Implementing AES 2000-2010: performance and security challenges

Emilia Käsper

Katholieke Universiteit Leuven

SPEED-CC
Berlin, October 2009
1. The AES Performance Challenge
   - The need for fast encryption
   - Inside AES
   - Implementing AES 2000-...

2. The AES Security Challenge
   - Cache-timing attacks
   - Countermeasures
   - Bitsliced implementations of AES 2007-...
   - A new bitsliced implementation

3. The Future
   - AES-NI instruction set
   - Lessons learnt
   - Implementing cryptography 2010-...
The Advanced Encryption Standard

- Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES in October 2000
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
- Software performance a key advantage
  - Runner-up Serpent arguably “more secure”, but over 2x slower
- AES in OpenSSL — implementation by Rijmen, Bosselaers, Barreto from 2000
- AES-128 at around 18 cycles/byte = 110 MB/s @ 2GHz
The AES performance challenge

- Is 110 MB/s fast enough?
- Popular example: Truecrypt transparent disk encryption
- Truecrypt only supports AES-256, so make that 80 MB/s
- At the same time, consumer (solid state) hard drives can read at over 200 MB/s
- Encryption becomes performance bottleneck
- Since March 2008, Truecrypt includes an optimized assembly implementation of AES
Optimized implementations on Intel processors

2000: Aoki and Lipmaa report 14.8 cycles/byte on Pentium II

\[ \ldots \]

2007: Matsui and Nakajima report 9.2 cycles/byte for AES-CTR on Core 2
  - Assuming data is processed in 2 KB blocks
  - Compatibility with existing implementations via an extra input/output transform

2008: Bernstein-Schwabe report 10.57 cycles/byte for AES-CTR on Core 2

2009: Käsper-Schwabe report 7.59 cycles/byte for AES-CTR on Core 2
Inside AES

Plaintext in 4x4 grid

AES Crib Sheet
(Handy for memorizing)

General Math

11B = AES Polynomial = (x^4 + x + 1)

Fast Multiply

x^4 + x^3 + x + 1

x - a(x) = (a << 1) @ (a^2 = 1)

log(x*y) = log(x) + log(y)

Use (x+1) = 03 for log base

S-Box (SRD)

SRD[a] = f(s)(a)

g(a) = a^{-1} mod m(x)

Think 53 03

S-box and 3D's [0100 0111] t

Key Expansion: Round Constants

First Column: 1.0000 ... 01

Mix Columns:

Inverse Mix

Prev Col + Col from previous round key

Emilia Käsper
Implementing AES 2000-2010
AES round structure

- **SubBytes** is an S-Box acting on individual bytes

\[
\begin{array}{cccc}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33} \\
\end{array}
\quad \rightarrow \quad
\begin{array}{cccc}
  b_{00} & b_{01} & b_{02} & b_{03} \\
  b_{10} & b_{11} & b_{12} & b_{13} \\
  b_{20} & b_{21} & b_{22} & b_{23} \\
  b_{30} & b_{31} & b_{32} & b_{33} \\
\end{array}
\]

- **ShiftRows** rotates each row by a different amount

\[
\begin{array}{cccc}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33} \\
\end{array}
\quad\leftarrow\quad
\begin{array}{cccc}
  a_{11} & a_{12} & a_{13} & a_{10} \\
  a_{22} & a_{23} & a_{20} & a_{21} \\
  a_{33} & a_{30} & a_{31} & a_{32} \\
  a_{33} & a_{30} & a_{31} & a_{32} \\
\end{array}
\]
**AES round structure (cont.)**

- **MixColumns** is a linear transformation on columns

\[
\begin{array}{cccc}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33} \\
\end{array}
\rightarrow
\begin{array}{cccc}
  b_{00} & b_{01} & b_{02} & b_{03} \\
  b_{10} & b_{11} & b_{12} & b_{13} \\
  b_{20} & b_{21} & b_{22} & b_{23} \\
  b_{30} & b_{31} & b_{32} & b_{33} \\
\end{array}
\]

- **AddRoundKey** XORs the 128-bit round key to the state

\[
\begin{array}{cccc}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33} \\
\end{array} +
\begin{array}{cccc}
  k_{r00} & k_{r01} & k_{r02} & k_{r03} \\
  k_{r10} & k_{r11} & k_{r12} & k_{r13} \\
  k_{r20} & k_{r21} & k_{r22} & k_{r23} \\
  k_{r30} & k_{r31} & k_{r32} & k_{r33} \\
\end{array}
\]
**Implementing an AES round**

- Store AES state in 4 column vectors
- Combine `SubBytes`, `ShiftRows` and `MixColumns`:
  - Each column vector depends on 4 bytes
- Do 4 8-to-32-bit table lookups and combine using XOR

```
\begin{align*}
  b_0 &= T_0[\text{a0} \gg 24] \quad \text{T1}[\text{a1} \gg 16] \& \text{0xff} \\
  b_1 &= T_0[\text{a1} \gg 24] \quad \text{T2}[\text{a2} \gg 8] \& \text{0xff} \\
  b_2 &= T_0[\text{a2} \gg 24] \quad \text{T2}[\text{a3} \gg 8] \& \text{0xff} \\
  b_3 &= T_0[\text{a3} \gg 24] \quad \text{T2}[\text{a0} \gg 8] \& \text{0xff} \\
\end{align*}
```

---

**Emilia Käsper**

Implementing AES 2000-2010 9/31
The Core 2 execution units

Instruction decode/schedule

- ALU Branch Shuffle
- ALU FP Add
- ALU FP Mul
- Load
- Store

L1 cache
The Core 2 execution units

- The Core 2 can do one load per clock cycle
- AES-128 needs 160 table lookups to encrypt 16 bytes
- 10 cycles/byte barrier using this technique
Foot-Shooting Prevention Agreement

I, ________, promise that once
Your Name
I see how simple AES really is, I will not implement it in production code even though it would be really fun.

This agreement shall be in effect until the undersigned creates a meaningful interpretive dance that compares and contrasts cache-based, timing, and other side channel attacks and their countermeasures.

Signature

Date

Emilia Käsper
Implementing AES 2000-2010
Cache attacks on AES implementations

- Core idea (Kocher, 1996): variable-time instructions manipulating the secret key leak information about key bits
- Table lookups take different time depending on whether the value was retrieved from cache or memory
- The case of AES: lookup table indices directly depend on the secret key
- First round of AES: $T[plaintext \oplus roundkey]$
- Knowing which part of the table was accessed leaks key bits
Cache attacks (cont.)

- A variety of attack models
  - Active cache manipulation via user processes — preload cache with known values and observe via timing if the cache was hit
  - Passive (remote) timing of cache “hits” and “misses” — shorter encryption time implies collisions in lookups
  - Power traces

- Example: passive timing attack (Bonneau, Mironov 2006)
  - Attacker runs timing code on target machine
  - Obtain timing data from $2^{14}$ random encryptions
  - Deduce when first-round collisions occur to recover 5 bits of each key byte (assuming 32-byte cache line)
  - Can be improved to recover the whole key by considering second/last round
Countermeasures against cache attacks

- Protecting vulnerable cipher parts (e.g., first and last round) in software — only thwarts current attacks
- Add variable-time dummy instructions — attacks still work with more data
- Cache warming (preload some values) — for 32-byte cache line, \( \frac{4 \cdot 256}{8} = 128 \) instructions to preload all tables
- Force all operations to take constant time — as good as having no cache
- Algorithm-specific constant-time implementations
Bitslicing AES

- Bitslicing (Biham, 1997): instead of using lookup tables, evaluate S-Boxes on the fly using their Boolean form
- Efficient if multiple S-boxes can be computed in parallel
- Serpent: bitsliced design, 32 $4 \times 4$-bit S-boxes in each round
- AES $8 \times 8$ S-box based on Galois field inversion, matrix multiplication: ?
  - 2007: Matsui shows an efficient implementation using 128 parallel blocks
  - 2008: Könighofer’s implementation on 64-bit processors, 4 parallel blocks, $< 20$ cycles/byte
Bitslicing AES on Core 2 (2009)

- Implementation of AES in counter mode
- Applicable to any other parallel mode
  - Counter mode particularly handy, as no need to implement decryption
- Hand-coded in GNU assembly/qhasm
- Constant-time, immune to all timing attacks

New speed record

7.59 cycles/byte for large blocks

- Also fast for packet encryption
Making the most out of Core 2

- 16 128-bit XMM registers
- SSE (Streaming SIMD Extension) instructions
  - followed by SSE2, SSE3, SSSE3 (Intel), SSE4 (Intel), SSE5 (AMD), AVX (Intel) etc.
- “native” 128-bit wide execution units
  - older Core Duo’s “packed” 128-bit instructions
- 3 ALU units – up to 3 bit-logical instructions per cycle
### The Bitslicing approach

- 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 [Can05, BP09]
- 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
- We recompute some values multiple times (alternative: use stack)
- Total 163 instructions — 15% shorter than previous results

<table>
<thead>
<tr>
<th></th>
<th>xor</th>
<th>and/or</th>
<th>mov</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>82</td>
<td>35</td>
<td>–</td>
<td>117</td>
</tr>
<tr>
<td>Software</td>
<td>93</td>
<td>35</td>
<td>35</td>
<td>163</td>
</tr>
</tbody>
</table>
Example: multiplication in $GF(2^2)$

\[(x_1, x_0) \otimes (y_1, y_0) \rightarrow (z_1, z_0)\]

\[
z_1 = (y_0 + y_1)x_0 + x_1y_0
\]

\[
z_0 = (x_0 + x_1)y_1 + x_1y_0
\]

```
movdqa \ x0, \ z0
movdqa \ x1, \ z1
movdqa \ y0, \ t0
pxor \ y1, \ t0
pand \ z0, \ t0
pxor \ z1, \ z0
pand \ y1, \ z0
pand \ y0, \ z1
pxor \ z1, \ z0
pxor \ t0, \ z1
```
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) = 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 - $8 \times 128$ bits per round, key expansion slower
  - SSSE3 available only on Intel, not on AMD processors
### Putting it all together

<table>
<thead>
<tr>
<th></th>
<th>xor/and/or</th>
<th>pshufb/d</th>
<th>xor (mem-reg)</th>
<th>mov (reg-reg)</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SubBytes</strong></td>
<td>128</td>
<td>–</td>
<td>–</td>
<td>35</td>
<td>163</td>
</tr>
<tr>
<td><strong>ShiftRows</strong></td>
<td>–</td>
<td>8</td>
<td>–</td>
<td>–</td>
<td>8</td>
</tr>
<tr>
<td><strong>MixColumns</strong></td>
<td>27</td>
<td>16</td>
<td>–</td>
<td>–</td>
<td>43</td>
</tr>
<tr>
<td><strong>AddRoundKey</strong></td>
<td>–</td>
<td>–</td>
<td>8</td>
<td>–</td>
<td>8</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>155</td>
<td>24</td>
<td>8</td>
<td>35</td>
<td>222</td>
</tr>
</tbody>
</table>

- One AES round requires 214 bit-logical instructions
- Last round omits MixColumns — 171 instructions
- Input/output transform 84 instructions/each
- Excluding data loading etc, we get a lower bound

\[
\frac{214 \times 9 + 171 + 2 \times 84}{3 \times 128} = 5.9 \text{ cycles/byte}
\]

- Actual performance on Core 2 7.59 cycles/byte
eStream benchmarks of AES-CTR-128

AES-CTR performance on Core 2 Q9550

- constant-time
- table-based

Emilia Käsper
Implementing AES 2000-2010
Even faster on the Core i7...

AES-CTR performance on Core i7 920

- Constant-time
- Table-based
3 logically equivalent instructions: `xorps`, `xorpd`, `pxor`

- On Core 2, we saw no performance difference
- On Core i7, using `xorps/xorpd` gave a 50% performance hit
Interlude: A little lesson...

- 3 logically equivalent instructions: `xorps`, `xorpd`, `pxor`
- On Core 2, we saw no performance difference
- On Core i7, using `xorps`/`xorpd` gave a 50% performance hit
- The reason: only one unit in Core i7 handles fp Boolean

**Lesson**

Always use the instruction appropriate for your data type!
Implementing AES 2010-...

- Intel has announced hardware support for AES in its next generation processors (AES-NI instruction set extension)

Implementation simplicity:

\[
\begin{align*}
    b_0 &= T_0[a_0 >> 24] \ ^\wedge T_1[(a_1 >> 16) & 0xff] \\
         & \ ^\wedge T_2[(a_2 >> 8) & 0xff] \ ^\wedge T_3[a_3 & 0xff] \ ^\wedge rk[4]; \\
    b_1 &= T_0[a_1 >> 24] \ ^\wedge T_1[(a_2 >> 16) & 0xff] \\
         & \ ^\wedge T_2[(a_3 >> 8) & 0xff] \ ^\wedge T_3[a_0 & 0xff] \ ^\wedge rk[5]; \\
    b_2 &= T_0[a_2 >> 24] \ ^\wedge T_1[(a_3 >> 16) & 0xff] \\
         & \ ^\wedge T_2[(a_0 >> 8) & 0xff] \ ^\wedge T_3[a_1 & 0xff] \ ^\wedge rk[6]; \\
    b_3 &= T_0[a_3 >> 24] \ ^\wedge T_1[(a_0 >> 16) & 0xff] \\
         & \ ^\wedge T_2[(a_1 >> 8) & 0xff] \ ^\wedge T_3[a_2 & 0xff] \ ^\wedge rk[7];
\end{align*}
\]

becomes

aesenc xmm1, xmm3  % xmm1 - data, xmm3 - key

- Mitigates side-channel attacks
- Performance
  - Straightforward 4.4 cycles/byte
  - Parallel/optimized 1.35 cycles/byte
Concluding remarks I

- Breaking the 10 cycles/byte barrier: 7.59 cycles/byte for AES (from 110 MB/s in OpenSSL to 260 MB/s @ 2GHz)

- A posteriori improvement — AES was designed to be implemented with lookup tables

- In comparison: Whirlpool hash function uses an $8 \times 8$ S-Box composed of 5 $4 \times 4$ S-Boxes

- Small by design: 101 gates in hardware

- Slightly inferior security: maximum differential probability $2^{-5}$ vs $2^{-6}$ for AES

- AES requires only 16 gates more!

- But this is a result of 10 years of optimization...
Concluding remarks II

- Dedicated instructions (Intel AES-NI) available soon, but...
- ...almost 10 years after standardization, 5+? years to become widespread
- A general lesson: trends in processor architecture/graphics processing in favour of fast crypto
- Next generation processors: 256-bit registers, three operand instructions
- The case of Serpent: processing 4 blocks in parallel (8 for 256-bit) could yield up to factor 4 (8) performance improvement
- In Practice: reports of factor 2.7 improvement over a previous implementation
QHASM implementations of bitsliced AES:

http://cryptojedi.org/crypto/#aesbs

GNU asm implementations:

http://homes.esat.kuleuven.be/~ekasper/#software

A Stick Figure Guide to the Advanced Encryption Standard:

http://www.moserware.com