Minimalism of Software Implementation

- Extensive Performance Analysis of Symmetric Primitives on the RL78 Microcontroller -

Mitsuru Matsui and Yumiko Murakami

Information Technology R&D Center
Mitsubishi Electric Corporation
5-1-1 Ofuna Kamakura Kanagawa, Japan

Matsui.Mitsuru@ab.MitsubishiElectric.co.jp
Murakami.Yumiko@cw.MitsubishiElectric.co.jp

Abstract. This paper studies state-of-the-art software implementation of lightweight symmetric primitives from embedded system programmer's standpoint. In embedded environments, due to many possible variations of ROM/RAM-size combinations, it is not always easy to obtain an entire performance picture of a given primitive and to create a fair benchmark from top speed records.
In this study we classify these size combinations into several categories and optimize operation speed in each category. We implemented on Renesas' RL78 microcontroller - a typical CISC embedded processor, four block ciphers and seven hash functions with various combinations of ROM and RAM sizes to make performance characteristics of these primitives clearer. We also discuss how to create an interface and measure size and speed of a given primitive from a practical point of view.
As a result, our AES encryption codes run at as fast as 3,855 cycles/block in the ROM-1KB RAM-64B category, and 6,622 cycles/block in the ROM-512B RAM-128B category. For another examples aiming at minimizing a ROM size, we have achieved 453-byte Keccak, 396-byte Skein-256 and 210-byte PRESENT encryption codes on this processor.

1 Introduction

Lightweight crypto has become one of hot topics in cryptography, with increasing market requirements of embedded security as a background. In the SHA-3 project, suitability to embedded applications was regarded as an important metric for selection, and ISO/IEC 29192 is standardizing lightweight cipher primitives. Lightweight crypto has been more often discussed in hardware contexts, such as low area and low power consumption, but some of recent studies focus on software implementation on low resource processors, which is, we believe, equally important since it is rather common in embedded systems that encryption is carried out in hardware, but decryption is done in software.

One of such activities is ECRYPT II block cipher and hash function projects [1][2], which have published performance evaluation results of many symmetric

primitives on the ATtiny45 processor. All codes were written in an assembly language, aiming at low-cost implementation. These works are effectively the first extensive benchmarking on a low-end microprocessor.

The paper also deals with assembly language programming of symmetric primitives on a low-end embedded processor, but takes different approaches. First of all, our target processor RL78 has an accumulator-based CISC architecture with 8 registers and read-modify instructions, while ATtiny is a RISC processor with 32 registers and a fixed instruction length. Looking at implementations of the same algorithm on different processor architectures will be of independent interest.

Secondly we aim at demonstrating various ROM/RAM-size and speed tradeoffs for each primitive, not only pursuing pin-point top speed records. Embedded system programmers often deal with a crypto routine as an almost black box and want to know beforehand whether given size and speed can be achieved or not on a target processor. One of our purposes is to give them information about what ROM/RAM-size combinations are possible or **impossible** to implement on this processor. To do this, we first classify the size combinations into several categories and optimize each primitive in each category. Additionally we show a code toward a fastest speed and another code focusing on a smallest ROM size, accepting (very) slow computation speed.

Also we discuss interface and metric issues of symmetric primitives for embedded applications. In particular we point out that currently there is no consensus of how to count a RAM size of a given program. We here again take embedded programmers' viewpoint. What they are interested in is the amount of resources that they must allocate for a primitive. In this regard, we count the entire temporary area internally used in the primitive as RAM bytes, say, argument area and stack consumption including callee save register storage with a standard subroutine interface.

Our target primitives are AES [3], Camellia [4] and Clefia [5] with 128-bit key and Present [6] with 80-bit key for block ciphers. Note that AES and Camellia are included in ISO/IEC 18033-3, and Clefia and Present have been recently adopted as ISO/IEC 29192-2, a standard of lightweight block ciphers. For hash functions, our choices are SHA-256, SHA-512 [7], Keccak-256 [8], Skein-256, Skein-512 [9] Grøstl-256 and Grøstl-512 [10], where Keccak-256, Skein-256 and Skein-512 denote Keccak[r=1088,c=512], Skein-256-256 and Skein-512-512, respectively.

As a result, it is shown that AES achieves excellent size-speed balances for all ROM/RAM combinations on this processor. It runs at the speed of 3,855 cycles/block in the ROM-1KB RAM-64B category. Its ROM size was able to be reduced down to 486 bytes. Camellia outperforms AES in decryption. It is also demonstrated that the key scheduling of Clefia is a bottleneck for minimizing a code and Present is slow due to its harware-oriented nature, but its simple structure contributes to creating a very small program; we were able to write its encryption code with 210 ROM bytes.

For hash functions, it is shown that SHA-256 and SHA-512 are still good choices from a performance point of view. For 256-bit hash functions SHA-256

is fastest if 1KB or more ROM is given, and for 512-bit hash functions Skein-512 is the only option if only 256-byte RAM is given. It is also demonstrated that Keccak and Skein can be implemented in a very compact way; our smallest codes of Keccak-256/Skein-256 had 453/396 ROM bytes, respectively.

2 The RL78 Microcontroller

RL78 is Renesas Electronics' next-generation low-power microcontroller family combining advanced features from both the 78K and R8C families [11] which have been widely used in embedded applications such as in-vehicle controlling and mobile communication systems. It supports a wide range of pin, package and memory size combinations, currently covering Flash-ROM/RAM size variations of low-end 2KB/256B up to 512KB/32KB.

RL78 has a typical CISC architecture with an 8-bit accumulator-based instruction set including a small number of 16-bit instructions. It has eight 8-bit general registers `a,x,b,c,d,e,h,l`, which can be also used as register pairs `ax,bc,de,hl`. Most instructions allow only register `a` as a destination register, and only register pair `hl` as a general address pointer. For instance, `xor a,[hl]` is a valid instruction, but `xor b,[hl]` and `xor a,[de]` are not. This often causes size and speed penalties in programming symmetric primitives.

On the other hand, an advantage of this architecture is that it supports read-modify instructions and its average instruction length is short. Most instructions of RL78 used in a small model i.e. all segments are within 64KB, are one- to three-byte long. For instance, `xor a,[hl]` is a read-modify one-cycle instruction whose length is one byte.

As for the memory access speed, reading from internal RAM takes only one cycle, but reading from ROM takes four cycles. Moreover when an address register is modified in the preceding instruction, an additional one-cycle delay happens due to an address generation interlock stall. Hence a table lookup can be costly on this processor.

Table 1 shows some of the instructions essential in our programming:

instruction	length	latency	comment
`addw ax,[hl+byte]`	3 bytes	1 cycle	16-bit add without carry-in (with carry-out)
`sknc`	2 bytes	1 cycle	skip next instruction if non-carry
`xor/and/or reg1,reg2`	1 byte	1 cycle	reg1 or reg2 must be register `a`
`shl/shr a/b/c,cnt`	2 bytes	1 cycle	8-bit left/right shift; `shr` accepts only `a`
`shlw/shrw ax/bc,cnt`	2 bytes	1 cycle	16-bit left/right shift; `shrw` accepts only `ax`
`rolc/rorc a,1`	2 bytes	1 cycle	8-bit rotate shift with carry
`rolwc ax/bc,1`	2 bytes	1 cycle	16-bit rotate shift with carry; left shift only
`push/pop regpair`	1 byte	1 cycle	push/pop a register pair to/from stack
`call 16bit-adrs`	3 bytes	3 cycles	stack pointer is subtracted by 4 bytes
`ret`	1 byte	6 cycles	stack pointer is added by 4 bytes

Table 1. Key Instructions on RL78 in Symmetric Programming.

The 16-bit add instruction `addw` is convenient but unfortunately only works without carry-in (its result affects the carry bit, though). Using `sknc`, however, a memory-to-memory 32-bit addition can be implemented as shown below, which is slightly shorter and faster than using an 8-bit add-with-carry instruction:

```
movw  ax,[mem1+2]
addw  ax,[mem2+2]
movw  [mem3+2],ax
movw  ax,[mem1]
sknc                   ; skip next instruction if no carry
incw  ax               ; ax = ax + 1
addw  ax,[mem2]
movw  [mem3],ax
```

On the other hand, no 16-bit operations are supported in logical instructions such as `xor,or,and`, and these instructions accept only register pair `hl` as an address pointer. Also note that a subroutine call is quite expensive in this processor. A `call/ret` pair takes a total of nine cycles, and consumes stack by four bytes. For comparison, AVR's `rcall/ret` pair (short call) takes seven cycles with two stack bytes [12]. A programmer must try to minimize the number of subroutine calls if aiming at a speed record on RL78. Interestingly, however, a `push/pop` pair is inexpensive and handy for avoiding register starvation.

3 Interface and Metrics

3.1 Interface

First of all, we have adopted a commonly accepted program interface in embedded systems; i.e. we implemented a target primitive as a subroutine callable from C language, which we believe is a portable and small-overhead choice. In the following we use the calling conventions described in [13]: (1) the first argument is passed by `ax`, (2) other arguments are passed through stack, and (3) `hl` must be recovered at the end of the subroutine (callee-save register).

To reduce register pressure, we use only the first argument, and `ax` points to the RAM area prepared by a caller, which includes a message block to be encrypted or hashed, secret key (if any), a flag indicating first/middle/last block, and temporary buffer for internal use. For instance, one of our AES encryption routines has the following argument format that consists of a total of 50 bytes:

```
Bytes 00-15: plaintext/ciphertext
Bytes 16-31: secret key
Bytes 32-33: flag (bit 0/1: active in the first/last block)
Bytes 34-49: buffer for internal use
```

The first 16-byte plaintext is replaced by its correspondent ciphertext after encryption. It is allowed to overwrite this area during encryption to minimize RAM usage. The secret key can be also destroyed, but our codes were designed so that a caller does not have to rewrite the same secret key every block when

encrypting a multiple number of blocks. Note that this does not always mean that the secret key area remains unchanged.

Our routines process one block in a single call. A caller is responsible for creating a block format from a target message and calling a routine block-by-block. This looks common in block cipher setting, but it is not obvious which side (caller or callee) should be responsible for formatting the last block including a padding in hash functions from performance point of view. In fact, in embedded applications message size is often fixed or at most varies within a small range and in such a case a fully general interface supporting an arbitrary length in the callee side could simply lead to an overhead. Therefore in this paper we have decided that, from a minimalist point of view, rather than pursuing a full black box design, the interface policy of hash functions should be the same as that of block ciphers.

3.2 ROM/RAM Count

There does not seem to exist a consensus of how to count ROM/RAM size of a given crypto subroutine, especially RAM bytes on an embedded processor. Since RAM is much more expensive than ROM, it is important to give unambiguous information about RAM consumption to an embedded system programmer.

For instance, out of three implementation papers on the AVR processor [14][15][16], the first one does not count mandatory parameters such as plaintext and key areas as RAM bytes, the first and second papers seem to have excluded stack consumption from the RAM count, and the second and third papers introduce an uncommon subroutine calling convention where a callee can destroy any register without restoration.

For another example, a code of Grøstl designed by Feichtner on the same processor [10] pushes/pops 20 callee-save registers out of a total of 32 registers at the beginning and end of the routine, which agrees with our code design policy. In our metric, the ROM/RAM size should indicate the entire resource consumption of a target subroutine, and hence, for example, we count the size of RAM that the following sample code consumes as (at least) twelve bytes:

```
_Encryption_Routine:   ; 4 bytes for calling this routine itself
    push hl            ; 2 bytes for storing hl, callee-save register
    movw hl,ax

    call _Leaf_Routine ; 4 bytes for calling this function
    ..
    pop  hl            ; restoring hl
    ret

_Leaf_Routine:
    push bc            ; 2 bytes for pushing bc
    ..
    pop  bc
    ret
```

A consequence of this is that 64-byte RAM is very restrictive for most 128-bit block ciphers with 128-bit key, because 32-byte RAM is a must (plaintext+key) and often we need additional 16 bytes for keeping temporary data. Moreover, as mentioned above, if we call an internal subroutine, 12-byte stack is needed. At this stage we have only four free RAM bytes.

3.3 Categorization as to Resources

One of the purposes of this paper is to give a system programmer practical information on size and speed trade-offs for each algorithm, not only pin-point record data. We expect that this approach will make performance characteristics of each algorithm much clearer, and in addition, will reveal that a specific size-speed combination is **impossible** to implement on this processor, which is also important information for a programmer.

On the other hand, it is not realistic to write codes for too many possible ROM/RAM size combinations. Hence in this paper, we introduce several categories as to given memory size. Specifically, we categorize ROM size variations into 512B, 1024B and 2048B and RAM size variations into 64B, 128B, 256B, 512B (first two are for block ciphers and latter two are for hash functions). Our interest is to find out in which category i.e. in which ROM/RAM combination, a target primitive is implementable or not, and if yes, what performance it can achieve within the amount of resources specified in the category.

Our complete implementation results are given in appendix A, but in the next section, we use the following type of diagram to illustrate a performance portfolio of each algorithm:

	ROM-Min (400B)	ROM-512B	ROM-1024B	ROM-2048B
RAM-128B	20,000	9,000	3,000	-
RAM-64B	x	x	4,000	3,500

Table 2. Portfolio of a Primitive (an example).

This table shows five different implementations for the target algorithm, one of which runs at 3,000 cycles/block with less than 1024 ROM bytes and 128 RAM bytes. If the given RAM resource is reduced down to 64 bytes, then its speed also goes down to 4,000 cycles/block. Also if only 512 ROM bytes is available, then its speed penalty becomes serious, 9,000 cycles/block. 'x' means that it is (or seems) impossible to implement in this category, and '-' denotes "satiated", i.e. already reached enough resource for achieving high speed and having further resource does not lead to significant speed-up as compared with other implementations (left or down whichever faster; 3,000 cycles/block in this case). From this table, we can deliver important messages to an application programmer such as:

– 1024B/128B are most reasonable ROM/RAM resources for this primitive.
– If ROM size is less than 1024 bytes, its speed rapidly worsens.
– If only 512B/64B are available, using this primitive should be given up.

In the next section, we skip rows and columns that contain only '-' or 'x' entries. A special case is the left-most column, which is a ROM-minimum implementation concentrating on minimizing ROM memory, lowering priority of speed. We believe that this information is also practically important. In fact, in some industrial systems, speed is not an issue since mechanical motion is usually far more time-consuming than cryptographic applications.

3.4 Portability

All of our codes were written in assembly language in the small model, but we tried them to be as portable as possible. Our codes are relocatable, i.e. independent of the address where the code/data are located in physical memory.

Also we took into consideration that our codes should not conflict with other modules, and therefore avoided to access system memory area. We used only a single bank (RL78 has four register banks, each of which has an independent register memory), and also we did not access 256-byte short address RAM, which is a special area where a fast and short instruction is available. This area is shared with system programs and use of this area could affect portability.

4 Implementations

4.1 Block Ciphers

AES: We implemented encryption-only and encryption-decryption versions separately for all block ciphers. For AES, all of our RAM-64B programs are based on "flat" implementation, i.e. its round function including a key scheduling step (due to the on-the-fly implementation) does not contain any loop/subroutine inside. These flat programs required 1KB ROM for encryption-only version and 2KB ROM for encryption-decryption version, respectively. To reduce the ROM size to 512B and 1KB, we introduced a loop inside `MixColumns`, having a single vector-matrix multiplication code, instead of having the entire matrix-matrix multiplication. As a result, the RAM size of these codes exceeded 64B.

On RL78, multiplying {02} can be done with the following simple sequence of instructions without a branch:

```
shl  a,1
sknc
xor  a,#01bh
```

In table 4, the first number of each entry denotes encryption cycles, and the second/third number shows decryption cycles for first/second (and later) block, respectively. Note that in decryption, the second and later blocks can be faster than the first block by skipping part of its key scheduling. The right most column is another implementation for aiming at maximum speed by unrolling non-critical parts, which exceeded 2048 ROM bytes.

It looks that around 3,800 and 5,700 cycles/block is the fastest speed of AES encryption and decryption on this processor, respectively.

	ROM-Min (486B)	ROM-512B	ROM-1024B
RAM-128B	7,288	6,622	-
RAM-64B	x	x	3,855

Table 3. AES128 Encryption-only Portfolio.

	ROM-Min (970B)	ROM-1024B	ROM-2048B	Fast (2380B)
RAM-128B	7,743:12,683/10,862	7,339:10,636/9,106	-	-
RAM-64B	x	x	3,917:6,804/5,911	3,865:6,541/5,706

Table 4. AES128 Encryption-Decryption Portfolio.

Camellia: The key scheduling part of Camellia has rotate shifts on 128-bit data whose shift counts are irregular. This irregularity and its FL functions lead to a penalty in terms of ROM size. However since Camellia has Feistel structure, its decryption is as fast as encryption and is faster than AES decryption. Main ROM-size and speed trade-offs come from the number of different S-boxes that the code contains, which can vary from one (256B) to four (1KB), and the number of independent rotate shift routines. Its speed converges to around 4,000 cycles/block for both encryption and decryption.

Several efforts were made to create ROM-minimum codes (800B for encryption-only and 1024B for encryption-decryption): The P matrix is stored in an 8-byte ROM table and its computation is done bit-by-bit. Also having only one rotate shift routine that shifts 128-bit data by one bit, an n-bit rotate shift is done by running the routine n times. Obviously these methods resulted in heavy performance penalty, but it should be again noted that we focused on minimizing ROM size, and hence this is a forget-the-speed option, unlike other categories.

	ROM-Min (800B)	ROM-1024B	ROM-2048B
RAM-128B	43,182/39,358	5,539/4,631	4,738/3,966
RAM-64B		5,733/4,820	4,918/4,125

Table 5. Camellia128 Encryption-only Portfolio.

	ROM-1024B	ROM-2048B
RAM-128B	43,190/39,357 : 175,417/152,023	4,978/4,125 : 5,255/4,244
RAM-64B	x	5,126/4,337 : 5,512/4,477

Table 6. Camellia128 Encryption-Decryption Portfolio.

Clefia: Clefia has two independent 256-byte S-boxes, two 4x4 Matrices and a 240-byte constant value used in its key scheduling part, which causes a heavy ROM size penalty. However the constant value can be generated on-the-fly. All implementations except ROM-2048B versions used this technique to reduce their code size.

The methodology for implementing Clefia is basically the same as that for AES. RAM-128B versions are a bit faster than RAM-64B version, which is mainly because the former codes were able to allocate more and enough memory for on-the-fly subkey. The ROM-2048B versions have unrolled their most critical parts but are not still flat programs. We have not written a fastest possible flat code by accepting more ROM bytes, but it looks that around 5,000 cycles/block is a maximal performance of this primitive.

	ROM-Min (961B)	ROM-1024B	ROM-2048B
RAM-128B	17,434	12,367	8,208/5,302
RAM-64B	x	x	9,142/6,194

Table 7. Clefia128 Encryption-only Portfolio.

	ROM-Min (1,309B)	ROM-2048B
RAM-128B	18,062 : 18,759	9,399/6,208 : 9,931/6,740
RAM-64B		11,388/7,768 : 11,419/7,799

Table 8. Clefia128 Encryption-Decryption Portfolio.

Present: Present is a 64-bit block cipher with 80-bit key, and a 64-byte memory is enough for its RAM size. On the other hand, this algorithm is heavily optimized for hardware and in software we have to compute its round function bit-by-bit. Main design trade-offs come from a 4-bit S-box v.s. an 8-bit S-box. Our ROM-1024B version for encryption-only and ROM-2048B version for encryption-decryption have a latter choice. Once an 8-bit output of the S-box is stored on register `x`, then the pLayer of Present can be implemented basically by a repetition of the following simple code:

```
mov   a,reg1
addw  ax,ax     (shrw ax,1  in decryption)
mov   reg2,a
```

As seen below, its fastest speed is around 9,000 cycles/block, significantly slower than other lightweight block ciphers. On the other hand, since the structure of Present is very simple, further reduction of code size is possible at the cost of speed. Our ROM-minimum implementation requires only 210 ROM bytes (encryption only), which runs at the speed of 144,879 cycles/block.

	ROM-Min (210B)	ROM-512B	ROM-1024B
RAM-64B	144,879	122,00	9,007

Table 9. Present80 Encryption-only Portfolio.

	ROM-512B	ROM-1024B	ROM-2048B
RAM-64B	61,634 : 104,902/60,834	13,883 : 16,046/14,014	9,007 : 10,823/8,920

Table 10. Present80 Encryption-Decryption Portfolio.

4.2 Hash Functions

SHA: It is obvious that SHA-256/SHA-512 cannot be implemented within 128/256 RAM bytes and we assume that 256/512-byte RAM is given. Then we have room for storing intermediate message words $W_i (0 \leq i \leq 15)$ doubly. This makes the message scheduling part simpler by arranging the double-size message buffer as $W_0||W_1||...||W_{15}||W_0||W_1||...||W_{15}$ where the first W_i and the second W_i always the same. All of our codes of SHA-256/SHA-512 use this method.

For SHA256, our ROM-2048B version has achieved a flat code - its step function is fully unrolled -, which actually needed only 1,239 ROM bytes. This ROM size was able to be reduced to 1024B by introducing a byte-wise loop within the `Ch` and `Maj` functions and making frequent xor-to-memory operations a subroutine. The ROM-minimum version has a single rotate shift routine that rotates by one bit (as that of Camellia). For SHA512, 2048 ROM bytes were not enough for creating a flat code and 2,499 bytes were needed. The implementation method for the ROM-2048B/ROM-minimum version of SHA-512 is the same as that for the ROM-1024B/ROM-minimum version of SHA-256, respectively.

	ROM-Min (796B)	ROM-1024B	ROM-2048B
RAM-256B	216,775/216,393	41,175/40,793	25,265/25,143

Table 11. Portfolio of SHA-256.

	ROM-Min (1,285B)	ROM-2048B	Fast (2499B)
RAM-512B	819,034/818,268	81,610/80,844	66,008/65,562

Table 12. Portfolio of SHA-512.

Keccak: Keccak can be implemented within 256 RAM bytes only if a message size is always within a single block. Hence we assume that 512-byte RAM is given. Our flat code slightly exceeded 2048B, and in order to create a smaller code, we had to deal with reduction of a total of 24 different rotate shift operations in the ρ function. Our ROM-1024B code has a 1-bit rotate shift routine and m-byte rotate shift routines $(1 \leq m \leq 7)$ independently, and a given $n = 8n_1 + n_2$-bit rotate shift is done by carrying out the n_1-byte shift routine and an n_2-time repetition of the 1-bit shift routine. Similarly our ROM-512B routine has a 1-byte rotate routine and a 1-bit rotate routine, and an n-bit shift is made by an n_1-time repetition of the former and an n_2-time repetition of the latter. The ROM-minimum version has a 1-bit rotate routine only and repeating it n times creates an n-bit rotate shift.

	ROM-Min (453B)	ROM-512B	ROM-1024B	ROM-2048B
RAM-512B	516,528/517,022	237,960/238,454	155,209/155,703	118,705/119,171
	Fast (2,214B)			
RAM-512B	110,185/110,651			

Table 13. Portfolio of Keccak.

Skein: An advantage of Skein is that it allows a very compact ROM/RAM implementation. 2048 ROM bytes and 256 RAM bytes are enough for creating a flat implementation of Skein-256. The methodology for reducing its ROM size is basically the same as that of Keccak. Our ROM-1024B/512B/Min versions correspond to Keccak's ROM-1024B/512B/Min versions, respectively.

For Skein-512, our ROM-2048B code contains n_1-byte rotate routines ($1 \leq n_1 \leq 7$) and n_2-bit rotate routines ($1 \leq n_2 \leq 7$) independently, and our ROM-1024B code uses four rotate routines (1-byte, 3-byte, 1-bit and 5-bit shifts) to create a given n-bit rotation. The ROM-512B version has a 1-byte rotate routine and a 1-bit rotate routine.

	ROM-Min (396B)	ROM-512B	ROM-1024B	ROM-2048B
RAM-256B	122,348/121,964	42,566/42,182	30,524/30,140	20,156/19,772

Table 14. Portfolio of Skein-256.

	ROM-Min (457B)	ROM-512B	ROM-1024B	ROM-2048B
RAM-256B	823,806/823,038	121,590/120,822	66,834/66,066	46,747/46,299

Table 15. Portfolio of Skein-512.

Grøstl: The most time-consuming part of Grøstl is obviously a computation of `MixBytes`. To minimize speed penalty, reducing this part must be the last option. Most of our ROM-2048B/1024B programs have an unrolled 8-dimensional vector-matrix multiplication code and ROM size reduction comes from `AddRoundConst`. Special implementations were made for the RAM-512B versions Grøstl-256. In these programs, the 256-byte S-box is copied from ROM to RAM before starting the first block for better performance, since reading from RAM is faster than reading from ROM. As a result, we achieved a small gain of performance.

In the following table, the cycle count of the output transformation Ω is included in that of the first block.

	ROM-Min (615B)	ROM-1024B	ROM-2048B
RAM-512B		95,271/63,286	73,011/47,746
RAM-256B	164,664/111,349	99,625/67,126	77,365/51,586

Table 16. Portfolio of Grøstl-256.

	ROM-Min (672B)	ROM-1024B	ROM-2048B
RAM-512B	452,122/306,713	277,626/188,889	215,634/144,159

Table 17. Portfolio of Grøstl-512.

5 Comparative Figures

This section briefly discusses performance comparison of our target algorithms. Throughout this section, left and right graphs correspond to low and high resources (1024 ROM bytes or less/2048 ROM bytes), respectively. The horizontal axis denotes message length (bytes) and the vertical axis shows speed (cycles/byte). We have excluded ROM-minimum implementations since they are not optimized for operation speed. Note that only points make sense as performance data. Lines are added only for visibility of these graphs.

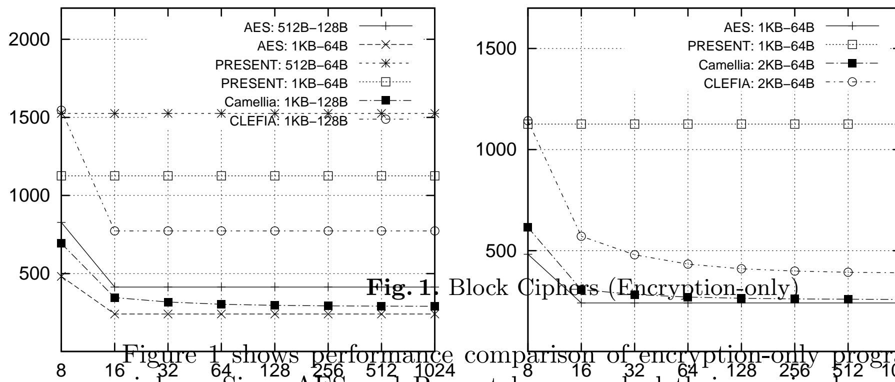

Fig. 1. Block Ciphers (Encryption-only)

Figure 1 shows performance comparison of encryption-only programs of block ciphers. Since AES and Present have reached their maximal speed with 1KB ROM, they are also included in the right graph. As easily seen, AES and Camellia are generally good choices for both low and high resources, and if ROM size is limited to 512 bytes, our options are limited to AES or Present.

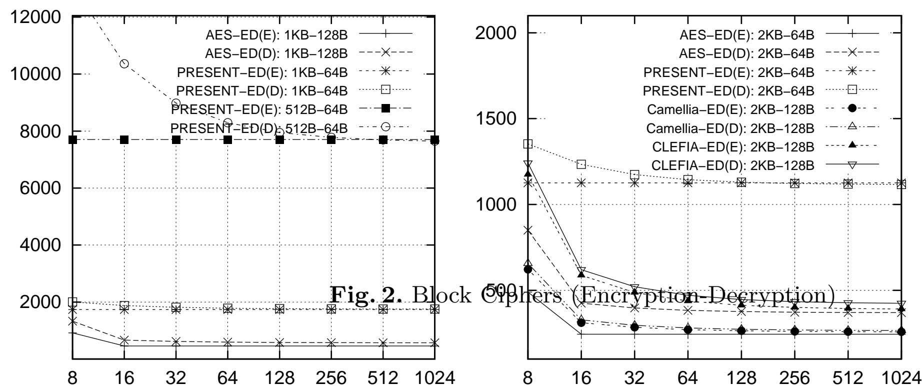

Fig. 2. Block Ciphers (Encryption-Decryption)

Figure 2 shows performance comparison of encryption-decryption programs of block ciphers, where `(E)/(D)` denotes encryption/decryption, respectively. Again AES achieves a good performance. In this case, if a given ROM size is 1KB, Camellia and Clefia are excluded. Note that however if 2KB ROM is given, Camellia's decryption speed outperforms AES.

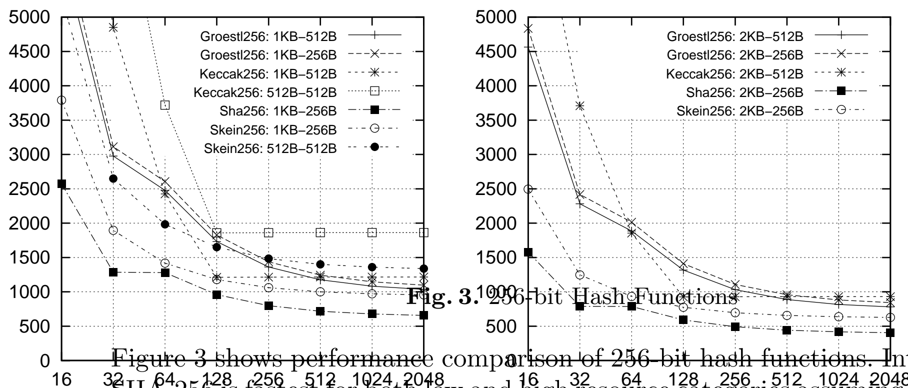

Fig. 3. 256-bit Hash Functions

Figure 3 shows performance comparison of 256-bit hash functions. Interestingly, SHA-256 is fastest for both low and high resource categories assuming 1KB ROM is given. However SHA-256 is excluded and Keccak-256 and Skein-256 survive when ROM size is limited to 512 bytes.

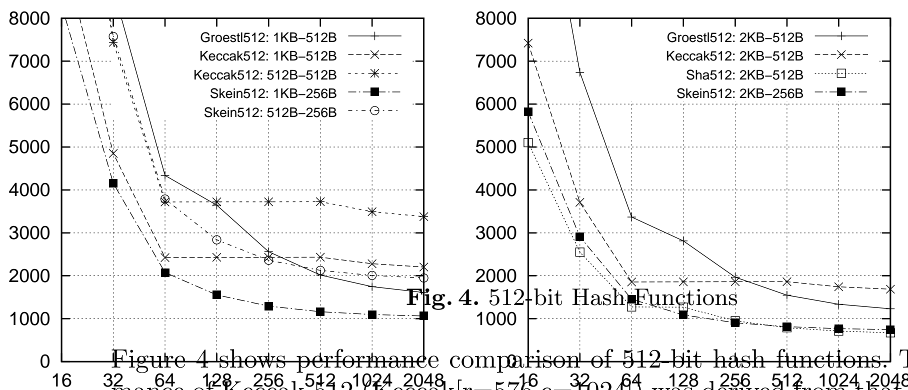

Fig. 4. 512-bit Hash Functions

Figure 4 shows performance comparison of 512-bit hash functions. The performance of Keccak-512 (Keccak[r=576,c=1024]) was derived from that of Keccak-256 (Keccak[r=1088,c=512]), since these codes can be almost the same except for input block sizes. This case Skein-512 is fastest on low resources but SHA-512 again remains a good choice when 2KB ROM is available. The SHA-3 winner

Keccak is not a high speed primitive, but is a low memory option with Skein on this processor.

6 Concluding Remarks

We here mention a couple of possibilities to further improve performance on this processor. As described in section 3.4, we did not access any short address RAM area to maintain portability of our programs. In general it is expected that utilizing this area could lead to a shorter code, but in our case, its gain seems to be limited unless we aim at a new ROM-minimum record.

Another possibility for speeding-up is to copy constant ROM data to RAM. RL78 takes one cycle to read a RAM byte/word to register, but takes four cycles to read from ROM. So if we copy ROM data to RAM before starting the routine or in the first block, then the overall performance could be improved in return for additional RAM resources. We tried this implementation in RAM-512B versions of Grøstl only, which resulted in 10% performance improvement, but obviously there are many possibilities of applying this method to other primitives.

Also we have found that minimizing a ROM size is a tricky puzzle. Reducing 10 bytes often makes a code 10 times slower. While our strategy in minimizing the ROM size was just to ignore speed, there must exist other various trade-offs between size and speed in exploring this extreme goal. Going deeper to this direction looks like another interesting topic.

References

[1] ECRYPT II, Implementations of low cost block ciphers in Atmel AVR devices, http://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/

[2] ECRYPT II, Implementations of hash functions in Atmel AVR devices, http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/

[3] Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, NIST, 2001.

[4] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, T. Tokita: The 128-Bit Block Cipher Camellia, IEICE Trans. Fundamentals, Vol.E85-A, No.1, pp.11-24, 2002.

[5] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata: The 128-bit Blockcipher CLEFIA, FSE 2007, LNCS 4593, Springer-Verlag, 2007.

[6] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, C. Vikkelsoe: PRESENT: An Ultra-Lightweight Block Cipher, CHES 2007, LNCS 4727, Springer-Verlag, 2007.

[7] Secure Hash Standard (SHS), Federal Information Processing Standards Publication 180-3, NIST, 2008.

[8] G. Bertoni, J. Daemen, M. Peeters, G.V. Assche: The Keccak sponge function family, http://keccak.noekeon.org/

[9] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, J. Walker: The Skein Hash Function Family, http://www.skein-hash.info/

[10] P. Gauravaram1, L.R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, S.S. Thomsen: Grøstl - a SHA-3 candidate, http://www.groestl.info/

[11] RL78 Family, Resesas Electronics, http://www.renesas.com/products/mpumcu/rl78/index.jsp

[12] AVR Instruction Set, Atmel Corporation, http://www.atmel.com/images/doc0856.pdf

[13] CodeSuite+ V1.02.00 Integrated Development Environment User's Manual: RL78,78K0R Coding, http://documentation.renesas.com/doc/products/tool/doc/r20ut0977ej0100_qscd78.pdf

[14] T. Eisenbarth, S. S. Kumar, C. Paar, A. Poschmann, L. Uhsadel: A Survey of Lightweight-Cryptography Implementations, IEEE Design & Test of Computers, 24(6):522-533, 2007.

[15] T. Eisenbarth, Z. Gong, T. Guneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, F.-X. Standaert, L. van Oldeneel tot Oldenzeel: Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices, Africacrypt 2012, LNCS 7374, 2012

[16] J. Balasch, B. Ege, T. Eisenbarth, B. Gerard, Z. Gong, T. Guneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Poppelmann, F. Regazzoni, F.-X. Standaert, G.V. Assche, R.V. Keer, L.v. Oldeneel tot Oldenzeel, I.v. Maurich: Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices, cryptology e-Print archive, report 2012/507.

Appendix: Summary of Our Implementation Results

Block Ciphers

Category		AES128 Enc-only			AES128 Enc,Dec		
ROM	RAM	ROM	RAM	cycles/block	ROM	RAM	cycles/block
1KB	128B	-	-	-	1,024	84	7,339 : 10,636/9,106
512B	128B	510	78	6,622	x	x	x
Min	128B	486	78	7,288	970	84	7,743 : 12,683/10,862
Fast	64B	-	-	-	2,380	64	3,865 : 6,541/5,706
2KB	64B	-	-	-	1,989	64	3,917 : 6,804/5,911
1KB	64B	1,021	60	3,855	x	x	x

Category		Camellia128 Enc-only			Camellia128 Enc,Dec		
ROM	RAM	ROM	RAM	cycles/block	ROM	RAM	cycles/block
2KB	128B	2,004	70	4,738/3,966	2,047	74	4,978/4,125 : 5,255/4,244
1KB	128B	1,023	70	5,539/4,631	1,020	78	43,190/39,357 : 175,417/152,023
Min	128B	800	74	43,182/39,358	x	x	x
2KB	64B	2,037	64	4,918/4,125	2,033	64	5,216/4,337 : 5,512/4,477
1KB	64B	1,024	64	5,733/4,820	x	x	x

Category		Clefia128 Enc-only			Clefia128 Enc,Dec		
ROM	RAM	ROM	RAM	cycles/block	ROM	RAM	cycles/block
2KB	128B	2,006	94	8,208/5,302	2,040	86	9,399/6,208 : 9,931/6,740
1KB	128B	1,024	74	12,367	x	x	x
Min	128B	961	76	17,434	1,309	76	18,062 : 18,759
2KB	64B	2,037	64	9,142/6,194	2,026	64	11,388/7,768 : 11,419/7,799

Category		Present80 Enc-only			Present80 Enc,Dec		
ROM	RAM	ROM	RAM	cycles/block	ROM	RAM	cycles/block
2KB	64B	-	-	-	1,855	48	9,007 : 10,823/8,920
1KB	64B	897	42	9,007	1,009	54	13,883 : 16,046/14,014
512B	64B	510	46	12,200	512,	62	61,634 : 104,902/60,834
Min	64B	210	54	144,879			

Hash functions

Category ROM	RAM	SHA256 ROM	SHA256 RAM	SHA256 cycles/block	SHA512 ROM	SHA512 RAM	SHA512 cycles/block
Fast	512B	-	-	-	2,499	420	66,008/65,562
2KB	512B	-	-	-	2,034	428	81,610/80,844
Min	512B	-	-	-	1,285	430	819,034/818,268
2KB	256B	1,239	216	25,265/25,143	x	x	x
1KB	256B	1,016	224	41,175/40,793	x	x	x
Min	256B	796	224	216,775/216,393	x	x	x

Category ROM	RAM	Keccak256 ROM	Keccak256 RAM	Keccak256 cycles/block
Fast	512B	2,214	392	110,185/110,651
2KB	512B	2,017	392	118,705/119,171
1KB	512B	1,024	392	155,209/155,703
512B	512B	512	392	237,960/238,454
Min	512B	453	392	516,528/517,022

Category ROM	RAM	Skein256 ROM	Skein256 RAM	Skein256 cycles/block	Skein512 ROM	Skein512 RAM	Skein512 cycles/block
2KB	256B	1,615	166	20,156/19,772	1,921	254	46,747/46,299
1KB	256B	1,015	166	30,524/30,140	1,024	252	66,834/66,066
512B	256B	502	144	42,566/42,182	509	252	121,590/120,822
Min	256B	396	144	122,348/121,964	457	252	823,806/823,038

Category ROM	RAM	Grøstl256 ROM	Grøstl256 RAM	Grøstl256 cycles/block	Grøstl512 ROM	Grøstl512 RAM	Grøstl512 cycles/block
2KB	512B	1,481	476	73,011/47,746	2,044	412	215,634/144,159
1KB	512B	1,023	476	95,271/63,286	1,015	412	277,627/188,889
Min	512B				672	412	452,122/306,713
2KB	256B	1,471	220	77,365/51,586	x	x	x
1KB	256B	1,019	220	99,625/67,126	x	x	x
Min	256B	615	230	164,664/111,349	x	x	x