

Cryptographic Engineering

Symmetric crypto in software

Radboud University, Nijmegen, The Netherlands



Spring 2019

Symmetric crypto overview

Primitives and algorithms

- ▶ **Block ciphers:** AES, Serpent, DES (and 3DES), IDEA, Present, LED, Prince, Klein, ...
- ▶ **Stream ciphers:** RC4, Salsa20, ChaCha20, HC-128, Rabbit, SOSEMANUK, Grain, MICKEY, Trivium, ...
- ▶ **Hash functions:** SHA-256, SHA-512, SHA-3, Blake, Blake2, ...
- ▶ **Authenticated encryption:** AES-GCM, Poly-1305, CAESAR, ...

Architectures and microarchitectures

- ▶ **Architectures:** x86, AMD64, ARMv6, ARMv7, ARMv8, AVR, 32-bit PowerPC, 64-bit PowerPC, SPARCv9, ...
- ▶ **Microarchitectures:** Pentium 4, Penryn, Nehalem, Sandy Bridge, Haswell, Cortex-A8, Cortex-A9, Cortex-A53, ...
- ▶ **Instruction-set extensions:** SSE, SSE2, SSE3, SSSE3, AVX, AVX2, AltiVec, NEON, ...

The Advanced Encryption Standard (AES)

- ▶ Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- ▶ Selected as AES by NIST in October 2000
- ▶ Block size: 128 bits (AES state: 4×4 matrix of 16 bytes)
- ▶ Key size 128/192/256 bits (resp. 10/12/14 rounds)
- ▶ AES with n rounds uses $n + 1$ 16-byte rounds keys K_0, \dots, K_n
- ▶ Four operations per round: SubBytes, ShiftRows, MixColumns, and AddRoundKey
- ▶ Last round does not have MixColumns

High-level pseudocode AES-128

Require: 128-bit input block B , 128-bit AES round keys K_0, \dots, K_{10}

Ensure: 128-bit block of encrypted output

$B \leftarrow \text{AddRoundKey}(B, K_0)$

for i from 1 to 9 **do**

$B \leftarrow \text{SubBytes}(B)$

$B \leftarrow \text{ShiftRows}(B)$

$B \leftarrow \text{MixColumns}(B)$

$B \leftarrow \text{AddRoundKey}(B, K_i)$

end for

$B \leftarrow \text{SubBytes}(B)$

$B \leftarrow \text{ShiftRows}(B)$

$B \leftarrow \text{AddRoundKey}(B, K_{10})$

return B

AES on 32-bit and 64-bit processors

- ▶ Idea from the AES proposal: Merge SubBytes, ShiftRows, and MixColumns
- ▶ Use 4 lookup tables T0, T1, T2, and T3 (1 KB each)

The first round of AES in C

- ▶ Input: 32-bit integers y_0, y_1, y_2, y_3
- ▶ Output: 32-bit integers z_0, z_1, z_2, z_3
- ▶ Round keys in 32-bit-integer array $rk[44]$

```
z0 = T0[ y0 >> 24          ] ^ T1[(y1 >> 16) & 0xff] \  
    ^ T2[(y2 >> 8) & 0xff] ^ T3[ y3          & 0xff] ^ rk[4];  
z1 = T0[ y1 >> 24          ] ^ T1[(y2 >> 16) & 0xff] \  
    ^ T2[(y3 >> 8) & 0xff] ^ T3[ y0          & 0xff] ^ rk[5];  
z2 = T0[ y2 >> 24          ] ^ T1[(y3 >> 16) & 0xff] \  
    ^ T2[(y0 >> 8) & 0xff] ^ T3[ y1          & 0xff] ^ rk[6];  
z3 = T0[ y3 >> 24          ] ^ T1[(y0 >> 16) & 0xff] \  
    ^ T2[(y1 >> 8) & 0xff] ^ T3[ y2          & 0xff] ^ rk[7];
```

What a machine is really doing

```
unsigned char rk[176], T0[1024], T1[1024], T2[1024], T3[1024];

z0 = *(uint32 *) (rk + 16);
z1 = *(uint32 *) (rk + 20);
z2 = *(uint32 *) (rk + 24);
z3 = *(uint32 *) (rk + 28);

z0 ^= *(uint32 *) (T0 + ((y0 >> 22) & 0x3fc)) \
      ^ *(uint32 *) (T1 + ((y1 >> 14) & 0x3fc)) \
      ^ *(uint32 *) (T2 + ((y2 >> 6) & 0x3fc)) \
      ^ *(uint32 *) (T3 + ((y3 << 2) & 0x3fc));
z1 ^= *(uint32 *) (T0 + ((y1 >> 22) & 0x3fc)) \
      ^ *(uint32 *) (T1 + ((y2 >> 14) & 0x3fc)) \
      ^ *(uint32 *) (T2 + ((y3 >> 6) & 0x3fc)) \
      ^ *(uint32 *) (T3 + ((y0 << 2) & 0x3fc));
z2 ^= *(uint32 *) (T0 + ((y2 >> 22) & 0x3fc)) \
      ^ *(uint32 *) (T1 + ((y3 >> 14) & 0x3fc)) \
      ^ *(uint32 *) (T2 + ((y0 >> 6) & 0x3fc)) \
      ^ *(uint32 *) (T3 + ((y1 << 2) & 0x3fc));
z3 ^= *(uint32 *) (T0 + ((y3 >> 22) & 0x3fc)) \
      ^ *(uint32 *) (T1 + ((y0 >> 14) & 0x3fc)) \
      ^ *(uint32 *) (T2 + ((y1 >> 6) & 0x3fc)) \
      ^ *(uint32 *) (T3 + ((y2 << 2) & 0x3fc));
```

AES instruction counts

- ▶ Each round has 20 loads, 16 shifts, 16 masks and 16 xors
- ▶ Last round is slightly different: Needs 16 more mask instructions
- ▶ 4 load instructions to load input, 4 stores for output
- ▶ In CTR mode: 4 xors with the key stream, incrementing the counter
- ▶ ... some more overhead
- ▶ Results in 720 instructions needed to encrypt a block of 16 bytes
- ▶ Specifically: 208 loads, 4 stores, 508 arithmetic instructions

Case study: AES on an UltraSPARC

(My first project as Ph.D. student)



- ▶ 64-bit architecture
- ▶ Up to 4 instructions per cycle
- ▶ At most 2 integer-arithmetic instructions per cycle
- ▶ At most 1 load/store instruction per cycle
- ▶ 24 integer registers available
- ▶ Previous AES speed:
 - ▶ 20.75 cycles/byte by Bernstein (public domain)
 - ▶ 16.875 cycles/byte (270 cycles/block) by Lipmaa (unpublished)

Making AES fast on an UltraSPARC

(My first project as Ph.D. student)

Computing a lower bound

Reminder: 208 loads, 4 stores, 508 integer instructions per 16-byte block

- ▶ Only one load or store per cycle (\Rightarrow at least 212 cycles)
- ▶ Only 2 arithmetic instructions per cycle (\Rightarrow at least 254 cycles)

Making it fast

- ▶ After quite some instruction scheduling: 269 cycles per block
- ▶ My supervisor's reaction:
“... this is no time to relax; you have to not just beat Lipmaa's code, but beat it to a bloody pulp and dance on its grave. :-)”
- ▶ After writing a simplified simulator and more instruction scheduling: 254 cycles/block, 15.98 cycles/byte
- ▶ What now? Is this already a bloody pulp?

Making AES fast on an UltraSPARC

Lowering the lower bound

- ▶ We have to reduce the number of (arithmetic) instructions
- ▶ Idea: The UltraSPARC is a 64-bit architecture, pad 32-bit values with zeros, i.e.,
0xc66363a5 becomes 0x0c60063006300a50
- ▶ Do that consistently for values in registers, the tables and the round keys
- ▶ Interleave entries in tables T0 and T1 and in T2 and T3
- ▶ Instruction set supports 32-bit shifts that zero out the upper 32 bits
- ▶ Apply some more optimizations

Without padded registers

```
t0 = (uint32) y0 >> 22
t1 = (uint32) y0 >> 14
t2 = (uint32) y0 >> 6
t3 = (uint32) y0 << 2
t0 &= 0x3fc
t1 &= 0x3fc
t2 &= 0x3fc
t3 &= 0x3fc
```

With padded registers

```
t0 = (uint64) y0 >> 48
t1 = (uint64) y0 >> 32
t2 = (uint64) y0 >> 16
t1 &= 0xff0
t2 &= 0xff0
t3 = y0 & 0xff0
t0 = (uint64) y0 >> 48
t1 = (uint64) y0 >> 32
t2 = (uint32) y0 >> 16
t1 &= 0xff0
t3 = y0 & 0xff0
```

Final result: AES in CTR mode on UltraSPARC III at 12.06 cycles/byte

More arithmetic tricks for AES I

Combined Shift-and-mask

- ▶ Some architectures have combined shift-and-mask instructions (e.g., PowerPC)
- ▶ Combine 160 shifts and 160 masks and save 160 instructions

Scaled-index loads

- ▶ Some architectures can combine shift and load (e.g., x86, AMD64)
- ▶ Use this to get rid of the mask instruction for top and shift instruction for bottom byte
- ▶ Overall save: 80 instructions

More arithmetic tricks for AES II

Various memory/arithmetic tradeoffs

- ▶ Can extract 4 bytes by one store and 4 loads
- ▶ Saves 160 mask instructions (or 320 if we have scaled-index loads)
- ▶ Costs 40 store and 160 load instructions

Counter-mode caching

- ▶ In CTR mode we encrypt a counter, then XOR keystream with plaintext
- ▶ Last counter byte only changes every 256 blocks
- ▶ Do computations depending on this byte in the first round only once, cache the state
- ▶ Similar in second round: only one 32-bit word changes every round
- ▶ Do computations depending on this word in the second round only once, cache the state
- ▶ Overall save: ≈ 100 instructions

Now forget everything I just said

Timing attacks

- ▶ The lookup-table-based approach is inherently vulnerable to cache-timing attacks
- ▶ Extensive literature on AES cache-timing attacks
- ▶ Osvik, Shamir, Tromer, 2006: Obtain AES-256 key in just 65 ms

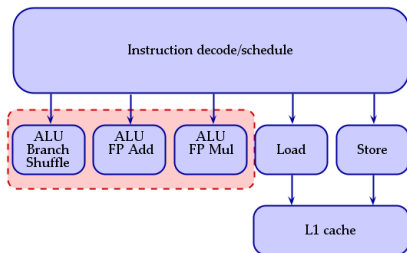
Then why did I tell you this?

- ▶ You have to be able to recognize and understand table-based AES implementations
- ▶ Optimizations show how to make best use of the instruction set
- ▶ General trick: Change your data representation

Looking for an alternative approach

- ▶ Remember bitslicing: vectorized “hardware emulation”
- ▶ Every algorithm can be implemented with bitslicing
- ▶ Bitslicing is inherently protected against timing attacks
- ▶ Efficiency depends on algorithm and micro-architecture
- ▶ Some crypto primitives are designed for efficient bitslicing
- ▶ AES was designed for table-based implementations
- ▶ Obvious question: Can bitsliced AES be fast?
- ▶ Common target for bitslicing AES: Intel Core 2

The Intel Core 2 processor



- ▶ 16 128-bit XMM vector registers
- ▶ 16 64-bit integer registers
- ▶ SSE (Streaming SIMD Extension) instructions
 - ▶ followed by SSE2, SSE3, SSSE3 (Intel), SSE4 (Intel), SSE5 (AMD), AVX, AVX2 (Intel) etc.
- ▶ Native 128-bit wide execution units
- ▶ 3 ALUs – up to 3 bit-logical instructions per cycle
- ▶ Some differences between 65 nm (Core) and 45 nm (Penryn)

Bitslicing AES on Intel Core 2 I

Matsui & Nakajima, 2007

- ▶ Process 128 blocks in parallel
- ▶ Performance: 9.2 cycles/byte
- ▶ Additional overhead for converting to/from bitsliced representation
- ▶ Great for, e.g., hard-disk encryption
- ▶ Bad for encryption of small Internet packets

Könighofer, 2008

- ▶ Process only 4 blocks in parallel
- ▶ Use 64-bit integer registers
- ▶ Performance: 19.6 cycles/byte

Bitslicing AES on Core 2 II

Käser & Schwabe, 2009

- ▶ Similar idea to Könighofer:
 - ▶ Most expensive operation in AES is SubBytes
 - ▶ SubBytes is already 16-times parallel
 - ▶ Exploit this parallelism and reduce number of required blocks
- ▶ Different from Könighofer:
 - ▶ Use 128-bit XMM registers instead of 64-bit registers
 - ▶ Factor-2 speedup for doing more bit ops per instruction
 - ▶ Different optimization (need to use SSE* instructions)
- ▶ Use CTR mode (parallel and does not need decryption)
- ▶ Corresponding decryption later implemented by Azad (2011)

The Bitslicing approach

row 0												row 3												
column 0				column 1				column 2				column 3				column 0				column 3			
block 0	block 1	...	block 7	block 0	block 1	...	block 7	block 0	block 1	...	block 7	block 0	block 1	...	block 7	block 0	block 1	...	block 7	block 0	block 1	...	block 7

- ▶ Process 8 AES blocks (= 128 bytes) in parallel
- ▶ Collect bits according to their position in the byte: i.e., the first register contains least significant bits from each byte, etc.
- ▶ AES state stored in 8 XMM registers
- ▶ Compute 128 S-Boxes in parallel, using bit-logical instructions
- ▶ For a simpler linear layer, collect the 8 bits from identical positions in each block into the same byte
- ▶ Never need to mix bits from different blocks – all instructions byte-level

Implementing the AES S-Box

- ▶ Start from the most compact hardware S-box, 117 gates (Canright 2005; Boyar, Peralta, 2009)
- ▶ Use equivalent 128-bit bit-logical instructions
- ▶ Problem 1: instructions are two-operand, output overwrites one input
- ▶ Hence, sometimes need extra register-register moves to preserve input
- ▶ Problem 2: not enough free registers for intermediate values
- ▶ Recompute some values multiple times (alternative: use stack)
- ▶ Total 163 instructions – 15% shorter than previous results

	xor	and/or	mov	TOTAL
Hardware	82	35	–	117
Software	93	35	35	163

Implementing the AES linear layer

- ▶ Each byte in the bitsliced vector corresponds to a different byte position in the AES state
- ▶ Thus, ShiftRows is a permutation of bytes
- ▶ Use SSSE3 dedicated byte-shuffle instruction `pshufb`
- ▶ Repeat for each bit position (register) \Rightarrow 8 instructions
- ▶ MixColumns uses byte shuffle and XOR, total 43 instructions
- ▶ AddRoundKey also requires only 8 XORs from memory
- ▶ Some caveats:
 - ▶ Bitsliced key is larger – 8×128 bits per round, key expansion slower
 - ▶ SSSE3 available only on Intel, not on AMD processors

Putting it all together

	xor/and/or	pshufb/d	xor (mem-reg)	mov (reg-reg)	TOTAL
SubBytes	128	–	–	35	163
ShiftRows	–	8	–	–	8
MixColumns	27	16	–	–	43
AddRoundKey	–	–	8	–	8
TOTAL	155	24	8	35	222

- ▶ One AES round requires 222 instructions
- ▶ Last round omits MixColumns: 171 instructions
- ▶ Input/output transform 84 instructions/each
- ▶ Excluding data loading etc, we get a lower bound

$$\frac{222 \times 9 + 171 + 2 \times 84}{3 \times (8 \cdot 16)} \approx 6.1 \text{ cycles/byte}$$

- ▶ Actual performance on Core 2 (Penryn): 7.58 cycles/byte

Back to (small) lookup tables

- ▶ Altivec offers a `vperm` instruction
 - ▶ 3 128-bit vector arguments: a , b , c
 - ▶ Replace each byte c_i in c by a byte from a or b , indexed by lowest 5 bits of c_i
- ▶ SSSE3 offers a `pshufb` instruction
 - ▶ 2 128-bit vector arguments: a , c
 - ▶ Shuffle bytes in a (in place) according to indices in c
- ▶ For constant indices in c these instruction implement a permutation
- ▶ For constant inputs a, b they implement a lookup table
 - ▶ 5-bit to 8-bit lookup for `vperm` (32 entries)
 - ▶ 4-bit to 8-bit lookup for `pshufb` (16 entries)

How do these lookup tables help?

- ▶ Idea by Hamburg (2009):
 - ▶ Use arithmetic representation of AES S-Box (inversion in \mathbb{F}_{2^8})
 - ▶ Represent \mathbb{F}_{2^8} as quadratic extension of \mathbb{F}_{2^4}
 - ▶ Use vector-permute lookup tables for arithmetic in \mathbb{F}_{2^4}
- ▶ Approach is fully constant time
- ▶ Not available on every architecture
- ▶ Can combine with counter-mode caching
- ▶ Performance:
 - ▶ 5.4 cycles/byte on Power G4 (CTR mode, 16 parallel blocks)
 - ▶ 21.8 cycles/byte on Core 2 (Core microarch, CTR, no parallel blocks)
 - ▶ 11.1 cycles/byte on Core 2 (Penryn microarch, CTR, no parallel blocks)

AES nowadays

```
pxor %xmm5, %xmm0
aesenc %xmm6, %xmm0
aesenc %xmm7, %xmm0
aesenc %xmm8, %xmm0
aesenc %xmm9, %xmm0
aesenc %xmm10, %xmm0
aesenc %xmm11, %xmm0
aesenc %xmm12, %xmm0
aesenc %xmm13, %xmm0
aesenc %xmm14, %xmm0
aesenclast %xmm15, %xmm0
```

- ▶ AESNI instructions on Intel processors
- ▶ Introduced with Westmere microarchitecture
- ▶ State in `%xmm0`
- ▶ Round keys in `%xmm5 ... %xmm15`
- ▶ Also instructions for key expansion, decryption
- ▶ AES instructions take constant time
- ▶ For parallel modes up to 0.625 cycles/byte (Ivy Bridge)

AES summary

- ▶ Best case: hardware support is available (e.g., AESNI)
- ▶ If not:
 - ▶ Bitslicing (performance highly depends on micro-architecture)
 - ▶ Vector-permute instructions (availability depends on architecture and instruction-set extensions; performance depends on micro-architecture)
 - ▶ Table-based approach is typically fast but vulnerable to timing attacks (almost everywhere)

Why was Rijndael chosen as AES?

- ▶ Faster than, e.g., SERPENT in software (for table-based implementations)
- ▶ From the Report on the Development of the Advanced Encryption Standard (AES), October 2000:

“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.”

AES on Cortex-A8 with NEON

Cortex-A8

- ▶ 32-bit ARMv7 core (2 instructions per cycle with various restrictions)
- ▶ NEON vector coprocessor working on 128-bit vectors
- ▶ Present in a large variety of mobile devices, e.g., Apple iPhone 3GS, Apple iPhone 4, 3rd generation Apple iPod touch (late 2009), Apple iPad 1, Nokia N9, Nokia N900, Palm Pre Plus, Samsung/Google Nexus S, Samsung Galaxy S
- ▶ Today very cheap (e.g., Allwinner A10 for \approx US\$5)

AES performance

- ▶ Table-based (ARM): 28.08 cycles/byte (C code, not optimized for ARM)
- ▶ Bitsliced (NEON): 18.94 cycles/byte
- ▶ Both numbers are for counter mode
- ▶ Vector permute: ???

From AES to Salsa20

- ▶ High-speed AES is typically for streaming modes (e.g., CTR)
- ▶ Simple reason: larger degree of parallelism
- ▶ If we don't need chaining modes, can also use stream cipher (should be faster)
- ▶ Traditional stream cipher: RC4
 - ▶ Broken by Fluhrer, Mantin, and Shamir in 2001
 - ▶ More broken by Klein in 2005
 - ▶ Broken even more by AlFardan, Bernstein, Paterson, Poettering, Schuldt in 2013
 - ▶ ... and even more by Garman, Paterson, and van der Merwe in 2015
 - ▶ ... and by Vanhoef and Piessens in 2015
 - ▶ ... and by Bricout, Murphy, Paterson, and van der Merwe in 2016

From AES to Salsa20

- ▶ High-speed AES is typically for streaming modes (e.g., CTR)
- ▶ Simple reason: larger degree of parallelism
- ▶ If we don't need chaining modes, can also use stream cipher (should be faster)
- ▶ Traditional stream cipher: RC4
 - ▶ Broken by Fluhrer, Mantin, and Shamir in 2001
 - ▶ More broken by Klein in 2005
 - ▶ Broken even more by AlFardan, Bernstein, Paterson, Poettering, Schuldt in 2013
 - ▶ ... and even more by Garman, Paterson, and van der Merwe in 2015
 - ▶ ... and by Vanhoef and Piessens in 2015
 - ▶ ... and by Bricout, Murphy, Paterson, and van der Merwe in 2016
- ▶ Better candidates are in eSTREAM portfolio:
 - ▶ Competition to find good stream ciphers organized by ECRYPT
 - ▶ Running from 2004–2008
 - ▶ Final decision: 3 ciphers in “hardware” portfolio; 4 in “software” portfolio
 - ▶ One cipher in the “software” portfolio: Salsa20 by Bernstein

Salsa20

- ▶ Generates random stream in 64-byte blocks, works on 32-bit integers
- ▶ Blocks are independent
- ▶ Per block: 20 rounds; each round doing 16 add-rotate-xor sequences, such as

```
s4 = x0 + x12
x4 ^= (s4 >>> 25)
```

- ▶ These sequences are 4-way parallel
- ▶ In ARM *without* NEON: 2 instructions, 1 cycle
- ▶ Sounds like total of $(20 \cdot 16)/64 = 5$ cycles/byte, but:
 - ▶ Only 14 integer registers (need at least 17)
 - ▶ Latencies cause big trouble
 - ▶ Actual implementations slower than 15 cycles/byte

A first approach in NEON

- ▶ Per round do 4× something like:

```
4x a0 = diag1 + diag0
```

```
4x b0 = a0 << 7
```

```
4x a0 unsigned >>= 25
```

```
diag3 ^= b0
```

```
diag3 ^= a0
```

- ▶ + some (free) shuffles
- ▶ Intuitive cycle lower bound:
 $(5 \cdot 4 \cdot 20)/64 = 6.25$ cycles/byte
- ▶ Problem: The above sequence has a 9-cycle latency, thus:
 $(9 \cdot 4 \cdot 20)/64 = 11.25$ cycles/byte

Trading parallelism

- ▶ Salsa20 rounds have 4-way data-level parallelism
- ▶ In a scalar implementations this turns into 4-way instruction-level parallelism
- ▶ Good for pipelined and superscalar execution
- ▶ The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left
- ▶ Bad for pipelined and superscalar execution
- ▶ Idea: Blocks are independent, use this to re-introduce instruction-level parallelism
- ▶ Lower bound when interleaving 2 blocks: 6.875 cycles/byte
- ▶ Lower bound when interleaving 3 blocks: 6.25 cycles/byte

Going even further

- ▶ NEON is basically a coprocessor to the ARM core
- ▶ ARM decodes instructions, forwards NEON instructions to the NEON unit
- ▶ Idea: Also keep the ARM core busy with Salsa20 computations
- ▶ New bottleneck: ARM core decodes at most 2 instructions per cycle
- ▶ Add-rotate-xor is only 2 ARM instructions
- ▶ Best tradeoff: One block on ARM, two blocks on NEON

A flavor of the code

```
4x a0 = diag1 + diag0
    4x next_a0 = next_diag1 + next_diag0
        s4 = x0 + x12
        s9 = x5 + x1
4x b0 = a0 << 7
    4x next_b0 = next_a0 << 7
4x a0 unsigned>>= 25
    4x next_a0 unsigned>>= 25
        x4 ^= (s4 >>> 25)
        x9 ^= (s9 >>> 25)
        s8 = x4 + x0
        s13 = x9 + x5
diag3 ^= b0
    next_diag3 ^= next_b0
diag3 ^= a0
    next_diag3 ^= next_a0
        x8 ^= (s8 >>> 23)
        x13 ^= (s13 >>> 23)
```

Result

5.47 cycles/byte for Salsa20 encryption on ARM Cortex-A8 with NEON

References

- ▶ Daniel J. Bernstein, Peter Schwabe. *New AES software speed records*. Indocrypt 2008.
<http://cryptojedi.org/papers/#aesspeed>
- ▶ Robert Könighofer. *A Fast and Cache-Timing Resistant Implementation of the AES*. CT-RSA 2008.
- ▶ Mitsuru Matsui, Junko Nakajima. *On the Power of Bitslice Implementation on Intel Core2 Processor*. CHES 2007.
www.iacr.org/archive/ches2007/47270121/47270121.ps
- ▶ Emilia Käsper, Peter Schwabe. *Faster and Timing-Attack Resistant AES-GCM*. CHES 2009.
<http://cryptojedi.org/papers/#aesbs>
- ▶ Mike Hamburg. *Accelerating AES with Vector Permute Instructions*. CHES 2009.
http://mikehamburg.com/papers/vector_aes/vector_aes.pdf
- ▶ Daniel J. Bernstein, Peter Schwabe. *NEON crypto*. CHES 2012.
<http://cryptojedi.org/papers/#neoncrypto>