications – WISA 2007, Revised Selected Papers*, volume 4867 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007.

[66] F. Mendel, T. Peyrin, C. Rechberger, and M. Schläffer. Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In M. J. Jacobson, V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography 2009, Proceedings*, volume 5867 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2009.

[67] F. Mendel, B. Preneel, V. Rijmen, H. Yoshida, and D. Watanabe. Update on Tiger. In R. Barua and T. Lange, editors, *Progress in Cryptology – INDOCRYPT 2006, Proceedings*, volume 4329 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2006.

[68] F. Mendel, C. Rechberger, and M. Schläffer. Cryptanalysis of Twister. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *Applied Cryptography and Network Security 2009, Proceedings*, volume 5536 of *Lecture Notes in Computer Science*, pages 342–353. Springer, 2009.

[69] F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and `Grøstl`. In O. Dunkelman, editor, *Fast Software Encryption 2009, Proceedings*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2009.

[70] F. Mendel and V. Rijmen. Cryptanalysis of the Tiger Hash Function. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 536–550. Springer, 2007.

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

[72] S. Miyaguchi, K. Ohta, and M. Iwata. Confirmation that Some Hash Functions Are Not Collision Free. In I. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90, Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 1991.

[73] National Institute of Standards and Technology. FIPS PUB 180-2, Secure Hash Standard. Federal Information Processing Standards Publication 180-2, U.S. Department of Commerce, August 2002.

[74] National Institute of Standards and Technology. FIPS PUB 198, The Keyed-Hash Message Authentication Code (HMAC). Federal Information Processing Standards Publication 198, U.S. Department of Commerce, March 2002.

[75] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, November 2007. Available: `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf` (2011/01/15).

[76] K. Nyberg. Differentially uniform mappings for cryptography. In T. Helleseth, editor, *Advances in Cryptology – EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 1994.

[77] K. Okeya. Side Channel Attacks Against HMACs Based on Block-Cipher Based Hash Functions. In *Australasian Conference on Information Security and Privacy – ACISP 2006, Proceedings*, volume 4058 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2006.

[78] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, University of Bristol, Department of Computer Science, June 2002. Available: `http://www.cs.bris.ac.uk/Publications/Papers/1000625.pdf` (2011/01/15).

[79] T. Peyrin. Cryptanalysis of Grindahl. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer, 2007.

[80] T. Peyrin. Improved Differential Attacks for ECHO and Grøstl. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010, Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 370–392. Springer, 2010.

[81] B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO '93, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1994.

[82] V. Rijmen, D. Toz, and K. Varici. Rebound Attack on Reduced-Round Versions of JH. In S. Hong and T. Iwata, editors, *Fast Software Encryption 2010, Proceedings*, volume 6147 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2010.

[83] M. Schläffer. Improved Collisions for Reduced ECHO-256. Cryptology ePrint Archive, Report 2010/588, 2010. `http://eprint.iacr.org/`.

[84] S. Tessaro. Personal communication, August 2009.

[85] S. Tillich. Bitsliced Implementation of `Grøstl`-256 without tweak, 2008. Personal communication, implementation written by Stefan Tillich and benchmarked in eBash.

[86] S. Tillich, M. Feldhofer, W. Issovits, T. Kern, H. Kureck, M. Mühlberghuber, G. Neubauer, A. Reiter, A. Köfler, and M. Mayrhofer. Compact Hardware Implementations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl, and Skein. Cryptology ePrint Archive, Report 2009/349, 2009. `http://eprint.iacr.org/2009/349`(2011/01/15).

[87] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely. High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Cryptology ePrint Archive, Report 2009/510, 2009. `http://eprint.iacr.org/2009/510`(2011/01/15).

[88] G. Tsudik. Message Authentication with One-Way Hash Functions. In *INFOCOM '92, Proceedings*, pages 2055–2059, 1992.

[89] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications (ISITA 2002), Proceedings*, October 2002.

[90] D. Wagner. A Generalized Birthday Problem. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

[91] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[92] Wikipedia. Close-mid front rounded vowel. `http://en.wikipedia.org/wiki/Close-mid_front_rounded_vowel` (2011/01/15).

[93] S. Wu, D. Feng, and W. Wu. Cryptanalysis of the LANE Hash Function. In M. J. Jacobson, V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography 2009, Proceedings*, volume 5867 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2009.

[94] S. Wu, D. Feng, and W. Wu. Practical Rebound Attack on 12-Round Cheetah-256. In *Information, Security and Cryptology - ICISC 2009, Proceedings*, volume 5984 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2009.

[95] K. Yasuda. "Sandwich" Is Indeed Secure: How to Authenticate a Message with Just One Hashing. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *Australasian Conference on Information Security and Privacy – ACISP 2007, Proceedings*, volume 4586 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2007.

A The name

Gröstl is an Austrian dish, usually made of leftover potatoes and pork, cut into slices. These are roasted on a pan together with onions and butterfat. The dish is often seasoned with salt, pepper, marjoram, cumin, and parsley, and served with a fried egg or *kraut* (cabbage). Hence, gröstl is somewhat similar to the American dish called hash.

The letter 'ö' was replaced by 'ø', which is a letter in the Danish alphabet that is pronounced in the same way as 'ö'. This way, the name, like the hash function itself, contains a mix of Austrian and Danish influences.

The pronunciation of `Grøstl` may seem challenging. If you think so, then think of the letter 'ø' as the 'i' in "bird". This letter is a so-called *close-mid front rounded vowel*, and if you need more examples of its pronunciation, or a sound sample, check out [92].

The letter 'ø' may not appear on your keyboard. It can be written in a number of word processing environments as follows:

Environment	Command for 'ø'
LaTeX	`{\o}`
HTML	`ø` or `ø`
Windows	Alt + 0248
Linux	AltGr + o *

(* does not work in all settings.)

B S-box

The s-box used in `Grøstl` is defined in Table 4.

Table 4: The `Grøstl` s-box (identical to the Rijndael/AES s-box). Given input x, find $x \wedge \mathsf{f0}$ in the first column ('$\wedge$' is logical *and*), and find $x \wedge \mathsf{0f}$ in the first row. Where the corresponding row and column meet, find the output $S(x)$ of the s-box.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16