Security in knowledge

Randomly Failed!

The State of Randomness in Current Java Implementations

Kai Michaelis, **Chris Meyer**, Jörg Schwenk

Horst Görtz Institute for IT-Security (HGI)

Chair for Network and Data Security

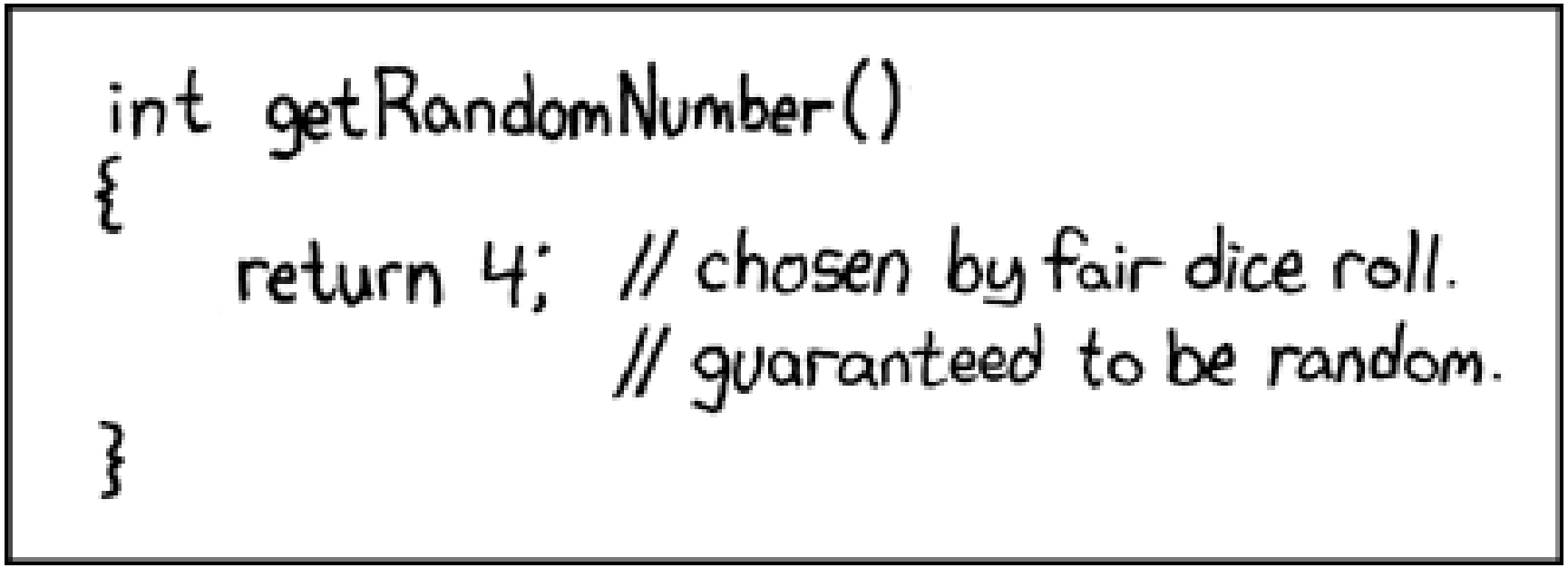
Ruhr-University Bochum, Germany

RSA CONFERENCE 2013

Session ID: CRYP-W25

Session Classification: Advanced

How *Random* are Java PRNGs?



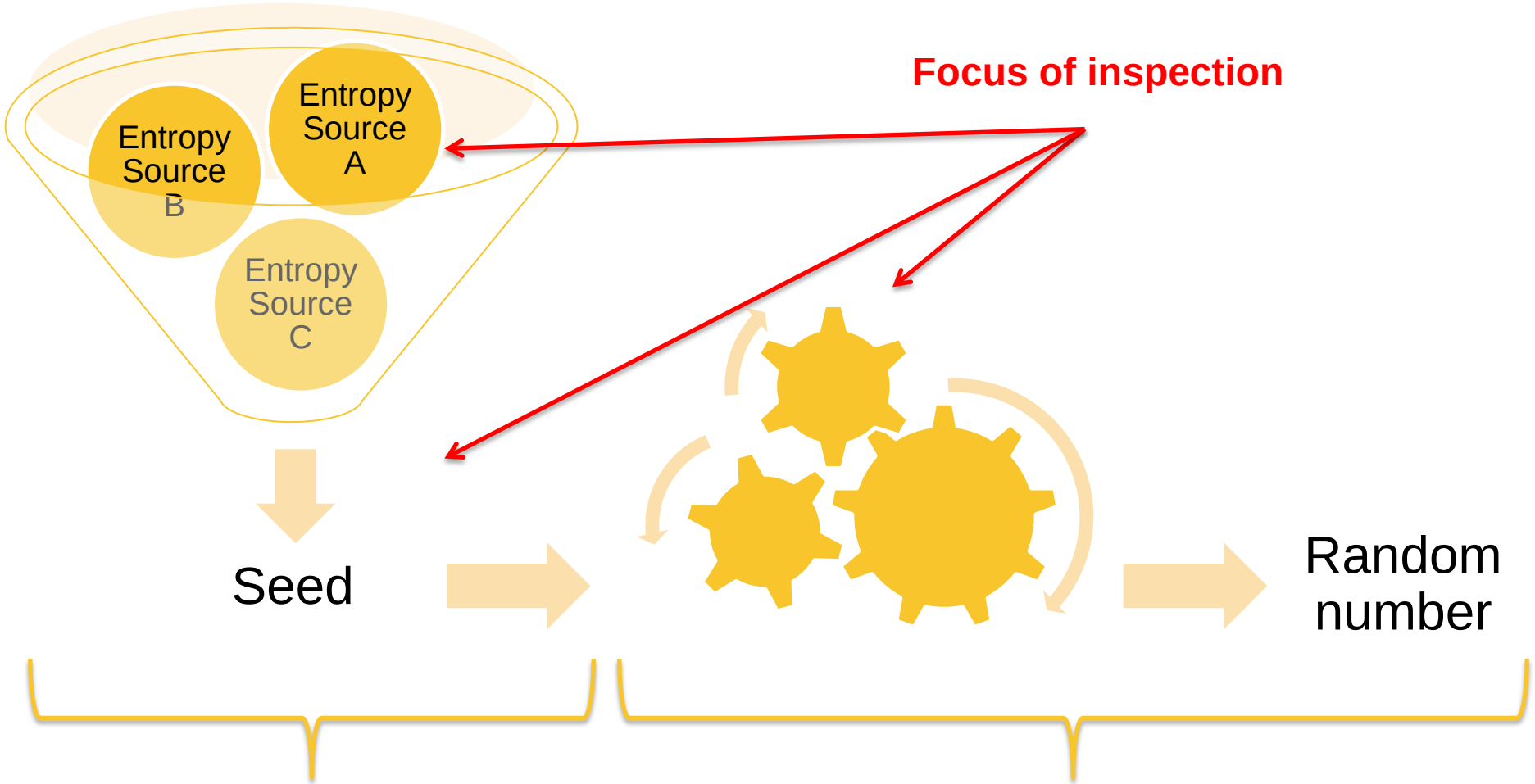
Source: http://www.xkcd.com

Spoiler Slide

- At least, thankfully not!
  - Inspected PRNGs
    - Apache Harmony
    - GNU Classpath
    - OpenJDK
    - The Legion of Bouncy Castle
  - Methodology
    - Code/algorithm inspection
    - Blackbox Tests (Dieharder, STS, …)
  - Broad range of code/algorithm quality
    - The good
    - The bad
    - And the ugly

Source: http://www.memegenerator.net

Operation Sketch of a PRNG



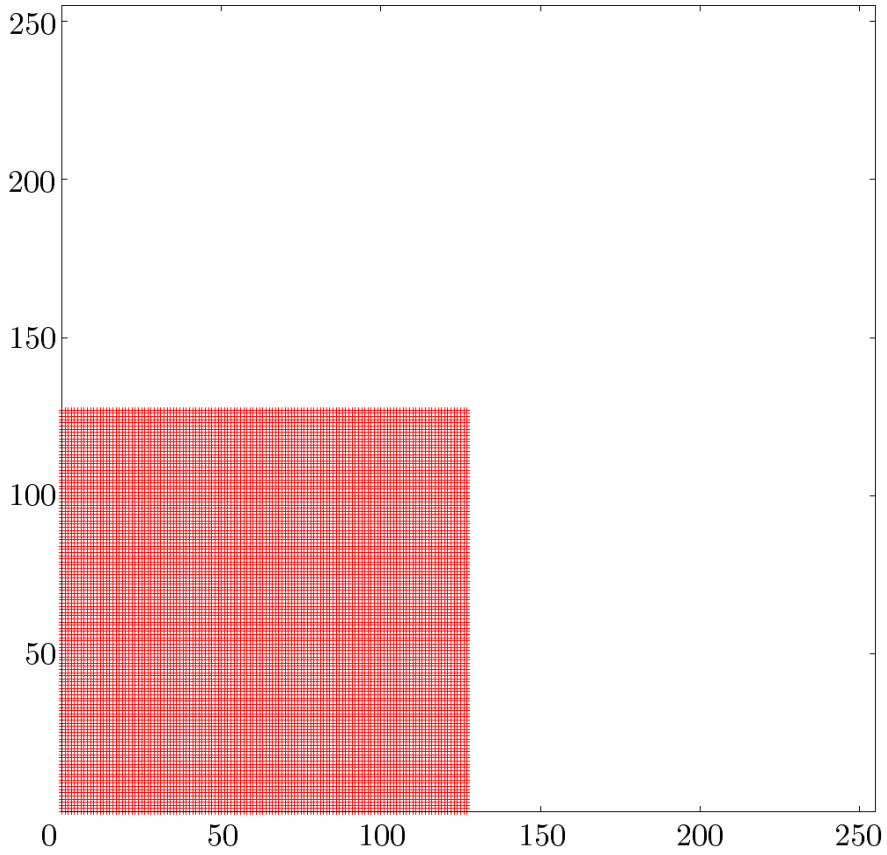
Results: Apache Harmony

- PRNG
  - SHA-1 based algorithm: *Seed, Counter, Internal State*
- Backup seeding facility (Entropy Collector)
  - (under Unix): *Unix-Time, Processor Time, Pointer value (Heap)*
- Weaknesses
  - Self-seeding *SecureRandom* suffers from implementation bug
    - Pointer into state buffer not properly adjusted
    - Entropy reduced to 64 bits
    - Directly affects the Android platform
  - Backup seeding facility (Entropy Collector) suffers from multiple implementation bugs
    - MSB set to 0 (reason for this remains unclear)
    - Inappropriate modular reduction (signed/unsigned integers)
    - Entropy reduced to 31 bits (worst case)

Results: Apache Harmony

Quality of Entropy Collector

- Worst-Case scenario
- 2 consecutive bytes mark a single point
- 10 MiB generated Seed



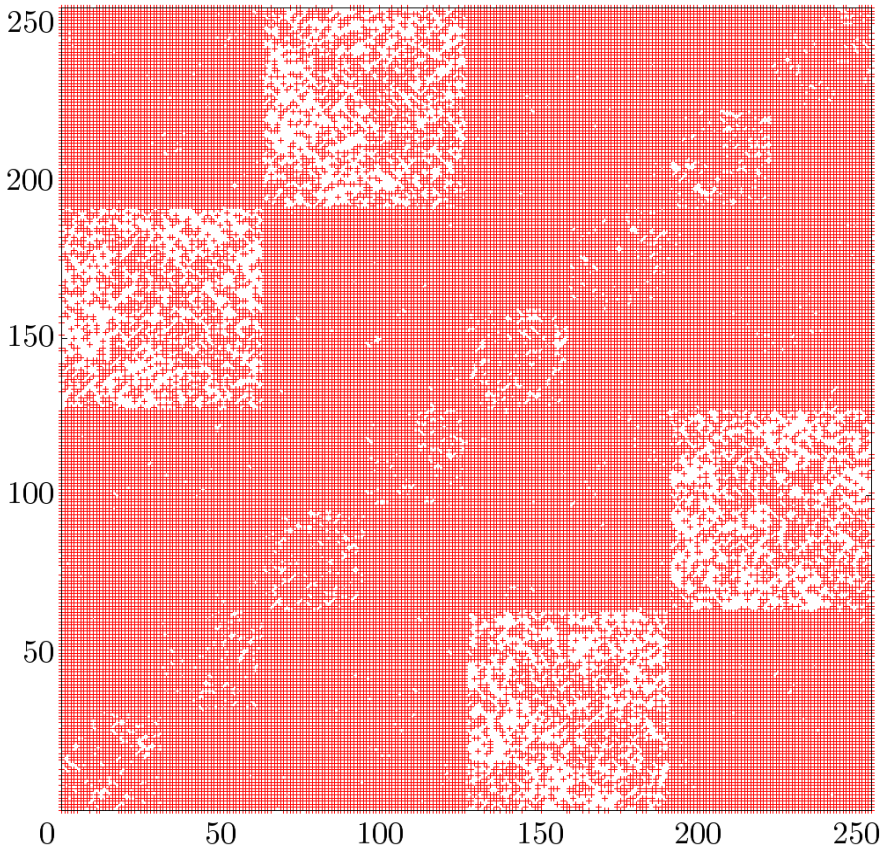
Results:GNU Classpath

- PRNG
  - SHA-1 based algorithm: *Seed, Internal State*
- Backup seeding facility (Entropy Collector)
  - 8 Threads increment independent counters (c1..c8)
  - Seed = c1 XOR c2 XOR …. c8
- Weaknesses
  - Repeated use of same IV
    - Predictable internal states: only 20 of 32 bytes unknown
  - Backup seeding facility (Entropy Collector) can be influenced
    - High load prevents threads execution

Results: GNU Classpath

Quality of Entropy Collector

- Worst-Case scenario
- 2 consecutive bytes mark a single point
- 10 MiB generated Seed



Results: OpenJDK

- PRNG
  - SHA-1 based algorithm: *Seed, Internal State, Fixed-state protection*
- Backup seeding facility (Entropy Collector)
  - Counter incrementation, *System.properties*
  - *Noise* threads (keep the scheduler busy)
  - S-Boxing counter
  - Enforcing mandatory runtime and counter incrementation
  - Slow….
- Weaknesses
    - No obvious weakness

Results: Bouncy Castle

- PRNG
  - Multiple *SecureRandom* replacements
    - SHA-1 based algorithm: Seed, Internal State, Counter
    - VMPC based algorithm
- Backup seeding facility (Entropy Collector)
  - Counter incrementation
  - Producer and Consumer Threads
  - *Slow* and *fast* operation mode
- Weaknesses
  - VMPC known to be vulnerable to distinguishing attacks

Conclusion

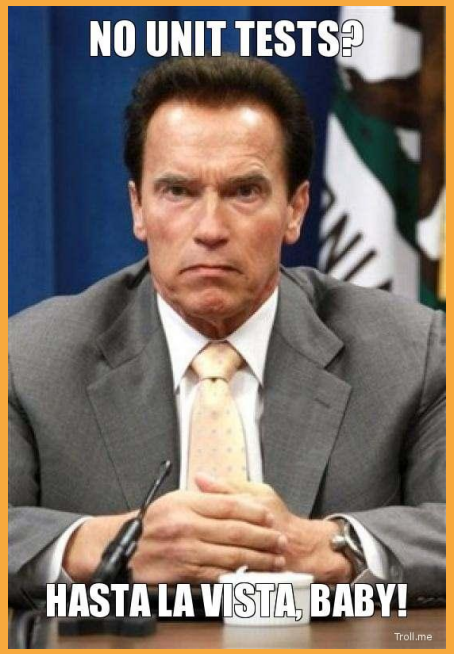
- PRNGs are only as good as the seed's entropy
- Software Entropy Collectors are mostly not suitable for cryptographic purposes
- Broad range of code/algorithm quality
- Fixed & limited size of internal states
- Some implementations are only susceptible to the outlined vulnerabilities if no OS entropy is available

Personal advice

In critical environments

- Prevent usage of PRNGs
- Exclusively rely on hardware ECs/RNGs

Random Questions?



Source: http://www.troll.me

hgi NDS **Chris Meyer**
Christopher.meyer@rub.de

http://armoredbarista.blogspot.com
http://www.nds.rub.de/chair/people/cmeyer

Security in knowledge

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

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

IAIK, Graz University of Technology, Austria
martin.schlaeffer@iaik.tugraz.at

Digital Security Group, Radboud University Nijmegen, Netherlands
Research Center for Information Technology Innovation, Academia Sinica, Taiwan
peter@cryptojedi.org

CT-RSA 2013

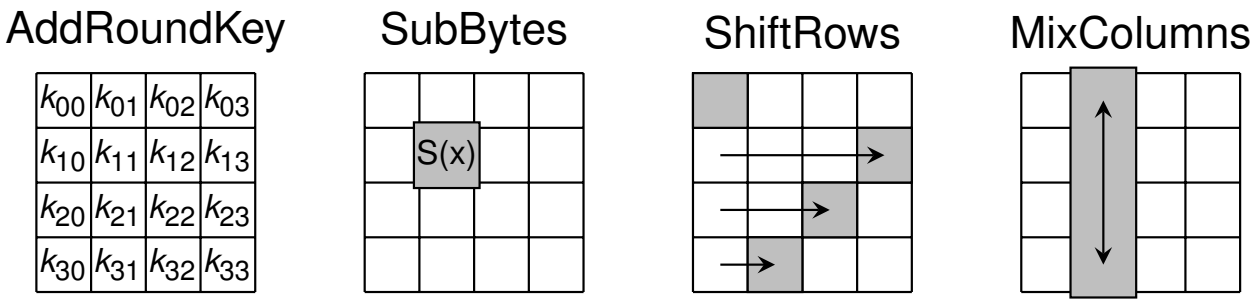
Contents

Outline

Motivation

- Many AES-based hash functions have been submitted to the NIST SHA-3 competition [Nat07]
- Fast using Intel AES-NI instructions, slow otherwise!?
- More difficult to implement (or to improve performance)?
- Learn lessons to improve new AES-based designs

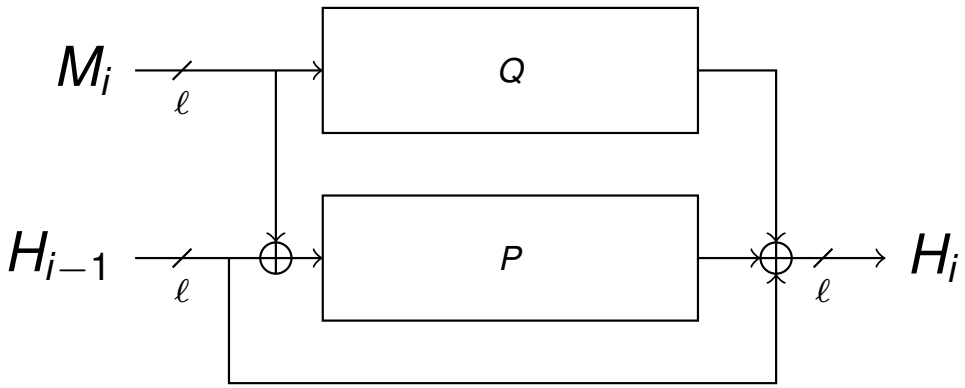
AES-Based Hash Functions



- AES [Nat01]: 4x4 state of bytes (128-bit)
- Advantages:
  - very well analyzed building blocks
  - proofs against linear and differential attacks
  - simple analysis (wide trails)
  - fast using AES instruction
- Disadvantages:
  - much larger state needed for 256-, 512-bit hash function
  - AES not so well analyzed in hash function setting
  - wrong usage of AES (just 1 round, ...)
  - slow without AES instruction, large tables, cache timing attacks

Outline

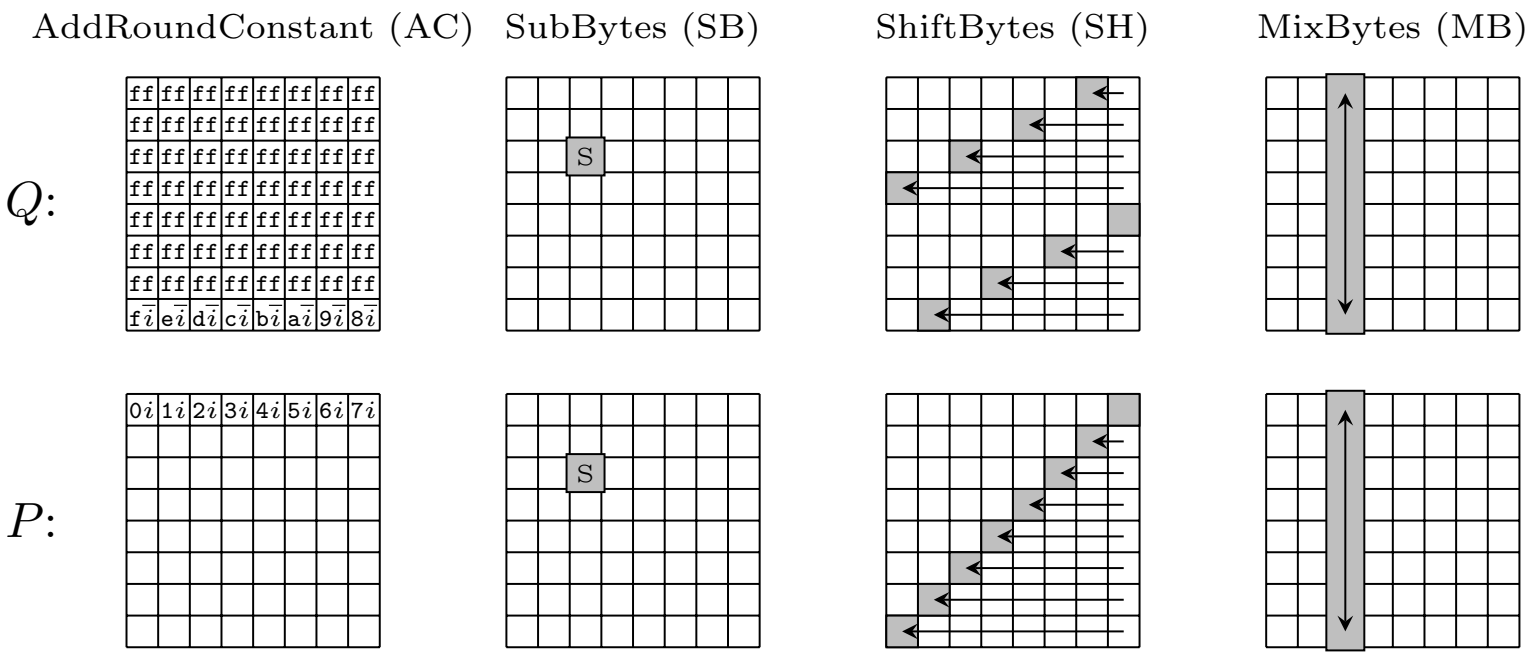
The SHA-3 Finalist Grøstl [GKM+11]



- Permutation based design
- Double-pipe compression function ($\ell \geq 2n$)
- AES-based hash function
- Designed by DTU (Denmark) and TU Graz (Austria)

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

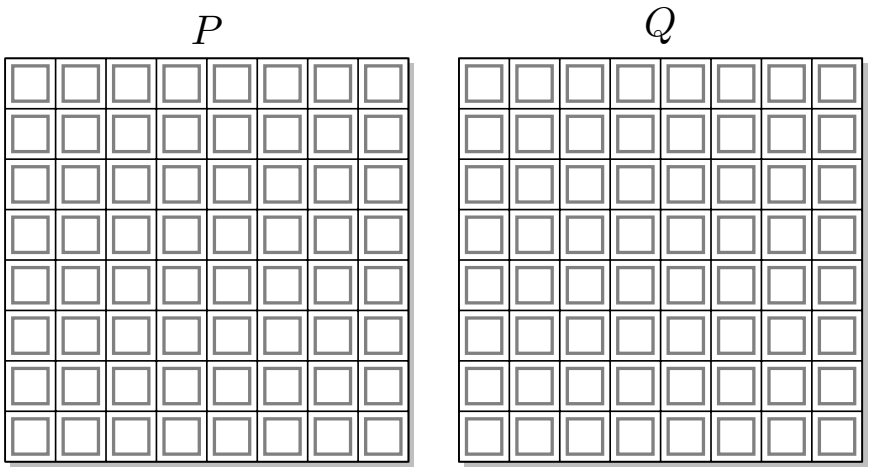
Permutations P and Q of Grøstl



- 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)

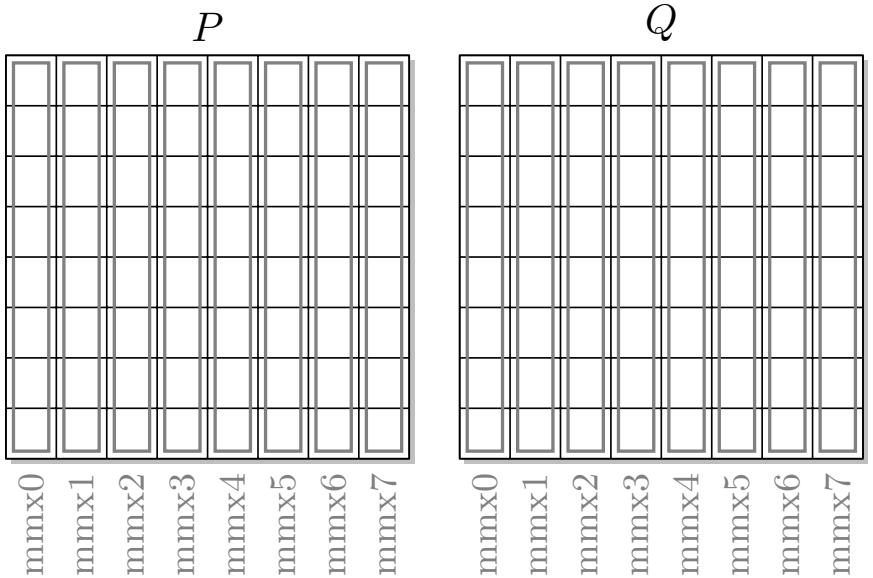
Outline

Ordering of the Grøstl State



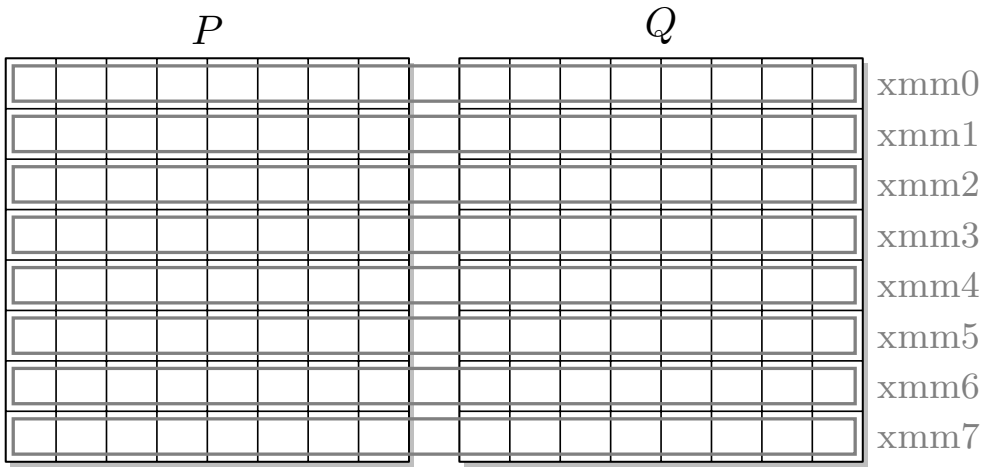
- Byte ordering (to avoid byte extractions, 8-bit implementation)

Ordering of the Grøstl State



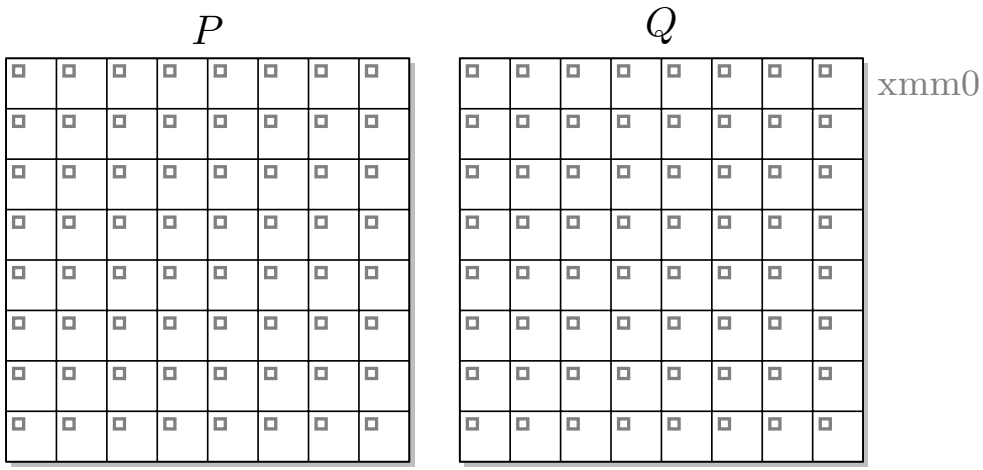
- Byte ordering (to avoid byte extractions, 8-bit implementation)
- Column ordering (T-table approach, [DR99, Sect. 5.2])

Ordering of the Grøstl State



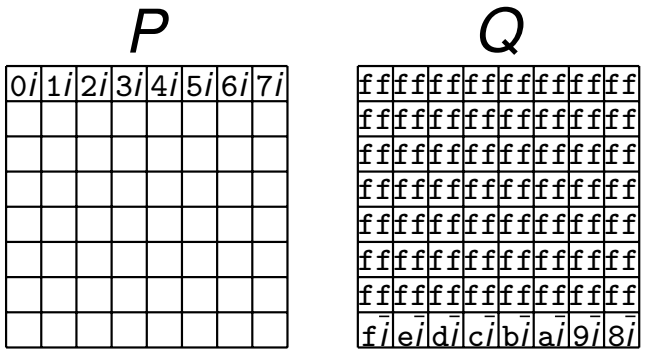
- Byte ordering (to avoid byte extractions, 8-bit implementation)
- Column ordering (T-table approach, [DR99, Sect. 5.2])
- Row ordering (byteslice implementation, [ARSS11])

Ordering of the Grøstl State



- Byte ordering (to avoid byte extractions, 8-bit implementation)
- Column ordering (T-table approach, [DR99, Sect. 5.2])
- Row ordering (byteslice implementation, [ARSS11])
- Bitslicing (to avoid table lookups, [Bih97])

AddRoundConstant



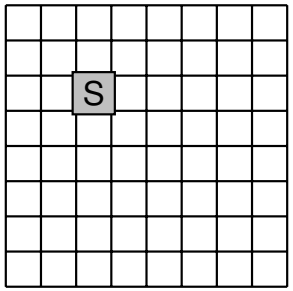
Description:

- round dependent row in P and Q
- full constant 0xff in Q

Implementation:

- load and XOR constant to the state
- implement 0xff using inversion or negative S-box indexing

SubBytes



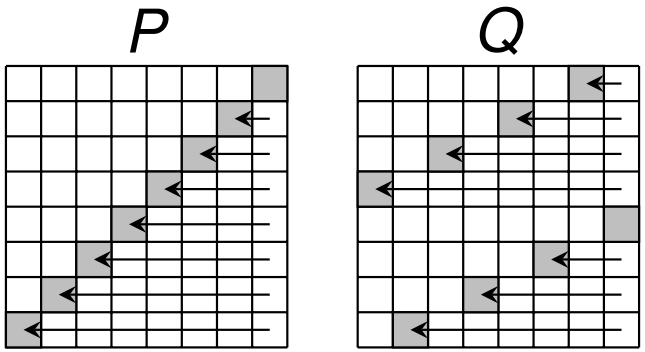
Description:

- 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 table lookups (or T-tables)
- Intel AES new instructions (AESENCLAST), [GI10]
- using byte shufflings (vperm), [Ham09]
- compute using optimized formulas (bitslicing), [Can05]

ShiftBytes



Description:

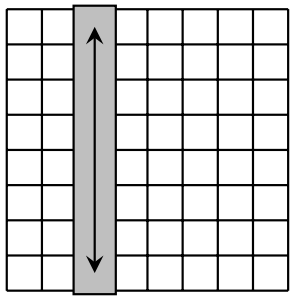
- rotate (shuffle) the bytes of each row
- different values for P and Q

Implementation:

- byte addressing (byte ordering)
- byte extractions (column ordering)
- byte shufflings/rotations (row ordering)
- bit shufflings/rotations (bitslice)

MixBytes

MixBytes (MB)



Definition:

- applied to 8-byte columns (input: a_i, output: b_i)

MixBytes

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

Definition:

- applied to 8-byte columns (input: a_i, output: b_i)
- multiplication with constant MDS matrix in $GF(2^8)$

MixBytes

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

Definition:

- applied to 8-byte columns (input: a_i, output: b_i)
- multiplication with constant MDS matrix in $GF(2^8)$

Implementation:

- T-tables: 8 byte extractions, 8 lookups, 7 XORs (S-box included)

MixBytes

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

Definition:

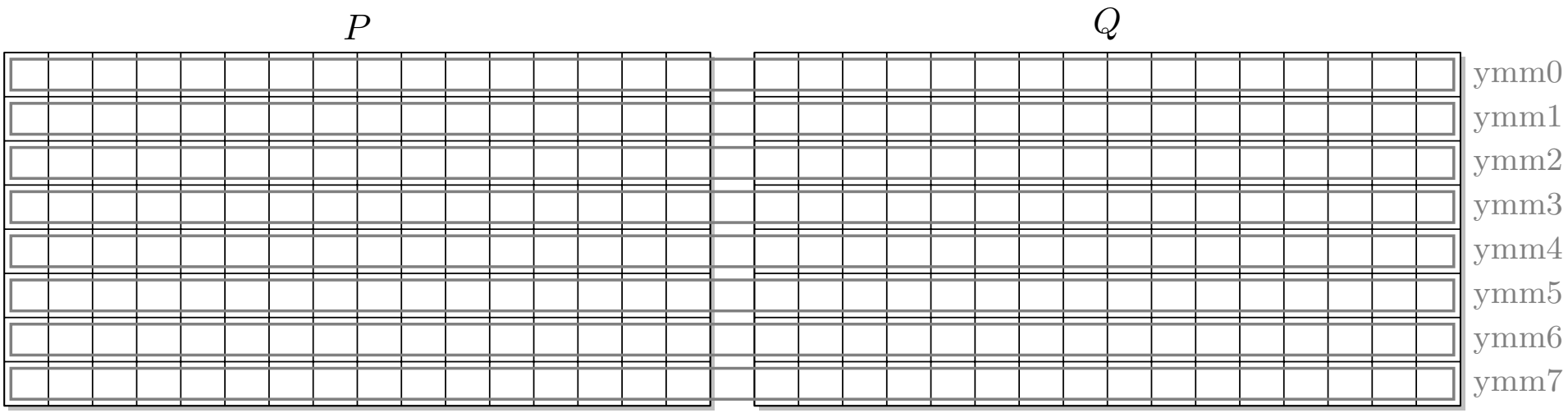
- applied to 8-byte columns (input: a_i, output: b_i)
- multiplication with constant MDS matrix in $GF(2^8)$

Implementation:

- T-tables: 8 byte extractions, 8 lookups, 7 XORs (S-box included)
- compute using optimized formulas (8-bit, byteslice, bitslice)

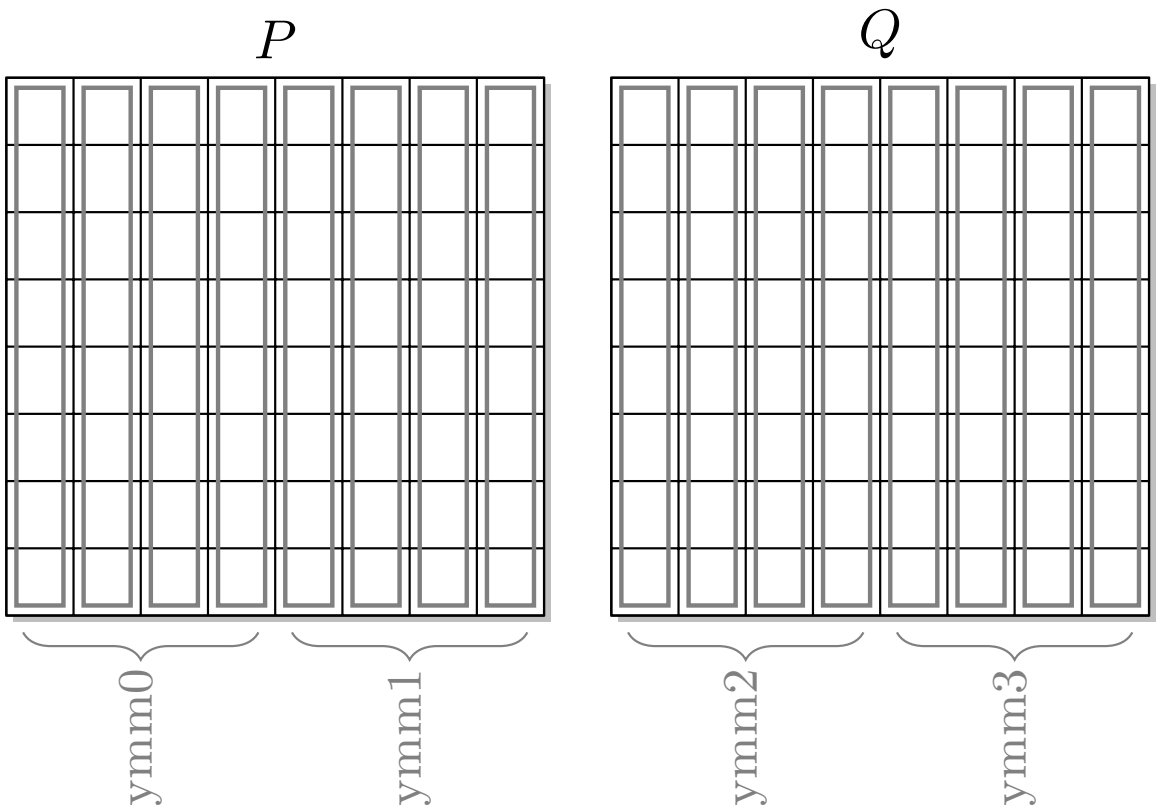
Outline

AVX2: Byteslicing Grøstl-512



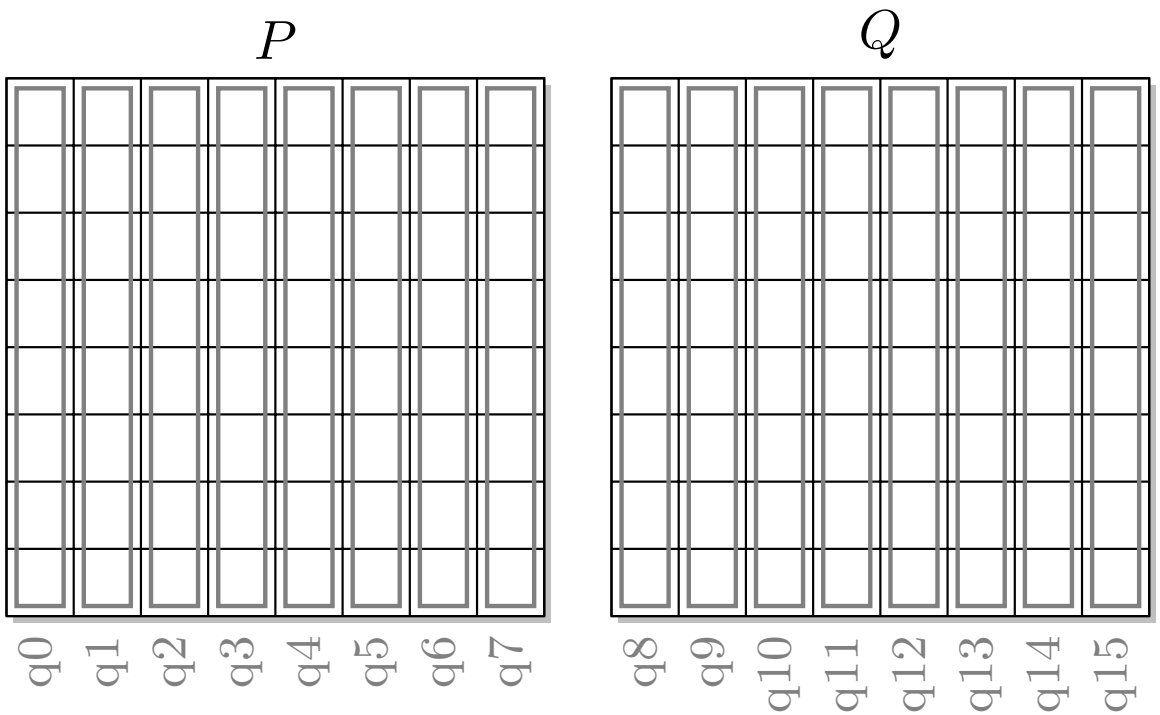
- Computing the whole Grøstl-512 state in parallel
- 32 columns in parallel using 256-bit AVX2 instructions
- Only need to extract for 128-bit AESENCLAST instruction
- 40% less instructions compared to AES-NI or AVX

AVX2: Parallel T-Table Lookups for Grøstl-256



- Store 4 columns in one 256-bit AVX2 register
- Perform 4 T-table lookups in parallel using VPGATHERQQ
- ShiftBytes more expensive: VPSHUFB, VPERMQ, VPBLENDD
- 15% less instructions compared to 64-bit implementation

NEON: Alternating T-Table Lookups for P and Q



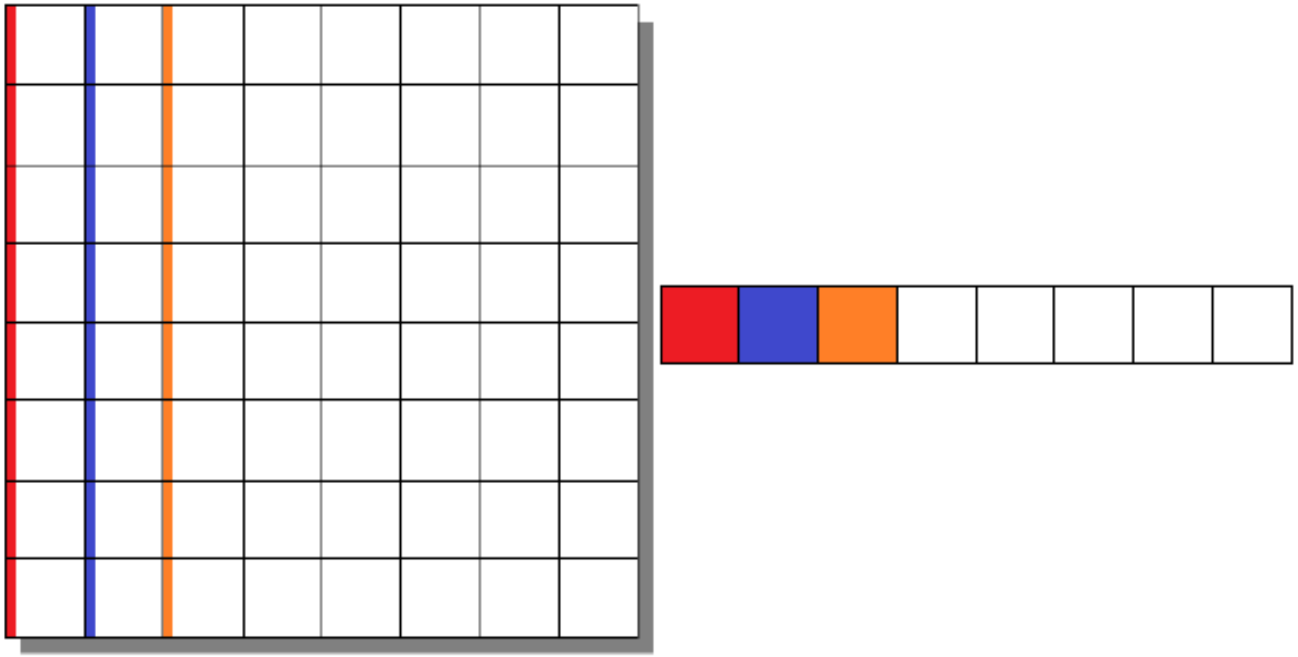
- Make use of 64-bit NEON loads (VLD1.64)
- Still need single byte ARM loads (LDRB) and address computation
- 20 cycle penalty when moving data between ARM and NEON
- Avoid by interleaving computation of P and Q
- 45.8 cycles/byte (previously: 76.9)

NEON: Alternating T-Table Lookups for *P* and *Q*

```
/* ROW 1 (SH+SB+MB) */
/* load state bytes */
/* T-table lookups  */
ldrb r0, [%[P], #9 ];
ldrb r1, [%[P], #17];
ldrb r2, [%[P], #25];
ldrb r3, [%[P], #33];
ldrb r4, [%[P], #41];
ldrb r5, [%[P], #49];
ldrb r6, [%[P], #57];
ldrb r7, [%[P], #1 ];
vld1.64 d8,  [r0, :64];
vld1.64 d9,  [r1, :64];
vld1.64 d10, [r2, :64];
vld1.64 d11, [r3, :64];
vld1.64 d12, [r4, :64];
vld1.64 d13, [r5, :64];
vld1.64 d14, [r6, :64];
vld1.64 d15, [r7, :64];
```

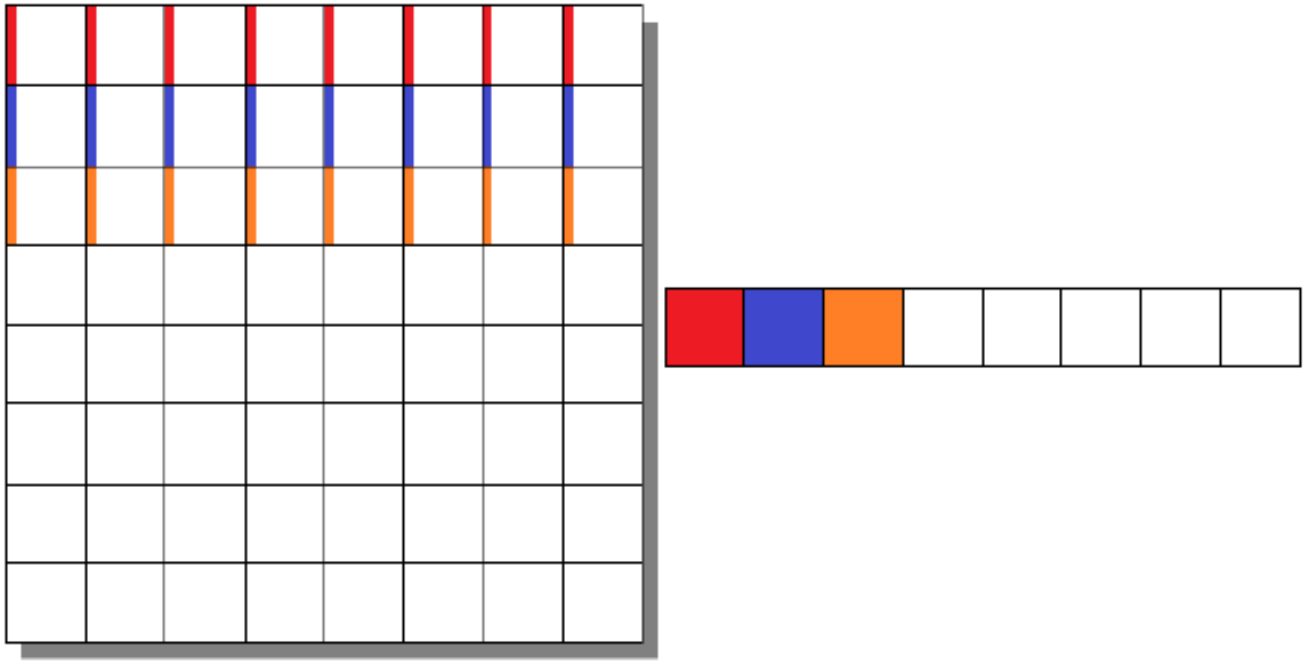
```
/* increase T-table address */
/* compute lookup address */
/* xor results */
add %[T], %[T], #2048;

add r0, %[T], r0, asl #3;
add r1, %[T], r1, asl #3;
add r2, %[T], r2, asl #3;
add r3, %[T], r3, asl #3;
add r4, %[T], r4, asl #3;
add r5, %[T], r5, asl #3;
add r6, %[T], r6, asl #3;
add r7, %[T], r7, asl #3;

veor q0, q0, q4;
veor q1, q1, q5;
veor q2, q2, q6;
veor q3, q3, q7;
```

NEON: Bitslice Implementation using VSHL and VEXT



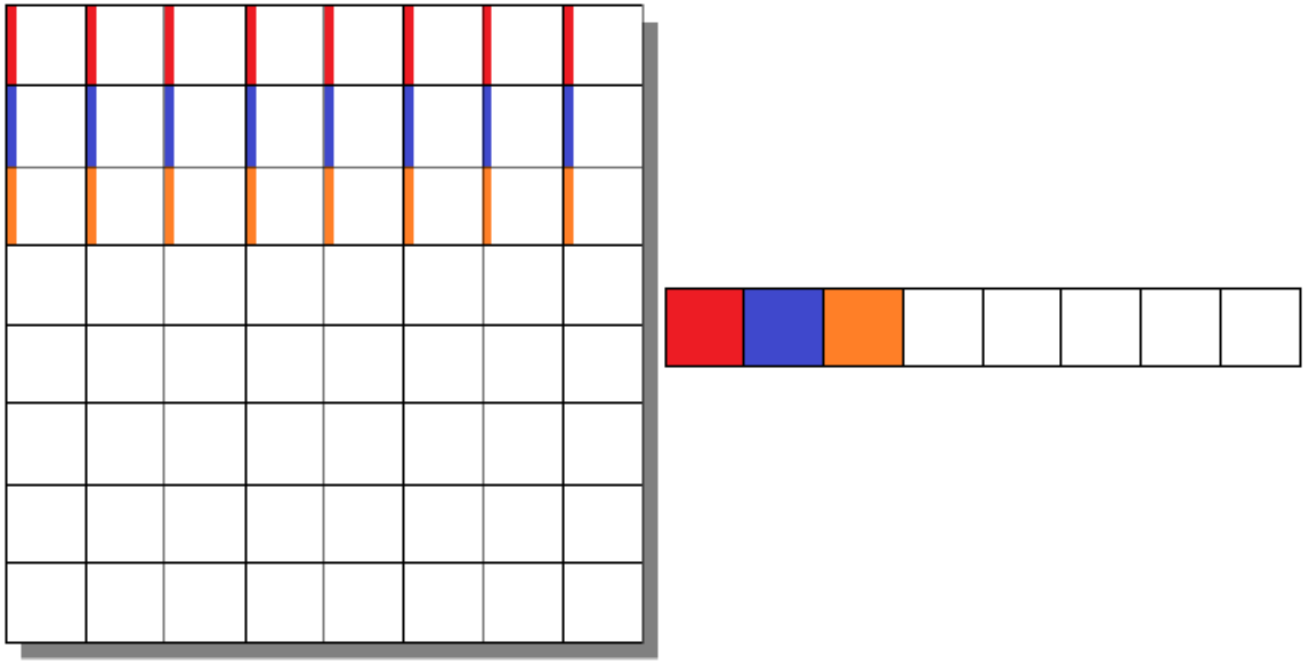
- Store the 8 bits of each byte in 8 separate 128-bit registers ($P||Q$)
- Efficiency strongly depends on arrangement of bits
  - combine columns in bytes

NEON: Bitslice Implementation using VSHL and VEXT



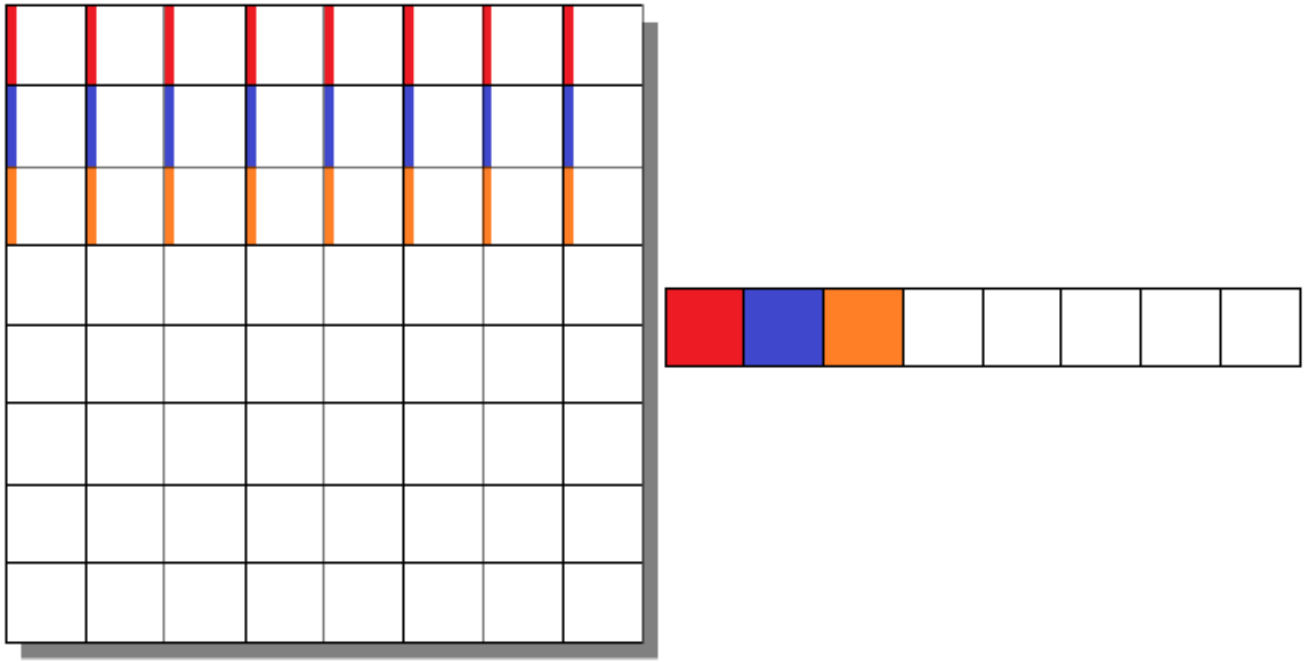
- Store the 8 bits of each byte in 8 separate 128-bit registers ($P||Q$)
- Efficiency strongly depends on arrangement of bits
  - combine columns in bytes
  - combine rows in bytes (more efficient using NEON)

NEON: Bitslice Implementation using VSHL and VEXT



- Store the 8 bits of each byte in 8 separate 128-bit registers ($P||Q$)
- Efficiency strongly depends on arrangement of bits
  - combine columns in bytes
  - combine rows in bytes (more efficient using NEON)
- ShiftBytes: variable rotation of bits within bytes (VSHL)
- MixBytes: to XOR rows, we first rotate bytes of registers (VEXT)

NEON: Bitslice Implementation using VSHL and VEXT

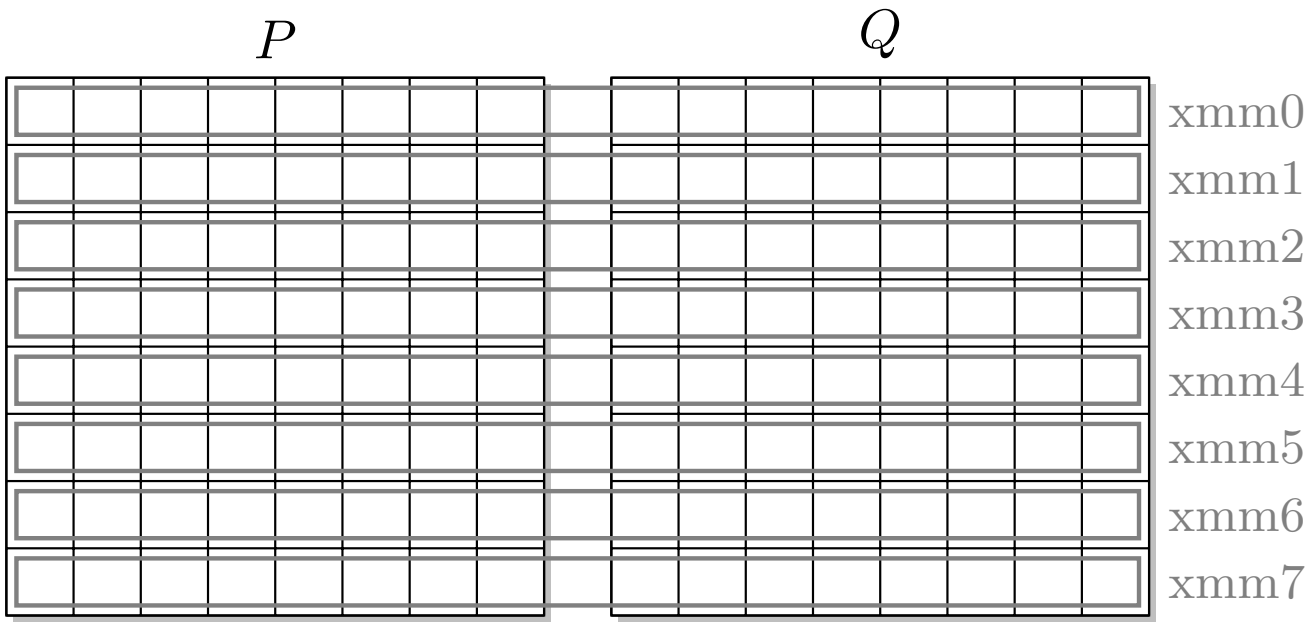


- Store the 8 bits of each byte in 8 separate 128-bit registers ($P||Q$)
- Efficiency strongly depends on arrangement of bits
  - combine columns in bytes
  - combine rows in bytes (more efficient using NEON)
- ShiftBytes: variable rotation of bits within bytes (VSHL)
- MixBytes: to XOR rows, we first rotate bytes of registers (VEXT)
- 48.5 cycles/byte (similar to table based)

NEON: Bitslice Implementation using VSHL and VEXT

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

NEON: Vperm Implementation



- Byteslice implementation using optimized MixBytes formulas
- Computing S-box using vperm approach
  - relatively expensive using NEON
  - improvements possible by optimizing dependency chains
  - base point for AES instruction implementation (ARM8)
- 92.0 cycles/byte

Cortex-M0: Low Memory Vector Implementation

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

- Many different improved implementations (T-table, byteslice)
- Best results using 32-bit byteslicing
  - only S-box tables needed (no large T-tables)
  - almost the speed of T-table implementation

Outline

Conclusion

- Many new implementations of AES-based hash function `Grøstl`
  - 2 AVX2 implementations
  - 3 NEON implementations
  - 4 low-mem Cortex-M0 implementations
- All implementations with significant improvements
- Ideas are applicable to any AES-based design
- Use results to avoid bottlenecks in new AES-based designs

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.


Eli Biham.
A fast new DES implementation in software.
In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *LNCS*, pages 260–272. Springer, 1997.
`http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1997/CS/CS0891.pdf`.


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.


Joan Daemen and Vincent Rijmen.
AES Proposal: Rijndael.
NIST AES Algorithm Submission, September 1999.
Available online:
`http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

References II


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).


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.


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


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