AES on the ARM Cortex-M3 and M4

Peter Schwabe and Ko Stoffelen
More AES software implementations?

- AES on AVR [OBSC10]
- AES on SPARC [BS08]
- AES on PowerPC [BS08]
- AES on NVIDIA GPU [OBSC10]
- AES on Cell [OBSC10]
- AES on x86-64 with AVX, SSE [Kön08, KS09]
- AES on Cortex-A with NEON [BGRV15]
- AES on StrongARM (ARMv4) [OBSC10]
- ...

Still, not so much to choose from on Cortex-M
Over 10 billion processors shipped by 2015 [ARM15]
Hardware coprocessor not always available
Target platforms

- Cortex-M3 and Cortex-M4
- 16 32-bit registers, 3 taken for pc, sp, (lr)
- ARMv7-M instruction set
  - `eor r2, r0, r1, ror #24`
- Thumb-2: mixed 16-bit and 32-bit instructions
- STM32L100C: M3, 256 KB flash, 16 KB RAM, 4 KB EEPROM
- STM32F407VGT6: M4, 1 MB flash, 192 KB RAM, TRNG

- Most arithmetic instructions: 1 cycle
- Simple store to memory: 1 cycle
- Loads from memory: $\geq 2$ cycles
- 3-stage pipeline
- $n$ loads can be pipelined to take $n + 1$ cycles
Approaches to implementing AES

Traditional
SubBytes as lookup table. Slow and cache attacks, but small.

T-tables
Combine SubBytes, ShiftRows, MixColumns in large table. Fast, but cache attacks.

Vector permute
As in [Ham09], but not applicable on this platform.

Bitslicing
Or byteslicing for AES. Spread bytes of state over 8 registers. Process multiple blocks in parallel for high throughput.
Our contribution

- Fastest T-table-based AES-{128,192,256}-CTR
- Fastest bytesliced AES-128-CTR
  - Exactly the same cycle count for random inputs, keys, nonces
- Fastest masked bytesliced AES-128-CTR
  - Exactly the same cycle count for random inputs, keys, nonces
- ARM-specific instruction scheduler and register allocator
- All software in public domain
## Making software fast – flash wait states

### Table 11. Number of wait states according to CPU clock (HCLK) frequency
(STM32F42xxx and STM32F43xxx)

<table>
<thead>
<tr>
<th>Wait states (WS) (LATENCY)</th>
<th>Voltage range 2.7 V - 3.6 V</th>
<th>Voltage range 2.4 V - 2.7 V</th>
<th>Voltage range 2.1 V - 2.4 V</th>
<th>Voltage range 1.8 V - 2.1 V Prefetch OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 WS (1 CPU cycle)</td>
<td>0 &lt; HCLK ≤ 30</td>
<td>0 &lt; HCLK ≤ 24</td>
<td>0 &lt; HCLK ≤ 22</td>
<td>0 &lt; HCLK ≤ 20</td>
</tr>
<tr>
<td>1 WS (2 CPU cycles)</td>
<td>30 &lt; HCLK ≤ 60</td>
<td>24 &lt; HCLK ≤ 48</td>
<td>22 &lt; HCLK ≤ 44</td>
<td>20 &lt; HCLK ≤ 40</td>
</tr>
<tr>
<td>2 WS (3 CPU cycles)</td>
<td>60 &lt; HCLK ≤ 90</td>
<td>48 &lt; HCLK ≤ 72</td>
<td>44 &lt; HCLK ≤ 66</td>
<td>40 &lt; HCLK ≤ 60</td>
</tr>
<tr>
<td>3 WS (4 CPU cycles)</td>
<td>90 &lt; HCLK ≤ 120</td>
<td>72 &lt; HCLK ≤ 96</td>
<td>66 &lt; HCLK ≤ 88</td>
<td>60 &lt; HCLK ≤ 80</td>
</tr>
<tr>
<td>4 WS (5 CPU cycles)</td>
<td>120 &lt; HCLK ≤ 150</td>
<td>96 &lt; HCLK ≤ 120</td>
<td>88 &lt; HCLK ≤ 110</td>
<td>80 &lt; HCLK ≤ 100</td>
</tr>
<tr>
<td>5 WS (6 CPU cycles)</td>
<td>150 &lt; HCLK ≤ 180</td>
<td>120 &lt; HCLK ≤ 144</td>
<td>110 &lt; HCLK ≤ 132</td>
<td>100 &lt; HCLK ≤ 120</td>
</tr>
<tr>
<td>6 WS (7 CPU cycles)</td>
<td></td>
<td>144 &lt; HCLK ≤ 168</td>
<td>132 &lt; HCLK ≤ 154</td>
<td>120 &lt; HCLK ≤ 140</td>
</tr>
<tr>
<td>7 WS (8 CPU cycles)</td>
<td></td>
<td>168 &lt; HCLK ≤ 180</td>
<td>154 &lt; HCLK ≤ 176</td>
<td>140 &lt; HCLK ≤ 160</td>
</tr>
<tr>
<td>8 WS (9 CPU cycles)</td>
<td></td>
<td></td>
<td>176 &lt; HCLK ≤ 180</td>
<td>160 &lt; HCLK ≤ 168</td>
</tr>
</tbody>
</table>
Making software fast – RAM

Figure 1. System architecture for STM32F405xx/07xx and STM32F415xx/17xx devices
Making software fast – alignment

- Only load/store full words at word-aligned locations
- Only branch to word-aligned destination
- Word-align instructions for instruction fetcher:

```
08003058 <somefunction>:
  8003058: ea80 6030           eor.w  r0, r0, r0, ror #24
  800305c: 4408                 add    r0, r1
  800305e: ea81 51f1           eor.w  r1, r1, r1, ror #23
  8003062: ea80 51b1           eor.w  r1, r0, r1, ror #22
  8003066: ea81 5070           eor.w  r0, r1, r0, ror #21
  800306a: 4770                 bx     lr
```
Making software fast – alignment

- Only load/store full words at word-aligned locations
- Only branch to word-aligned destination
- Word-align instructions for instruction fetcher:

08003058 <somefunction>:

8003058: ea80 6030 eor.w r0, r0, r0, ror #24
800305c: 4408 add r0, r1
800305c: eb00 0001 add.w r0, r0, r1
8003060: ea81 51f1 eor.w r1, r1, r1, ror #23
8003064: ea80 51b1 eor.w r1, r0, r1, ror #22
8003068: ea81 5070 eor.w r0, r1, r0, ror #21
800306c: 4770 bx lr
Making software fast – more tricks

- Pipeline loads
  
  ldr r0, [r1]; ldr r1, [r2]; add r0, #1

  instead of

  ldr r0, [r1]; add r0, #1; ldr r1, [r2]

- Caches
- Prefetch buffers
- Data locality: adr instead of ldr
Making AES software fast – T-tables

See [BS08]

Combined shift-and-mask
Scaled-index loads
Second-byte instructions
Padded registers
32-bit shift of padded registers
Byte loads
Two-byte loads

Masked tables
Combined mask-and-insert
Combined load-and-xor
Byte extraction via loads
Round-key recomputation
Round-key caching
Counter-mode caching
Making AES software fast – T-tables

See [BS08]

Combined shift-and-mask
Scaled-index loads
Second-byte instructions
Padded registers
32-bit shift of padded registers
Byte loads
Two-byte loads

Masked tables
Combined mask-and-insert
Combined load-and-xor
Byte extraction via loads
Round-key recomputation
Round-key caching
Counter-mode caching
## Making AES software fast – results T-table

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speed (cycles)</th>
<th>ROM (bytes)</th>
<th>RAM (bytes)</th>
<th>I/O</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M3</td>
<td>M4</td>
<td>Code</td>
<td>Data</td>
<td>I/O</td>
</tr>
<tr>
<td>AES-128 KS</td>
<td>276.9</td>
<td>284.9</td>
<td>862</td>
<td>1024</td>
<td>176</td>
</tr>
<tr>
<td>AES-128</td>
<td><strong>639.5</strong></td>
<td><strong>644.7</strong></td>
<td><strong>1970</strong></td>
<td>1024</td>
<td><strong>176 + 2m</strong></td>
</tr>
<tr>
<td>AES-128-CTR</td>
<td><strong>531.2</strong></td>
<td>537.5</td>
<td>2128</td>
<td>1024</td>
<td><strong>192 + 2m</strong></td>
</tr>
<tr>
<td>AES-192 KS</td>
<td>258.8</td>
<td>264.2</td>
<td>778</td>
<td>1024</td>
<td>208</td>
</tr>
<tr>
<td>AES-192-CTR</td>
<td>649.1</td>
<td>656.0</td>
<td>2512</td>
<td>1024</td>
<td><strong>224 + 2m</strong></td>
</tr>
<tr>
<td>AES-256 KS</td>
<td>353.8</td>
<td>357.9</td>
<td>1114</td>
<td>1024</td>
<td>240</td>
</tr>
<tr>
<td>AES-256-CTR</td>
<td>767.9</td>
<td>774.6</td>
<td>2896</td>
<td>1024</td>
<td><strong>256 + 2m</strong></td>
</tr>
<tr>
<td>AES-128</td>
<td>1463</td>
<td>1816</td>
<td><strong>AES_128_128_V06</strong></td>
<td><strong>in FELICS</strong></td>
<td></td>
</tr>
<tr>
<td>AES-128-ECB</td>
<td>1066.7</td>
<td>4179.1</td>
<td><strong>SharkSSL</strong></td>
<td><strong>NXP AN11241</strong></td>
<td></td>
</tr>
<tr>
<td>AES-128-CTR</td>
<td>1247.4</td>
<td>32</td>
<td><strong>mbed TLS v2.3.0</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Protecting against cache attacks

- Byteslice, process 2 blocks in parallel
- Conversion: 48 1-cycle instructions
- ShiftRows: 104 1-cycle instructions
- MixColumns: 27 1-cycle instructions
- SubBytes: ?
Bitsliced SubBytes

- Smallest S-box: 113 gates [BP10]

\[
\begin{align*}
  y_{14} &= U_3 + U_5 \\
  y_{13} &= U_0 + U_6 \\
  y_9 &= U_0 + U_3 \\
  y_8 &= U_0 + U_5 \\
  t_0 &= U_1 + U_2 \\
  y_1 &= t_0 + U_7 \\
  y_4 &= y_1 + U_3 \\
  y_{12} &= y_{13} + y_{14} \\
  y_2 &= y_1 + U_0 \\
  y_5 &= y_1 + U_6 \\
  y_3 &= y_5 + y_8 \\
  t_1 &= U_4 + y_{12} \\
  y_{15} &= t_1 + U_5 \\
  y_{20} &= t_1 + U_1 \\
  y_6 &= y_{15} + U_7 \\
  y_{10} &= y_{15} + t_0 \\
  y_{11} &= y_{20} + y_9 \\
  y_{17} &= y_{10} + y_{11} \\
  y_{19} &= y_{10} + y_8 \\
  y_7 &= U_7 + y_{11} \\
  y_{16} &= t_0 + y_{11} \\
  y_{21} &= y_{13} + y_{16} \\
  y_{18} &= U_0 + y_{16} \\
  t_2 &= y_{12} \times y_{15} \\
  t_3 &= y_3 \times y_6 \\
  t_4 &= t_3 + t_2 \\
  t_5 &= y_4 \times U_7 \\
  t_6 &= t_5 + t_2 \\
  t_7 &= y_{13} \times y_{16} \\
  t_8 &= y_5 \times y_1 \\
  t_9 &= t_8 + t_7 \\
  t_{10} &= y_2 \times y_7 \\
  t_{11} &= t_{10} + t_7 \\
  t_{12} &= y_9 \times y_{11} \\
  t_{13} &= y_{14} \times y_{17} \\
  t_{14} &= t_{13} + t_{12} \\
  t_{15} &= y_8 \times y_{10} \\
  t_{16} &= t_{15} + t_{12} \\
  t_{17} &= t_4 + y_{20} \\
  t_{18} &= t_6 + t_{16} \\
  t_{19} &= t_9 + t_{14} \\
  t_{20} &= t_{11} + t_{16} \\
  t_{21} &= t_{17} + t_{14} \\
  t_{22} &= t_{18} + y_{19} \\
  t_{23} &= t_{19} + y_{21} \\
  t_{24} &= t_{20} + y_{18} \\
  t_{25} &= t_{21} + t_{22} \\
  (\ldots) 
\end{align*}
\]
Why compilers are not ideal

- Compilers aim to produce fast binaries on average
- Compilers aim to run reasonably fast on large code bases
- Compilers only do one attempt
- Compilers are complicated
- So do it yourself!
Our scheduler and register allocator

- Focus only on ARM’s three-operand instructions
- Multiple strategies implemented, designed to ‘play round’
- Nondeterministic due to hash randomization
- First reschedule, decrease the length of live ranges
  - Push down based on left-hand side
  - Push up based on right-hand side
- Then allocate greedily, keep output in registers
- If registers are full
  - Free register with expired variable
  - Otherwise, free register with longest distance until reuse
- Detect direct recomputation, can be cheaper than loading from memory
Comparison

- Results for 113-instruction S-box

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loads</strong></td>
<td>46</td>
<td>32</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td><strong>Stores</strong></td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

- GCC 6.2, Clang 3.8.1, ARM Compiler 5.06, ‘best’ sets of flags
- Other compilers also insert arithmetic and move instructions
Protecting against cache attacks

- Byteslice, process 2 blocks in parallel
- Conversion: 48 1-cycle instructions
- ShiftRows: 104 1-cycle instructions
- MixColumns: 27 1-cycle instructions
- SubBytes: 113-gate S-box
  Custom scheduler: $113 + 16 \text{ ldr} + 16 \text{ str}$
- Slowdown: $\times 2.9$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speed (cycles)</th>
<th>ROM (bytes)</th>
<th>RAM (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M3</td>
<td>M4</td>
<td>Code</td>
</tr>
<tr>
<td>AES-128 KS</td>
<td>1027.8</td>
<td>1033.8</td>
<td>3434</td>
</tr>
<tr>
<td>AES-128-CTR</td>
<td>1616.6</td>
<td>1617.6</td>
<td>12120</td>
</tr>
</tbody>
</table>
Protecting against 10 SCA

- Boolean masking with Trichina gate [Tri03]
  - Compute $a \cdot b$ with $\bar{a} = (a \oplus r_a)$, $\bar{b} = (b \oplus r_b)$, $r_a$, $r_b$, $r$ masks:
    $$(a \cdot b) \oplus r = ((\bar{a} \cdot \bar{b}) \oplus ((r_a \cdot \bar{b}) \oplus ((r_a \cdot r_b) \oplus r))) \oplus (r_b \cdot \bar{a})$$
  - $1 \text{ and} \Rightarrow 4 \text{ eor}, 4 \text{ and}, 1 \text{ ldr}$
  - $1 \text{ eor} \Rightarrow 2 \text{ eor}$

- Generate 328 random words with hardware RNG
- Double active data set does not fit anymore
  - Operate on one share throughout linear layer
  - Swap to other share as late as possible
- Need both shares for SubBytes, use our tool to minimize overhead

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>330</td>
<td>185</td>
<td>332</td>
<td>135</td>
</tr>
<tr>
<td>Stores</td>
<td>126</td>
<td>145</td>
<td>132</td>
<td>99</td>
</tr>
</tbody>
</table>

- (Excluding 32 loads for randomness)
Protecting against 1O SCA – results

- Slowdown: $\times 4.6$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speed (cycles)</th>
<th>ROM (bytes)</th>
<th>RAM (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M3</td>
<td>M4</td>
<td>Code</td>
</tr>
<tr>
<td>AES-128 KS</td>
<td>1027.8</td>
<td>1033.8</td>
<td>3434</td>
</tr>
<tr>
<td>AES-128-CTR</td>
<td>N/A</td>
<td>7422.6</td>
<td>39916</td>
</tr>
</tbody>
</table>

- Generating and storing random words: 2132.5 cycles
- All the rest: 5290.1 cycles
- Experimental validation should be performed before trusting that this implementation is really secure
Thanks...  

...for your attention  

Paper and code at  
https://ko.stoffelen.nl/  
https://cryptojedi.org/
References I

Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede.
DPA, bitslicing and masking at 1 GHz.

Joan Boyar and René Peralta.
A new combinational logic minimization technique with applications to cryptology.

Daniel J. Bernstein and Peter Schwabe.
New AES software speed records.
http://cryptojedi.org/users/peter/#aesspeed.
References II

Mike Hamburg.
Accelerating AES with vector permute instructions.

Robert Könighofer.
A fast and cache-timing resistant implementation of the AES.

Emilia Käsper and Peter Schwabe.
Faster and timing-attack resistant AES-GCM.
https://cryptojedi.org/papers/#aesbs.

Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright.
Fast software AES encryption.
References III

ARM Holdings plc.

Elena Trichina.
Combinational logic design for AES SubByte transformation on masked data.