Grøstl Implementation Guide

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

contact@groestl.info

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

3rd SHA-3 Conference

Contents

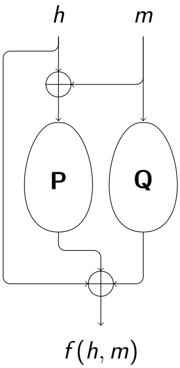
Outline

Motivation

- many ways to implement Grøstl
- different strategies and tricks for different platforms
- new optimizations developed recently
- share knowledge with implementers

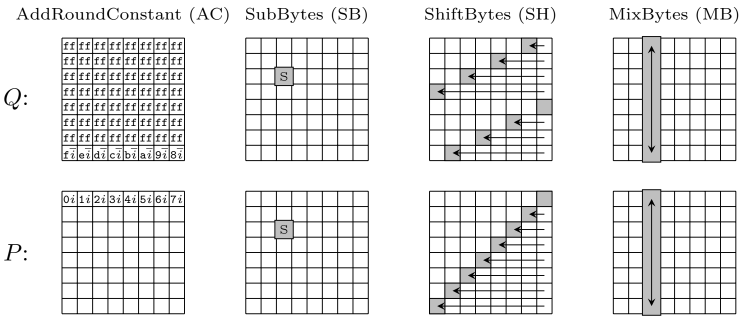
Outline

The SHA-3 Candidate Grøstl [GKM+11]



- iterated hash function with output transformation
- wide-pipe compression function
- permutation based design
- round transformation follow AES design principle

Permutations P and Q of Grøstl



- AES like round transformations
  - 8×8 state and 10 rounds for Grøstl-256
  - 8×16 state and 14 rounds for Grøstl-512
- differences between P/Q and Grøstl-256/Grøstl-512
  - heavier round transformations are the same (SB,MB)
  - lightweight round transformations differ (AC,SH)

AddRoundConstant

P

0i	1i	2i	3i	4i	5i	6i	7i

Q

ff	ff	ff	ff	ff	ff	ff	ff
ff	ff	ff	ff	ff	ff	ff	ff
ff	ff	ff	ff	ff	ff	ff	ff
ff	ff	ff	ff	ff	ff	ff	ff
ff	ff	ff	ff	ff	ff	ff	ff
ff	ff	ff	ff	ff	ff	ff	ff
ff	ff	ff	ff	ff	ff	ff	ff
f$\bar{i}$	e$\bar{i}$	d$\bar{i}$	c$\bar{i}$	b$\bar{i}$	a$\bar{i}$	9$\bar{i}$	8$\bar{i}$

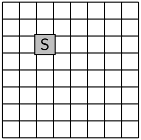
Definition

- XORs a constant to the state
- round dependent row in P and Q
- full constant 0xff in Q

Implementation

- word-size XOR with hard coded constant (8 to 256 bits)
- 0xff in Q: negative indexing of subsequent table lookup (ARM) or inversion in hardware

SubBytes



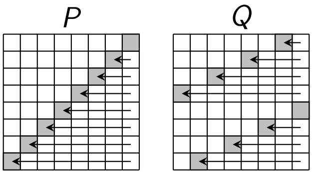
Definition

- substitute each byte using AES S-box
- based on inversion in finite field $GF(2^8)$
- $S(x) = A \cdot x^{-1} + b$

Implementation

- 8-bit lookup table
- AES new instructions (AESENCLAST [GI10])
- using byte shufflings (vperm implementation [Ham09])
- bitslice (optimized formulas by Canright [Can05])

ShiftBytes



Definition

- cyclically rotate the bytes of each row
- transposition of bytes

Implementation

- byte addressing (stored byte wise)
- byte extractions (column ordering)
- byte shufflings (row ordering)
- byte shufflings (bitslice)

MixBytes

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

MDS matrix multiplication

- applied to 8-byte columns (input: a_i, output: b_i)
- multiply with constants 2,3,4,5,7 in finite field $GF(2^8)$
- irreducible polynomial: 0x11b

MixBytes

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

Multiplication by constants

- $2 \cdot x$: byte shift left by 1; if carry is set, XOR 0x1b
- other factors using double-and-add:

  e.g. $7 \cdot x = (2 \cdot 2 \cdot x) \oplus (2 \cdot x) \oplus x$

MixBytes Implementations

Very flexible to implement

- Reference implementation (naive approach, do not use)
- Lookup tables for some multipliers
- T-table approach (includes S-box for free)
- Compute using optimized formulas (different variants)
- Byteslicing: 16x (SSE) or 32x (AVX2) in parallel
- Bitslicing: 16x (SSE) or 32x (AVX2) in parallel

Outline

How to Implement MixBytes?

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

Naive Approach (do not use)

- 64 byte-wise multiplications in finite field $GF(2^8)$
- 56 byte-wise XORs

MixBytes using Precomputed Tables

$$b_0 = 2a_0 \oplus 2a_1 \oplus 3a_2 \oplus 4a_3 \oplus 5a_4 \oplus 3a_5 \oplus 5a_6 \oplus 7a_7$$
$$b_1 = 7a_0 \oplus 2a_1 \oplus 2a_2 \oplus 3a_3 \oplus 4a_4 \oplus 5a_5 \oplus 3a_6 \oplus 5a_7$$
$$b_2 = 5a_0 \oplus 7a_1 \oplus 2a_2 \oplus 2a_3 \oplus 3a_4 \oplus 4a_5 \oplus 5a_6 \oplus 3a_7$$
$$b_3 = 3a_0 \oplus 5a_1 \oplus 7a_2 \oplus 2a_3 \oplus 2a_4 \oplus 3a_5 \oplus 4a_6 \oplus 5a_7$$
$$b_4 = 5a_0 \oplus 3a_1 \oplus 5a_2 \oplus 7a_3 \oplus 2a_4 \oplus 2a_5 \oplus 3a_6 \oplus 4a_7$$
$$b_5 = 4a_0 \oplus 5a_1 \oplus 3a_2 \oplus 5a_3 \oplus 7a_4 \oplus 2a_5 \oplus 2a_6 \oplus 3a_7$$
$$b_6 = 3a_0 \oplus 4a_1 \oplus 5a_2 \oplus 3a_3 \oplus 5a_4 \oplus 7a_5 \oplus 2a_6 \oplus 2a_7$$
$$b_7 = 2a_0 \oplus 3a_1 \oplus 4a_2 \oplus 5a_3 \oplus 3a_4 \oplus 5a_5 \oplus 7a_6 \oplus 2a_7$$

T-table approach (64-bit, 32-bit)

- 8-to-64 bit tables: $T_0(a_0) = 2a_0 \| 7a_0 \| 5a_0 \| 3a_0 \| 5a_0 \| 4a_0 \| 3a_0 \| 2a_0$
- combine with S-box lookup to compute 1 column of AC,SB,SH,MB
- 8 byte extractions, 8 table lookups, 8 XORs

MixBytes using Precomputed Tables

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

T-table approach (64-bit, 32-bit)

- 8-to-64 bit tables: $T_0(a_0) = 2a_0 \| 7a_0 \| 5a_0 \| 3a_0 \| 5a_0 \| 4a_0 \| 3a_0 \| 2a_0$
- combine with S-box lookup to compute 1 column of AC,SB,SH,MB
- 8 byte extractions, 8 table lookups, 8 XORs
- TODO: optimize for NEON (lower bound: 35-40c/b)

MixBytes using Optimized Formulas

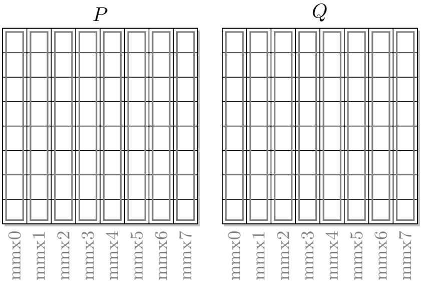
$$x_i = a_i \oplus a_{i+1}$$
$$y_i = x_i \oplus x_{i+3}$$
$$z_i = x_i \oplus x_{i+2} \oplus a_{i+6}$$
$$b_i = 2 \cdot (2 \cdot y_{i+3} \oplus z_{i+7}) \oplus z_{i+4}$$

Computing MixBytes (8-bit, 128-bit, 256-bit)

- 16 MUL2, 48 XORs [ARSS11] (for any register size)
- easy parallelism
  - 16 times in parallel (SSSE3, NEON)
  - 32 times in parallel (AVX, AVX2)
  - hardware implementations!
  - tree mode
- TODO: find optimal formulas

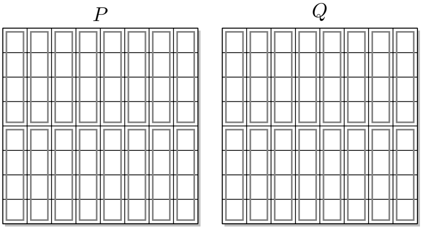
Outline

Ordering of the Grøstl State



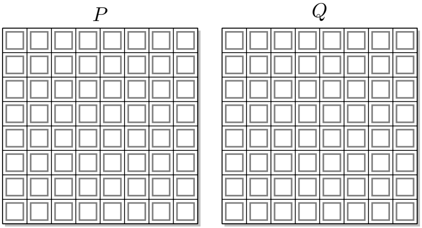
- column ordering (64-bit)

Ordering of the Grøstl State



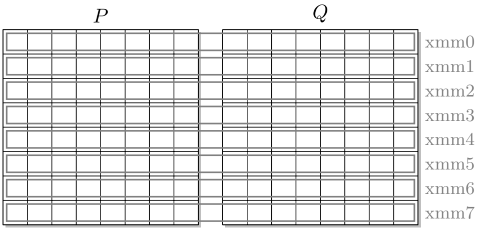
- column ordering (64-bit)
- column ordering (32-bit)

Ordering of the Grøstl State



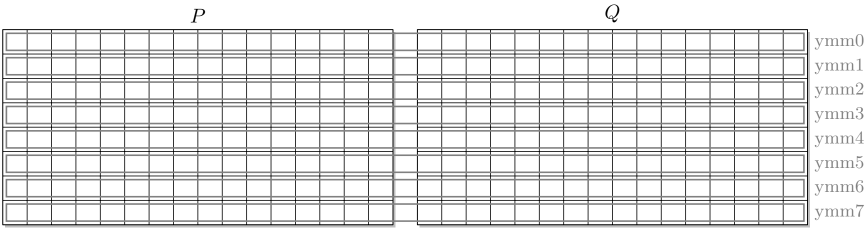
- column ordering (64-bit)
- column ordering (32-bit)
- byte ordering (8-bit, to avoid byte extractions)

Ordering of the Grøstl State



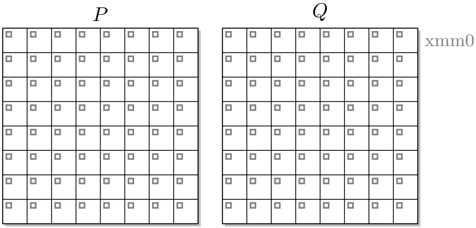
- column ordering (64-bit)
- column ordering (32-bit)
- byte ordering (8-bit, to avoid byte extractions)
- row ordering (128-bit)

Ordering of the Grøstl State



- column ordering (64-bit)
- column ordering (32-bit)
- byte ordering (8-bit, to avoid byte extractions)
- row ordering (128-bit)
- row ordering (256-bit)

Ordering of the Grøstl State



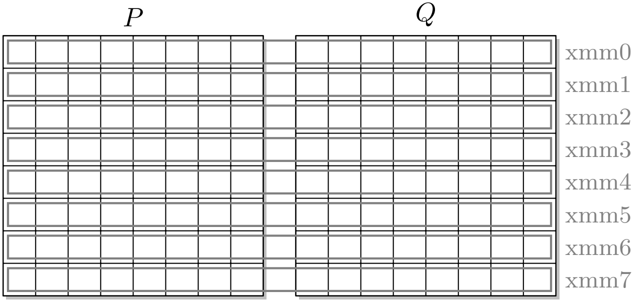
- column ordering (64-bit)
- column ordering (32-bit)
- byte ordering (8-bit, to avoid byte extractions)
- row ordering (128-bit)
- row ordering (256-bit)
- bitslice (128-bit, 256-bit)

Outline

Implementation overview

- Reference implementation
- T-table implementation
  - 64-bit CPUs
  - 32-bit CPUs
- Byte slice implementation
  - AES instruction
  - vperm implementation
  - 8-bit implementation
- Bitslice implementation
- (Hardware implementations)

Byte Slice Implementation



- storing the Grøstl-256 state in row ordering
- compute all round transformations 16x in parallel
- same algorithm for 8-bit and hardware

AES-NI Implementation (Pseudo Code)

```
pxor        xmm0, [CONST0]  // AC
pshufb      xmm0, [SIGMA0]  // SH (with AES ShiftRowsInv)
aesenclast  xmm0, xmm8      // SB (with AES ShiftRows)
movdqa      xmm14, xmm0;    // MB (y_i = a_{i+6})
pxor        xmm0, xmm1;     // MB (t_i = a_i + a_{i+1})
pxor        xmm8, xmm4;     // MB (y_i = a_{i+6} + t_i)
pxor        xmm14, xmm4;    // MB (y_i = y_i + t_{i+2})
pxor        xmm0, xmm3;     // MB (x_i = t_i + t_{i+3})
MUL2       (xmm0, xmm8);    // MB (z_i = 02 * x_i)
pxor        xmm2, xmm10;    // MB (w_i = z_i + y_{i+4})
MUL2       (xmm0, xmm8);    // MB (v_i = 02 * w_i)
pxor        xmm13, xmm0;    // MB (b_i = v_{i+3} + y_{i+4})
```

- 8x for one round of Grøstl-256
    - single instruction for AC,SB,SH
    - use optimized formulas for MB

AES-NI Implementation (Pseudo Code)

```
pxor        xmm0, [CONST0]  // AC
pshufb      xmm0, [SIGMA0]  // SH (with AES ShiftRowsInv)
aesenclast  xmm0, xmm8      // SB (with AES ShiftRows)
movdqa      xmm14, xmm0;    // MB (y_i = a_{i+6})
pxor        xmm0, xmm1;     // MB (t_i = a_i + a_{i+1})
pxor        xmm8, xmm4;     // MB (y_i = a_{i+6} + t_i)
pxor        xmm14, xmm4;    // MB (y_i = y_i + t_{i+2})
pxor        xmm0, xmm3;     // MB (x_i = t_i + t_{i+3})
MUL2       (xmm0, xmm8);    // MB (z_i = 02 * x_i)
pxor        xmm2, xmm10;    // MB (w_i = z_i + y_{i+4})
MUL2       (xmm0, xmm8);    // MB (v_i = 02 * w_i)
pxor        xmm13, xmm0;    // MB (b_i = v_{i+3} + y_{i+4})
```

- 8x for one round of Grøstl-256
  - single instruction for AC,SB,SH
  - use optimized formulas for MB
- MUL2: 5 instructions

AES-NI Implementation (Pseudo Code)

```
pxor        xmm0, [CONST0]  // AC
pshufb      xmm0, [SIGMA0]  // SH (with AES ShiftRowsInv)
aesenclast  xmm0, xmm8      // SB (with AES ShiftRows)
movdqa      xmm14, xmm0;    // MB (y_i = a_{i+6})
pxor        xmm0, xmm1;     // MB (t_i = a_i + a_{i+1})
pxor        xmm8, xmm4;     // MB (y_i = a_{i+6} + t_i)
pxor        xmm14, xmm4;    // MB (y_i = y_i + t_{i+2})
pxor        xmm0, xmm3;     // MB (x_i = t_i + t_{i+3})
MUL2       (xmm0, xmm8);    // MB (z_i = 02 * x_i)
pxor        xmm2, xmm10;    // MB (w_i = z_i + y_{i+4})
MUL2       (xmm0, xmm8);    // MB (v_i = 02 * w_i)
pxor        xmm13, xmm0;    // MB (b_i = v_{i+3} + y_{i+4})
```

- 8x for one round of Grøstl-256
  - single instruction for AC,SB,SH
  - use optimized formulas for MB
- MUL2: 5 instructions
- MUL2: half the number of instructions per round

VPERM Implementation

- AES-NI not (yet) widely available
- can we still compute 16 AES S-box lookups in parallel?
- yes, using byte-shuffling instructions (vperm) [Ham09]
- MUL2: almost for free
- only SSSE3 needed (or NEON)
- as fast as T-table implementation
- constant time!

VPERM Implementation

- AES-NI not (yet) widely available
- can we still compute 16 AES S-box lookups in parallel?
- yes, using byte-shuffling instructions (vperm) [Ham09]
- MUL2: almost for free
- only SSSE3 needed (or NEON)
- as fast as T-table implementation
- constant time!
- TODO: optimize for Intel Atom
- TODO: implement using NEON

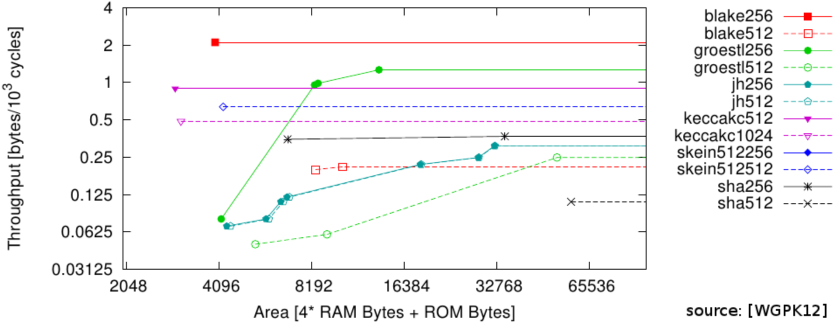
Bitslice Implementation

- Grøstl-0: 29c/b on Core2Duo (Tillich)
- Grøstl-0: 4x in parallel: 24c/b (Calik)
- AES implementation: 7.6c/b (Käsper,Schwabe)
  - lower bound: 2x AES speed
- new upcoming implementations for NEON and x86
  - using optimized MixBytes formulas

Bitslice Implementation

- Grøstl-0: 29c/b on Core2Duo (Tillich)
- Grøstl-0: 4x in parallel: 24c/b (Calik)
- AES implementation: 7.6c/b (Käsper,Schwabe)
  - lower bound: 2x AES speed
- new upcoming implementations for NEON and x86
  - using optimized MixBytes formulas
- TODO: improve bitslice implementations

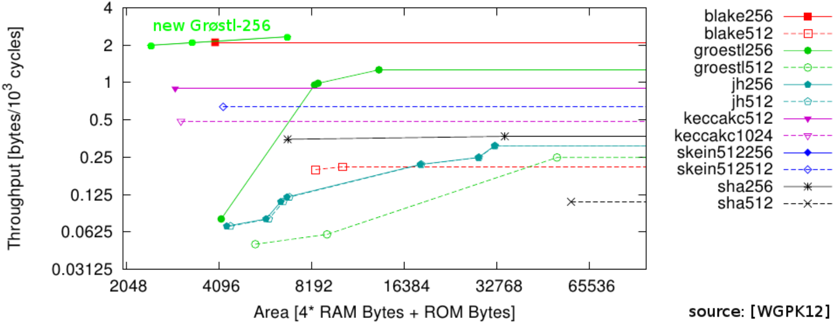
Optimized 8-bit Implementations



source: [WGPK12]

- new implementations by Johannes Feichtner [Fei11]
  - SPEED: 447-506 c/b (for 1516 bytes)
  - RAM: 256-512 bytes (192 by loading message twice)
  - ROM: 1526-4990 bytes

Optimized 8-bit Implementations



source: [WGPK12]

- new implementations by Johannes Feichtner [Fei11]
  - SPEED: 447-506 c/b (for 1516 bytes)
  - RAM: 256-512 bytes (192 by loading message twice)
  - ROM: 1526-4990 bytes

Outline

New Instructions?

- AVX2: all instructions 256-bit wide except AESENCLAST
  - fresh new implementation for `Grøstl`-512
  - 165 instead of 271 instructions per round (-40%)

New Instructions?

- AVX2: all instructions 256-bit wide except AESENCLAST
  - fresh new implementation for `Grøstl-512`
  - 165 instead of 271 instructions per round (-40%)
- AVX2: VPGATHERDD/DQ
  - 4 parallel 64-bit table lookups
  - efficient parallel byte manipulation and extractions
  - removes current bottleneck in T-table implementations

New Instructions?

- AVX2: all instructions 256-bit wide except AESENCLAST
  - fresh new implementation for Grøstl-512
  - 165 instead of 271 instructions per round (-40%)
- AVX2: VPGATHERDD/DQ
  - 4 parallel 64-bit table lookups
  - efficient parallel byte manipulation and extractions
  - removes current bottleneck in T-table implementations
- MUL2: simpler than byte addition (paddb)
  - assuming same properties as paddb:
  - 6.8c/b instead of 9.6c/b

New Instructions?

- AVX2: all instructions 256-bit wide except AESENCLAST
  - fresh new implementation for Grøstl-512
  - 165 instead of 271 instructions per round (-40%)
- AVX2: VPGATHERDD/DQ
  - 4 parallel 64-bit table lookups
  - efficient parallel byte manipulation and extractions
  - removes current bottleneck in T-table implementations
- MUL2: simpler than byte addition (paddb)
  - assuming same properties as paddb:
  - 6.8c/b instead of 9.6c/b
- multiplication by constants in AES field
  - MULx xmm0, [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
  - Grøstl, AES and many other designs will benefit
  - reusable and more lightweight than full AES round

New Instructions?

- AVX2: all instructions 256-bit wide except AESENCLAST
  - fresh new implementation for Grøstl-512
  - 165 instead of 271 instructions per round (-40%)
- AVX2: VPGATHERDD/DQ
  - 4 parallel 64-bit table lookups
  - efficient parallel byte manipulation and extractions
  - removes current bottleneck in T-table implementations
- MUL2: simpler than byte addition (paddb)
  - assuming same properties as paddb:
  - 6.8c/b instead of 9.6c/b
- multiplication by constants in AES field
  - MULx xmm0, [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
  - Grøstl, AES and many other designs will benefit
  - reusable and more lightweight than full AES round

⇒ Grøstl benefits best from ISE [CBG12]

Conclusion

- main results of this work
  - best 8-bit finalist (including SHA-2) in all categories
  - faster than SHA-2 on high-end Intel platforms
  - hardware and other platforms: work in progress

Conclusion

- main results of this work
  - best 8-bit finalist (including SHA-2) in all categories
  - faster than SHA-2 on high-end Intel platforms
  - hardware and other platforms: work in progress
- Grøstl has lots of implementation options
  - still many improvements possible
  - great implementation flexibility
  - adaptable to new upcoming (unknown) instructions

Conclusion

- main results of this work
  - best 8-bit finalist (including SHA-2) in all categories
  - faster than SHA-2 on high-end Intel platforms
  - hardware and other platforms: work in progress
- Grøstl has lots of implementation options
  - still many improvements possible
  - great implementation flexibility
  - adaptable to new upcoming (unknown) instructions

⇒ Grøstl is designed for the future, not the past!

Conclusion

- main results of this work
  - best 8-bit finalist (including SHA-2) in all categories
  - faster than SHA-2 on high-end Intel platforms
  - hardware and other platforms: work in progress
- Grøstl has lots of implementation options
  - still many improvements possible
  - great implementation flexibility
  - adaptable to new upcoming (unknown) instructions

⇒ Grøstl is designed for the future, not the past!

Thank you for your attention!

References I

Kazumaro Aoki, Günther Roland, Yu Sasaki, and Martin Schläffer.

Byte Slicing Grøstl – Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Grøstl.

In Javier Lopez and Pierangela Samarati, editors, *SECRYPT 2011, Proceedings*, pages 124–133. SciTePress, 2011.

David Canright.

A Very Compact S-Box for AES.

In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *LNCS*, pages 441–455. Springer, 2005.

Jeremy Constantin, Andreas Burg, and Frank K. Gurkaynak.

Investigating the Potential of Custom Instruction Set Extensions for SHA-3 Candidates on a 16-bit Microcontroller Architecture.

Cryptology ePrint Archive, Report 2012/050, 2012.

`http://eprint.iacr.org/`.

References II

Johannes Feichtner.

Effcient Grøstl-256 Implementations for the AVR 8-bit Microcontroller Architecture, 2011.

http://www.groestl.info/groestl-avr8asm.pdf.

Shay Gueron and Intel Corp.

Intel® Advanced Encryption Standard (AES) Instructions Set, 2010.

Retrieved December 21, 2010, from
http://software.intel.com/en-us/articles/
intel-advanced-encryption-standard-aes-instructions-set/.

Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen.

Grøstl – a SHA-3 candidate.

Submission to NIST (Round 3), 2011.

Available: http://www.groestl.info (2011/11/25).

References III

Mike Hamburg.

Accelerating AES with Vector Permute Instructions.

In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 18–32. Springer, 2009.

Christian Wenzel-Benner, Jens Gräf, John Pham, and Jens-Peter Kaps.

XBX Benchmarking Results January 2012.

The Third SHA-3 Candidate Conference, 2012.