SDU
University of
Southern Denmark

Cryptographic Engineering

Optimizing Symmetric Cryptography in Software

Ruben Niederhagen
(based on content by Peter Schwabe)

Department of Mathematics and Computer Science (IMADA)

Department of Mathematics
and Computer Science (IMADA)

Primitives and algorithms:

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

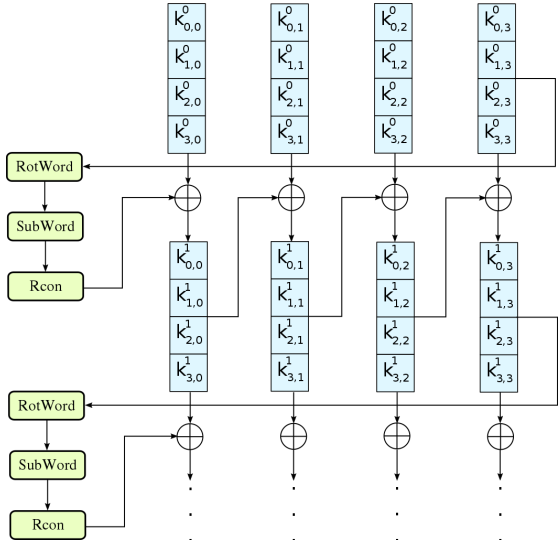
Architectures and microarchitectur:

- 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, . . .
- Inst.-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 round keys $K - 0, \ldots, K_n$.
- Four operations per round: SubBytes, ShiftRows, MixColumns, and AddRoundKey.
- Last round does not have MixColumns.

AES Key Expansion



High-Level Pseudocode AES-128

Require: 128-bit input block B, 128-bit AES round keys $K_0, \ldots, K_{10}$
Ensure: 128-bit block of encrypted output
  $B \leftarrow$AddRoundKey(B, K_0)
  for i from 1 **to** 9 **do**
    $B \leftarrow$Sub*B*ytes(B)
    $B \leftarrow$ShiftRows(B)
    $B \leftarrow$MixColumns(B)
    $B \leftarrow$AddRoundKey(B, K_i)
  end for
  $B \leftarrow$Sub*B*ytes(B)
  $B \leftarrow$ShiftRows(B)
  $B \leftarrow$AddRoundKey(B, K_{10})
  return B

Department of Mathematics
and Computer Science (IMADA)

Each element $b_{i,j}$ in the block B is replaced by the value of the S-box: $b_{i,j} \leftarrow \mathrm{S}(b_{i,j})$

The S-box is actually the inverse function in $\mathbb{F}_{2^8}$ with reduction by $x'^8 + x'^4 + x'^3 + x' + 1$ (elements in $\mathbb{F}_{2^8}$ represented as polynomials in $\mathbb{F}_2[x']$).

So, the S-box can for example be implemented using a lookup table with 256 entries or by explicitly implementing the finite field inversion.

ShiftRows in Detail

$$\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{bmatrix}$$

Each column $(a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})^T$ is multiplied by $c(x)$.

$c(x) = \quad 3 \quad x^3 + 1x^2 + 1x + \quad 2 \quad$ is a fixed polynomial of degree 4 over $\mathbb{F}_{2^8}$, so the columns are interpreted as polynomials as well.

Polynomial multiplication in $GF(2^8)[x]$ is performed modulo $x^4 + 1$.

Let $b(x) = c(x) \cdot a(x)$. Then:

$(3x^3$	$+1x^2$	$+1x$	$+2)$	$\cdot$	$(a_{3,j}x^3$	$+a_{2,j}x^2$	$+a_{1,j}x$	$+a_{0,j})$	
					$(3x^3$	$+1x^2$	$+1x$	$+2)$	$\cdot a_{0,j}$
					$(1x^3$	$+1x^2$	$+2x$	$+3)$	$\cdot a_{1,j}$
					$(1x^3$	$+2x^2$	$+3x$	$+1)$	$\cdot a_{2,j}$
					$(2x^3$	$+3x^2$	$+1x$	$+1)$	$\cdot a_{3,j}$
					$b_{3,j}x^3$	$b_{2,j}x^2$	$b_{1,j}x$	$b_{0,j}$	

Department of Mathematics
and Computer Science (IMADA)

Each column $(a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})^T$ is multiplied by $c(x)$.

$c(x) = \quad 3 \quad x^3 + 1x^2 + 1x + \quad 2 \quad$ is a fixed polynomial of degree 4 over $\mathbb{F}_{2^8}$, so the columns are interpreted as polynomials as well.

Polynomial multiplication in $GF(2^8)[x]$ is performed modulo $x^4 + 1$.

Let $b(x) = c(x) \cdot a(x)$. Then:

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix}$$

(Element-wise multiplication in $\mathbb{F}_{2^8}$.)

Department of Mathematics
and Computer Science (IMADA)

SubBytes:

$$b_{i,j} = S[a_{i,j}], \quad 0 \leq i, j < 4.$$

ShiftRows:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j+0 \bmod 4} \\ b_{1,j+1 \bmod 4} \\ b_{2,j+2 \bmod 4} \\ b_{3,j+3 \bmod 4} \end{bmatrix}$$

MixColumns:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$$

Department of Mathematics
and Computer Science (IMADA)

All together:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} S[a_{0,j+0 \bmod 4}] \\ S[a_{1,j+1 \bmod 4}] \\ S[a_{2,j+2 \bmod 4}] \\ S[a_{3,j+3 \bmod 4}] \end{bmatrix}$$

Matrix multiplication component-wise:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} S[a_{0,j+0 \bmod 4}] \oplus \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix} S[a_{1,j+1 \bmod 4}] \oplus \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix} S[a_{2,j+2 \bmod 4}] \oplus \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix} S[a_{3,j+3 \bmod 4}]$$

Department of Mathematics
and Computer Science (IMADA)

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

Define lookup tables:

$$T_0[a] = \begin{bmatrix} 2 \cdot S[a_{0,j+0 \bmod 4}] \\ 1 \cdot S[a_{0,j+0 \bmod 4}] \\ 1 \cdot S[a_{0,j+0 \bmod 4}] \\ 3 \cdot S[a_{0,j+0 \bmod 4}] \end{bmatrix}, T_1[a] = \begin{bmatrix} 3 \cdot S[a_{1,j+1 \bmod 4}] \\ 2 \cdot S[a_{1,j+1 \bmod 4}] \\ 1 \cdot S[a_{1,j+1 \bmod 4}] \\ 1 \cdot S[a_{1,j+1 \bmod 4}] \end{bmatrix},$$

$$T_2[a] = \begin{bmatrix} 1 \cdot S[a_{2,j+2 \bmod 4}] \\ 3 \cdot S[a_{2,j+2 \bmod 4}] \\ 2 \cdot S[a_{2,j+2 \bmod 4}] \\ 1 \cdot S[a_{2,j+2 \bmod 4}] \end{bmatrix}, T_3[a] = \begin{bmatrix} 1 \cdot S[a_{3,j+3 \bmod 4}] \\ 1 \cdot S[a_{3,j+3 \bmod 4}] \\ 3 \cdot S[a_{3,j+3 \bmod 4}] \\ 2 \cdot S[a_{3,j+3 \bmod 4}] \end{bmatrix}.$$

Department of Mathematics
and Computer Science (IMADA)

The first round of AES in C:

- Input: 32-bit integers `y0`, `y1`, `y2`, `y3`; Output: 32-bit integers `z0`, `z1`, `z2`, `z3`
- 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 + ((y3 >>  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

(Bernstein, Schwabe: ``New AES Software Speed Records'', 2008)

- 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).

Department of Mathematics
and Computer Science (IMADA)

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
- After writing a simplified simulator and more instruction scheduling: 254 cycles/block, 15.98 cycles/byte .
- What now? Is this as good as it gets?

Department of Mathematics
and Computer Science (IMADA)

Lower 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 T_0 and T_1 and in T_2 and T_3.
- Instruction set supports 32-bit shifts that zero out the upper 32 bits.
- Apply some more optimizations.
- Final result: AES in CTR mode on UltraSPARC III at 12.06 cycles/byte.

Department of Mathematics
and Computer Science (IMADA)

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.

Department of Mathematics
and Computer Science (IMADA)

Various memory/arithmetic trade-offs:

- Can extract 4 bytes by one store and 4 loads.
- Saves 160 mask instructions.
- Costs 40 store and 160 load instructions.

Counter-mode caching:

- In CTR mode we encrypt a counter, then XOR key-stream 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 state.
- Overall save: $\approx$ 100 instructions.

Small Break

Department of Mathematics
and Computer Science (IMADA)

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.

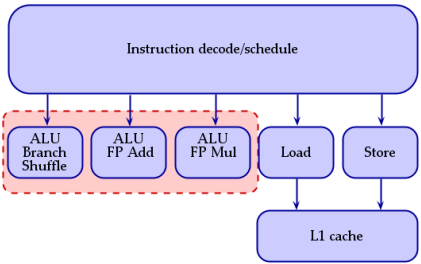
Then why did we discuss this?

- 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.

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).

Department of Mathematics
and Computer Science (IMADA)

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.

Käsper & 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).

Bitslicing Approach

Department of Mathematics
and Computer Science (IMADA)

row 0																.....	row 3												
column 0				column 1				column2				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

Department of Mathematics
and Computer Science (IMADA)

- Each byte in the bitsliced vector corresponds to a different byte 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 of

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

- 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 be combined with counter-mode caching.
- Performance:
  - 5.4 cyles/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).

- 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.

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 iPhones and iPads, Nokia and Samsuns smart phones, . . .
- Today very cheap ($\approx$ 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.