Speed-, Space-, and Code Size Optimisation of AES-128 in ATmega16 micro controller environments

DTU - The Technical University of Denmark

01409 - Lightweight Cryptography E15

Project I

Student name and id: Mikkel Ole Rømer (s113408)

1 Introduction

The following paper seeks to investigate Cryptography issues in within the domain of embedded systems. In specific this report will conduct three individual performance investigations of the AES-128 encryption environment on a ATmega16 micro controller. The dimensions of this analysis will be, RAM consumption, Code Size, and finally Execution Time, all of which will be implemented using the low level programming language C, and analysed using AVR Studio 6.
Knowledge about AES and Cryptography in general is assumed, as well as basic programming skills in C. However, this article will provide a short introduction towards AES-128 in terms of algorithmic specifications.

1.1 AES-128 - Algorithm Summary

As mentioned the AES cryptography system that this article will focun on is the AES-128 environment with 10 rounds. The operations of AES and their order is given in table 1.

In short terms, the `AddRoundKey` operation simply XOR's the message with the key from the key schedule.
The Substitution Bytes operation simply substitutes the message value with the value of the SBox table. That is the value of the message byte becomes the index.
Shiftrows operates by shifting every row x positions to the left, where x is the row number. Eg: row 0 stays the same, row 1 is shifted 1 position to the left, and so on.
Finally, the slightly more complex Mix Columns operation, works by multiplying the matrix M with the each column such that the keep their order.

AES Encryption over n rounds	
AddRoundKey	Pre-whitening
SubBytes ShiftRows MixColumns AddRoundKey	For n-1 rounds
SubBytes ShiftRows AddRoundKey	The Final Round

Table 1: The general AES algorithm structure

2 AES-128 optimized for Speed

Throughout the following section, this paper will introduce an C implementation of a AES-128 cryptographic-environment optimized for speed. This implementation will ground in knowledge about loop unrolling, in-line computations and in memory (RAM) table lookups.

From previous chapters the AES-128 algorithm and specification is described in further details, thus this section will only point out the implementation on each step.

2.1 Summarise

The below table 2, summarises the performance of the implementation, using the C optimisation standard `-O0`, which in none. By allowing the C compiler to optimise further on the code, using for instance `-O1`, a RAM and Program decrease of $(1-(284/316)\cdot 100 = 10\%$ and $(1-(852/2594)\cdot 100 = 67\%$ respectively was achieved.

Optimization dimention	Value	Configuration Level
Program Memory	2594 Bytes	-O0
RAM Consumption	316 Bytes	-O0
Clock Cycles	21499 Clocks	-O0

Table 2: Optimisation for speed on an ATmega16 Micro Controller using -O0 configuration

2.2 Round key generation

From Figure 2 the round key implementation is illustrated. This function is created as a static function, which takes the round number as argument ie: $0, 1, \cdots, 10$, and modifies the Round Key accordingly:

1 If round number is 0 then set round key equal to the key

2 If round number is not 0 then modify the previous round key according to algorithm summary 1.1.

```
static void roundKeyGeneration(uint8_t i)
{
  if(i == 0)
  {
    roundKey[0] = key[0];
    roundKey[1] = key[1];
    roundKey[2] = key[2];
    roundKey[3] = key[3];
    roundKey[4] = key[4];
    roundKey[5] = key[5];
    roundKey[6] = key[6];
    roundKey[7] = key[7];
    roundKey[8] = key[8];
    roundKey[9] = key[9];
    roundKey[10] = key[10];
    roundKey[11] = key[11];
    roundKey[12] = key[12];
    roundKey[13] = key[13];
    roundKey[14] = key[14];
    roundKey[15] = key[15];
  }
  else
  {
    uint8_t temp[4];
    temp[0] = sBox[roundKey[13]] ^ rCon[i];
    temp[1] = sBox[roundKey[14]];
    temp[2] = sBox[roundKey[15]];
    temp[3] = sBox[roundKey[12]];
    roundKey[0] ^= temp[0];
    roundKey[1] ^= temp[1];
    roundKey[2] ^= temp[2];
    roundKey[3] ^= temp[3];
    roundKey[4] ^= roundKey[0];
    roundKey[5] ^= roundKey[1];
    roundKey[6] ^= roundKey[2];
    roundKey[7] ^= roundKey[3];
    roundKey[8] ^= roundKey[4];
    roundKey[9] ^= roundKey[5];
    roundKey[10] ^= roundKey[6];
    roundKey[11] ^= roundKey[7];
    roundKey[12] ^= roundKey[8];
    roundKey[13] ^= roundKey[9];
    roundKey[14] ^= roundKey[10];
    roundKey[15] ^= roundKey[11];
  }
}
```

Figure 1: Round key implementation optimized for speed.

2.3 Pre-whitening

The pre-whitening operation is performed, as a set of in-line operations in the main method, in order to waste clock- cycles on function calls. The in addition the encryption of the 128 bit message with the 128 bit round key is carried out without looping through the byte array[]. The implementation is viewable from figure **??**

2.4 The n-1 rounds

The main part of this AES algorithm is the n-1, in this case 9, rounds. Each round follows the previously described algorithm, see algorithm summary 1.1, Once again, the loops is unrolled and performed in-line, in order to minimise the clock count. In general the implemented algorithm consists of two parts, see below, the implementation is visible from figure 3.

1 The first part is the Subbytes and Shiftrows operations, which is both performed at the same time, for each byte in the message. That is, the bytes are shifted and looked up in the S-Box in the same step.

2 The second part of the algorithm, is the mixcolumn and the add round key operations, which

```
roundKeyGeneration(round);
message[0] ^= roundKey[0];
message[1] ^= roundKey[1];
message[2] ^= roundKey[2];
message[3] ^= roundKey[3];
message[4] ^= roundKey[4];
message[5] ^= roundKey[5];
message[6] ^= roundKey[6];
message[7] ^= roundKey[7];
message[8] ^= roundKey[8];
message[9] ^= roundKey[9];
message[10] ^= roundKey[10];
message[11] ^= roundKey[11];
message[12] ^= roundKey[12];
message[13] ^= roundKey[13];
message[14] ^= roundKey[14];
message[15] ^= roundKey[15];
```

Figure 2: Pre-whitening implementation optimized for speed.

is likewise performed together for each message byte. Note that this part utilises 4 temporary registers, in order to enforce data consistency throughout the calculations.

2.5 The final round

The final round is actually implemented as the first part in the (n-1) round loop. That is, the Subbytes and Shiftrows is calculated in the same step. However, this time we need to add the round key as well. Thus yielding the implementation of Figure 4

```
while(round < 10)
{
  // Subbytes + Shiftrows
  message[0] = sBox[message[0]];
  message[4] = sBox[message[4]];
  message[8] = sBox[message[8]];
  message[12] = sBox[message[12]];

  temp = message[1];
  message[1] = sBox[message[5]];
  message[5] = sBox[message[9]];
  message[9] = sBox[message[13]];
  message[13] = sBox[temp];

  temp = message[2];
  temp1 = message[6];
  message[2] = sBox[message[10]];
  message[6] = sBox[message[14]];
  message[10] = sBox[temp];
  message[14] = sBox[temp1];

  temp = message[11];
  message[11] = sBox[message[7]];
  message[7] = sBox[message[3]];
  message[3] = sBox[message[15]];
  message[15] = sBox[temp];

  // MixColumns + AddRowKey
  roundKeyGeneration(round);
  temp = message[0];
  temp1 = message[1];
  temp2 = message[2];
  temp3 = message[3];
  message[0] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[0];
  message[1] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[1];
  message[2] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[2];
  message[3] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[3];

  temp = message[4];
  temp1 = message[5];
  temp2 = message[6];
  temp3 = message[7];
  message[4] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[4];
  message[5] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[5];
  message[6] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[6];
  message[7] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[7];

  temp = message[8];
  temp1 = message[9];
  temp2 = message[10];
  temp3 = message[11];
  message[8] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[8];
  message[9] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[9];
  message[10] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[10];
  message[11] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[11];

  temp = message[12];
  temp1 = message[13];
  temp2 = message[14];
  temp3 = message[15];
  message[12] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[12];
  message[13] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[13];
  message[14] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[14];
  message[15] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[15];

  // Prepare next round
  round = round + 1;
}
```

Figure 3: Implementation of the (n-1)-round loop, optimized for speed.

```
roundKeyGeneration(round);
message[0] = (sBox[message[0]]) ^ roundKey[0];
message[4] = (sBox[message[4]]) ^ roundKey[4];
message[8] = (sBox[message[8]]) ^ roundKey[8];
message[12] = (sBox[message[12]]) ^ roundKey[12];

temp = message[1];
message[1] = (sBox[message[5]]) ^ roundKey[1];
message[5] = (sBox[message[9]]) ^ roundKey[5];
message[9] = (sBox[message[13]]) ^ roundKey[9];
message[13] = (sBox[temp]) ^ roundKey[13];

temp = message[2];
temp1 = message[6];
message[2] = (sBox[message[10]]) ^ roundKey[2];
message[6] = (sBox[message[14]]) ^ roundKey[6];
message[10] = (sBox[temp]) ^ roundKey[10];
message[14] = (sBox[temp1]) ^ roundKey[14];

temp = message[11];
message[11] = (sBox[message[7]]) ^ roundKey[11];
message[7] = (sBox[message[3]]) ^ roundKey[7];
message[3] = (sBox[message[15]]) ^ roundKey[3];
message[15] = (sBox[temp]) ^ roundKey[15];
```

Figure 4: Implementation of the (n) iteration, optimized for speed.

3 AES-128 optimized for Code Size

The following section will illustrate tricks and exemplification upon how to optimise C-Code in the dimension of Code-Size. That is, by minimising the Program Memory consumption. Again, this optimisation is done on the AES-128 Cryptography Algorithm.

3.1 Summarise

As observed it was possible to do an astonishing $(1-(1888/2594))\cdot 100 = 27\%$ reduction in program memory compared to the Speed optimisation. However, this comes with a trade off. From table 3 it is readily seen, that the clock count of this optimisation strategy suffered great impact. If we compare this result to the speed-dimension, an $45490/21499 = 2.12$ factor difference, meaning that this programme executes more than twice as slow compared to the speed optimisation. Moreover, it is noticed that the RAM consumption merely changes between these two implementation strategies.

Optimization dimention	Value	Configuration Level
Program Memory	1888 Bytes	-O0
RAM Consumption	350 Bytes	-O0
Clock Cycles	45490 Clocks	-O0

Table 3: Optimisation for Program Memory on an ATmega16 Micro Controller using -O0 configuration

3.2 Round key generation

As for the previous implementation, the round key generation of this strategy consists of two parts, see below. Moreover, the function takes the round number as input parameter. The implementation is illustrated in figure 5. From the figure, it is seen how this approach utilises the usage of loops in order to achieve the same functionality. This is done, since loops reduces code size, both visually and in terms of program memory.

1 If round number is 0 then set round key equal to the key

2 If round number is not 0 then modify the previous round key according to Algorithm **??**.

3.3 Pre-whitening

It really becomes visual how loops reduces the code size, when an example like the pre-whitening implementation below, figure 6, is provided. What previously consisted of 16 lines of code is combined into only 4 lines, thus removing 1/4 of the visual complexity. Furthermore, from the implementation, line 4, it is seen that the pre-whitening procedure is used to load the input message into our global static AES state variable.

```
static void roundKeyGeneration(uint8_t i)
{
  if(i == 0)
  {
    for(int i = 0; i < 16; i++)
    {
      roundKey[i] = key[i];
    }
  }
  else
  {
    uint8_t temp[4] =
    {
      sBox[roundKey[13]] ^ rCon[i], sBox[roundKey[14]], sBox[roundKey[15]], sBox[roundKey[12]]
    };

    for(int i = 0; i < len; i++)
    {
      roundKey[i] ^= (i < 4) ? temp[i] : roundKey[i-4];
    }
  }
}
```

Figure 5: Round key implementation optimized for speed.

```
roundKeyGeneration(round);
for (int i = 0; i < len; i++)
{
  message[i] = in_message[i] ^ roundKey[i];
}
```

Figure 6: Round key implementation optimized for program memory.

3.4 The n-1 rounds

From figure 7 the implementation of the AES-128, n-1 round iterations is illustrated. Again, a loop has been utilized to minimise visual and memory complexity. Moreover, a observation is made in this implementation. If the strategy did not utilise the `temp[4]` byte array to store the result from the substitution and shift row operation, and thus simply recalculated them every time in-line in the Mix Column operations, the programme memory is increased to 2660 bytes, which is even more than the space optimisation implementation. Finally, it is worthwhile noting that the encryption of the very message and the round key, in this strategy, is provided in its own function, figure 8.

```
while(round < 10)
{
  for(int i = 0; i < 4; i++)
  {
        // Subbytes + Shiftrows + MixColumns
    temp[0] = sBox[message[(i*4) % 16]];
    temp[1] = sBox[message[(i*4+5) % 16]];
    temp[1] = sBox[message[(i*4+10) % 16]];
    temp[1] = sBox[message[(i*4+15) % 16]];
    temporaryState[i*4] = (x2(temp[0]) + (x2(temp[1]) ^ temp[1]) + temp[1] + temp[1]);
    temporaryState[i*4+1] = (temp[0] + x2(temp[1]) + (x2(temp[1]) ^ temp[1]) + temp[1]);
    temporaryState[i*4+2] = (temp[0] + sBox[temp[1]] + x2(temp[1]) + (x2(temp[1]) ^ temp[1]));
    temporaryState[i*4+3] = ((x2(temp[0]) ^ temp[0]) + temp[1] + temp[1] + x2(temp[1]));
  }
  // AddRowKey
  roundKeyGeneration(round);
  encrypt();
  round = round + 1;
}
```

Figure 7: The n-1, 9, rounds of subbytes, shiftrows, mixcolumns, and addkey operations of AES-128, optimised for code size.

```
static void encrypt()
{
  for (int i = 0; i < len; i++)
  {
    message[i] = temporaryState[i] ^ roundKey[i];
  }
}
```

Figure 8: The Add Key implementation of the AES-128, optimised for code size.

3.5 The final round

Figure 9, provides the strategy of the final round, that is the 10th round of the algorithm. It is readily seen, that the implementation utilises a loop in order to minimise the code complexity. In addition, this loop iterates over four bytes at a time. This implementation is chosen since it allows for a small speed optimisation. In specific this trick allows for us to do the shiftrows and the byte substitution operation concurrently.

```
// Subbytes + Shiftrows
for(int i = 0; i < 4; i++)
{
  // Subbytes + Shiftrows
  temporaryState[i*4] = sBox[message[(i*4) % 16]];
  temporaryState[i*4+1] = sBox[message[(i*4+5) % 16]];
  temporaryState[i*4+2] = sBox[message[(i*4+10) % 16]];
  temporaryState[i*4+3] = sBox[message[(i*4+15) % 16]];
}
// AddRoundKey
roundKeyGeneration(round);
encrypt();
```

Figure 9: The final round, optimised for code size.

4 AES-128 optimized for Ram Consumption

The final dimension of this study, is concerning minimising of RAM memory. This implementation is based upon the speed implementation in the beginning of this article. The difference however it to find in the introduction of FLASH Memory. That is in this exercise, the huge S-Box table used in the substitution bytes operation, will be moved to the FLASH memory, by using the `PROGMEM` interface. This is illustrated in figure 10.

4.1 Summarise

By storing the S-Box data table in the flash memory of the ATmega16 processor, the total ram size needed is reduced $(1-(60/350))\cdot = 82\%$, thus really minimising the ram consumption. But as expected this is slower than having the data available in the RAM. And from table 4 this reduce is really visible. Since the RAM optimisation is directly based on the speed implementation, we denote that by introducing FLASH memory, the execution time is growing a little less than 8000 clocks. Moreover, it is noticed that the Program Memory needed in this implementation grew approximately by a factor $5780/1888 = 3$, which is again expected since it the code now need interaction with the FLASH via the BUS.

Optimization dimention	Value	Configuration Level
Program Memory	5780 Bytes	-O0
RAM Consumption	60 Bytes	-O0
Clock Cycles	29355 Clocks	-O0

Table 4: Optimisation for speed on an ATmega16 Micro Controller using -O0 configuration

```
const uint8_t sBox[256] PROGMEM=
{
        0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
        0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
        0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
        0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
        0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
        0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
        0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
        0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
        0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
        0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
        0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
        0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
        0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
        0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
        0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
        0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
        0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
```

Figure 10: The S-Box is stored in the Flash Memory.