Efficient Implementations of AES-128 and Grøstl-256 for the AVR 8-bit Microcontroller Architecture

Johannes Feichtner

TU Graz, Institute for Applied Information Processing and Communications
Inffeldgasse 16a, 8010 Graz, Austria
`johannes.feichtner@student.tugraz.at`

Abstract. As embedded systems are more and more used for security-critical applications, data integrity needs to be assessed using efficient implementations of cryptographic primitives. In this paper, assembly implementations of the block cipher AES-128 and the AES-based hash function Grøstl-256 are presented and special properties of an ATmega128 microcontroller are taken into consideration. The different implementations try to focus on effective usage of memory and computation resources and thereby yield results, which outperform existing solutions.

1 Introduction

Efficient implementations of cryptographic primitives are essential in all fields of application. In particular, the word "efficiency" attains a high position if an embedded system is used as execution platform. Due to limited resources, cryptographic services have to be implemented very carefully, regarding their runtime-performance and memory-usage. In order to integrate an embedded device within a critical environment, such as the Internet of Things, requirements like secure authentication, data integrity, confidentiality, and non-repudiation must be met. These demands can be achieved by using symmetric-key ciphers (e.g. AES), public-key ciphers (e.g. Elliptic Curve Cryptography (ECC)), and hash functions (e.g. Grøstl).

This paper describes how the block cipher AES [14], a widely used standard for symmetric key cryptography, and the AES-based hash function Grøstl [8], with a message digest size of 256 bits, can be implemented very effectively for 8-bit microcontrollers in order to fulfill the constraints of practical application. The target architecture during development was an AVR ATmega128 [3]. For simulation purposes, the AVR simulator `simulavr` [17] suited very well. As known so far, the Grøstl implementations presented in this paper are the first made in avr-gcc assembly only.

This paper is organized as follows. Section 2 discusses related work and refers to other implementations. Besides, multiple approaches for efficient implementations are considered. Characteristics of the ATmega128 processor are shown in Section 3. Section 4 introduces to the block-cipher AES, followed by advices on how to implement AES effectively. Section 5 explores the hash function Grøstl and comes up with various implementation approaches. Performance results of the AES and Grøstl implementations are then summarized in Section 6. Section 7 concludes this work.

2 Related Work

Implementing cryptographic primitives effectively for embedded devices has been a goal for years. Since the standardization of the Advanced Encryption Standard (AES) [14] in 2001, it has been tried to optimize the most time-consuming functional parts *MixColumns* and *InvMixColumns*. Hua Li *et al.* [11] focus on an efficient architecture of *MixColumns*. By analyzing the data dependencies, they are able to reduce the multiplications in the finite field $\mathbb{F}_{256}$ significantly. Fischer *et al.* [7] pursue this approach and decompose the *InvMixColumns* operation. They point out which components of *MixColumns* are shared with *InvMixColumns*. Due to the fact that *InvMixColumns* can be performed by a data pre-processing and a subsequent *MixColumns* operation, redundant code parts can be omitted and thus the code size is reduced.

In terms of efficient implementations, the work of Eisenbarth *et al.* [6] provides compact implementation strategies and a performance evaluation of various block-ciphers, all apted for 8-bit microcontrollers. Notable in this context are the well-performing AES-128 implementations of Poettering *et al.* [15] in AVR assembly and a byte oriented implementation in C by Gladman *et al.* [9]. Although focussed on an efficient hardware design, the work of Feldhofer *et al.* [12] contains valuable information for low cost implementations in general. Tillich *et al.* [20] go one step ahead and propose to implement AES with some custom instructions (instruction set extensions). This hardware implementation outperforms any software implementation due to the fact that lower energy consumption, smaller code size, lower RAM usage and short latency can be assured.

Compared to AES, Grøstl requires much more resources. Aoki *et al.* [2] have shown how to reduce the plentitude of operations in Grøstl's most expensive round transformation *MixBytes* significantly. Therefore, multiplications in $\mathbb{F}_{256}$ are rewritten as multiplications by two and additions, and then performed with a minimized amount of instructions. The Grøstl implementations presented in this work also follow their approach.

Grøstl-0-256 has been implemented efficiently for an ATmega163 microcontroller by Roland [16]. Due to the circumstance that these implementations were targeted for an ATmega163 only, extensive adaption is necessary for an execution on other 8-bit devices. Compared to the work of Roland, the implementations presented in this paper follow different approaches and thereby excel the former work. Also notable is a sophisticated Grøstl implementation of Ipsen [10] in C which has been submitted to the hash function benchmarking project eBASH [1].

3 ATmega128 Processor

Developed in the 1990s, the AVR series by Atmel became one of the most popular 8-bit processors ever manufactured. A series of high-performance, low-power Atmel AVR processors evolved to the popular ATmega series. The ATmega series supports 133 instructions and comes with a wide range of peripherals. For the process of implementing AES and Grøstl, the ATmega128 [3] has been used as target model. As illustrated in Fig. 1, it possesses an 8-bit RISC processor with Harvard architecture, 32 general-purpose registers, an on-chip multiplier, 128 kbyte of Flash, 4 kbyte of EEPROM, and 4 kbyte of SRAM. It also supports boundary-scan JTAG as well as several timers, ADCs, UARTs, and an SPI and a TWI interface.

The registers R26 to R31 have some added functions to their general purpose usage. They can be used as 16-bit address pointers for indirect addressing. Hence these registers allow fixed displacement (*i.e.* for looking up a specific value in the Rijndael S-box), automatic increment and decrement [3].

Regarding the rather long execution time of certain available instructions, their usage should be well considered in performance-critical applications. So e.g. it takes 3 cycles to load a value from program memory (`LPM`) whereas a RAM access (`LD`) is possible within just two cycles. A tradeoff between code size and speed has also to be made if functions need to be called repetively, such as *MixBytes* in the Grøstl permutation routines. If a function is embedded multiple times, the code size increases proportionally but in return an expensive `RCALL` (3 cycles) and a `RET` (4 cycles) instruction can be saved.

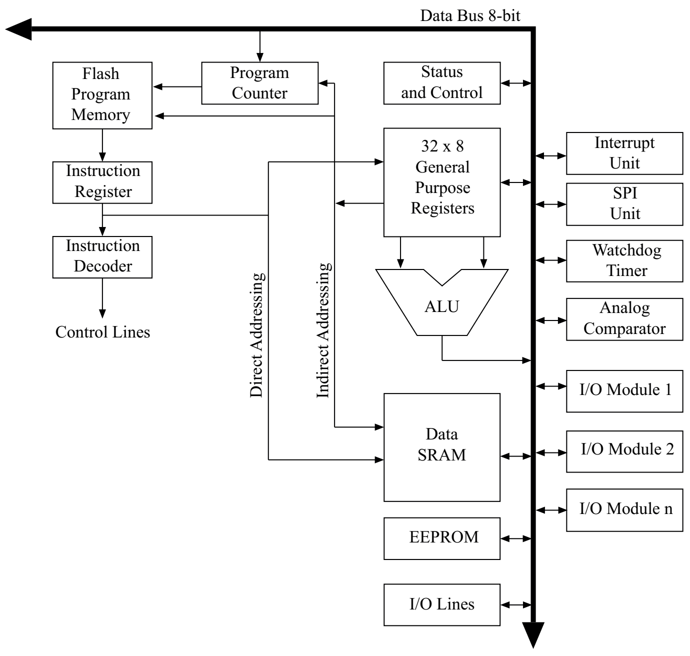


Fig. 1. Block diagram of the AVR MCU architecture [3]

4 AES

AES [14] was standardized in 2001 as a replacement of DES. Since then it has been used in a magnitude of applications and has become the most investigated symmetric-key cipher. AES is a round-based block cipher, operating on 128-bit data blocks, with an internal 128-bit state matrix. For encryption, the state is modified by iteratively applying four round transformations, known as *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. Each iteration round needs a proper key which is derived from the secret key using a key scheduling routine.

In short, the algorithm can be summarized in 4 steps as follows:

- Key schedule: A 128-bit key is expanded to 11 round keys
- State initialization: Initial state $\leftarrow$ Plaintext block XOR the 128-bit key
- Round transformations: Apply the round function on the state 9 times
- Final round: Apply the round function without the *MixColumns* transformation

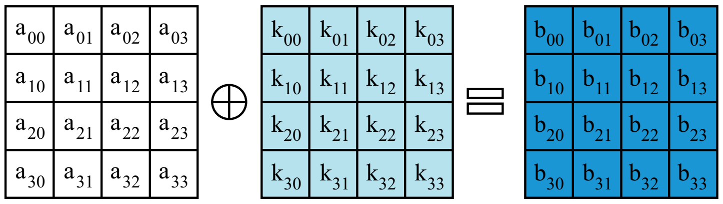
The decryption routine follows the same procedure but performs the round transformations inverted and in reverse order.

Depending on the required security level, one can select from AES-128, AES-192 and AES-256, which vary on the size of the key and the number of applied round transformations. Having embedded applications in mind, this work puts the focus on AES-128 which uses a 128-bit key.

4.1 Round transformations

AES-128 performs the round transformations 10 times. Within one iteration, the four steps *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns* are executed consecutively. For decryption, the steps *SubBytes*, *ShiftRows* and *MixColumns* are replaced by their inverted counterparts *InvSubBytes*, *InvShiftRows* and *InvMixColumns*.

AddRoundKey First, the key of the current round is added to the state in the *AddRoundKey* step. As shown in Fig. 2 this is done by applying a bitwise XOR operation to each byte of the state and to the corresponding byte of the current round key.



a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
$\oplus$
k_{00} k_{01} k_{02} k_{03}
k_{10} k_{11} k_{12} k_{13}
k_{20} k_{21} k_{22} k_{23}
k_{30} k_{31} k_{32} k_{33}
=
b_{00} b_{01} b_{02} b_{03}
b_{10} b_{11} b_{12} b_{13}
b_{20} b_{21} b_{22} b_{23}
b_{30} b_{31} b_{32} b_{33}

Fig. 2. The *AddRoundKey* transformation

SubBytes Subsequently, the *SubBytes* operation substitutes each byte of the state by its entry in a lookup table, named S-Box. For these substitutions, AES uses Rijndael's S-Box [4] which was specifically designed to be resistant to linear and differential cryptanalysis [18]. Fig. 3 illustrates the process of substitution.

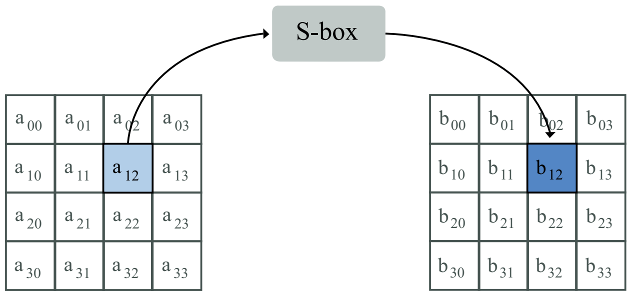


Fig. 3. The *SubBytes* transformation

InvSubBytes For decryption, *InvSubBytes* processes the state by an inverse S-box which technically means that for an input value, the inverse affine transformation is calculated, followed by its multiplicative inverse in $\mathbb{F}_{256}$. This finite field is defined by the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus$ x $\oplus$ 1 over $\mathbb{F}_2$.

ShiftRows *ShiftRows* cyclically shifts the bytes within a row (except for the first) of the state matrix to the left by a number of positions. As it can be seen in Fig. 4, the second row is shifted by one, the third row is shifted by two and the fourth row is shifted by three bytes to the left. It is worth to mention that the *InvShiftRows* performs this shifting in reverse direction. As a consequence of this step, each column of the output state matrix is composed of bytes from each column of the input state.

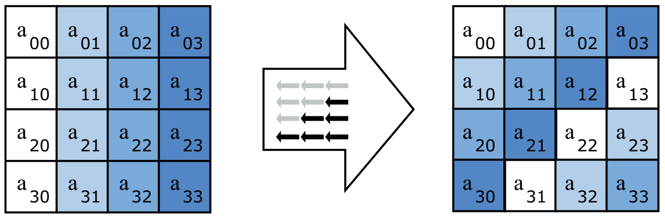


Fig. 4. The *ShiftRows* transformation

MixColumns Finally, in the *MixColumns* step each column of the state is transformed independently and multiplied by a constant 4×4 matrix in $\mathbb{F}_{256}$

(see Fig. 5). The matrix to multiply with is circulant, implying that each row is equal to the row above, rotated right by one position. By the combination of *SubBytes*, *ShiftRows* and *MixColumns* a substitution-permutation-network is constructed, providing a sophisticated extent of diffusion [19].

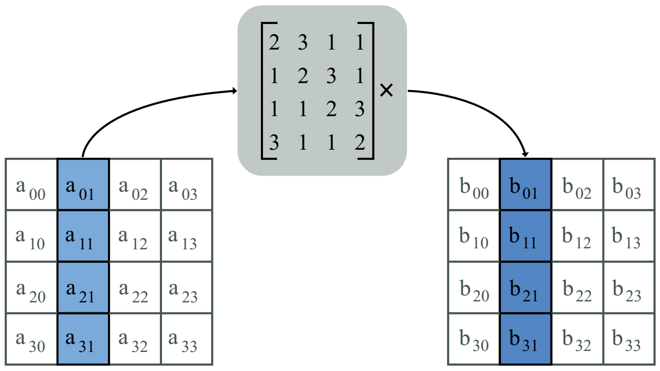


Fig. 5. The *MixColumns* transformation

InvMixColumns Concerning *InvMixColumns*, Fischer *et al.* [7] have shown that this step can share resources with *MixColumns*, resulting in reduced code size overall. Therefore, the *InvMixColumns* step is decomposed and expressed as a set of polynomials: $d(X) = c(X) \cdot f(X)$. The inverse multiplication of one state matrix column X is denoted as $d(X)$. $c(X)$ describes the forward multiplication, which is performed during the *MixColumns* step. The polynomial $f(X)$ can be represented as: $f(X) = c^2(X) = \{04\}X^2 + \{05\}$ whereas $\{04\}$ and $\{05\}$ indicate multiplications by 4 and 5 in $\mathbb{F}_{256}$. Due to the fact that the present implementations operate on byte-level, the $f(X)$ function would equate for the first output byte b_0 in the following way: $b_0 = \{05\}b'_0 + \{04\}b'_2 = \{04\}(b'_0 + b'_2) + b'_0$ where b'_0 and b'_2 are the first and the third byte. These two bytes can share the term $\{04\}(b'_0 + b'_2)$ so that an addition of b'_2 would result b_2. Using this approach, the second and the fourth byte can be expressed in the same manner.

4.2 Rijndael key schedule

AES uses a key schedule to expand a secret key into separate keys for each round. For the first round key, the last column of the initial key matrix is used. It is then cyclically rotated to the left of the key matrix, similar to the *ShiftRows* round transformation. Subsequently, the values of this column are substituted by S-Box values, followed by an XOR-addition to the first column of the key matrix. To conclude the key expansion of the first column, a round constant is added. This value, named Rcon by the Rijndael documentation [4], is the exponentiation of two to a specific input value in $\mathbb{F}_{256}$. For illustration purposes, it can be rewritten as $Rcon(i) = x^{(254+i)}$ where i denotes the round constant iteration value.

The computation of the second column is done by XOR-ing values of the first and the second column. Similarily, the third column is XOR-ed with the second and the same manner applies to the fourth column.

4.3 Implementation

AES-128 uses a 128-bit key and performs the round transformations 10 times. The 128-bit state is constantly held in the registers R0-R15. In order to avoid loading the key twice, the key expansion step and the *AddRoundKey* transformation have been merged into a single function. Thus, expensive `RCALL` and `RET` instructions can be evaded too. Values from the S-box can either be located in RAM or program memory. RAM access (`LD`) needs one cycle less than access to program memory (`LPM`) and hence provides faster encryption/decryption overall. Nevertheless, one could tend to use the program memory instead if the sparsely available RAM is needed for other purposes.

Due to the fact that *MixColumns* applies a transformation on each column of the state matrix, this is the most expensive part of AES. For an efficient implementation of *MixColumns*, the needed amount of `XOR` and multiplication operations has to be cut down to a minimum. A multiplication by 2 is performed by consecutively executing an `ADD`, `BRCC` and `EOR` operation. `ADD` is used to do a right shift and depending on the carry register, a reduction using the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ is performed. This sequence of operations can be performed in constant time. The *MixColumns* transformation is implemented using 48 `XOR` operations, 16 multiplications by 2, and 36 `MOV` instructions.

The implementation of *InvMixColumns* follows the approach of Fischer *et al.* [7] (see Section 4.1 for details). This step is implemented as a combination of pre-processing and a subsequent *MixColumns* transformation. Instead of multiplying the sum of two bytes by 4, two multiplications by 2 achieve the same result. Consequently, the use of a multiplication table can be omitted and thereby the code size is kept small. The pre-processing operations for *InvMixColumns* require 24 `XOR` operations, 16 multiplications by 2, and 8 `MOV` instructions.

5 Grøstl

Grøstl [8] is an iterated hash function operating on two distinct permutations P and Q and a compression function f in a variant of the Merkle-Damgård construction [5, 13]. The input of the compression function is a message block m and a 512-bit chaining value h, , initially defined as $IV = \{00, ..., 00, 01, 00\}$. The construction of f is shown in Fig. 6. It uses the following definition: $f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h$. The final hash value is calculated from the last state of h using an output transformation $\Omega = trunc_{256}(P(x) \oplus x)$ where $trunc_{256}(x)$ truncates all but the 256 least significant bits of x.

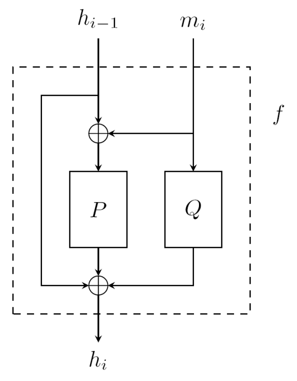


Fig. 6. The compression function f of Grøstl [8]

5.1 Permutations

The permutations P and Q are based on AES and thus consist of four transformations, known as *AddRoundConstant*, *SubBytes*, *ShiftBytes*, and *MixBytes*, which are applied 10 times in Grøstl-256. Based on the given similarities, it comes to mind to actually combine the implementations of AES and Grøstl. In fact it is possible that AES and Grøstl share an S-box lookup table and temporary data memory. Apart from that, the assembly implementations are kept completely separated.

The *AddRoundConstant* step adds a constant to one row of the state by XORing. It is to note that the constants and the rows where to add them are different for P and Q. Further, an independent constant `0xff` is added to every byte in Q. The *SubBytes* transformation substitutes the state with values from the Rijndael S-box. Similar to *ShiftRows* in AES, *ShiftBytes* cyclically moves the bytes to the left by a specific position offset. These rotation offsets are different for P and Q. The *MixBytes* transformation is the most expensive part in Grøstl as it involves a matrix multiplication in $\mathbb{F}_{256}$. Each column of the state is multiplied with a constant circulant 8×8 matrix $B = circ(02, 03, 04, 05, 07)$. Due to limited resources, only one column can be processed at once. This constraint causes 80 invocations of *MixBytes* per permutation and leads to the conclusion that any speed-up potential for this transformation needs to be exploited.

5.2 Implementation

For efficient implementations of Grøstl, performance critical components have to be optimized as far as possible. Due to the fact that the permutation routines P and Q as well as the round transformation *MixBytes* are the most expensive parts, it will now be briefly described how they can be implemented. Furthermore, two ways for the integration of the *ShiftBytes* step are covered.

Permutations The matrices P and Q are permuted in 10 rounds. For convenience and to save cycles, the round transformations *AddRoundConstant*, *Sub-*

Bytes and *ShiftBytes* are directly implemented within the permutation routines. This allows to perform the operations without additional RAM access and also reduces the memory footprint. *SubBytes* is applied to every single byte of the state, so if *MixBytes* is not repetively embedded using a macro, the S-box substitution can also be moved there to lower the code size.

The permutation routines steadily add the round constants to a set of 8 state values (R0-R7), which are then handed over to *MixBytes*. Unfortunately, *MixBytes* can not be calculated in-place as it combines the mentioned value set by multiplication in $\mathbb{F}_{256}$. In order to store a finally transformed column, the byte storing can be moved to *MixBytes* and a 16-bit address pointer is increased within every store operation (`ST`). At the end of a permutation round, the pointer then needs to be set back by 64 positions.

ShiftBytes The *ShiftBytes* step can be implemented nearly transparently if two matrices X and Y are used for one permutation round. According to the Grøstl specification, the bytes would be shifted and then processed column-wise by *MixBytes*. This step can be enhanced if bytes are directly selected at those positions where they would have been shifted to (diagonal addressing). This results in a significant performance gain because load and store instructions, which would occur within the classical *ShiftBytes* implementation, are entirely omitted. The disadvantage is that two matrices are needed, one to read the input bytes for *MixBytes* and another one to store the transformation output. After each round, the matrices are swapped to proceed the next permutation round with the output previously gained from *MixBytes*. Instead of doing this using a temporary variable (*i.e.* the third 16-bit address register Z), the XOR swap algorithm is used.

Another implementation of *ShiftBytes* makes use of no longer needed state values. For example, for the *MixBytes* step of the first column in a P permutation, the values at state matrix indices $0, 9, 18, 27, 36, 45, 54, 63$ can be selected by diagonal addressing and loaded to the registers R0-R7. As *MixBytes* would overwrite the whole column, the old state values are backuped to the no longer needed offsets, *i.e.* $1 \rightarrow 63,\ 2 \rightarrow 54,\ 3 \rightarrow 45,\ 4 \rightarrow 36,\ 5 \rightarrow 27,\ 6 \rightarrow 18,\ 7 \rightarrow 9$. Pursuing this approach for the other columns too, *ShiftBytes* can be implemented in each permutation routine with only 28 `LDD` and `STD` instructions and no additional overhead.

MixBytes Due to high values in the multiplication matrix B, the *MixBytes* transformation is the most expensive part of Grøstl. When *MixBytes* is invoked, the registers R0-R7 hold 8 state values to be multiplied (denoted as a_i). The registers R8-R15 are used for temporary calculations. After the transformation, the output bytes b_i are stored at the corresponding position in the state matrix. In order to reduce the needed amount of `XOR` and multiplication-by-2 operations, the approach of Aoki et al [2] is followed. Instead of multiplying the bytes of a column in $\mathbb{F}_{256}$ by 2, 3, 4, 5 and 7, only multiplications by 2 are performed. As described in Section 4.3 this multiplication is possible with no additional overhead. Many terms of the form $a_i + a_{i+1}$ are needed for each factor of 1, 2 and 4 and thus can be re-used from temporary results t_i, x_i and y_i. The formulas for this calculation are denoted as follows:

$$\begin{aligned}
t_i &= a_i + a_{i+1} \\
x_i &= t_i + t_{i+3} \\
y_i &= t_i + t_{i+2} + a_{i+6} \\
b_i &= 2 \cdot (2 \cdot x_{i+3} + y_{i+7}) + y_{i+4}
\end{aligned} \tag{1}$$

Per column, *MixBytes* needs 48 `XOR` instructions, 9 `MOV` instructions and 16 multiply-by-2 operations.

6 Results

The implementations of AES and Grøstl were made in assembly for avr-gcc and follow a common path of execution. Due to the fact that the main crypto routine is not a C function (as opposed to former implementations), call-saved registers (R2-R17, R28-R29) and fixed registers (R0, R1) need to be pushed and popped only once.

In the following, several results which have been accumulated, are discussed. First, the characteristics of the AES implementations are brought to attention. Second, the Grøstl implemenations are compared and their performance is pointed out.

6.1 AES Results

Three implementations of AES-128, one in C, another two in assembly language have been realized. Basically, AES can either be implemented with a pre-encryption round key generation or an on-the-fly key generation. The advantage of the first method is that all needed round keys are generated once before performing the normal round transformations. Thus, the key can be loaded to the registers once and then be expanded 10 times (for AES-128) which requires 176 bytes of RAM overall. The second method refrains from occupying additional RAM for the key. Instead, the keys are expanded per round which inherits that the key has to be loaded into the registers over and over again. Without dispute, this has a negative impact on the performance but having the limited RAM size of a tiny microcontroller in mind, the on-the-fly key generation seems perfectly reasonable.

Both assembly implementations have in common that all loops are unrolled and no macro embedding is used. Due to the fact that the key expansion as well as the *AddRoundKey* step need the round key in registers, the two steps have been merged. Further, *InvMixColumns* does the pre-processing and subsequently proceeds with *MixColumns*. Sharing resources between *InvMixColumns* and *MixColumns* lowers the code size because redundant code is omitted. Apart from that, the state matrix is consistently held in registers R0-R15. Stack usage is mainly caused by initial `PUSH` instructions because when *encrypt* or *decrypt* is called from inside a C environment, the callee needs to save and preserve certain registers.

Table 1. Comparison of AES-128 implementations.

Version	**ROM**	**RAM**	**Runtime** [Cycles]	
	[Bytes]	[Bytes]	Enc.	Dec.
C version	6,814	858	6,502	7,938
High-speed	1,980	294	3,290	4,534
Low RAM	2,048	38	3,490	4,630
Eisenbarth *et al.* [6]	1,659	33	4,557	7,015
Poettering *et al.* [15]	1,568	192	3,629	4,462

Table 1 shows runtime, ROM, and RAM requirements of our and related AES implementations. The C version is given as reference in order to underline the margin exhausted by assembly implementations. In the high-speed design, an S-box lookup table was loaded to the RAM. By keeping the S-box within the program memory it is possible to decrease the expensive RAM requirement from 294 to 38 bytes. In terms of runtime, even the RAM optimized implementation is still faster than Eisenbarth *et al.* [6] and Poettering *et al.* [15]. As usual for AES, the decryption takes longer than the encryption. Aside from the *InvMixColumns* overhead, the last round key needs to be computed from the secret key before the decryption can start inverting the encryption.

6.2 Grøstl Results

In the following, six highly-optimized assembly implementations of Grøstl-256 are presented. They focus on efficient usage of memory and computation resources and thus yield results, which outperform existing solutions. Due to repetitive invocations, the permutation rounds and the *MixBytes* transformation have to be optimized as far as possible in order to achieve the highest degree of performance. The six versions have several components in common, such as the *MixBytes* computation or the initial message read-in where padding bytes are appended if needed. Further, all versions need 64 bytes for the h matrix and 64 bytes for a message buffer. Where possible, consistently needed values are retained in registers, such as the irreducible polynomial (in R17) or the S-box (in Z).

The main routine reads the message to hash block-wise into a 64 bytes buffer. If more than 64 bytes are remaining, the buffer is entirely filled with a message part, if less than a full block is left, the buffer is filled with the last message segment and the remainder is padded. If the message length is zero, the buffer contains only padding bytes. Due to the fact that the buffer, the h matrix and, depending on the implementation, other statically-allocated variables are not

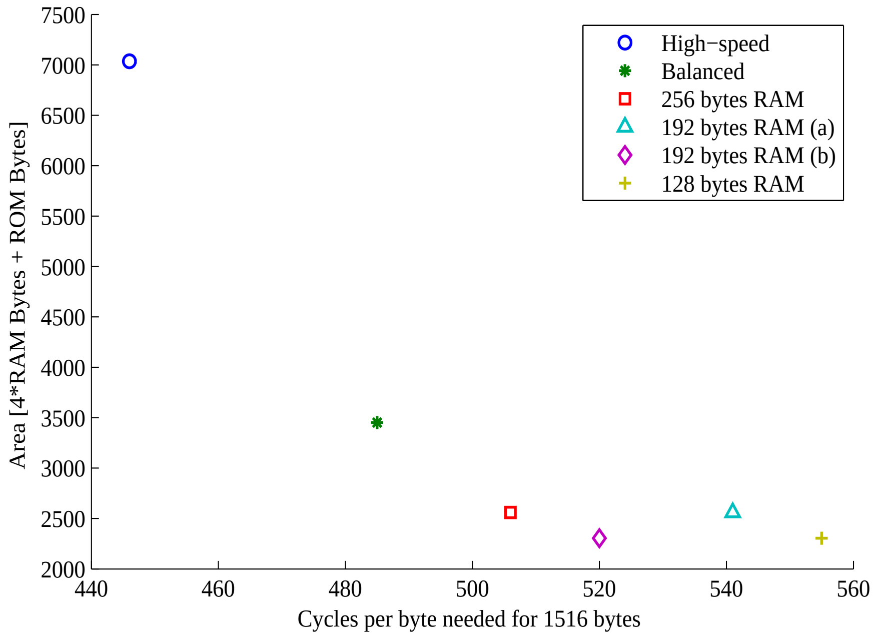


Fig. 7. Performance of the six Grøstl implementations

used with initialized values, they can be moved to the *.bss* segment. The advantage is that for these elements no space is occupied in program memory, resulting in a reduced code size. As soon as the program is loaded, the loader will automatically initialize the memory for that section in RAM with zeros.

Below, the characteristics of all versions are denoted briefly and highlighted in Table 2. Fig. 7 illustrates the area and cycles/byte needed for a test message with 1516 bytes.

Table 2. Comparison of Grøstl-256 implementations

Version	**ROM** [Bytes]	**RAM** [Bytes]	**Runtime** [Cycles] 55 bytes	2776 bytes	**Cycles / Byte** 55 bytes	2776 bytes
High-speed	4,988	534	41,222	1,230,282	**749**	**443**
Balanced	**1,404**	536	44,820	1,337,037	815	482
Low RAM 256	1,536	280	46,740	1,393,997	850	502
Low RAM 192 (a)	1,792	216	50,006	1,490,917	909	537
Low RAM 192 (b)	1,536	218	47,665	1,434,721	867	517
Low RAM 128	1,792	**154**	50,928	1,531,509	926	552
Ipsen [10]	4,684	602	87,643	2,636,561	1,594	950
Roland [16]	4,228	994	50,105	1,301,944	911	469
Roland [16]	2,336	164	70,896	2,102,066	1,289	757

A speed-optimized version makes extensive use of assembly macros and omits expensive `RCALL` (3 cycles) and `RET` (4 cycles) instructions. A disadvantage of this technique is that the code size increases to 4,988 bytes as the *MixBytes* routine is embedded 16 times. It is to mention, that due to this embedding, the subroutine *PermP* becomes so large that it needs to be invoked by its absolute address, respectively `CALL` (4 cycles) from inside the main crypto routine. Before the *MixBytes* operation is executed in the permutation routines, an S-box lookup is performed for each of the 8 state values in registers R0-R7. Due to the elimination of unused overhead, a hash calculation for a message with 2776 bytes needs only 443 cycles/byte.

The balanced version is very similar to the high-speed version but does not embed *MixBytes* using assembly macros. Instead, it is invoked by an `RCALL` instruction from the permutation routines. Compared with the speed-optimized version, the needed cycles/byte increase to 482 for the same test message. Like *MixBytes*, the *SubBytes* step is applied on all columns and therefore can be integrated within *MixBytes* in order to evade redundant code. As a consequence, the code size minimizes to 1,406 bytes.

In contrast to the other versions, a low-RAM version, occupying 256 bytes of RAM, is especially suited for devices with small RAM, because the S-box values are loaded from program memory. Therefore, the RAM access instruction `LD` (2 cycles) is replaced by the instruction `LPM` (3 cycles) which accesses the program memory. As a result of this, the cycles per byte raise up to 502 for a long message but the consumed RAM reduces to 280 bytes.

Two versions have been implemented which need only 192 bytes of RAM. The first, annotated by a trailing *a*, varies from the version with 256 bytes by a different implementation of *ShiftBytes*. The permutation rounds are performed with only one matrix and an optimized *ShiftBytes* step (see Section 5.2 for details). Nevertheless, 28 additional `LDD` and `STD` instructions raise the needed cycles/byte up to 537 for the long test message. The second version, annotated by a trailing *b*, only differs from the version with 256 bytes by the fact that the input message is read-in twice. This is necessary because with 192 bytes only, the original buffer can not be held unchanged during the permutation routines. However, the buffer is XOR-ed with h before the P permutation and so can not be used as input for the Q permutation. Hence the input message is read twice and a flag decides whether to perform the P or the Q permutation. This overhead has a remarkable effect on the performance and so in case of a long message, the cycles/byte adds up to 517.

The last version combines RAM lowering features from the other versions and thus consumes only 128 bytes. In terms of performance, this version is clearly the slowest one needing 552 cycles/byte for a test message with 2776 bytes. By unifying the characteristics of both 192 bytes versions, the message is read-in twice and the permutation routines are performed with only one matrix. However, compared with the high-speed version this implementation performs only 40% slower for a short and 25% for a long test message. Regarding the significantly different RAM usage, the performance of the slowest version is still impressive.

7 Conclusion

In this paper, multiple approaches have shown how cryptography can be implemented efficiently for a standard microprocessor. Current AES and Grøstl implementations have been improved in terms of speed and memory footprint and evaluated on the ATmega128 microcontroller. The presented results are manifold. On the one hand, two well-performing AES implementations with on-the-fly key generation have been introduced. On the other hand, six sophisticated assembly implementations of Grøstl have been exposed, each one having its virtues. The high-speed version can run with only 443 cycles/byte on this very limited target hardware, the balanced version allocates just 1,404 bytes of ROM and the slowest version, using 128 bytes of RAM, performs still reasonable with 552 cycles/byte. Further, it can be seen that by taking advantage of sophisticated design concepts, an application is easily adjusted to fit various requirements such as run-time, RAM and program memory usage.

Due to its well-conceived design, Grøstl provides many possibilites for efficient implementations. In future work a significant performance gain could be achieved if the *MixBytes* transformation is further optimized.

References

1. eBASH: ECRYPT Benchmarking of All Submitted Hashes. Available online at `http://bench.cr.yp.to/ebash.html`.
2. K. Aoki, G. Roland, Y. Sasaki, and M. Schläffer. Byte Slicing Grøstl - Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Grøstl. 2011. in press.
3. Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, August 2007.
4. J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.
5. I. B. Damgård. A design principle for hash functions. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 416–427, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
6. T. Eisenbarth, T. Gneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Reggazoni, F.-X. Standaert, and L. van Oldeneel tot Oldenzeel. Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices. In *ECRYPT Workshop on Lightweight Cryptography 2011, November 28-29, Louvain-la-Neuve, Belgium*, November 2011.
7. V. Fischer, M. Drutarovský, P. Chodowiec, and F. Gramain. Inv mix column decomposition and multilevel resource sharing in AES implementations. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(8):989–992, Aug. 2005.

8. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, and S. S. T. Martin Schläffer. Grøstl – a SHA-3 candidate. Document version 2.0, available online at `http://www.groestl.info/Groestl.pdf`, March 2011.
9. B. Gladman. Byte Oriented AES Implementation, November 2006. Available online at `http://gladman.plushost.co.uk/oldsite/AES/`.
10. M. S. Ipsen. C-Implementation of Grøstl Optimized for 8-bit Architectures, December 2011. submitted to ebash.
11. H. Li and Z. Friggstad. An efficient architecture for the AES mix columns operation. volume 5, pages 4637–4640, May 2005.
12. J. W. Martin Feldhofer and V. Rijmen. AES Implementation on a Grain of Sand. volume 152, pages 13–20, October 2005.
13. R. C. Merkle. One way hash functions and DES. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 428–446, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
14. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.
15. B. Poettering. AVRAES: The AES block cipher on AVR controllers, March 2007. Available online at `http://point-at-infinity.org/avraes/`.
16. G. Roland. Efficient Implementation of the Grøstl-256 Hash Function on an ATmega163 Microcontroller. June 2009.
17. K. Rudolph, J. Wunsch, E. Weddington, and J. Sherrill. Simulavr: an AVR simulator, November 2011. Available online at `http://http://www.nongnu.org/simulavr/`.
18. M. Schläffer. *Cryptanalysis of AES-Based Hash Functions*. PhD thesis, 2011.
19. C. E. Shannon. Communication Theory of Secrecy Systems. volume 28, pages 656–715, 1949.
20. S. Tillich and C. Herbst. Boosting AES Performance on a Tiny Processor Core. 2008.