Software Evaluation of Grain-128AEAD for Embedded Platforms

Alexander Maximov[1] and Martin Hell[2]

[1] Ericsson AB, Lund, Sweden, alexander.maximov@ericsson.com
[2] Lund University, Lund, Sweden, martin.hell@eit.lth.se

Abstract. Grain-128AEAD is a stream cipher supporting authenticated encryption with associated data, and it is currently in round 2 of the NIST lightweight crypto standardization process. In this paper we present and benchmark software implementations of the cipher, targeting constrained processors. The processors chosen are the 8-bit (AVR) and 16-bit (MSP) processors used in the FELICS-AEAD framework. Both high speed and small code size implementations are targeted, giving us in total 4 different implementations. Using the FELICS framework for benchmarking, we conclude that Grain-128AEAD is competitive to other algorithms currently included in the FELICS framework. Our detailed discussion regarding particular implementation tricks and choices can hopefully be of use for the community when considering optimizations for other ciphers.

Keywords: Grain-128AEAD · stream cipher · software implementation · NIST · optimizations

1 Introduction

The stream cipher Grain-128AEAD is currently a round 2 candidate of the NIST lightweight crypto standardization process. Its specification is closely based on Grain-128a, introduced in 2011, which has, already for several years, been analyzed in the literature. To benefit from the maturity of the Grain family, the design of Grain-128AEAD is very closely based on Grain-128a, with as small changes as possible. This allows us to argue for the security of the cipher based on previous results on Grain-128a.

Grain-128a is in turn based on Grain v1 and Grain-128, which have both been extensively analyzed, providing much insight into the security of the design approach. All Grain stream ciphers also allow the throughput to be increased by adding additional copies of the Boolean functions involved.

Grain-128AEAD can be very suitable in Internet of things (IoT) and embedded systems. Strong advantages of Grain-128AEAD and its precedent versions can be seen in its industrial relevance.

The design of Grain-128AEAD has previously been given in [HJM+19a] and [HJM+19b], while details related to hardware implementations are given in [SHSK19].

In this paper, we give details of software implementations targeting constrained processors, namely the 8-bit and 16-bit processors used in the FELICS-AEAD framework [CdSGB19]. This, together with the hardware implementation results in [SHSK19], provides an understanding of how Grain-128AEAD performs on constrained devices, both in software and in hardware.

In [BMA+18], it was shown that lightweight stream ciphers are typically more suitable than lightweight block ciphers for energy optimization when encrypting longer messages, in particular when the speed can be increased at the expense of moderate extra hardware.

Thus, in these cases, Grain-128AEAD can provide authenticated encryption with low energy consumption. Our implementations targeting embedded processors show that also short messages can be handled efficiently by Grain-128AEAD.

2 Algorithm specification

The full specification of the algorithm can be found in [HJM+19b]. Here, we only give a very brief overview of the overall design in order to introduce some of the challenges we are facing when implementing the cipher on constrained embedded processors.

Grain-128AEAD consists of two main building blocks. The first is a pre-output generator, which is constructed using a Linear Feedback Shift Register (LFSR), a Non-linear Feedback Shift Register (NFSR) and a pre-output function, while the second is an authenticator generator consisting of a shift register and an accumulator.

2.1 Building blocks and functions

The pre-output generator generates a stream of pseudo-random bits, which are used for encryption and the authentication tag. It is depicted in Fig. 1. The content of the 128-bit

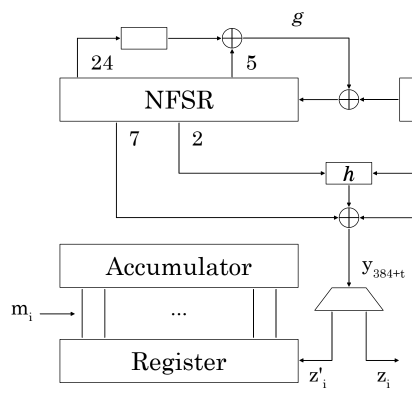

Figure 1: An overview of the building blocks in Grain-128AEAD.

LFSR is denoted $S_t = [s_0^t, s_1^t, \dots, s_{127}^t]$ and the content of the 128-bit NFSR is similarly denoted $B_t = [b_0^t, b_1^t, \dots, b_{127}^t]$. These two shift registers represent the 256-bit state of the pre-output generator.

The update function of the LFSR is given by

$$s_{127}^{t+1} \quad = \quad s_0^t + s_7^t + s_{38}^t + s_{70}^t + s_{81}^t + s_{96}^t = \mathcal{L}(S_t),$$

and update function for the NFSR is given by

$$\begin{aligned} b_{127}^{t+1} &= s_0^t + b_0^t + b_{26}^t + b_{56}^t + b_{91}^t + b_{96}^t + b_3^t b_{67}^t + b_{11}^t b_{13}^t + b_{17}^t b_{18}^t + b_{27}^t b_{59}^t + b_{40}^t b_{48}^t \\ &+ b_{61}^t b_{65}^t + b_{68}^t b_{84}^t + b_{22}^t b_{24}^t b_{25}^t + b_{70}^t b_{78}^t b_{82}^t + b_{88}^t b_{92}^t b_{93}^t b_{95}^t = s_0^t + \mathcal{F}(B_t). \end{aligned}$$

Nine state variables are taken as input to a Boolean function $h(x)$,

$$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8,$$

where the variables $x_0, \dots, x_8$ correspond to, respectively, the state variables b_{12}^t, s_8^t, s_{13}^t, s_{20}^t, b_{95}^t, s_{42}^t, s_{60}^t, s_{79}^t and s_{94}^t.

The output of the pre-output generator, is then given by the pre-output function

$$y_t = h(x) + s_{93}^t + \sum_{j\in\mathcal{A}} b_j^t,$$

where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$.

The authenticator generator consists of a shift register, storing the most recent 64 odd bits from the pre-output, and an accumulator. Both are of size 64 bits. We denote the content of the accumulator at instance i as $A_i = [a_0^i, a_1^i, \ldots, a_{63}^i]$. Similarly, the content of the shift register is denoted $R_i = [r_0^i, r_1^i, \ldots, r_{63}^i]$.

2.2 Key and nonce initialization

Denote the key bits as k_i, $0 \le i \le 127$ and the nonce (IV) bits as IV_i, $0 \le i \le 95$. Then the state is initialized as follows. The 128 NFSR bits are loaded with the bits of the key $b_i^0 = k_i$, $0 \le i \le 127$ and the first 96 LFSR elements are loaded with the nonce bits, $s_i^0 = IV_i$, $0 \le i \le 95$. The last 32 bits of the LFSR are filled with 31 ones and a zero, $s_i^0 = 1, 96 \le i \le 126,\ s_{127}^0 = 0$. Then, the cipher is clocked 256 times, feeding back the pre-output function and XORing it with the input to both the LFSR and the NFSR, i.e.,

$$\begin{aligned} s_{127}^{t+1} &= \mathcal{L}(S_t) + y_t, \quad 0 \le t \le 255, \\ b_{127}^{t+1} &= s_0^t + \mathcal{F}(B_t) + y_t, \quad 0 \le t \le 255. \end{aligned}$$

Once the pre-output generator has been initialized, the authenticator generator is initialized by loading the register and the accumulator with the pre-output keystream as

$$a_j^0 = y_{256+j} \quad \text{and} \quad r_j^0 = y_{320+j}, \qquad 0 \le j \le 63.$$

When the register and the accumulator are initialized, the key is simultaneously shifted into the LFSR,

$$s_{127}^{t+1} = \mathcal{L}(S_t) + k_{t-256}, \quad 256 \le t \le 383,$$

while the NFSR is updated as

$$b_{127}^{t+1} = s_0^t + \mathcal{F}(B_t), \quad 256 \le t \le 383.$$

Thus, when the cipher has been fully initialized the LFSR and the NFSR states are given by S_{384} and B_{384}, respectively, and the register and accumulator are given by R_0 and A_0, respectively. The initialization procedure is summarized in Fig 2.

2.3 Operating mode

For a message $\boldsymbol{m}$ of length L, denoted $m_0, m_1, \ldots, m_{L-1}$, set $m_L = 1$ as padding in order to ensure that $\boldsymbol{m}$ and $\boldsymbol{m}\|0$ have different tags.

After initializing the pre-output generator, the pre-output is used to generate keystream bits z_i for encryption and authentication bits z_i' to update the register in the accumulator generator. The keystream is generated as

$$z_i = y_{384+2i},$$

i.e., every even bit (counting from 0) from the pre-output generator is taken as a keystream bit. The authentication bits are generated as

$$z_i' = y_{384+2i+1},$$

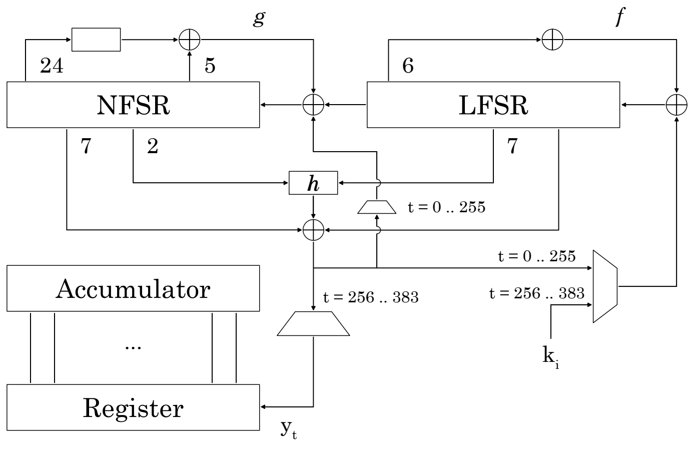

Figure 2: An overview of the initialization of Grain-128AEAD. Note that, in hardware, the accumulator initialization is realized by first loading 64 pre-output bits into the register, followed by moving them to the accumulator.

i.e., every odd bit from the pre-output generator is taken as an authentication bit. The message is encrypted as

$$c_i = m_i \oplus z_i, \quad 0 \le i < L.$$

The accumulator is updated as

$$a_j^{i+1} = a_j^i + m_i r_j^i, \qquad 0 \le j \le 63, \quad 0 \le i \le L,$$

and the shift register is updated as

$$r_{63}^{i+1} = z_i' \quad \text{and} \quad r_j^{i+1} = r_{j+1}^i, \qquad 0 \le j \le 62.$$

2.3.1 Using the NIST API

For the specific case of the NIST software API, the input (ad, ad length, message, message length) is mapped to a string m' as

$$m' = \text{Encode(ad length)}||\text{ad}||\text{m}||\text{0x80},$$

where Encode() $= y$ denotes the message length encoded in the DER format as used by, e.g., X.509. If the first byte in y starts with a 0, the remaining 7 bits contain an encoding of the number of bytes in the associated data (up to 127 bytes). If the first byte in y starts with a 1, the remaining 7 bits are, instead, an encoding of the number of forthcoming bytes that are used to describe the length (in bytes) of the associated data. In y, this first byte is then followed by the bytes giving the length.

3 Software implementations on constrained processors

In this section, we discuss software implementations targeting two constrained processors. The provided discussion and implementation details serve mainly two purposes. First, some of our optimizations might be useful also for other algorithms, and second, by giving the details of the rationale behind the optimizations, others might more easily be able to further optimize the code. Re-use of specific optimizations and transparency of implementations allows more fair comparisons between algorithms. A fair comparison obviously also require comparable results in terms of the processors used as implementation targets. Here, we have decided to use the FELICS-AEAD framework for the benchmarking [CdSGB19]. The framework defines three different processors, targeting resource constrained devices. We have made optimized implementations for the two smallest of these processors, namely the AVR ATmega 128 and the MSP430F1611. The two processors are very different in several aspects. A brief comparison is given in Table 1.

Table 1: Some details and comparison of AVR and MSP targets

	AVR target [Atm06]	**MSP target** [Ins11, Ins06]
	CPU Characteristics	
Platform	8-bit RISC (Harvard)	16-bit RISC (Von Neumann)
Version	AVR ATmega 128 @ 16MHz	MSP430F1611 @ 8MHz
Flash / SRAM	128 KB / 4 KB	48 KB / 10 KB
Registers	8-bit R0-R31 where R26-R31 are for three 16-bit pointers	16-bit R0-R15 where R0-R3 are reserved for Control Regs

It can be noted that both processors are little-endian machines, which will be used in the implementations. The main properties, as suggested by FELICS-AEAD, are the code size, the RAM consumption, and the execution time. We have implemented Grain-128AEAD in four different versions: **AVR-Small** – small code size for AVR; **AVR-Fast** – fast execution time for AVR; **MSP-Small** – small code size for MSP; **MSP-Fast** – fast execution time for MSP. [1]

3.1 Implementation details

Neither the AVR, nor the MSP, is suitable for bit-shifts. They can only perform a 1-bit shift to the left or to the right, with or without carry propagation, or a swap of a half-word (2x4-bit swap in AVR and 2x8 bit swap in MSP). This means that in the implementation of Grain we must take care to reduce the number of bit-shifts needed as much as possible.

The two processors differ in several critical aspects and a programmer must take the specific properties and limitations into account. In the MSP, we can only use 11 16-bit registers for Grain logic, while 1 register is reserved for storing the pointer to the state data. This requires a more careful treatment than in the AVR case, where we can use 26 8-bit registers for logic (or, 13 2x8-bit registers), and the Z-registers (R30-R31) for storing the pointer to the Grain state. In the MSP, we should reduce the usage of instructions with immediate constants, and also limit memory read/write as much as possible. In addition, the usage of the stack should be reduced as far as possible, since the RAM$\leftrightarrow$registers transfer time is rather expensive (latency 3-4). An overview of the latency and code size in the MSP can be found in Table 2. In the AVR, this problem is not present as most of the instructions have latency 1 and for RAM$\leftrightarrow$registers transfers it is only 2.

Table 2: Code size and the number of cycles for MSP430F1611, based on input arguments.

src→dst	#cycles	code size	Example
reg→reg	1	2	MOV R4, R5
reg→mem	4	4	AND R4, 6(R5)
mem→reg	3	4	XOR 6(R5), R4
mem→mem	6	6	ADD 6(R5), 3(R4)
const→reg	2	4	MOV #10, R5
const→mem	5	6	XOR #10, 3(R5)

[1]All four implementations, and also some alternative variants, can be found in the FELICS virtual machine accessible at https://www.cryptolux.org/index.php/FELICS

Several implementation optimizations have been considered. Here, we provide an overall description of our implementation approach, together with code samples to highlight some particular implementation choices.

3.1.1 Data structures and main sub-routines

For the 8-bit AVR implementation, the state of Grain is defined as

```
typedef struct GrainState_st
{ uint8_t lfsr[16], nfsr[16], A[8], R[8], z1;
} GrainState;
```

and for the 16-bit variant, it is

```
typedef struct GrainState_st
{ uint16_t lfsr[8], nfsr[8], A[4], R[4], z1;
} GrainState;
```

In both cases, the order of data in the structure GrainState_st is highly important, as will be seen later. The members of the structure correspond to the state of LFSR, NFSR, Accumulator, and Register in the design description of Grain-128AEAD. 1 (or 2) byte of extra information (.z1) is added to the structure GrainState_st for efficiency reasons. This contains the odd bits from the keystream and will be used when authenticating each message byte.

In the implementation of both the 8- and 16-bit variants, we utilize four sub-routines.

```
uint8/16_t grain_update(GrainState * g);
void       grain_auth  (GrainState * g, uint8_t msg);
uint8_t    grain_getz  (GrainState * g);
void       grain_encdec(uint8_t *state, uint8_t *message,
                        uint32_t message_length, uint8_t mask);
```

The sub-routines implement the following tasks.

- grain_update() performs LFSR and NFSR updates, computes and returns the y value. For the AVR platform this is an 8-bit value and for the MSP it is 16 bits.

- grain_getz() calls the function grain_update() 1 (MSP) or 2 (AVR) times and splits the returned 16 bits into two bytes, z_0 and z_1, containing even and odd bits. Then 8 odd bits are stored in the state of Grain in .z1, and is used later in the authentication, while the 8 even bits serve as a keystream byte, which is returned to the caller.

- grain_auth() receives a single byte of a message, and updates the "Accumulator" .A and the "Register" .R given the message byte msg and the 8 (odd) bits saved in the Grain state as .z1.

- grain−encdec() is a combined function for both encryption and decryption, since the only difference is the order of the authentication and the XORing of the input message with the keystream.

3.1.2 Implementing the FELICS API

The Grain-128AEAD implementation in the FELICS-AEAD framework must follow a certain pre-defined API, which has six functions, ProcessPlaintext(), ProcessCiphertext(), Finalize (), TagGeneration(), Initialize (), and ProcessAssociatedData().

In this section, we discuss the implementation of the functions for this API. Using the above-mentioned sub-routines, we get a very simple implementation of the first four API functions.

```c

void ProcessPlaintext(uint8_t *state, uint8_t *message, uint32_t
    message_length)
{ grain_encdec(state, message, message_length, 0xff); }

void ProcessCiphertext(uint8_t *state, uint8_t *message, uint32_t
    message_length)
{ grain_encdec(state, message, message_length, 0x00); }

void Finalize(uint8_t *state, uint8_t *key)
{   /* Do nothing */  }

void TagGeneration(uint8_t *state, uint8_t *tag)
{ GrainState * g = (GrainState *)state;
  uint8_t i;
  for(i=0; i<4; ++i)
    ((uint16_t*)tag)[i] = g->A[i] ^ g->R[i];
}
```

Note that the 32-bit length of the message (uint32_t message_length) is not suitable for 8 and 16 bit platforms. Instead of introducing a 32-bit counter, we use the message_length variable directly and decrement it while incrementing the data pointer (uint8_t * message). This is done in the sub-routines grain_encdec() and ProcessAssociatedData(), described later.

The API function Initialize () is quite straight-forward. It only differs between 8- and 16-bit implementations in the size of the returned y from the sub-routine grain_update(). For the 16-bit version it is given as follows.

```c
1  void Initialize(uint8_t *state, const uint8_t *key, const uint8_t *nonce)
2  {
3    GrainState * g = (GrainState *)state;
4    uint8_t i;
5
6    memcpy(g->nfsr, key, 16);
7    memcpy(g->lfsr, nonce, 12);
8    g->lfsr[6] = 0xffff;
9    g->lfsr[7] = 0x7fff;
10
11   for(i=0; i<16; ++i)
12   { uint16_t y = grain_update(g);
13     g->lfsr[7] ^= y;
14     g->nfsr[7] ^= y;
15   }
16
17   for(i=0; i<8; ++i)
18   { g->A[i] = grain_update(g);
19     g->lfsr[7] ^= ((const uint16_t*)key)[i];
20   }
21 }
```

Here we take advantage of the order of .A[] and .R[] in the structure GrainState_st. On line 18 above, for i>=4 it will actually update g->R[i-4] with no extra expense in code size and time.

The remaining function from the FELICS-AEAD API is ProcessAssociatedData(). The main complication there is the DER encoding of the message length, and since the length in the API is a 32-bit integer, this also has to be taken into consideration. The smallest and most efficient DER encoding is a byte-oriented solution, and here we utilize the fact that AVR and MSP are little-endian machines. The 16-bit implementation then looks as follows.

```c
void ProcessAssociatedData(uint8_t *state, uint8_t *associatedData, uint32_t
     associated_data_length)
{
  GrainState * g = (GrainState *)state;
  uint8_t der[5], k, der_len;
  *(uint32_t*)(der + 1) = associated_data_length;

  der[0] = 0x80;
  for(der_len=4; !der[der_len]; --der_len);

  /* Alt1: if(!((der[1] & 0x80) | (der_len>>1))) */
  /* Alt2: if(!((der_len>>1) | (der[1]>>7))) */
  /* Alt3: if(!((der_len & 0xfe)|(der[1] & 0x80))) */
  if((der_len<=1) && (der[1]<128))
  { der[0] = der[1];
    der_len = 0;
  }
  else
    der[0] |= der_len;

  for(k=0; k <= der_len; k++)
  { grain_getz(g);
    grain_auth(g, der[k]);
  }

  while(associated_data_length--)
  { grain_getz(g);
    grain_auth(g, *(associatedData));
    associatedData++;
  }
}
```

It can be noted that the implementations for both 8- and 16-bit targets are byte oriented in terms of how we process the message and the authentication data. This removes the need for handling odd lengths. Moreover, for the 16-bit case, having 16-bit keystream chunks is not justified since in that case we would have to update the LFSR/NFSR with 32 clocks at a time. Thus, for both the AVR and MSP architectures, the approach to handle the message byte-wise seems most efficient.

3.1.3 Sub-routine grain_encdec()

The implementation of the combined encryption/decryption is very straight-forward, and does not contain any particular processor oriented optimizations. Recall that the mask is set to 0xff for encryption, meaning that the message before XORing with the keystream is sent to the authentication sub-routine. For decryption, the mask is instead 0x00, resulting in a decrypted message on line 6, before the plaintext is used for authentication.

```
1  void grain_encdec(uint8_t *state, uint8_t *message,
2                   uint32_t message_length, uint8_t mask)
3  { GrainState * g = (GrainState *)state;
4    while(message_length--)
5    { uint8_t z0 = grain_getz(g);
6      *message ^= z0 & ~mask;
7      grain_auth(g, *message);
8      *message ^= z0 & mask;
9      message++;
10   }
11 }
```

3.1.4 Sub-routine grain_getz()

As noted, for the 8-bit AVR target, the function grain_getz() must call grain_update() two times in order to receive 16 bits of y. In the 16-bit implementation the call to grain_update() is only performed once since grain_update() then returns 16 bits of y.

The next step is to deinterleave the received 16 bits into two bytes, z_0 and z_1, containing even and odd bits, respectively.

Deinterleaving may be done in several ways. For the 8-bit case, we can first deinterleave the 8 bits in each of the 2 bytes of y independently, then mix the results to further get the full split in z_0, z_1.

```
// AVR compiler will convert this into a single instruction 'swap'
static inline uint8_t SWAP(uint8_t x)
{ return (x>>4)|(x<<4); }

static uint8_t deinterleave(uint8_t x)
{ uint8_t tmp;
  tmp = (x ^ (x >> 1)) & 0x22; x ^= tmp ^ (tmp << 1);
  tmp = (x ^ (x >> 2)) & 0x0c; x ^= tmp ^ (tmp << 2);
  return x;
}

uint8_t grain_getz(GrainState * g)
{ uint8_t r0 = deinterleave(grain_update(g));
  uint8_t r1 = SWAP(deinterleave(grain_update(g)));
  uint8_t t = (r0 ^ r1) & 0xf0;
  g->z1 = SWAP(r1 ^ t);
  return r0 ^ t;
}
```

This approach gives a small code. However, it is not very fast since it requires several bit shifts. A faster approach is to use a lookup table, 256 bytes, which deinterleaves a single byte, where even bits are collected as low 4 bits of the result and odd bits are placed as high 4 bits of the resulting byte. This helps to speed up the execution but requires larger code size.

For the 16-bit MSP case we implemented the deinterleaving directly on the 16-bit response y, as follows.

```
uint8_t grain_getz(GrainState * g)
{ uint16_t tmp, x = grain_update(g);
  tmp = (x ^ (x>>1)) & 0x2222; x ^= tmp ^ (tmp<<1);
  tmp = (x ^ (x>>2)) & 0x0c0c; x ^= tmp ^ (tmp<<2);
  tmp = (x ^ (x>>4)) & 0x00f0; x ^= tmp ^ (tmp<<4);
  g->z1 = x >> 8;
  return (uint8_t)x;
}
```

For the fast case on the MSP target we also implemented this function in inline assembly, so that we could ignore the carry flag cleanups, optimizing this function even further.

3.1.5 Sub-routine grain_auth()

Authenticating one message byte in Grain-128AEAD will require a loop of in total 64 steps. For the AVR processor, reading from and writing to RAM is not very expensive, so an efficient implementation can be achieved as follows.

```
void grain_auth(GrainState * g, uint8_t msg)
{ uint8_t i;
  for(i=0; i<8; ++i)
  { uint8_t j, mask = -(msg & 1);
    msg >>=1;
    for(j=0; j<8; ++j)
    { g->A[j] ^= g->R[j] & mask;
      g->R[j] = (uint8_t)((*(uint16_t*)(g->R + j))>>1);
    }
    g->z1 >>= 1;
  }
}
```

Note that we here also take advantage of the fact that `.z1` is located in the data structure right after `.R[]`, so that we avoid buffer overflow.

A similar implementation on the 16-bit MSP target is not efficient due to the comparatively slow instructions for reading from and writing to the memory. Here, it is more efficient to load the state of `.A[]` and `.R[]` into registers, operate on the registers, and then store them back at the very end of the procedure. In addition, inline assembly can be used in shifting `.R[]` by 1, since there we can effectively use the instruction `rrc` that gets in and pushes out the carry value. Thus, the shifting of $\leftarrow R \leftarrow z_1$ is only 5 instructions. The overall optimization for the 16-bit platform can be implemented as follows.

```
void grain_auth(GrainState * g, uint8_t msg)
{
  uint16_t r0 = g->R[0], r1 = g->R[1], r2 = g->R[2], r3 = g->R[3], z=g->z1;
  uint16_t a0 = g->A[0], a1 = g->A[1], a2 = g->A[2], a3 = g->A[3], i;

  for(i=0; i<8; ++i)
  { uint16_t j, mask = -(uint16_t)(msg & 1);
    msg >>= 1;
__asm__ __volatile__(
  "mov  %3, %10    \t\n"
  "and  %9, %10    \t\n"
  "xor  %10, %8    \t\n"
  "mov  %2, %10    \t\n"
  "and  %9, %10    \t\n"
  "xor  %10, %7    \t\n"
  "mov  %1, %10    \t\n"
  "and  %9, %10    \t\n"
  "xor  %10, %6    \t\n"
  "mov  %0, %10    \t\n"
  "and  %9, %10    \t\n"
  "xor  %10, %5    \t\n"
  "rrc  %4         \t\n"
  "rrc  %3         \t\n"
  "rrc  %2         \t\n"
  "rrc  %1         \t\n"
  "rrc  %0         \t\n"
  : "+r"(r0), "+r"(r1), "+r"(r2), "+r"(r3), "+r"(z), "+r"(a0), "+r"(a1), "+r
    "(a2), "+r"(a3), "+r"(mask), "=r" (j) : );
  }

  g->R[0] = r0; g->R[1] = r1; g->R[2] = r2; g->R[3] = r3;
  g->A[0] = a0; g->A[1] = a1; g->A[2] = a2; g->A[3] = a3;
}
```

Note that in both approaches above we do `mask = -(uint16_t)(msg & 1)` which effectively generates the mask `0x0000` or `0xffff` in just 4 instructions (e.g., `mov`, `and`, `inv`, `inc`), based on the bit of the message. This approach is branchless and executes in constant time independently on the message. This helps protecting against message dependent side-channel timing attacks on the authentication part of the implementation.

3.1.6 Sub-routine grain_update()

The update sub-routine function is the main core of the Grain-128AEAD algorithm. The function clocks the two registers 8 or 16 times respectively for the AVR and MSP targets, and returns 8 or 16 bits of y. This is the main function to consider for speed and area optimizations, and where most effort should be done. As will also be shown, we can additionally gain a significant code size reduction by carefully analyzing the function. This section will discuss several optimization approaches for this function and the different options that were considered.

Let a "word" be an 8-bit integer for the AVR, and a 16-bit integer for the MSP. Further, a "double-word" is a 16-bit integer for the AVR, and a 32-bit integer for the MSP. Let us define an LFSR/NFSR "double-word" at a *byte offset* `i` as follows:

```
#define LF(i) (*(uint16/32_t*)((uint8_t*)g->lfsr + i))
```

```
#define NF(i) (*(uint16/32_t*)((uint8_t*)g->nfsr + i))
```

where we use the type conversion to `uint16_t` for the 8-bit AVR and `uint32_t` for the MSP.

In the function, the main goal is to compute a new value for the LFSR, NFSR, and the value of y. These will be referred to as `uint8/uint16_t ln, nn, y`, respectively. Now, `ln, nn, y` can be expressed in terms of the `LF`, `NF` macros. For the 16-bit case this can be done as follows.

```
uint16_t grain_update(GrainState * g)
{
  uint16_t ln, nn, y;

  ln = LF(0) ^ LF(12) ^ (LF(0)>>7) ^ (LF(4)>>6) ^ (LF(8)>>6) ^ (LF(10)>>1);

  y  = (LF(1)>>5) & (LF(2)>>4);
  y ^= (LF(7)>>4) & (LF(9)>>7);
  y ^= (NF(11)>>7) & (LF(5)>>2) ^ (NF(11)>>1);
  y ^= (NF(1)>>4) & (NF(11)>>7) & (LF(11)>>6);
  y ^= (NF(1)>>4) & LF(1);
  y ^= (LF(11)>>5) ^ (NF(0)>>2) ^ (NF(1)>>7) ^ (NF(4)>>4);
  y ^= (NF(5)>>5) ^ NF(8) ^ (NF(9)>>1) ;

  nn  = LF(0) ^ NF(0) ^ NF(7) ^ NF(12) ^ (NF(5) & NF(6));
  nn ^= NF(11) & (NF(11)>>4) & (NF(11)>>5) & (NF(11)>>7) ^ (NF(11)>>3);
  nn ^= (NF(0)>>3) & (NF(8)>>3);
  nn ^= (NF(1)>>3) & (NF(1)>>5);
  nn ^= (NF(2)>>1) & (NF(2)>>2);
  nn ^= (NF(3)>>3) & (NF(7)>>3);
  nn ^= (NF(3)>>2);
  nn ^= (NF(7)>>5) & (NF(8)>>1);
  nn ^= (NF(8)>>4) & (NF(10)>>4);
  nn ^= (NF(2)>>6) & NF(3) & (NF(3)>>1);
  nn ^= (NF(8)>>6) & (NF(9)>>6) & (NF(10)>>2);

  memcpy(g->lfsr, g->lfsr + 1, 30);
  g->lfsr[7] = ln;
  g->nfsr[7] = nn;
  return y;
}
```

While this is in general a rather efficient way of updating bit-oriented shift registers, a problem with the above code is that it needs many shifts of double-words. A single shift by 1 to the right (left) takes at least 2 instructions, i.e., just a single term `NF(11)>>7` would normally take 14 instructions, excluding the time of loading the double-word `NF(11)`.

Another problem is that every call to the macros `LF`, `NF` would mean loading the 2 words from RAM into 2 registers. This may be acceptable for the AVR case, but in the MSP it becomes quite expensive.

So, our speed optimization goal is to

- reduce the number of shifts required, and
- reduce the number of register to/from RAM transfers.

The latter part becomes particularly challenging on the MSP target as there we only have 11 registers at our disposal.

For reducing the number of shifts, one approach could be to switch the order of operations. As an example,

```
  nn ^= ((NF(0)>>3) & (NF(8)>>3)) ^ ((NF(3)>>3) & (NF(7)>>3));
```

requires 4 "double-word" shifts, taking 2(^)+2(&)+4*6(>>3)=28 instructions. Instead,

```
  nn ^= (NF(0) & NF(8)) ^ (N(3) & NF(7)) >>3;
```

would require only 13 instructions. Another approach is that shifts can be nested. For example

```
nn ^= ((NF(1)>>3) & (NF(1)>>5)) ^ ((NF(2)>>1) & (NF(2)>>2)) ^ ((NF(3)>>3)
  & (NF(7)>>3)) ^ (NF(3)>>2);
```

requires 45 instructions, while the nested expression needs only 25 instructions:

```
nn ^= (((((NF(1)>>2) & NF(1)) ^ (NF(3) & NF(7)))>>1) ^ NF(3))>>1) ^ ((NF
  (2)>>1) & NF(2)))>>1;
```

These ideas can be applied to the AVR by utilizing 21 registers, but they do not work nicely for the MSP, where we only have 11 registers. Loading of the LFSR state already occupies 7 16-bit registers, and thus we only have 4 remaining registers for operations and to keep intermediate values during the evaluation of `ln`, `nn`, `y`.

A better approach for the MSP is to load the LFSR state into 7 registers – the last 16 bits are not used in the update function and thus we have 4 extra registers to work with. Then we are shifting these 7 registers by 1 bit in parallel. If some expression contains an `NF(x)>>1`, `LF(x)>>1`, we pick that value from the shifted state almost directly, and use it in the expression. If there is an `&` operator, then we have to save the value somewhere and AND it with the next operand as soon as the state will be shifted by the right amount. Then we repeat shifting the 7 state registers by 1 yet another 7 times until all arguments in the functions are met.

However, since registers in MSP are 16-bit long, there are two possible situations that need to be handled somewhat differently. Assume we are shifting the state to the right; if the needed argument in `NF/LF(x)` has even `x`, then the value is achieved directly from the register `R(x/2)`. However, if `x` is odd, then the value is spanned over two 16-bit registers `(R(x/2+1)|R(x/2))>>8`, which need to be extracted.

The following assembly code is used for such an extraction, where `B|A` is the 32-bit value in two 16-bit registers `A` and `B`, and where we need to extract the middle 16 bits into the register `R`. This can be either a third register or the register `B`. The both fastest and smallest code use only 4 instructions on the MSP target.

```
#define get8(B, A, R)          \
    "mov.b  " B " , " R "  \n\t"\
    "xor.b  " A " , " R "  \n\t"\
    "xor    " A " , " R "  \n\t"\
    "swpb   " R "          \n\t"
```

The above ideas provide an efficient implementation of the update function, but it is not necessarily very small. If code size is crucial and speed is of much less importance, we here outline an approach for minimizing the code size, but at the expence of speed.

By the above, it is clear that the update function is implemented as a sequence of AND and XOR operations, where the arguments are basically the bit-offsets from the beginning of the `GrainState_st` structure. Each invocation of `NF(i)/LF(i)` is a number of assembly instructions, where each instruction is 2 bytes long. Based on this observation the following approach can be taken to dramaticaly reduce the code size:

1. Encode the sequence of the functions' evaluation steps in a shorter form, say as a vector of bytes. Each byte points to the bit-offset of the argument in some well-defined order of the evaluation of an expression.

2. Encode the sequence of operations. Here we introduce `result` as being the result of an expression evaluation, and `product` being the intermediate AND-product of the input arguments. The sequence of the evaluation is thus binary, where '1' means that the argument has to be ANDed to the product, and '0' means that the product must be XORed to the total `result`. The values are initialized as `result=0`, `product=−1`, and when the `product` is XORed to the `result`, it is then initialized again as `product=−1`.

3. Write a mini-RISC CPU that can process the given program with the provided arguments.

This approach effectively substitutes the code for each &/^ LF/NF(i)>>j operation by just 1 byte of the encoded command with one argument to the mini-RISC CPU implemented within the Grain code itself. As an exampe, for the 16-bit case, that implementation is done as follows.

```
static const uint8_t program16[54] =
{ // program for LFSR update
  0x60, 0x51, 0x46, 0x26, 0x07, 0x00,
  // program for NFSR update
  /*0x00,*/ 0x80, 0x9a, 0xb8, 0xdb, 0xe0, 0xd8, 0xdc, 0xdd, 0xdf, 0x96, 0x98
    , 0x99, 0xc6, 0xce, 0xd2, 0x83, 0xc3, 0x8b, 0x8d, 0x91, 0x92, 0x9b, 0xbb
    , 0xa8, 0xb0, 0xbd, 0xc1, 0xc4, 0xd4,
  // program for y
  0x8c, 0x08, 0x0d, 0x14, 0xdf, 0x2a, 0x3c, 0x4f, 0x8c, 0xdf, 0x5e, 0x5d, 0
    x82, 0x8f, 0xa4, 0xad, 0xc0, 0xc9, 0xd9
};

// Mini-RISC CPU
static uint16_t execute_program(const uint16_t * data, uint16_t command,
    uint16_t pc, uint16_t pc_end)
{ uint16_t result = 0x0000, product = 0xffff;
  for(; pc < pc_end; ++pc, command>>=1)
  { uint16_t offset = program16[pc]>>4;
    uint16_t shift = program16[pc] & 15;
    product &= (uint16_t)(*((uint32_t*)(data + offset)) >> shift);
    if(command & 1) continue;
    result ^= product;
    product = 0xffff;
  }
  return result;
}

uint16_t grain_update(GrainState * g)
  uint16_t nn, y, i;
  nn  = execute_program(g->lfsr, 0x6dc0, 5, 21);
  nn ^= execute_program(g->lfsr, 0x1555, 21, 35);
  y   = execute_program(g->lfsr, 0x0355, 35, 54);
  g->nfsr[0] = execute_program(g->lfsr, 0x0000, 0, 6);

  memcpy(g->lfsr, g->lfsr + 1, 30);
  g->nfsr[7] = nn;
  return y;
}
```

The “program” for nn has 30 instructions, and we have to call the mini-RISC CPU twice. We also utilize the fact that after 16 shifts of the command 0x0355 it will produce only zeroes, thus normally the processing will XOR all the arguments to the result. Therefore, the computation of y can be done with only a single call to the processor while there are (54-35)=19 commands being executed.

3.2 Benchmarking results

The FELICS-AEAD framework defines several scenarios. One set of scenarios targets the use case given by IEEE 802.15.4, which has a standard packet size of 127 bytes, including at most 25 byte header. Assuming a 16 byte authentication tag, this allows for 86 bytes of plaintext. The other set of scenarios is based on IPv6 with a MTU of 1280 bytes. This includes a fixed 40 byte header. Thus, there is a 1224 byte plaintext (since 16 bytes are used for the MAC). Here, we focus on the scenarios with authenticated encryption.

- Scenario 1c. Authenticated encryption of 86 bytes of payload and 25 bytes of header (associated data);
- Scenario 2c. Authenticated encryption of 1224 bytes of payload and 40 bytes header (associated data).

At the time of writing, very few ciphers are implemented in the FELICS-AEAD framework. These are ACORN, AES-GCM, ASCON, Ketje-Jr, and NORX. Of these, ASCON, is also in round 2 of the NIST LWC standardization process. Each cipher comes with a few different implementations and, in addition, different compiler options are used. The best performing implementation/option for each scenario is used in the comparison. In particular, the comparison with AES-GCM is of interest since a LWC cipher is expected to perform better than AES in the constrained environments.

The simulation results given in Table 3 are divided into 4 targets, namely minimizing code size and minimizing total time for AVR and MSP, respectively. For each target, we also provide alternative (Alt.) choices that are next best for this target (code size or total time). The idea is to show that a small price in the primary metric can lead to a significantly better performance in other metrics. For example, in "AVR-Small" we can see that (Alt.) NORX has a very small increase in code size but a much better performance in terms of the total time.

RAM usage. Most of the ciphers have a relatively small and stable usage of RAM for the state and stack, except Ketje-Jr whose stack usage heavily depends on the message length (about 1350 bytes for scenario 2c). On the smaller end, Grain and ACORN have state and stack RAM usage of about 100 bytes, where Grain uses the least RAM resources among all of the presented ciphers (83-94 bytes, in different scenarios/targets).

AVR/MSP-Small. Grain outperforms all the presented algorithms regarding the code size in all 4 target groups. On AVR, Grain has the smallest code size of 1100 bytes, with the next smallest being ACORN with 1868 bytes. On MSP, Grain has the smallest code size of 926 bytes. The next smallest, again being ACORN, is 1744 bytes.

AVR-Fast. When targeting fast code for AVR, Grain is faster than most of the compared ciphers, except NORX and ASCON. Alt. Grain is 3.8 times faster than AES-GCM, while still being the smallest of all ciphers in code size. NORX demonstrates x2 speed of Grain but then the code size is x2-3 times larger than Grain. ASCON has x4.6 times faster speed but the code size is x11-14 times larger than Grain's (24590 vs 2372/1734 bytes). Alt.ASCON with a decent code size (but still larger than Grain) has a speed performance 28% slower than Grain.

MSP-Fast. When targeting fast code for MSP, Grain still has the smallest code size and performs $\sim$x12.2 times faster than AES-GCM. ASCON is faster, but its code size is again enormously large (46174 vs 1436 bytes). Alt.ASCON with a decent code size becomes $\sim$x2.4 times slower than Grain. NORX seems to have 2x faster speed than Grain but the code size is 3-4 times larger, again.

Balanced choice. The code size and the total time of a chosen cipher heavily depend on the platform, implementation, and the compiler's optimization options. Minimizing both code size and total time require a balanced metric for that. Here, we compute the product of the code size and the total time and pick the cipher variant with the smallest such metric. These "balanced" variants are given in Table 4. The initialization requirement for stream ciphers typically makes them less favorable for very short messages. Still, we have chosen Scenario 1c for this metric in order to show that Grain outperforms the compared ciphers and their implementations even in this scenario. For ASCON, the best balanced choice has a very large code size (24590 bytes), so we also provide the next best metric with smaller code size. On both targets, AVR and MSP, Grain has the smallest metric of the balance between the code size and the total time. Moreover, Grain has the smallest code size, and the total time is a lot faster than AES-GCM (about x5.6 and x12,

Table 3: Simulation results on FELICS-AEAD

Name	Ver	CO	RAM	Code	RAM	Total time	RAM	Total time
			state	size	stack	(cycles)	stack	(cycles)
AVR-Small (minimizing code size)					**Scenario 1c**		**Scenario 2c**	
ACORN	v2	-Os	37	1868	64	692884	64	2947063
AES-GCM	v1	-O2	228	2336	96	1463142	96	14700113
ASCON	v3	-O1	40	3724	122	362589	124	3849045
Grain	v1	-Os	49	1100	34	1647100	34	19026431
(Alt.) Grain	v3	-O1	50	1306	38	1505809	38	17407445
Ketje-Jr	v2	-O2	25	3022	197	3149313	1335	29903034
NORX	v4	-Os	64	5024	202	178095	202	1192691
(Alt.) NORX	v3	-O2	64	5126	203	123769	203	814237
AVR-Fast (minimizing total time)					**Scenario 1c**		**Scenario 2c**	
ACORN	v1	-Os	37	3024	81	396798	81	1614759
(Alt.) ACORN	v2	-O2	37	1916	66	661208	66	2811932
AES-GCM	v2	-O3	228	6578	111	975184	111	9812008
(Alt.) AES-GCM	v1	-O3	228	5944	111	985390	111	9919910
ASCON	v2	-O3	40	24590	63	53272	63	598098
(Alt.) ASCON	v2	-O2	40	24086	63	54931	63	596774
(Alt.) ASCON	v3	-O1	40	3724	122	362589	124	3849045
Grain	v2	-O3	49	2372	38	255936	38	3000359
(Alt.) Grain	v2	-O2	49	1734	38	263101	38	3090000
Ketje-Jr	v2	-O3	25	5156	190	311949	1328	3007966
(Alt.) Ketje-Jr	v1	-O3	25	4562	189	409286	1327	3948612
NORX	v3	-O2	64	5126	203	123769	203	814237
MSP-Small (minimizing code size)					**Scenario 1c**		**Scenario 2c**	
ACORN	v2	-Os	37	1744	74	892381	74	3781122
(Alt.) ACORN	v2	-O1	37	1750	64	676228	64	2863923
AES-GCM	v1	-Os	228	1874	116	2331871	116	23407855
ASCON	v3	-Os	40	5572	336	417711	338	4420289
(Alt.) ASCON	v1	-Os	40	5578	336	603756	330	6866631
Grain	v3	-Os	50	926	42	1612326	42	18515129
(Alt.) Grain	v4	-Os	50	1358	44	184104	44	2193116
Ketje-Jr	v2	-Os	25	2574	212	6753573	1350	64134071
(Alt.) Ketje-Jr	v2	-O2	25	2602	206	6135738	1344	58219121
NORX	v4	-Os	64	4214	214	100988	214	687123
(Alt.) NORX	v3	-O2	64	4308	218	71368	218	480123
MSP-Fast (minimizing total time)					**Scenario 1c**		**Scenario 2c**	
ACORN	v1	-O3	37	6454	76	435022	76	1749651
(Alt.) ACORN	v1	-O2	37	3136	92	525851	92	2182308
AES-GCM	v1	-O2	228	2020	126	2126367	126	21372940
ASCON	v2	-Os	40	46174	68	117690	68	1265628
(Alt.) ASCON	v3	-O1	40	5688	332	415098	334	4391449
Grain	v4	-O1	50	1436	40	174338	40	2071091
Ketje-Jr	v2	-O3	25	6248	196	335624	1334	3242592
(Alt.) Ketje-Jr	v2	-O2	25	2602	206	6135738	1344	58219121
NORX	v3	-O3	64	8112	208	69360	208	449837
(Alt.) NORX	v3	-O2	64	4308	218	71368	218	480123

respectively). Of the compared designs, the closest to Grain in respect to this metric is NORX, which has about a 2x faster speed but about 3x larger code.

Table 4: Balanced choice of algorithms. The smallest Code-size × min total time is used.

Name	Ver	CO	RAM state	Code size	RAM stack	Total time (cycles)	Code size × Total time
AVR balanced choice (Sc.1c)							
ACORN	v1	-Os	37	3024	81	396798	$2^{30.16}$
AES-GCM	v2	-O2	228	2338	96	1460677	$2^{31.67}$
ASCON	v2	-O3	40	24590	63	53272	$2^{30.29}$
(Alt.) ASCON	v3	-O1	40	3724	122	362589	$2^{30.33}$
Grain	v2	-O2	49	1734	38	263101	$2^{28.77}$
Ketje-Jr	v2	-O3	25	5156	190	311949	$2^{30.58}$
NORX	v3	-Os	64	5028	201	124062	$2^{29.22}$
MSP balanced choice (Sc.1c)							
ACORN	v2	-O1	37	1750	64	676228	$2^{30.14}$
AES-GCM	v2	-O1	228	1952	126	2174330	$2^{31.98}$
ASCON	v3	-Os	40	5572	336	417711	$2^{31.12}$
Grain	v4	-Os	50	1358	44	184104	$2^{27.90}$
Ketje-Jr	v2	-O3	25	6248	196	335624	$2^{30.97}$
NORX	v3	-Os	64	4216	212	71419	$2^{28.17}$

4 Conclusions

We have presented a software implementation of the Grain-128AEAD stream cipher for embedded processors. The two processors that were targeted are the two smallest defined in the FELICS-AEAD framework, namely the AVR ATmega 128 and the MSP430F1611. Since these two processors have very different characteristics, the optimizations require different approaches. Moreover, we have shown that the code size can be made extremely small by implementing a mini-RISC CPU for the clocking of the registers and the computation of the pre-output keystream. The benchmarking results show that Grain-128AEAD is competitive when implemented for embedded platforms.

References

[Atm06] Atmel. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash ATmega128 ATmega128L, 2006. http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf.

[BMA+18] Subhadeep Banik, Vasily Mikhalev, Frederik Armknecht, Takanori Isobe, Willi Meier, Andrey Bogdanov, Yuhei Watanabe, and Francesco Regazzoni. Towards low energy stream ciphers. *IACR Transactions on Symmetric Cryptology*, 2018(2):1–19, Jun. 2018.

[CdSGB19] Luan Cardoso dos Santos, Johann Großschädl, and Alex Biryukov. FELICS-AEAD: Benchmarking of lightweight authenticated encryption algorithms, 2019. Lightweight Cryptography Workshop.

[HJM+19a] Martin Hell, Thomas Johansson, Willi Meier, Jonathan Sönnerup, and Hirotaka Yoshida. An AEAD variant of the Grain stream cipher. In Claude Carlet, Sylvain Guilley, Abderrahmane Nitaj, and El Mamoun Souidi, editors, *Codes, Cryptology and Information Security*, pages 55–71. Springer International Publishing, 2019.

[HJM+19b] Martin Hell, Thomas Johansson, Willi Meier, Jonathan Sönnerup, and Hirotaka Yoshida. Grain-128AEAD - a lightweight AEAD stream cipher,

2019. https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates.

[Ins06] Texas Instruments. MSP430x1xx Family User's Guide, 2006. http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[Ins11] Texas Instruments. MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller, 2011. http://www.ti.com/lit/ds/symlink/msp430f1611.pdf.

[SHSK19] Jonathan Sönnerup, Martin Hell, Mattias Sönnerup, and Ripudaman Khattar. Efficient hardware implementations of Grain-128AEAD. In *Progress in Cryptology – INDOCRYPT 2019*. Springer International Publishing, 2019. Accepted.