AES Smaller Than S-Box

Minimalism in Software Design on Low End Microcontrollers

Mitsuru Matsui and Yumiko Murakami(✉)

Information Technology R&D Center, Mitsubishi Electric Corporation, Chiyoda-ku, Japan
Matsui.Mitsuru@ab.MitsubishiElectric.co.jp,
Murakami.Yumiko@cw.MitsubishiElectric.co.jp

Abstract. This paper explores state-of-the-art software implementations of "size-minimum" AES on low-end microcontrollers. In embedded environments, reducing memory size often has priority over achieving faster speed. Some recent lightweight block ciphers can be implemented in 200 to 300 ROM bytes, while the smallest software implementation of AES including key scheduling, encryption and decryption is, as far as we know, around 1 K ROM bytes.

The first purpose of this study is to see how small AES could be. To do this, we aggressively minimize code and data size of AES by introducing a ring multiplication for computing the S-box without any lookup table, a compact algorithm for embedding MixColumns into InvMixColumns, and a tiny loop for processing AddRoundKey, ShiftRows and SubBytes at the same time. As a result, we achieve a 192-byte AES encryption-only code and a 326-byte AES encryption-decryption code on the RL78 microcontroller. We also show that an AES-GCM core can be implemented in 429 bytes on the same microcontroller. These codes include on-the-fly key scheduling to minimize RAM size and their running time is independent of secret information, i.e. timing-attack resistant.

The second purpose of this research is to see what processor hardware architecture is suitable for implementing lightweight ciphers from a minimalist point of view. A simple-looking algorithm often results in very different size and speed figures on different low-end microcontrollers in practice, even if their instruction sets consist of similar primitive operations. We show concrete code examples implemented on four low-end microcontrollers, RL78, ATtiny, Cortex-M0 and MSP430 to demonstrate that slight differences of processor hardware, such as carry flag treatment and branch timing, significantly affect size and speed of AES.

1 Introduction

Lightweight is one of the recent keywords in cryptography, with increasing market requirements of embedded security as a background. A lot of new lightweight symmetric ciphers and hash functions have been proposed, aiming at achieving low resource occupation and at the same time maintaining high level security.

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 51–66, 2015.
DOI: 10.1007/978-3-319-16363-5_4

Lightweight cryptography is in many cases studied in the context of hardware lightweight such as low energy consumption and small circuit area, but software lightweight is also getting paid attention. Some recent researches concentrate on extensive software implementation of lightweight ciphers on an embedded microcontroller [1–3].

In embedded environments, reducing memory size often has priority over achieving faster speed and it has been reported that some lightweight block ciphers can be implemented on an embedded microcontroller in extremely small 200 to 300 ROM bytes [3–5]. This paper goes deep into this direction for AES. As far as we know, the smallest software AES with 128-bit key, including key scheduling, encryption and decryption, still requires 1 K ROM bytes [3]. In fact, to create an AES code within 1. 5K ROM bytes, loop rolling is necessary inside its round function, which leads to heavy performance penalty. This explains why most of known AES implementations require at least 1.5 K ROM bytes.

Our aim is to see how small AES could be, and to achieve this goal, we aggressively try to minimize code and data size of AES. Our code does not use any lookup tables for the S-box. It is instead computed with a Galois field inversion and a matrix multiplication as in its original definition. While the matrix multiplication is not a Galois field operation, we point out that a Galois field multiplication included in the Galois field inversion is "essentially the same" as the matrix multiplication since the former is a ring operation on $GF(2)[x]/(x^8+x^4+x^3+x+1)$ and the latter is that on $GF(2)[x]/(x^8+1)$. This observation leads to a new compact logical S-box code. Note that the fact that the matrix is circular is essential.

We next show that `MixColumns` can be fully embedded in `InvMixColumns` not only in hardware [6] but in software in a very simple and compact manner. In fact, `InvMixColumns` also works as `MixColumns` by just adding one conditional jump indicating encryption or decryption into `InvMixColumns`. This greatly contributes to code reduction of AES containing both encryption and decryption. In addition, it is demonstrated that `AddRoundKey`, `ShiftRows` and `SubBytes` can be merged into a tiny loop of around 20 bytes, except the S-box logic.

As a result, we achieve a 192-byte AES encryption-only code and a 326-byte AES encryption-decryption code on the RL78 microcontroller. We also show that an AES-GCM core can be implemented in 429 bytes on RL78. These algorithms are implemented on the ATtiny microcontroller as well, and it is seen that our resultant codes are a bit larger but much faster than those on RL78. All of these codes include on-the-fly key scheduling to minimize RAM size, and their running time is independent of secret information such as key and text.

The second purpose of this research is to see what processor hardware is suitable for implementing lightweight ciphers from a minimalist point of view. It is rather common that a code based on the same algorithm exhibits very different size and speed figures on different low-end microcontrollers, even if their instruction sets consist of similar primitive operations. Many types of low-end microcontrollers have been used in real-world embedded applications, but their comparative research from a cryptographic point of view seems missing.

We show concrete examples extracted from our AES codes and implemented on four low-end microcontrollers, RL78, ATtiny, Cortex-M0 and MSP430, and demonstrate that slight-looking differences of hardware, in particular carry flag treatment or branch timing, significantly affect size and speed of a target code. We believe that this information is beneficial to not only programmers but also designers of a cryptographic algorithm.

2 How to Minimize AES in Software

In this section we show several techniques to minimize code size of AES. For the specification of AES and the notations, see [7]. Throughout this section, we use C language notations in describing implementation algorithms.

2.1 SubBytes and InvSubBytes

We implement S-box $S(x)$ and the inverse S-box $IS(x)$ as their original formula shown below, using an inversion over $GF(2^8)$, a matrix multiplication on $GF(2)^8$ and an xor of a constant value without any lookup tables.

$$S(x) = \begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix} \cdot x^{-1} + 63, \qquad IS(x) = (x^{-1} + 63) \cdot \begin{pmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{pmatrix}$$

The inversion is an operation on $GF(2)[X]/(X^8+X^4+X^3+X+1)$ and the matrix multiplication can be regarded as an operation on $GF(2)[X]/(X^8+1)$ since it is circular. In other words, a multiplication routine on $GF(2^8)$ can also compute a matrix multiplication on $GF(2)^8$ by just replacing the polynomial.

In general, a (random) matrix calculation is expensive in software, but in our case, the matrix multiplication above becomes almost free by sharing it to the Galois multiplication. This observation leads to the following simple algorithm for computing $S(x)$ and $IS(x)$ as follows:

```
Input x, Output SubBytes(x)
  01:  x = INV8(x)                   : Galois inversion
  02:  x = MUL8(x,0x1f,0x101,0x63) ; matrix multiplication
  03:  return x                      ; (0x101 denotes X^8+1)

Input x, Output InvSubBytes(x)
  04:  x = MUL8(x,0x4a,0x101,0x05) ; matrix multiplication
  05:  x = INV8(x)                   ; Galois inversion
  06:  return x                      ; (0x05 = 0x63 * 0x4a)
```

```
Input x, Output INV8(x)          ; x^254 using a binary method
  07:  c = 0, y = 1
  08:    y = MUL8(y,x,0x11b,0)     ; Galois multiplication
  09:    y = MUL8(y,y,0x11b,0)     ; Galois multiplication
  10:    c = c+1                   ; (0x11b denotes X^8+X^4*X^3+X+1)
  11:    if(c != 7) goto 08
  12:  return y

Input x,y,f,v, Output MUL8(x,y,f,v) ; v=v+(x*y) on GF(2)[X]/(f)
  13:  c = 0
  14:    if((x&1) == 1) v = v^y
  15:    x = x>>1
  16:    y = y<<1
  17:    if(y > 255) y = y^f
  18:    c = c+1
  19:    if(c != 8) goto 14
  20:  return v
```

2.2 AddRoundKey+ShiftRows+SubBytes

AddRoundKey, ShiftRows and SubBytes can be combined into in a very simple loop by noting that ShiftRows moves its i-th input byte to the ($i*13$ mod 16)-th output byte ($i = 0, 1, 2, .., 15$). It is easy to see that InvShiftRows, InvSubBytes and AddRoundKey for decryption can be written in a similar way:

```
Input  x[0]..x[15],k[0]..k[15]
Output y[0]..y[15]=(AddRoundKey+ShiftRows+SubBytes)(x,k)
  22:  c = 0, d = 0
  23:    a = x[c]^k[c]       ; AddRoundKey
  24:    y[d] = SubBytes(a)  ; SubBytes (S-box)
  25:    d = d+13 mod 16     ; ShiftRows
  26:    c = c+1
  27:    if(c != 8) goto 23  ; equivalently, if(d == 0) goto 23
  28:  return y
```

2.3 Sharing MixColumns with InvMixColumns

Another trick to minimize AES is to share MixColumns with InvMixColumns using the following equation, where the middle/left matrix is the one defined in MixColumns/InvMixColumns, respectively. This decomposition was implicitly used in [6] in the hardware context. Note that all entries in the middle matrix have active bits at bit 0 and/or 1, and all entries in the right matrix have active bits at bit 2 and/or 3.

$$\begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} + \begin{pmatrix} 0c & 08 & 0c & 08 \\ 08 & 0c & 08 & 0c \\ 0c & 08 & 0c & 08 \\ 08 & 0c & 08 & 0c \end{pmatrix}$$

Using this fact, we can embed `MixColumns` into `InvMixColumns` in the following simple way, where only one additional instruction - a conditional branch - is necessary for detecting encryption/decryption. This algorithm consists of a double loop to minimize its code size, where the inner loop (lines 32 to 44) computes only one vector of the entire four vectors contained in the matrices. We are again using the fact that they are circular.

```
Input  x[0]..x[15]
Output y[0]..y[15]=MixColumns(x) or InvMixColumns(x)
  29:  c0 = 0                                ; matrix number
  30:    a0 = a1 = a2 = a3 = 0
  31:    c1 = 0                              ; vector number
  32:      t = x[c0*4+c1]
  33:      a0 = a0^t, a1 = a1^t, a2 = a2^t   ; 0th bit (ENC)
  34:      t = t*2 on GF(256)
  35:      if ENCRYPTION, then go to 41
  36:        a2 = a2^t, a3 = a3^t            ; 1st bit (DEC)
  37:        t = t*2 on GF(256)
  38:        a1 = a1^t, a3 = a3^t            ; 2nd bit (DEC)
  39:        t = t*2 on GF(256)
  40:        a0 = a0^t, a1 = a1^t            ; 3rd bit (DEC)
  41:      a2 = a2^t, a3 = a3^t              ; 1st/3rd bit (ENC/DEC)
  42:      t = a0, a0 = a1, a2 = a3, a3 = t  ; rotate shift
  43:      c1 = c1+1
  44:      if(c1 != 4) goto 32
  45:    y[c0*4] = a0, y[c0*4+1] = a1, y[c0*4+2] = a2, y[c0*4+3] = a3
  46:    c0 = c0+1
  47:    if(c0 != 4) goto 30
  48:  return y
```

Note that [8] reports another decomposition of the `InvMixColumns` matrix as a multiplication of two matrices, not an addition, one of which is the `MixColumns` matrix. Implementing this form in software leads to a bigger code than the above because of the matrix multiplication.

3 Implementation on RL78 and ATtiny

In this section, we discuss our implementations of AES on two low-end microcontrollers, RL78 [9] and ATtiny [10]. RL78 is a typical accumulator-based CISC processor and ATtiny is a typical register-symmetrical RISC processor.

We believe that looking at software implementation on processors with utterly different architecture is of its own interest.

On each processor, we implement three instances: AES encryption-only (AES-E), AES encryption/decryption (AES-ED) and AES Galois counter mode (AES-GCM) [11], based on the algorithms shown in the previous section. Our top priority is to minimize ROM size, and we also try to reduce RAM usage, which is equally important in practice. All codes presented in this section include on-the-fly key scheduling.

3.1 RL78 and ATtiny Microcontrollers

In this subsection we briefly introduce the two microcontrollers, RL78 and ATtiny. More detailed architectural comparison with actual code examples will be discussed in next section.

RL78 has eight 8-bit general-purpose registers `a,x,b,c,d,e,h,l`, of which many instructions accept only `a` or `ax` as a destination. A limited number of instructions have a 16-bit form with register pairs `ax,bc,de,hl`. Unfortunately an xor instruction, frequently used in block ciphers, does not have a 16-bit form [12].

We hence often suffer from "register starvation" on this microcontroller, which requires extra instructions and memory to save/restore data on an accumulator, but a big advantage of RL78 is that code size tends to be short due to its support of read-modify instructions. For instance, `'xor a,[hl]'`, – read from an address pointed by `hl` and xor to `a` –, is a one-cycle instruction with one-byte length. This significantly contributes to code size reduction.

ATtiny has thirty-two 8-bit general-purpose registers `r0` to `r31`. Most instructions have "register symmetry", i.e. accept any register as a destination. Some instructions dealing with immediate data accept only `r16` to `r31`, but this seldom causes trouble to a minimalist. Three register pairs `(r26,r27)`, `(r28,r29)`, and `(r30,r31)` are used as address registers `X`, `Y`, and `Z`, respectively [13].

Almost all instructions of ATtiny are two-byte long and no read-modify instructions are supported. Hence a code size of this RISC microcontroller tends to be bigger than that of CISC RL78, but in general creating a faster code is possible due to less memory accesses and faster jump instructions. The latter is very important because a minimum-size code often consists of many small loops.

Our coding and performance measurement is done on RL78-G12 (ROM 8 K bytes and RAM 768 bytes) and ATtiny85 (ROM 8K bytes and RAM 512 bytes), respectively.

3.2 Interface and Metrics

Defining a software interface clearly is particularly important for discussing a minimal code. For instance, some implementations on ATtiny allow programmers to destroy all registers without restoration [1,2]. Other implementations [14,15] follow the function call conventions described by Atmel [16]. For the latter case, a subroutine code that destroys all registers has to save/restore 18 registers at the entry and exit of the code, which requires additional 72-byte ROM and 36-byte RAM by pushing/popping these registers. Obviously this overhead is not ignorable in our context.

While our goal is to obtain a size minimum code, we also keep a practical and usable code in mind. We hence adopt the latter approach; that is, we create a function callable from a high-level language and count all resources occupied by a code in referring to ROM/RAM size, as discussed in [3]. The following is our coding and measurement policy in this paper, balancing minimalism, usability and security.

1. Code is described as a subroutine callable from C-language.
2. Code processes one-block data.
3. Code should be relocatable.
4. Execution time is independent of secret information (text and key).
5. ROM size includes instruction code and constant data.
6. RAM size includes plaintext/ciphertext, key, stack, and temporary memory.
7. Plaintext area is shared with ciphertext area.
8. Key area can be destroyed but is recovered at the end of the code.

RAM memory for locating parameters such as text, key and other necessary information, is allocated in consecutive area within a callee code and its address is passed to a caller program as an external variable.

We should clarify policy 2 in the case of AES-GCM. In AES-GCM, we introduce a switch, which we call MODE, that indicates which part of AES-GCM should be carried out, as shown in Fig. 1. Our code reads MODE included in the parameters and executes an appropriate component of the AES-GCM algorithm.

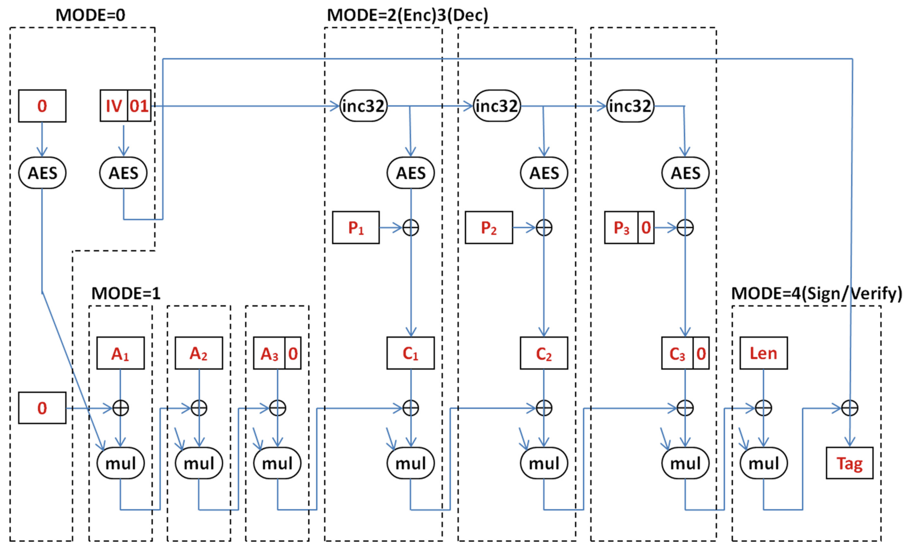

Fig. 1. Our implementation blocks of AES-GCM

Also, strictly speaking we apply policy 3 to our codes of ATtiny in a slightly relaxed way. It is assumed that the RAM memory (excluding stack) resides within the same 256-byte block without crossing a 256-byte aligned boundary. This is because otherwise updating address registers becomes very costly.

3.3 Implementation Results

Table 1 shows our implementation results of AES-E, AES-ED and AES-GCM on RL78 and ATtiny. We followed the coding policy described in the previous subsection and aimed at a minimum memory size. In this table, (E) and (D) denote speed in encryption and decryption, respectively, and (n) in AES-GCM means MODE shown in Fig. 1.

Table 1. Size minimum implementation of AES-E, AES-ED and AES-GCM

Algorithm	Controller	ROM	RAM	Speed (cycles/block)
AES-E	RL78	192	88	369901
AES-E	ATtiny	214	78	262061
AES-ED	RL78	326	103	374016 (E), 449408 (D)
AES-ED	ATtiny	356	78	264302 (E), 318800 (D)
AES-GCM	RL78	429	182	741088 (0), 40620 (1), 412608 (2,3), 40937 (4)
AES-GCM	ATtiny	522	165	525882 (0), 30536 (1), 293506 (2,3), 30876 (4)

We achieved a 192-byte AES encryption code, including on-the-fly key scheduling, on the RL78 microcontroller. It is again noted that it runs in a constant time. As far as we know, this is the smallest AES ever made in software. Of course, its speed is very slow, but still makes sense in non timing-critical applications since it runs in 18.5 ms/block in 20 MHz clock.

For comparison with ATtiny, RL78 is smaller but slower than ATtiny, as expected. More specifically ATtiny is 30 % faster, but 10 % larger for AES-E and AES-ED and 20 % larger for AES-GCM. There are two reasons why ATtiny is much larger in AES-GCM. One is that RL78 has a 16-bit addition instruction and a multiple-bit shift instruction, which are missing in ATtiny. These instructions are efficiently used for counting and accumulating text length in AES-GCM.

Another and more serious reason of this is that ATtiny only allows a 6-bit displacement in register indirect addressing. This means that an address register must be updated every time when it points a new address that is distant from a current address by 64 or more bytes. This restriction does not cause a problem in AES-E and AES-ED since their RAM size is close to 64 bytes, but that of AES-GCM is much larger, which results in visible penalty.

We will discuss more detailed software implementation issues depending on processor hardware architecture in the next section. Here we only illustrate code examples of `AddRoundKey+ShiftRows+SubBytes` to show how a small loop can be implemented on these microcontrollers in Table 2, where we use the same assembler syntax for RL78 and ATtiny for readers' convenience. It is seen in `AddRoundKey(AR)` that a missing read-modify instruction on ATtiny is compensated by its post increment addressing mode. It is also noted that ATtiny requires more instructions to modify/restore the destination address register at the end of `SubBytes(SB)`, and instead RL78 needs more instructions to do `ShiftRows(SR)` due to its accumulator-based architecture; i.e. no instruction exists such as `'sub c,3'` on RL78.

Table 2. AddRoundKey+ShiftRows+SubBytes on RL78 and ATtiny

```
[RL78: 19byte long]
      clr   c        ; c=0
    loop:
(AR)  mov   a,[hl]
(AR)  xor   a,[hl+KEY]
(AR)  inc   hl
(SB)  call  Sbox     ; a->a
(SB)  mov   TMP[c],a
(SR)  mov   a,c
(SR)  sub   a,3
(SR)  and   a,15
(SR)  mov   c,a
(SR)  jnz   loop
```

```
[ATtiny: 22byte long]
      clr   r20        ; r20=0
    loop:
(AR)  mov   r21,[Z+KEY]
(AR)  mov   r22,[Z++]
(AR)  xor   r21,r22
(SB)  call  Sbox       ; r21->r0
(SB)  add   XL,r20
(SB)  mov   [X],r0     ; X=TMP+r20
(SB)  sub   XL,r20
(SR)  sub   r20,3
(SR)  and   r20,15
(SR)  jnz   loop
```

3.4 Variations

It is common in a minimum-size approach to see that a slight modification of a source code significantly affects its performance. To see this, we unroll a performance critical loop and measure the size and speed of resultant codes. The performance bottleneck of our AES codes is of course computation of S-box. In particular a multiplication on $GF(2)[X]/(f)$, corresponding to `MUL8` shown in the implementation algorithm of `SubBytes` and `InvSubBytes`, is the critical routine. It consists of a loop with an eight-time iteration (a code example of this routine will be shown in the next section). Unrolling this performance-critical loop improves speed at the cost of a small increase in ROM size as illustrated in Table 3. This table shows that our 520-byte code of AES-GCM on RL78 outperforms the 522-byte code on ATtiny. It should be noted that if we see performance of these microcontrollers with the same ROM size, the lead of ATtiny is not so big.

Table 4 shows another variation of our codes where the S-box and its inversion routines are replaced with normal lookup tables, including previous smallest implementations [3]. We think that ours are still a minimum record of AES, while not a minimalist approach. In the implementation on ATtiny, we put these lookup tables on a 256-byte address boundary for faster memory access, as most implementations of AES on an AVR processor do [1,17,18].

Table 3. Loop unrolled codes of AES-E, AES-ED and AES-GCM on RL78

Algorithm	Controller	#iterations	ROM	RAM	Speed (cycles/blocks)
AES-E	RL78	4	206	88	309901
AES-E	RL78	2	234	88	279901
AES-E	RL78	1	283	88	246901
AES-ED	RL78	4	340	103	314016 (E), 377408 (D)
AES-ED	RL78	2	368	103	284016 (E), 341408 (D)
AES-ED	RL78	1	417	103	251008 (E), 301792 (D)
AES-GCM	RL78	4	442	182	621088 (0), 352608 (2,3)
AES-GCM	RL78	2	471	182	561088 (0), 322592 (2,3)
AES-GCM	RL78	1	520	182	495104 (0), 289600 (2,3)

Table 4. Lookup table implementation of AES-E, AES-ED and AES-GCM

Algorithm	Controller	ROM	RAM	Speed (cycles/block)
AES-E [3]	RL78	486	78	7288
AES-E	RL78	399	78	8704
AES-E	ATtiny	428	82	8870
AES-ED [3]	RL78	970	84	7743 (E), 10362 (D)
AES-ED	RL78	776	85	9847 (E), 13634 (D)
AES-ED	ATtiny	814	82	9624 (E), 13869 (D)
AES-GCM	RL78	642	172	19695 (0), 40640 (1), 51904 (2,3), 40928 (4)
AES-GCM	ATtiny	730	165	19486 (0), 30536 (1), 40308 (2,3), 30876 (4)

In this implementation, ATtiny is 5–10 % larger than RL78 but its speed is comparable with RL78 for AES-E and AES-ED because the performance gain in S-box of ATtiny, which will be demonstrated in the next section, is lost. On the other hand, ATtiny is much faster in all modes of AES-GCM except MODE=0. This is because GHASH of AES-GCM, more specifically a multiplication on $GF(2^{128})$ runs much faster on ATtiny than on RL78. This will be also discussed in the next section.

Lastly, we mention that it is possible to further reduce the code size of our 192-byte program on RL78 by relaxing (or ignoring) the coding policies shown in this section. This is not recommended in general, but might make sense in certain situations. The first possibility is to allow to destroy key data without restoration. Our code copies key to temporary area before starting actual encryption, and hence removing this part reduces code size by 10 bytes. Another possibility is to remove timing-attack protection. In the `MUL8` routine, which is in the bottom of the S-box, we can quit the loop as soon as the shifted multiplier becomes zero, without iterating eight times. The resultant code no longer runs in constant time, but reduces register pressure and saves 4 bytes. Ignoring the function call convention gains another 2 bytes. Also without violating any policy, x^{254} can be computed by simply multiplying x 253 times instead of using a binary method, which reduces further 4 bytes. We did not adopt this because it makes the code too slow. Applying all these reduces its ROM size down to 172 bytes. It will be a less practical code, though.

4 Minimalism from Hardware Viewpoints

There are various types of microcontrollers currently available in the market, of which low-end ones usually have a similar instruction set consisting of only basic operations such as read/write, arithmetic, logical and branch. However, in practice, minor-looking differences of these instructions often lead to a significant impact on size and speed. This section takes Cortex-M0 [19] and MSP430 [20], in addition to RL78 and ATtiny, as target microcontrollers and demonstrates this fact using concrete code examples for AES. An architectural comparison of these microcontrollers is summarized in the appendix.

Our first example is `MUL8`, a multiplication on $GF(2)[X]/(f)$ used in S-box. The following examples illustrate code minimum implementation of `MUL8` on these four processors. `(1)` corresponds to lines 15 and 16, and `(2)` corresponds to lines 17 and 18 in the sequence shown in Sect. 2.1 (Tables 5 and 6).

Table 5. MUL8 on RL78 (left) and ATtiny (right)

```
[RL78:  b=a*x+b (c=f)]                      [ATtiny: r23=r21*r22+r23 (r24=f)]
[19 bytes, 103 cycles]                      [18 bytes, 72 cycles]
     mov   d,8                                   mov   r25,8
   loop:                                       loop:
(1)  xch   a,x ;exchange a with x           (1)  shr   r22,1
(1)  shr   a,1                              (1)  jnc   L1
(1)  xch   a,x                              (1)  xor   r23,r21
(1)  sknc      ;skip next if non carry        L1:
(1)  xor   b,a
(2)  shl   a,1                              (2)  shl   r21,1
(2)  sknc      ;skip next if non carry      (2)  jnc   L2
(2)  xor   a,c                              (2)  xor   r21,r24
                                               L2:
     dec   d                                     dec   r25
     jnz   loop                                  jnz   loop
```

The simplest code is on ATtiny. On RL78, an overhead for creating backup of the accumulator is unavoidable. On the other hand RL78's `sknc` instruction (replace next instruction with `nop` if a condition is met) works fine as a faster alternative of `jnc`. A conditional taken/not-taken jump of Cortex-M0 takes three/one cycles, respectively. This means that a redundant `nop` instruction must be inserted for constant time execution. Also since all registers of

Table 6. MUL8 on Cortex-M0 (left) and MSP430 (right)

```
[Cortex-M0 r6=r4*r5+r6 (r7=f)]
[22 bytes, 102 cycles]
     mov   r1,8
   loop:
(1)  shr   r5,r5,1
(1)  jnc   L1
(1)  xor   r6,r6,r4
(1)  nop   ;protection from
           ;timing attack

   L1:
(2)  shl   r4,r4,1
(2)  shl   r2,r4,24 ;creating carry
(2)  jnc   L2
(2)  xor   r4,r4,r7
(2)  nop   ;protection from
           ;timing attack
   L2:
     sub   r1,r1,1
     jnz   loop
```

```
[MSP430 r12=r11*r10+r12 (r13=f)]
[26 bytes, 121 cycles]
     mov   r14,8
   loop:
(1)  shr   r11,1
(1)  jc    L1a ;protection from
(1)  nop       ;timing attack
  L1a:
(1)  jnc   L1b
(1)  xor   r12,r10
   L1b:
(2)  shl   r10,1
(2)  jc    L2a ;protection from
(2)  nop       ;timing attack
   L2a:
(2)  jnc   L2b
(2)  xor   r10,r13
   L2b:
     sub   r14,1
     jnz   loop
```

Cortex-M0 are 32-bit long only, an extra instruction is required to create the carry flag. Interestingly, a conditional jump of MSP430 always takes two cycles, and hence a special care is need to create a timing-attack protected code. To do this we insert a dummy conditional jump instruction with an opposite logic for each branch, which causes a heavy size and performance penalty.

The next example is a multiplication on $GF(2^{128})$ that appears in GHASH of AES-GCM. The following codes show part of one iteration of the multiplication. More specifically, the code consists of two functions: (1) If carry (or the highest bit of a register) is active, then $A = A$ xor B, (2) Rotate right shift B by one bit. A and B are 128-bit data pointed by an address register.

The most straightforward code is RL78. Note that in the first loop the carry flag must be checked every time to make the code run in constant time. Since both loops handle the carry flag independently, it is not trivial to combine them into a single loop. However for ATtiny, thanks to its `sbrc` instruction (replace next instruction with `nop` if a bit on a register is non active), this can be done in a very simple way (Table 7).

An obstacle of Cortex-M0 and MSP430 is that they do not have a decrement instruction that does not touch the carry flag. In general carry-free `dec/inc` instructions are frequently used for arithmetic of long integers. Moreover Cortex-M0 does not have a rotate-shift-with-carry instruction. Hence we have to create a rather tricky code to simulate it. This causes a significant penalty. MSP430 again suffers a speed overhead for timing adjustment, while the code is very simply described due to its abundant addressing mode.

Table 7. Multiplication on $GF(2^{128})$ on RL78 (left) and ATtiny (right)

```
[RL78]
[23 bytes, 302 cycles]
     mov   bc,#1010h
   loop1:
     dec   hl
(1)  mov   a,[hl+disp]
(1)  sknc  ;check carry
(1)  xor   a,[hl]
(1)  mov   [hl+disp],a
     dec   b
     jnz   loop1

     cmp0  a ;reset carry
   loop2:
(2)  mov   a,[hl]
(2)  rorc  a,1 ;rotation with carry
(2)  mov   [hl],a
     inc   hl
     dec   c
     jnz   loop2
```

```
[ATtiny]
[22 bytes, 225 cycles]
     clc            ;reset carry
     mov   r22,16
   loop1:
(1)  mov   r20,[Z]
(1)  mov   r21,[Z+disp]
(1)  sbrc  r23,7    ;skip next if 7th bit is 0
(1)  xor   r20,r21
(2)  rorc  r21,1    ;rotation with carry
(2)  mov   [Z+disp],r21
(1)  mov   [Z++],r20
     dec   r22
     jnz   loop1
```

Table 8. Multiplication on $GF(2^{128})$ on Cortex-M0 (left) and MSP430 (right)

```
[Cortex-M0]
[32 bytes, 320 cycles]
     mov   r1,0      ;reset carry
     mov   r4,16
   loop:
(1)  mov   r6,[r7]
(1)  shl   r5,r2,25 ;check 7th bit
(1)  mov   r5,[r7+16]
(1)  xor   r6,r6,r5
(1)  jnc   L1
(1)  mov   [r7],r6
   L1:
(2)  shl   r1,r1,7  ;r1=00 or 80
(2)  shr   r5,r5,1
(2)  xor   r5,r5,r1
(2)  mov   [r7+16],r5
(2)  adc   r1,r1,r1 ;r1=00 or 01
     add   r7,r7,1
     sub   r4,r4,1
     jnz   loop
```

```
[MSP430]
[30 bytes 291 cycles]
     mov   r13,0           ;reset carry
     mov   r14,16
   loop:
(1)  test  r12             ;check 7th bit
(1)  jl    L1              ;protection from
(1)  xor   [r11+0],[r11]   ;timing attack
   L1:
(1)  jge   L2
(1)  xor   [r11+16],[r11]
   L2:
(2)  rorc  r13,1           ;save carry
(2)  rorc  [r11++],1
(2)  rolc  r13,1           ;restore carry
     sub   r14,1
     jnz   loop
```

For Cortex-M0 and MSP430, we can write a much faster code by fully using their 32-bit/16-bit wide registers. However this results in an increase in code size because we need extra byte-swap instructions due to little-endianness of these microcontrollers (Table 8).

5 Concluding Remarks

In this paper we explored minimalism in software implementation of AES on various modern low-end microcontrollers. As far as the authors know, this is the first extensive analysis of embedded software coding of symmetric primitives toward memory size reduction with comparative viewpoints of processor hardware. As concluding remarks, we mention some lessons we learned which could be beneficial to programmers and designers of symmetric primitives for low-end microcontrollers.

Use left shifts. Availability and efficiency of shift instructions greatly depends on processor hardware. Some do not support shift with carry instructions. Then `adc` (addition with carry) instruction can be an alternative of a left rotation with carry.

Be aware of locality of RAM access. ATtiny accepts only 6-bit displacement in its register indirect addressing, which is often restrictive. An order of parameters such as text, key, temporary subkey etc., can affect code size and performance.

Why not little endian. Most modern processors have a little endian hardware, but most modern symmetric encryption algorithms are suitable to a big endian architecture. RL78, ATtiny, Cortex-M0, MSP430 are all little endian.

Matrix should be circular. A circular matrix significantly contributes to code reduction. First of all, a matrix multiplication with a circular matrix can be described using a vector-wise loop, and also can be regarded as a multiplication on a ring, as shown in this paper.

Appendix: Low End Microcontrollers Comparison Chart

This table is not intended to be exhaustive, but to illustrate typical cases in implementing lightweight symmetric ciphers for readers' convenience.

	RL78	ATtiny	CortexM0	MSP430
Hardware Registers				
- Register Size	8,16	8	32	8,16
- Number of General Registers	8	32	13	12
Addressing Modes				
- Number of Operands	2	2	2,3	2
- Read-Modify(-Write) Instructions	R-M	No	No	R-M-W
- Post-Increment Addressing	No	Yes	No	Yes
Code Length (bytes)				
- Operation equivalent to `xor reg,[mem]`	1-3	4	4	2,4
- Conditional Short Jump	2	2	2	2
- Subroutine Call	3	2	4	4
Instruction Latency (cycles)				
- Read from Memory (RAM/ROM)	1/4	2/3	2	2
- Operation equivalent to `xor reg,[mem]`	1	2	2	2-3
- Conditional Short Jump (taken/not-taken)	4/2	2/1	3/1	2/2
- Call+Return	9	7	7	7
Supported Instructions				
- Shift with multiple counts	Yes	No	Yes	No
- Rotate Shift without carry	Yes	No	Yes	No
- Rotate Shift with carry	Yes	Yes	No	Yes
- Carry preserving increment/decrement	Yes	Yes	No	No
- Conditional Skip	Yes	Yes	No	No

References

1. Eisenbarth, T., et al.: Compact implementation and performance evaluation of block ciphers in ATtiny devices. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 172–187. Springer, Heidelberg (2012)
2. Balasch, J., et al.: Compact implementation and performance evaluation of hash functions in ATtiny devices. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 158–172. Springer, Heidelberg (2013). http://eprint.iacr.org/2012/507.pdf
3. Matsui, M., Murakami, Y.: Minimalism of software implementation-extensive performance analysis of symmetric primitives on the RL78 microcontroller. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 393–409. Springer, Heidelberg (2014)
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. http://eprint.iacr.org/2013/404.pdf
5. Papagiannopoulos, K., Verstegen, A.: Speed and size-optimized implementations of the PRESENT cipher for tiny AVR devices. In: Hutter, M., Schmidt, J.-M. (eds.) RFIDsec 2013. LNCS, vol. 8262, pp. 161–175. Springer, Heidelberg (2013)

6. Fischer, V., Drutarovsky, M., Chodowiec, P., Gramain, F.: InvMixColumn decomposition and multilevel resource sharing in AES implementations. IEEE Trans. VLSI Syst. **13**(8), 989–992 (2005)
7. Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, NIST (2001)
8. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer, Heidelberg (2002)
9. Renesas Electronics, RL78 Family. http://am.renesas.com/products/mpumcu/rl78/index.jsp?campaign=gn_prod
10. Atmel, tinyAVR Microcontrollers. http://www.atmel.com/products/microcontrollers/avr/tinyavr.aspx
11. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Special Publication 800–38D, NIST (2007)
12. RL78 Family, User's Manual. http://documentation.renesas.com/doc/products/mpumcu/doc/rl78/r01us0015ej0210_rl78.pdf
13. 8-bit AVR Instruction Set http://www.atmel.com/Images/doc0856.pdf
14. AVR-Crypto-Lib Wiki. http://www.das-labor.org/wiki/AVR-Crypto-Lib/en
15. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl - a SHA-3 candidate. http://www.groestl.info/
16. Mixing Assembly and C with AVRGCC. http://www.atmel.com/Images/doc42055.pdf
17. Bos, J.W., Osvik, D.A., Stefan, D.: Fast Implementations of AES on Various Platforms. http://eprint.iacr.org/2009/501.pdf
18. Poettering, B.: Rijndael Furious. http://perso.uclouvain.be/fstandae/lightweight_ciphers/source/AES_furious.asm
19. ARM Cortex-M0 core MCUs. http://www.nxp.com/products/microcontrollers/cortex_m0_m0/
20. Overview for MSP430 Ultra-Low Power 16-bit MCUs. http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/overview.page