AES on the ARM Cortex-M3 and M4

Peter Schwabe and Ko Stoffelen

iCIS | Digital Security
Radboud University

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: ≥ 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

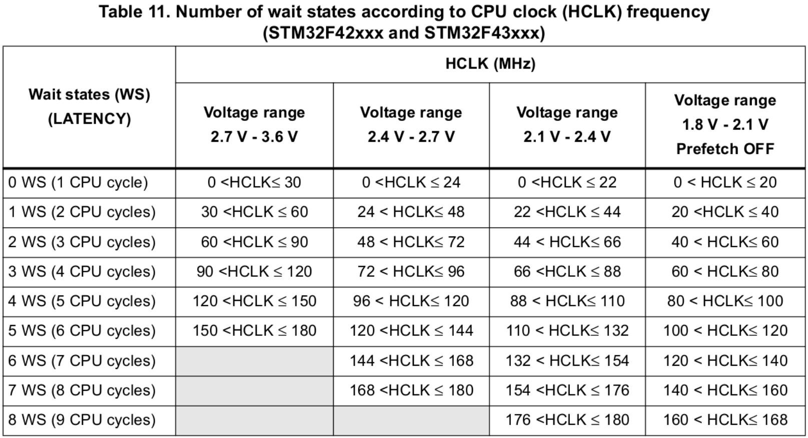

Table 11. Number of wait states according to CPU clock (HCLK) frequency (STM32F42xxx and STM32F43xxx)

Wait states (WS) (LATENCY)	HCLK (MHz)			
	Voltage range 2.7 V - 3.6 V	Voltage range 2.4 V - 2.7 V	Voltage range 2.1 V - 2.4 V	Voltage range 1.8 V - 2.1 V Prefetch OFF
0 WS (1 CPU cycle)	0 <HCLK≤ 30	0 <HCLK ≤ 24	0 <HCLK ≤ 22	0 < HCLK ≤ 20
1 WS (2 CPU cycles)	30 <HCLK ≤ 60	24 < HCLK≤ 48	22 <HCLK ≤ 44	20 <HCLK ≤ 40
2 WS (3 CPU cycles)	60 <HCLK ≤ 90	48 < HCLK≤ 72	44 < HCLK≤ 66	40 < HCLK≤ 60
3 WS (4 CPU cycles)	90 <HCLK ≤ 120	72 < HCLK≤ 96	66 <HCLK ≤ 88	60 < HCLK≤ 80
4 WS (5 CPU cycles)	120 <HCLK ≤ 150	96 < HCLK≤ 120	88 < HCLK≤ 110	80 < HCLK≤ 100
5 WS (6 CPU cycles)	150 <HCLK ≤ 180	120 <HCLK ≤ 144	110 < HCLK≤ 132	100 < HCLK≤ 120
6 WS (7 CPU cycles)		144 <HCLK ≤ 168	132 < HCLK≤ 154	120 < HCLK≤ 140
7 WS (8 CPU cycles)		168 <HCLK ≤ 180	154 <HCLK ≤ 176	140 < HCLK≤ 160
8 WS (9 CPU cycles)			176 <HCLK ≤ 180	160 < HCLK≤ 168

Making software fast – RAM

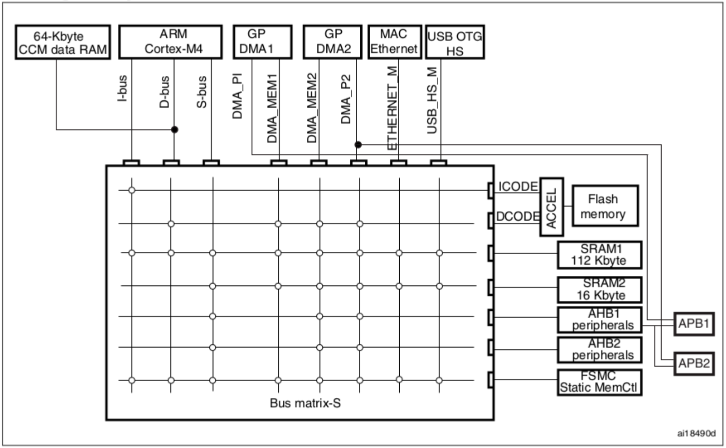

Figure 1. System architecture for STM32F405xx/07xx and STM32F415xx/17xx devices

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

Algorithm	**Speed (cycles)**		**ROM (bytes)**		**RAM (bytes)**	
	M3	**M4**	**Code**	**Data**	**I/O**	**Stack**
AES-128 KS	276.9	284.9	862	1024	176	32
AES-128	**639.5**	644.7	1970	1024	$176 + 2m$	44
AES-128-CTR	**531.2**	537.5	2128	1024	$192 + 2m$	72
AES-192 KS	258.8	264.2	778	1024	208	32
AES-192-CTR	649.1	656.0	2512	1024	$224 + 2m$	72
AES-256 KS	353.8	357.9	1114	1024	240	32
AES-256-CTR	767.9	774.6	2896	1024	$256 + 2m$	72

AES-128	1463	Cryptovia
AES-128	1816	`AES_128_128_V06` in FELICS
AES-128-ECB	1066.7	SharkSSL
AES-128-ECB	4179.1	NXP AN11241
AES-128-CTR	1247.4	mbed TLS v2.3.0

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]

```
y14 =  U3 +  U5
y13 =  U0 +  U6
 y9 =  U0 +  U3
 y8 =  U0 +  U5
 t0 =  U1 +  U2
 y1 =  t0 +  U7
 y4 =  y1 +  U3
y12 = y13 + y14
 y2 =  y1 +  U0
 y5 =  y1 +  U6
 y3 =  y5 +  y8
 t1 =  U4 + y12
y15 =  t1 +  U5
y20 =  t1 +  U1
 y6 = y15 +  U7
y10 = y15 +  t0
```

```
y11 = y20 +  y9
 y7 =  U7 + y11
y17 = y10 + y11
y19 = y10 +  y8
y16 =  t0 + y11
y21 = y13 + y16
y18 =  U0 + y16
 t2 = y12 x y15
 t3 =  y3 x  y6
 t4 =  t3 +  t2
 t5 =  y4 x  U7
 t6 =  t5 +  t2
 t7 = y13 x y16
 t8 =  y5 x  y1
 t9 =  t8 +  t7
t10 =  y2 x  y7
```

```
t11 = t10 +  t7
t12 =  y9 x y11
t13 = y14 x y17
t14 = t13 + t12
t15 =  y8 x y10
t16 = t15 + t12
t17 =  t4 + y20
t18 =  t6 + t16
t19 =  t9 + t14
t20 = t11 + t16
t21 = t17 + t14
t22 = t18 + y19
t23 = t19 + y21
t24 = t20 + y18
t25 = t21 + t22
(...)
```

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

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

- 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 `ldr` + 16 `str`
- Slowdown: ×2.9

Algorithm	**Speed (cycles)**		**ROM (bytes)**		**RAM (bytes)**	
	M3	**M4**	**Code**	**Data**	**I/O**	**Stack**
AES-128 KS	1027.8	1033.8	3434	1036	368	188
AES-128-CTR	1616.6	1617.6	12120	12	$368 + 2m$	108

Protecting against 1O 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 `and` ⇒ 4 `eor`, 4 `and`, 1 `ldr`
  - 1 `eor` ⇒ 2 `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

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	330	185	332	135
Stores	126	145	132	99

- (Excluding 32 loads for randomness)

Protecting against 1O SCA – results

- Slowdown: ×4.6

Algorithm	**Speed (cycles)**		**ROM (bytes)**		**RAM (bytes)**	
	M3	**M4**	**Code**	**Data**	**I/O**	**Stack**
AES-128 KS	1027.8	1033.8	3434	1036	368	188
AES-128-CTR	N/A	7422.6	39916	12	$368 + 2m$	1588

- 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.
In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *LNCS*, pages 599–619. Springer, 2015.

Joan Boyar and René Peralta.
A new combinational logic minimization technique with applications to cryptology.
In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010.
`http://eprint.iacr.org/2009/191/`.

Daniel J. Bernstein and Peter Schwabe.
New AES software speed records.
In Dipanwita Roy Chowdhury and Vincent Rijmen, editors, *Progress in Cryptology – INDOCRYPT 2008*, volume 5365 of *LNCS*, pages 322–336. Springer, 2008.
`http://cryptojedi.org/users/peter/#aesspeed`.

References II

Mike Hamburg.
Accelerating AES with vector permute instructions.
In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *LNCS*, pages 18–32. Springer, 2009.
`http://mikehamburg.com/papers/vector_aes/vector_aes.pdf`.

Robert Könighofer.
A fast and cache-timing resistant implementation of the AES.
In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *LNCS*, pages 187–202. Springer, 2008.

Emilia Käsper and Peter Schwabe.
Faster and timing-attack resistant AES-GCM.
In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009.
`https://cryptojedi.org/papers/#aesbs`.

Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright.
Fast software AES encryption.
In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption*, volume 6147 of *LNCS*, pages 75–93. Springer, 2010.

References III

ARM Holdings plc.
ARM's Cortex-M and Cortex-R embedded processors, 2015.
http://www.arm.com/zh/files/event/2_2015_ARM_Embedded_Seminar_Ian_Johnson.pdf.

Elena Trichina.
Combinational logic design for AES SubByte transformation on masked data.
Cryptology ePrint Archive, Report 2003/236, 2003.
http://eprint.iacr.org/2003/236/.