All the AES You Need on Cortex-M3 and M4

Peter Schwabe and Ko Stoffelen

Radboud University, Digital Security Group,
Nijmegen, The Netherlands
peter@cryptojedi.org, k.stoffelen@cs.ru.nl

Abstract. This paper describes highly-optimized AES-{128, 192, 256}-CTR assembly implementations for the popular ARM Cortex-M3 and M4 embedded microprocessors. These implementations are about twice as fast as existing implementations. Additionally, we provide the fastest bitsliced constant-time and masked implementations of AES-128-CTR to protect against timing attacks, power analysis and other (first-order) side-channel attacks. All implementations, including an architecture-specific instruction scheduler and register allocator, which we use to minimize expensive loads, are released into the public domain.

Keywords: AES, software implementation, ARM Cortex-M, constant-time, bitslicing, masking

1 Introduction

AES was published as Rijndael in 1998 and standardized in FIPS PUB 197 in 2001. Highly optimized implementations have been written for most common architectures, ranging from 8-bit AVR microcontrollers to x86-64 and NVIDIA GPUs. See, for example, [4,17,24]. Implementing optimized AES on any of these architectures essentially requires to start from scratch to find out which implementation approach is going to be the most efficient. The past decades have seen a large shift toward ARM architectures and while we have seen efficient AES implementations for high-end processors used in modern smartphones [5] and for older microprocessors used in smart cards [1,6], there is little to choose from for modern low-end embedded devices and Internet of Things applications.

Sometimes an embedded device contains a coprocessor that can perform AES encryption in hardware, but such a coprocessor is not always available. It makes a device more expensive and it can increase the power consumption of a device. Simply compiling an existing implementation written in, for example, the C programming language, is unlikely to produce optimal performance. Even worse, embedded systems are typical targets for timing attacks, power analysis attacks,

* This work was supported by the European Commission through the Horizon 2020 program under project number ICT-645622 (PQCRYPTO) and by Netherlands Organization for Scientific Research (NWO) through Veni 2013 project 13114. Permanent ID of this document: 9fc0b970660e40c264e50ca389dacd49. Date: October 19, 2016

and other forms of side-channel attacks, so software for those devices typically needs to include adequate protection against such attacks.

We fill these gaps by providing highly optimized AES software implementations for two of the most popular modern microprocessors for constrained embedded devices, the ARM Cortex-M3 and the Cortex-M4. Our implementations of AES-$\{128, 192, 256\}$-CTR are more than twice as fast as existing implementations. We also provide a single-block AES-128 implementation, a constant-time AES-128-CTR implementation and a masked implementation that is secure against first-order power analysis attacks. All of them are the fastest of their kind. They are put into the public domain and available at `https://github.com/Ko-/aes-armcortexm`.

The results of this paper are not only interesting for "stand-alone" AES encryption. In the ongoing CAESAR competition for authenticated encryption schemes, 14 out of the 29 remaining second round candidates are based on AES or the AES round function. Our implementations will be helpful to speed up those candidates on embedded ARM microcontrollers.

Organization of the paper. In Section 2, we will first discuss AES and give an outline of the different implementation approaches. We will also provide an overview of the target architecture and what features we can benefit from when optimizing software for speed. Section 3 then discusses our fastest AES implementations, based on the T-tables approach, while Section 4 and 5 consider our constant-time bitsliced and our masked implementation, respectively. We report performance benchmarks and provide a comparison to related work at the end of each of the Sections 3–5.

2 Preliminaries

2.1 Implementing AES

AES is a substitution-permutation network that operates on 128-bit blocks. Key sizes of 128, 192, and 256 bits are supported. Depending on the key size, the network has 10, 12, or 14 rounds, respectively. The nonlinear substitution layer consists of the SubBytes step, where an 8-bit S-box is applied to each byte of the state. The linear permutation layer consists of ShiftRows and MixColumns, to provide diffusion. In the beginning, between all rounds, and at the end, the AddRoundKey step xors the state with round keys that are derived from the main key during a key schedule. MixColumns is omitted in the final round [12]. In software, there are four main implementation approaches:

Traditional: All steps are implemented "as is"; typically SubBytes is implemented through a 256-byte lookup table.

T-tables: SubBytes, ShiftRows, and MixColumns are combined in 4 1024-byte lookup tables. Each AES round then consists of 16 masks, 16 loads from the lookup tables and 4 loads from the round keys, and 16 `XOR`s. This leads to very efficient implementations on platforms with a word size of at least

32 bits. At the cost of extra rotations, only 1 lookup table is required. This strategy was already suggested in the original Rijndael proposal [11]. Our fastest implementations in Section 3 are based on this approach.

Vector permute: The disadvantage of the T-tables approach is that key- and data-dependent lookups open the door for timing attacks on architectures with cache. See, for example, [3,25,33]. Another approach to implementing AES, which avoids such data-dependent lookups, uses vector-permute instructions [15]. However, such instructions are unavailable on our target platform, which is why we do not go into more detail on this strategy.

Bitslicing: Another approach that does not require lookup tables is bitslicing, originally introduced for DES by Biham [7]. The core idea is that data is split over multiple registers, but that other blocks are used to fill the registers. Multiple blocks can then be processed in parallel in a SIMD fashion. This approach is especially beneficial for architectures with large registers. For AES, the 128-bit state is usually bytesliced over 8 registers, as this allows for an efficient linear layer. Various papers describe bitsliced implementations of AES on Intel processors [19,20,21]; the most recent one by Käsper and Schwabe from 2009 is still the software speed-record holder [17]. Our implementations in Sections 4 and 5 are also using bitslicing.

2.2 ARM Cortex-M

The Cortex-M is a family of 32-bit processors by ARM meant for use in embedded microcontrollers. They are designed to be cheap and to be energy efficient, while still being powerful enough to offer adequate performance in applications such as automotive systems, medical instruments, the Internet of Things, or other consumer products. As of 2015, over 10 billion of these processors have been shipped [27].

The Cortex-M3 was announced in 2004, while the Cortex-M4 is a more recent successor from 2010. Both microprocessors have 16 32-bit registers, of which three are reserved for program counter, stack pointer, and link register. The link pointer can be pushed to the stack to free another register. Both microprocessors support the ARMv7-M architecture and the Thumb-2 technology, but the Cortex-M4 supports additional instructions for digital signal processing, i.e., the ARMv7E-M architecture. However, we do not use these extensions.

Bitwise and arithmetic instructions take one cycle on these architectures, except for divisions or writes to the program counter. Branches, loads, and stores may take more cycles, which is why they can easily bottleneck the performance. A distinguishing feature of the ARM architecture is the availability of barrel-shifting registers. This means that we can do arithmetic on rotated or shifted registers, without any additional cost for the rotation or shift.

We used the STM32L100C and STM32F407VG development boards. The first comes with 256 KB of flash memory, 16 KB of RAM, and 4 KB of EEPROM. It can run a Cortex-M3 core at up to 32 MHz. The second is more powerful and has a 168 MHz Cortex-M4 core, 1024 KB of flash memory, 192 KB of RAM, and a true-random-number generator.

2.3 Accelerating memory access

Memory access can be expensive in terms of CPU cycles. Additionally, there are a lot of ways to introduce penalty cycles. Carefully optimized software therefore avoids as many potential delays as possible. Here we list a number of generic strategies related to memory access to reduce the cycle count of programs running on the Cortex-M3 and M4. A siginifcant portion of our speedups of AES stem from a combination of these strategies.

Flash. The instructions and tables are typically stored in flash memory. Accessing flash can introduce a number of wait states, depending on the relative clock frequency of the microprocessor core and the memory chip. For our development boards, the STM32L100C and STM32F407VG, STMicroelectronics describes in its documentation when it is possible to have zero wait states [29, p. 59, tbl. 13][30, p. 80, tbl. 10]. For example, on the STM32L100C, the CPU clock can only run at 16 MHz for a supply voltage of 3.3 V. To be able to compare the performance of implementations across different devices or boards, it is important to be in this scenario.

RAM. Something similar holds for accessing RAM, where the stack is stored. On the STM32F407VG, four different regions of RAM are available: `SRAM1`, `SRAM2`, `SRAM3`, and `CCM`. In our case it turned out to be faster to use the core coupled memory (CCM), as it uses the D-bus directly.

Alignment. The Cortex-M3 and M4 support Thumb-2 technology, which means that 16-bit and 32-bit encodings of instructions can freely be mixed. However, consider the case that a 16-bit instruction starts at a word-aligned address, followed by one or more 32-bit instructions. The 32-bit instructions are then no longer word-aligned, which may cause penalty cycles for the instruction fetcher, which fetches multiple instructions at a time. In this case, forcing the use of a 32-bit encoding for the first instruction by adding a `.w` suffix can improve the instruction alignment and reduce the cycle count. Our implementations take this into consideration. Penalty cycles may also be introduced when branching to addresses that are not word-aligned, when loading from memory at addresses that are not word-aligned or when not loading full words from memory. Implementers needs to take care of the alignment themselves. Our implementations carefully avoid these penalty cycles.

Pipelining loads. Most `str` instructions take 1 cycle, because of the availability of a write buffer, but `ldr` instructions generally take at least 2 cycles. However, n `ldr` instructions can be pipelined together to be executed in $n + 1$ cycles if there are no address dependencies and the program counter remains untouched. An instruction such as `ldm` pipelines all of its loads together, but when it is followed by an `ldr`, those will not be pipelined together. For our implementations, we pipeline as many loads as possible.

Caches and prefetch buffers. The Cortex-M3 and M4 by themselves do not have any caches. However, caches can be added in embedded devices or development boards to boost the performance. For example, the STM32F407VG

contains 64 128-bit lines of instruction cache memory and 8 128-bit lines of data cache memory [30, p. 90]. It also contains an instruction prefetch buffer to reduce the experienced number of wait states when a microprocessor running at a high clock frequency accesses flash memory to fetch 128 bits of instructions [30, p. 82]. The STM32L100C supports a similar prefetch buffer when 64-bit flash access is enabled [29, p. 59].

Data location. When one wants to read data that is stored in the flash memory, one first needs to load the address of the data block before one can load the data itself. However, when data is located within 4096 bytes of the value of the program counter, the first load instruction can be replaced by an `adr` pseudo-instruction that is really an addition or subtraction of the program counter, which may save one cycle, depending on whether the load could be pipelined. It is therefore useful to store data close to where the data is being used.

3 Making AES fast

Ever since Rijndael was standardized as AES, a lot of effort has been put into making fast and secure software implementations for a large range of platforms and architectures. Numerous optimization tricks have been suggested to improve the performance. For T-table-based implementations, the majority is summarized in [4]. In this section we discuss which strategies are useful to apply on the Cortex-M3 and M4.

Using the T-table-based approach, AES-128-CTR can typically be implemented in 720 instructions: 208 loads, 4 stores, 160 shifts, 176 masks, 168 xors and 4 others [4]. Thanks to ARM's barrel-shifting registers, we can do combined shifts and masks, saving 160 instructions. [4] also mentions *scaled-index loads* and *second-byte instructions*. A scaled-index load is the option to shift the offset of a load instruction for free, while a second-byte instruction allows for extracting the second byte of a register in one instruction. Both features are supported by our architecture, but as all shifts are already fully subsumed, these optimizations no longer yield any additional advantage.

Byte loads and *two-byte loads* could save another 8 instructions by not requiring an additional mask, but loads that are not word-aligned cause a penalty cycle, so for speed these optimizations are of little use. Other potential optimization strategies, such as combining *masks and inserts* or *loads and xors*, are not possible in a single instruction on these platforms. Being able to do *byte extraction via loads* allows to exchange arithmetic instructions for load instructions, but loads are either as fast or slower, so this strategy gives no advantage either.

With *round-key recomputation*, only one out of four round-key words is stored for all rounds except the first. During encryption, the other parts of the round keys can be recomputed on the fly, exchanging 30 loads for 30 xors. However, in our case the loads can be fully pipelined and the round keys from the previous round would not fit into registers anymore, so this would also not reduce the total number of cycles. *Round-key caching*, where all round keys are kept in registers

when encrypting multiple blocks, would require even more registers. Another technique called *padded registers* exists, where a 32-bit value is stored in a 64-bit register in such a way that combing shifts and masks can be done a bit more cleverly. However, our registers are too small to use anything like this.

However, *counter-mode caching* helps to save another 81 instructions in the main loop. In counter mode, for 256 consecutive blocks, only 1 byte of the input changes. This means that through the first and second AES round, computations that do not depend on this one byte can be cached and reused. Starting from the third round, everything will depend on all input bytes. While there is some additional overhead involved in storing and retrieving the cached values, this trick already leads to a speedup when only 2 blocks are processed.

3.1 Our implementations

Our implementations of AES-128 encryption, AES-128-CTR, AES-192-CTR, and AES-256-CTR use one 1024-byte lookup table. The extra rotates that this would normally cause come for free thanks to ARM's barrel shifting registers. Using four tables would save another 40 1-cycle instructions in the key schedule, and 16 1-cycle instructions in the final round for encryption, but as there is typically little memory available on microcontrollers and the improvement in speed is only marginal, we decided that this trade-off was not worth it. AES-128 decryption needs two 1024-byte lookup tables. On the other hand, the 16 mask instructions in the final round are no longer required.

Key expansion is performed separately, as the round keys can be reused for multiple blocks. In our implementation of counter mode, there is a 32-bit counter and a 96-bit nonce. The reason is that then we do not have to deal with a carry from the counter and a conditional `add` for the second counter word, which gives another small speedup. We consider a 32-bit counter, providing a maximum stream length of $2^{32} \cdot 16 = 68719476736$ bytes, to be large enough in a typical microcontroller environment.

The performance of our speed-optimized implementations is summarized in Table 1. All results are averages over 10000 runs with random keys, inputs, and, if applicable, nonces. For encryption in counter mode, the number of cycles reflects the average number of cycles per block when processing 256 blocks, or 4096 bytes. Loops are fully unrolled, so the code size can be reduced drastically with only a small performance penalty. Note that data in ROM is typically shared by key expansion and encryption/decryption, so it has to be in memory only once. Under RAM usage, I/O refers to the amount of RAM that is required to store the input and output for the functions, e.g., $192 + 2m$ means that we require 4 bytes for the counter, 12 for the nonce, 176 for all round keys, m for our m-byte input, and m for the m-byte output. Again, I/O data is typically shared by key expansion and encryption/decryption and the same stack space can be reused for the encryption/decryption function call. It turns out that the same code runs in slightly fewer cycles on the Cortex-M3, which is most likely caused by the different way that instructions are fetched.

Table 1. Performance of unprotected AES

Algorithm	Speed (cycles)		ROM (bytes)		RAM (bytes)	
	M3	M4	Code	Data	I/O	Stack
AES-128 key expansion encryption	289.8	294.8	902	1024	176	32
AES-128 key expansion decryption	1180.0	1174.6	3714	2048	176	176
AES-128 single block encryption	659.4	661.7	2034	1024	$176 + 2m$	44
AES-128 single block decryption	642.5	648.3	1974	2048	$176 + 2m$	44
AES-128-CTR	546.3	554.4	2192	1024	$192 + 2m$	72
AES-192 key expansion	264.9	272.2	810	1024	240	32
AES-192-CTR	663.2	673.0	2576	1024	$224 + 2m$	72
AES-256 key expansion	364.8	371.8	1166	1024	240	32
AES-256-CTR	786.9	791.7	2960	1024	$256 + 2m$	72

3.2 Comparison to existing implementations

There are few publicly available AES implementations optimized for the Cortex-M3 and M4:

- In the SharkSSL crypto library, a speed of 1066.7 cycles per block is claimed for AES-128-ECB on the Cortex-M3 [28]. CTR mode is unavailable.
- A company called Cryptovia sells an implementation that does AES-128 on a single block in 1463 cycles [10], also on the Cortex-M3.
- The latest version of mbed TLS [26], formerly known as PolarSSL, contains a table-based AES-128-CTR implementation that takes 1247.4 cycles per block on the M3, while AES-128 key expansion takes 41545 cycles[1].
- NXP hosts the AN11241 AES library [23], but its implementation is very slow. AES-128-ECB runs in 4179.1 cycles per block on the M3, while the AES-128 key expansion takes 1089 cycles[1].
- The fastest implementation currently listed by the FELICS benchmarking framework [13] encrypts a single block with AES-128 in 1816 cycles on a Cortex-M3. The fastest key scheduling takes 724 cycles[2].

We therefore claim that our CTR-mode implementations are about twice as fast as existing implementations. We also require fewer cycles than optimized implementations for older yet similar ARM architectures [1], even though in [1] heavy use is made of the fact that the full lookup tables fit in the data cache on a StrongARM-1110, which does not hold for our platforms.

[1] We used `gcc -O3 -funroll-loops -fno-schedule-insns` with GCC 6.1.1 for these benchmarks, the best set of compiler flags we could find, based on all sets that are tried in the SUPERCOP benchmarking framework.

[2] `AES_128_128_V06` in scenario 0 with `-Os` and with `-O3`, respectively.

3.3 Benchmarking with FELICS

The FELICS framework [13] has been proposed as an open system to benchmark the performance of implementations of lightweight cryptographic systems on three different microprocessors, one of them being the ARM Cortex-M3. Cycle counts and memory usage are measured for three usage scenarios. Scenario 0 deals with single-block encryption, where the round keys are stored in RAM. In scenario 1, 128 bytes are encrypted in CBC mode. In scenario 2, 128 bits are encrypted in CTR mode.

This choice of scenarios means that our implementation needs to be adapted to fit in the framework. In particular, counter-mode caching can no longer be used and needs to be removed, which impacts the performance. Furthermore, the decryption algorithm and decryption key expansion are now required as well in scenarios 0 and 1. But most importantly, the FELICS framework does not set the number of wait states, which means that a load from memory will cost more than 2 cycles and that reported cycle counts are biased toward implementations with less load instructions. This greatly slows down the overall performance of our implementation.

The framework reports 1641 cycles for our encryption in scenario 0 and 578 cycles for our key schedule. Although this is still faster than currently listed results, the margin is smaller. This also holds for scenarios 1 and 2.

4 Protecting against timing attacks

While the availability of caches allows for speedups on platforms with relatively slow memory, it also makes table-based AES implementations vulnerable to cache-timing attacks [3,18]. A popular technique for writing a constant-time AES implementation that is still reasonably fast, is by applying bitslicing. Of course, caches can be simply disabled when performing cryptographic operations, but this implementation also serves as a step toward the masked implementation.

Bitslicing is often explained as a technique where every bit of the state is stored in a separate register, such that we can do operations on the bits independently and such that we can process 32 blocks in parallel on 32-bit machines. However, in the case of AES this is not the fastest way to bitslice, as most operations are byte-oriented. Full bitslicing would also increase the amount of registers needed to store the state by a factor of 32. There are very few architectures that have enough registers to keep the bitsliced state in registers, so there would be a lot of overhead in storing and loading data to other types of memory.

Könighofer suggested in [19] to 'byteslice' and to process 4 blocks in parallel on an architecture with 64-bit registers. Käsper and Schwabe were able to process 8 blocks in parallel using 128-bit registers [17]. Unfortunately, the Cortex-M3 and M4 only have 32-bit registers, so we can only process 2 blocks in parallel while still retaining an efficient implementation of the linear layer.

4.1 Our implementation

After key expansion, the round keys are stored in their bitsliced representation. To transform to bitsliced representation, we require 12 SWAPMOVE operations [22].

```
SWAPMOVE(a,b,n,m) {
  t = ((a ≫ n) ⊕ b) & m
  b = b ⊕ t
  a = a ⊕ (t ≪ n)
}
```

Due to ARM's barrel shifter, we can implement SWAPMOVE in just 4 1-cycle instructions, which gives a transformation overhead of 48 cycles.

```
eor t, b, a, lsl #n
and t, m
eor b, t
eor a, a, t, lsr #n
```

During encryption, the AES state is first transformed to bitsliced representation. AddRoundKey is then again just a matter of xoring the bitsliced round keys with the bitsliced state.

For SubBytes, a lot of research has been done on an efficient hardware implementation of the AES S-box [9]. These results are also very useful for bitsliced software implementations. Boyar and Peralta found a circuit with only 115 gates [8], which was later improved to 113: 32 `AND` gates, 77 `XOR` gates, and 4 `XNOR` gates. This is the smallest known implementation, which is why we used it as a basis for our implementation. However, with only 14 available registers, it is impossible to implement it directly in 113 instructions. We need more instructions to deal with storing values on the stack or with recomputation of values. We wrote an ad hoc combined instruction scheduler and register allocator that is tailored to our microprocessors.

Scheduling. Both instruction scheduling and register allocation are hard problems, as is the combined problem. Compilers usually implement a graph coloring algorithm and/or linear-scan allocation. They aim to schedule well on average, but do not necessarily generate the most efficient assembly for a specific part of code.

Existing compilers do not provide a lot of options to play with different scheduling and allocation strategies, which is why we decided to write an ARM-specific instruction scheduler and register allocator. This allows us to focus on ARM's three-operand instructions and to try several approaches. We aim to minimize the number of loads and stores and the usage of the stack. We first reschedule instructions to reduce the size of the active data set, by pushing instructions down based on their left-hand side and by pushing instructions up based on their right-hand side. Then we allocate registers in a greedy fashion, where we insert loads and stores when necessary and try to leave the output in

registers. A more thorough overview of the tool is provided in [31], including a comparison against the compilers GCC, Clang, and the ARM Compiler.

Our tool is nondeterministic because of hash randomization in Python, so we try several scheduling strategies multiple times and only use the best result. With our scheduler we are able to compute the AES S-box in 145 instructions: the 113 original operations, 16 loads and 16 stores. It is unknown whether this is optimal.

ShiftRows on a bitsliced state can be computed very efficiently on modern Intel CPUs using 8 SSSE3 byte-shuffling instructions [17]. However, something like this is unavailable on the Cortex-M3 and M4. We use the `ubfx` and `uxtb` bitfield instructions, together with `eor` on shifted registers, to compute ShiftRows in $8 \cdot 13 = 104$ 1-cycle instructions. The code below performs ShiftRows on `r9`, while `r12` and `r5` are used as temporary registers.

```
uxtb.w r12, r9
ubfx r5, r9, #14, #2
eor r12, r12, r5, lsl #8
ubfx r5, r9, #8, #6
eor r12, r12, r5, lsl #10
ubfx r5, r9, #20, #4
eor r12, r12, r5, lsl #16
ubfx r5, r9, #16, #4
eor r12, r12, r5, lsl #20
ubfx r5, r9, #26, #6
eor r12, r12, r5, lsl #24
ubfx r5, r9, #24, #2
eor r9, r12, r5, lsl #30
```

In contrast, the barrel shifters allow us to compute MixColumns in just 27 `eor` instructions on shifted registers, which is even more efficient than in [17].

To update the counter for the next blocks, one can either store the bitsliced representation and operate on this, or one can use the original representation and transform this to bitsliced representation every two blocks. While the first may appear to be faster, we implemented both and it turned out that the latter is in fact more efficient. This is due to overhead caused by the limited way in which you can do conditional execution with `IT`-blocks on these microprocessors.

Table 2 contains performance benchmarks of our implementation. Again, speed is measured as the average number of cycles per block when encrypting 256 consecutive blocks, which explains the decimal for the encryption. The amount of cycles is exactly equal for all 10000 combinations of random nonces, keys, and inputs that we tried. We see a slowdown of roughly a factor 2.9 compared to our previous implementation. Note, however, that when one can disable the caches during the AES execution or when caches are not available at all, our previous faster implementations are also constant-time and should be favored. We verified that after disabling caches, the cycle counts are exactly equal for

random combinations of inputs and keys. There is little related work that would make a fair comparison.

Table 2. Performance of constant-time AES

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

5 Protecting against side-channel attacks

Microprocessors are typical targets for side-channel attacks such as differential power analysis or differential electromagnetic analysis. A well-known countermeasure against first-order side-channel attacks that is used in practice is by Boolean masking, where a secret intermediate value a is split into two statistically independent shares, i.e., r_a and $\bar{a} = (a \oplus r_a)$, where r_a is called a random mask. Linear operations can be computed on both shares independently. After a linear operation, the shares can be xored together to unmask the result. Nonlinear operations are more difficult to mask securely. Trichina suggested the following provably secure method to mask $a \cdot b$ [32], where $\bar{a} = (a \oplus r_a)$, $\bar{b} = (b \oplus r_b)$, and r_a, r_b, r are random masks:

$$((\bar{a} \cdot \bar{b}) \oplus ((r_a \cdot \bar{b}) \oplus ((r_a \cdot r_b) \oplus r))) \oplus (r_b \cdot \bar{a}).$$

This means that every `AND` operation requires 4 `AND` operations, 4 `XOR` operations, and 1 load (of r) to mask.

We added first-order Boolean masking using Trichina gates to our constant-time bitsliced implementations to find out how much this additional security would cost on common microprocessors.

5.1 Our implementation

To generate the masks, we need a source of randomness. The STM32F407VG contains a random number generator (RNG) that guarantees a new 32-bit random word every 40 periods of the RNG clock. In the case of AES, 8 random words are required to mask the input, as two blocks are processed in parallel, and 320 random words are required for a single encryption, as SubBytes contains 32 `AND` operations and is executed in all 10 rounds. While interleaving randomness generation and executing instructions can decrease the waiting time, the performance of the

implementation will greatly depend on the performance of the RNG and the relative clock frequency between the core and the RNG.

All other operations are linear, so at least a factor of 2 slowdown can be expected there. However, because the size of the active data set doubles and will not fit in 14 registers anymore, a lot of overhead is created by additional loads and stores. Our scheduler manages to generate a securely masked bitsliced SubBytes implementation in $2 \cdot 83 + 4 \cdot 32 = 294$ `XOR`s, $4 \cdot 32 = 128$ `AND`s, 99 stores and 167 loads, that are pipelined as much as possible. Once more, the speed is measured as the average number of cycles per block when encrypting 256 consecutive blocks. The cycle counts are precisely equal for all combinations of inputs, keys, and nonces.

Table 3. Performance of masked constant-time AES

Algorithm	**Speed (cycles)**		**ROM (bytes)**		**RAM (bytes)**	
	M3	**M4**	**Code**	**Data**	**I/O**	**Stack**
AES-128-CTR masked constant-time	N/A	7422.6	39916	12	$368 + 2m$	1588

The performance of the final implementation is summarized in Table 3. Note that of these 7422.6 cycles per block, 2132.5 are spent on generating random words and pushing them to the stack, while all the rest takes 5290.1 cycles per block. A faster RNG could significantly boost the total speed. Of the 1588 bytes on the stack, 1312 are taken by the 328 random words.

5.2 Comparison to existing implementations

Balasch et al. [2] showed at CHES 2015 that adding first-order Boolean masking with Trichina gates slows the implementation down by roughly a factor of 5 on the Cortex-A8. On the Cortex-M4, we see something similar compared to the unmasked bitsliced implementation, with a factor 4.6, although a faster RNG could reduce this to almost a factor of 3.5. Furthermore, we require less randomness because we based ourselves on the 113-gate SubBytes implementation.

Goudarzi and Rivain [14] investigated the performance of different approaches to higher-order masking based on the ISW masking scheme [16] by implementing masked versions of AES and PRESENT on the ARM7TDMI-S microprocessor, a somewhat older architecture from 2001 that is still widely deployed. For first-order masking, their fastest implementation takes 49329 cycles [14, tbl. 16, standard AES with parallel Kim-Hong-Lim S-box, 2 shares], which is a factor 5.6 more than ours, but that comparison is not entirely fair as we do not support higher-order masking. However, instruction timings appear to be similar between the two architectures.

6 Conclusion and outlook

This paper presented various speed-optimized AES software implementations for multiple use case scenarios, including side-channel attack protection, for the ARM Cortex-M3 and M4. All of them are the fastest of their kind. Additionally, we provide an ARM-specific instruction scheduler and register allocator that is of independent interest to optimize other software for these platforms. All software is put into the public domain, which also may benefit the performance of (AES-based) CAESAR candidates on modern embedded microcontrollers.

We admit that the 'all the AES you need' claim in our tittle does not hold for use cases that need to protect against *higher-order* side-channel attacks. We plan to have an assembly generator for higher-order masked AES implementations, although one then may want to resort to masking schemes other than gate-level masking.

References

1. Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. Efficient AES implementations for ARM based platforms. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 841–845. ACM, 2004.
2. 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.
3. Daniel J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.
4. 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.
5. Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, 2012. https://cryptojedi.org/papers/#neoncrypto.
6. Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient software implementation of AES on 32-bit platforms. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *LNCS*, pages 159–171. Springer, 2003.
7. Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *LNCS*, pages 260–272. Springer, 1997. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1997/CS/CS0891.pdf.
8. 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/.

9. D. Canright. A very compact S-box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *LNCS*, pages 441–455. Springer, 2005.
10. Cryptovia. AES algorithms for ARM CPU. http://www.cryptovia.com/ARM_AES.html.
11. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.
12. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES – the Advanced Encryption Standard*. Springer, 2013.
13. Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the Internet of Things. Cryptology ePrint Archive, Report 2015/209, 2015. http://eprint.iacr.org/2015/209/.
14. Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? Cryptology ePrint Archive, Report 2016/264, 2016. http://eprint.iacr.org/2016/264/.
15. 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.
16. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003. https://www.cs.berkeley.edu/~daw/papers/privcirc-crypto03.pdf.
17. 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.
18. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996. http://www.cryptography.com/public/pdf/TimingAttacks.pdf.
19. 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.
20. Mitsuru Matsui. How far can we go on the x64 processors? In Matthew Robshaw, editor, *Fast Software Encryption*, volume 4047 of *LNCS*, pages 341–358. Springer, 2006. http://www.iacr.org/archive/fse2006/40470344/40470344.pdf.
21. Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *LNCS*, pages 121–134. Springer, 2007.
22. Lauren May, Lyta Penna, and Andrew Clark. An implementation of bitsliced DES on the Pentium MMX^{TM} processor. In E. P. Dawson, A. Clark, and Colin Boyd, editors, *Information Security and Privacy*, volume 1841 of *LNCS*, pages 112–122. Springer, 2000.
23. NXP Semiconductors N.V. AN11241: AES encryption and decryption software on LPC microcontrollers. https://www.lpcware.com/content/nxpfile/an11241-aes-encryption-and-decryption-software-lpc-microcontrollers.

24. 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.
25. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. http://eprint.iacr.org/2005/271/.
26. ARM Holdings plc. mbed TLS v2.3.0. https://tls.mbed.org/.
27. 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.
28. RealTimeLogic. SharkSSL/RayCrypto v2.4 crypto library – benchmarks with ARM Cortex-M3. https://realtimelogic.com/products/sharkssl/Cortex-M3/.
29. STMicroelectronics. RM0038 reference manual, 2015. http://www2.st.com/content/ccc/resource/technical/document/reference_manual/cc/f9/93/b2/f0/82/42/57/CD00240193.pdf/files/CD00240193.pdf/jcr:content/translations/en.CD00240193.pdf.
30. STMicroelectronics. RM0090 reference manual, 2015. http://www2.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf.
31. Ko Stoffelen. Instruction scheduling and register allocation on ARM Cortex-M. In *Software performance enhancement for encryption and decryption, and benchmarking – SPEED-B*, 2016. http://ccccspeed.win.tue.nl/papers/armscheduler-final.pdf.
32. 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/.
33. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf.