Feasibility and practicability of standardized cryptography on 4-bit micro controllers

Jacob, Nisha; Saetang, Sirote; Chen, Chien-Ning; Kutzner, Sebastian; Ling, San; Poschmann, Axel

2013

Jacob, N., Saetang, S., Chen, C. N., Kutzner, S., Ling, S., & Poschmann, A. (2013). Feasibility and Practicability of Standardized Cryptography on 4-bit Micro Controllers. 19th International Conference, Selected Areas in Cryptography (SAC) 2012, 7707, pp.184-201.

<https://hdl.handle.net/10356/99650>

https://doi.org/10.1007/978-3-642-35999-6_13

© 2013 Springer-Verlag Berlin Heidelberg. This is the author created version of a work that has been peer reviewed and accepted for publication by 19th International Conference, Selected Areas in Cryptography (SAC) 2012, Springer-Verlag Berlin Heidelberg. It incorporates referee's comments but changes resulting from the publishing process, such as copyediting, structural formatting, may not be reflected in this document. The published version is available at: [http://dx.doi.org/10.1007/978-3-642-35999-6_13].

Downloaded on 26 Apr 2023 19:25:59 SGT

Feasibility and Practicability of Standardized Cryptography on 4-bit Micro Controllers

Nisha Jacob¹, Sirote Saetang¹, Chien-Ning Chen²,
Sebastian Kutzner², San Ling¹, and Axel Poschmann^{1,2}

School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
{`njacob,sirote.tang,lingsan,aposchmann`}@ntu.edu.sg

Physical Analysis & Cryptographic Engineering (PACE)
Nanyang Technological University, Singapore
{`chienning,skutzner`}@ntu.edu.sg

Abstract. Myriads of ultra-constrained 4-bit micro controllers (MCUs) are deployed in (mostly) legacy devices, some in security sensitive applications, such as remote access and control systems or all sort of sensors. Yet the feasibility and practicability of standardized cryptography on 4-bit MCUs has been mostly neglected. In this work we close this gap and provide, to the best of our knowledge, the first implementations of ECC and SHA-1, and the fastest implementation of AES on a 4-bit MCU. Though it is not the main focus of this paper, we have investigated the SCA resistance trade-offs for ECC by implementing a variety of countermeasures. We hope that our comprehensive, highly energy-efficient crypto library—that even outperforms all previously published implementations on low-power 8-bit MCUs—will give rise to a variety of security functionalities, previously thought to be too demanding for these ultra-constrained devices.

Keywords: 4-bit MCU, AES, Elliptic Curve Cryptography, SHA-1, Lightweight Cryptography.

1 Introduction

Current market figures for embedded processors are hard to obtain in the open literature, nevertheless a book from 2003 indicates that 98 percent of all processors are embedded [49] and a publication from 1997 states with regards to the market shares of 4-, 8- and 16-bit micro controllers (MCUs): “*Like any ecosystem, the smallest creatures appear in the greatest numbers*” [48]. Though these market shares have certainly changed over the last 15 years—away from 4-bit MCUs and towards 8-, 16-, and 32-bit MCUs—it also indicates that myriads of 4-bit MCUs are deployed in legacy devices, e.g. watches and toys, but also security sensitive applications such as remote access and control systems, car immobilizers [46], one-time password generators, and all sorts of sensors.

8-bit MCUs have been long used as the platform of choice to evaluate the efficiency of cryptographic algorithms in embedded devices. On one hand this is a perfectly sound forward-looking approach, while on the other hand, this approach excludes myriads of 4-bit legacy devices, many of which could benefit from security functionalities. It is these legacy devices which make us believe that it is worth to evaluate the feasibility and practicability of cryptographic algorithms on 4-bit MCUs.

The feasibility of block cipher implementations on 4-bit MCUs has been shown in [51] using PRESENT [4] as an example. Shortly after, HUMMINGBIRD [13]—already broken [40] and replaced by a successor called HUMMINGBIRD-2 [14]—was implemented on the same 4-bit MCU [15]. Recently, also AES [35] as the first standardized algorithm¹ and PRINTcipher [25] have been implemented on the same platform [24].

In order to enable widespread security functionalities—such as legally binding sensor readings or secure firmware updates—not only block ciphers but also cryptographic hash functions and public key cryptography are required. To the best of our knowledge, no implementation on a 4-bit MCU has been published for the latter two categories so far. Hence, we close this gap by providing the first implementations of the SHA-1 hash function [36] and Elliptic Curve Cryptography (ECC) using the 160-bit curve secp160r1 [7]. We have also implemented AES, yielding the fastest implementation on a 4-bit MCU. All our highly-optimized implementations even outperform implementations on low-power 8-bit MCUs.

As side-channel attacks (SCA), such as Timing attacks [28], Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [29], are a major concern for embedded devices, we have implemented basic countermeasures against them, though it is beyond the scope of this paper. As a matter of fact, our implementations of AES and SHA-1 are timing attack resistant, and for ECC we have investigated different DPA resistance-security trade-offs.

The remainder of this work is organized as follows. In Section 2 the target platform is briefly introduced before various implementations with different optimization goals of the AES and SHA-1 are presented in Sections 3.1 and 4, respectively. Subsequently, in Section 5 elliptic curve cryptography is treated. A wide variety of optimization tricks are described and different countermeasures against timing attacks, SPA and DPA are implemented. Then, in Section 6 our results are—where available—compared with previous implementations on 4-bit MCUs, and for the sake of completeness with implementations on 8-bit MCUs. Finally, Section 7 concludes this paper.

2 Target Platform and Design Flow

The Epson S1C63 family of MCUs was introduced in 2011 and is one of the most recent 4-bit low-power architectures. All members of the S1C63 family have a 4-bit core along with ROM, RAM, LCD drivers, I/O ports and a wide instruction

¹ PRESENT has been recently standardized as well [20].

set (mostly 1-2 cycles) with a linear addressing space without pages. It also has a two-stage pipeline (fetch and execute) and a maximum of 15 and 63 hardware and software interrupt vectors respectively, depending on the MCU being used. The MCUs differ mainly in the memory size ranging from 6kB to 50kB and on-board components, such as UART or hardware multiplier [42]. Our goal was to use the most constrained S1C63003 which has just 6kB of code ROM, 128 bytes of RAM and no data ROM. However, since this is not sufficient for the extensive computations of ECC, we moved to S1C63016 (only for ECC), which has 26kB of code ROM, 1kB of RAM and 2kB of data ROM. S1C63016 also has an integer multiplier/divider which is absent in S1C63003.

There are 47 basic types of instructions with 8 addressing modes (411 instructions in all). It has two 4-bit data registers A and B; two 16-bit index registers X, Y for 4-bit indirect data access; two stack pointers SP1 (16-bit) and SP2 (8-bit); a 4-bit condition flag register F which consists of extension E, interrupt I, carry C and zero flag Z; an 8-bit extend register EXT for extended addressing mode. Table 1 gives a list of the most frequently used instruction and its instruction cycles. Each instruction cycle is equal to 2 clock cycles².

Table 1. Frequently used instructions [42] of EPSON S1C63 family MCU.

Mnemonic*	Cycles
LD [%ir]+,%r; LD %r,[%ir]+; ADC %r,[%ir]; ADC %r, [%ir]+	1
AND %F,imm4; OR %F,imm4; CMP %r,[%ir]+; CMP [%ir]+,%r	1
JR sign8; JRNC sign8; JRNZ sign8; CALR imm8; RET	1
LDB % EXT, %BA; LDB %rr, imm8; ADD %ir, %BA; ADD %ir,sign8	1
LD [%ir]+,[%ir]+; LDB [%X]+,%BA; ADC [%ir]+,%r;	2
INC [addr6]; DEC [addr6]; XOR [%ir]+,%r; EX %r,[%ir]+	2
RETD	3

*ir = index register (X or Y); r= data register (A, B or F); addr6= 6-bit absolute data address; imm8= 8-bit immediate data; sign8= signed 8-bit digit; rr= XL,XH,YL,YH.

Details of the design flow of this MCU can be found in [45]. For debugging we use a software simulator on the PC that can even simulate an LCD (Fig.1(a)), and a hardware tool called *In-Circuit Emulator (ICE)*, an FPGA-based emulation board (Fig.1(b)) [44]. The code is first tested on the software simulator or on the ICE and can then be burned on the target board. The advantage of using the ICE over software simulator is to ensure the proper operation of the system before burning it onto the target board (Figure 1(c)). The software simulator and the debugger are used to get the cycle count and code size of the tested functions which are the two most common performance metrics for embedded platforms. Furthermore, we provide estimated energy consumptions based on public datasheets as an additional performance metric.

² In the remainder of the paper we refer to instruction cycles as cycles

Fig. 1. S1C63 family development tools [44].

3 The Advanced Encryption Standard

In this section we will first give a brief introduction to the Advanced Encryption Standard (AES). Subsequently, starting from a naïve implementation, we will describe several tricks applied for either code size or speed optimization. Table 2 summarizes our AES implementation results.

3.1 Introduction to AES

The Advanced Encryption Standard [35] is a symmetric block cipher with a block size of 128 bits and key sizes of 128, 192, and 256 bits, called AES-128, AES-192, and AES-256, respectively. The input and round keys are organized as a 4×4 state array, with each element being one byte. Encryption and decryption require $N_r = 10, 12$, and 14 rounds for AES-128, AES-192, and AES-256, respectively. Each round of AES is composed of the following four operations: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. For details please refer to [11, 35].

3.2 Naïve Implementation of AES

To implement AES on a 4-bit architecture, every byte of the state array and the key state array, $state_{rc}$ and key_{rc} , should be split into two 4-bit values. We use $state_{rcn}$ and key_{rcn} , where $n = 0$ or 1 denotes the high or low nibble. We now give a brief overview of our naïve AES implementation on Epson’s 4-bit MCU.

SubBytes: In a straight forward implementation, each byte of the state array is substituted by using a look-up table, called S-Box, that consists of 256 8-bit entries. We split the S-Box into 16 tables, $SBOX_i$ with $0 \leq i \leq 15$, each

containing 16 entries. According to the manual of our target platform [42], table look-ups are realized by a combination of “LD”, “JR %BA” and “RETD” instructions. It takes 11 cycles to substitute one byte, and in total *SubBytes* takes 163 cycles, i.e., 11 cycles for the first look-up, 10×15 cycles for the remaining look-ups (only 10 cycles/look-up because of the post-increment functionality), and 2 cycles for the call/return instruction and has a code size of 475 bytes.

ShiftRows: We exchange the location of elements in rows 1 to 3 of the state array nibble by nibble. For example, when exchanging $state_{100,101}$ and $state_{130,131}$, we have to copy $state_{100,101}$ to registers A and B, then copy $state_{130,131}$ to $state_{100,101}$. Finally, the $state_{130,131}$ is replaced by registers A and B. In total 90 cycles and 36 bytes of code size are required for *ShiftRows* and *InvShiftRows*.

MixColumns: This step is a matrix multiplication, using coefficients {01}, {02}, and {03} for encryption ({09}, {0B}, {0D} and {0E} for decryption). The element multiplication can be implemented by using only shifts and XORs. In case the higher nibble of the multiplication result is greater than 7, we need a modulo reduction, that is an XOR with the constant “0x1B”. To achieve a constant time implementation, we use a dummy reduction (XOR with “0x00”) in case the higher nibble is less or equal than 7. In our constant time implementation, in total *MixColumns* requires 1297 cycles and 555 bytes of code size, and *invMixColumns* requires 6207 cycles and 1554 bytes of code size.

AddRoundKey: It performs a nibble-wise XOR between the state array and key state arrays. In total 113 cycles and 43 bytes of code size are required for the *AddRoundKey* operation. After every *AddRoundKey* operation, the *Key Schedule* is performed to generate the next round key. In total, *Key Schedule* requires 204 cycles and 114 bytes of code size for encryption (276 cycles and 150 bytes of code size for decryption).

3.3 Memory-Optimized Implementation of AES

In order to reduce the memory requirement, we have applied the following optimization tricks.

Code Structure Optimization Process: We need the extended register for call and jump instructions if the relative address is more than 8 bits. These extended instructions consist of two “regular” instructions, thus doubling the code size. Hence, we rearranged parts of the code in memory to avoid these extended instructions. Furthermore, we combined *MixColumns*, *Key Expansion*, and *AddRoundKey* operations in the same function for AES-128 and AES-256, but not for AES-192 due to the different Key Expansion routine. We also combined *SubBytes* and *ShiftRows* into one function to save code space.

MixColumns/InvMixColumns Optimization Process: We have implemented the technique proposed in [11] but did not achieve significant code savings for encryption. Hence, we stuck to the direct multiplication. For decryption, this technique will reduce the code size by 30%, and it is 3.5 times faster than the direct multiplication.

Key Expansion Optimization Process: In our naïve implementation, every byte of the last column of the key state array is loaded from RAM to process

substitution, rotation, and XOR with the round constants and then stored back to the key state array. The “EX” instruction is used to exchange values between RAM and the registers A and B. When using this “EX” instruction efficiently one can combine a *load from RAM* and *store to RAM* into one instruction, hence halving the number of instructions required for loading and saving the state.

3.4 Speed-Optimized Implementation of AES

In order to speed up the naïve implementation as much as possible, we have applied the following optimization tricks.

Code Structure Optimization Process: To speed up our implementation we used two well-known techniques called loop-unrolling and inline expansion. When unwinding a loop, the instructions are replicated as many times as the loop will be processed, to avoid the overhead due to conditional branching. Inline expansion is a similar technique: it replaces the function call by the actual contents of the to-be-called function, thus saving the function call.

MixColumns/InvMixColumns Optimization Process: By using look-up tables for multiplication by {02}, {03}, {09}, {0B}, {0D} and {0E} we could speed up *MixColumns* by 29% and *invMixColumns* by a factor of more than 3. These gains come at an additional memory cost of 72% and 32%, respectively.

3.5 Implementation Results

All implementations of AES are tailored for the most constrained member of the MCU family, the Epson S1C63003 MCU running at 3 Volt. Table 2 compares our naïve and optimized implementations. For encryption, we could reduce the code size up to 15% for the memory-optimized implementation. For the speed-optimized version, we could boost the speed of the algorithm by almost 22%. For decryption, we could reduce the code size by almost 15% for the memory-optimized version and could boost the speed by almost 72% for the speed-optimized one. In addition, we also calculated the throughput of all versions and the energy consumption for computing one block of AES. The power consumptions were estimated based on information from datasheets or from information directly received from the manufacturer by using the formula $\text{voltage} * \text{current} * (\text{cycles} * 2) / \text{frequency}$.

We have also AES implementations providing both encryption and decryption functionality. The memory-optimized version requires 2625, 2999, and 2895 bytes (about 10% more than the respective decryption-only variant), and the speed-optimized variant requires 4674, 4962, 5698 bytes (about 12% more than the respective decryption-only variant).

4 SHA-1

In this section we will first give a brief introduction into the Secure Hash Algorithm (SHA-1). Subsequently, starting from a naïve implementation, we will

Table 2. Implementation results of AES.

Encryption		AES-128			AES-192			AES-256		
		Naïve	Mem Opt	Spd Opt	Naïve	Mem Opt	Spd Opt	Naïve	Mem Opt	Spd Opt
Code size [bytes]		1,426	1,294	2,645	1,715	1,468	2,942	1,677	1,458	3,280
Cycles		17,675	17,347	13,729	21,188	20,883	16,524	24,979	24,771	19,474
T'put [kbps]	@32.768KHz	0.1159	0.1181	0.1492	0.0967	0.0981	0.1239	0.0820	0.0827	0.1052
	@4MHz	14.144	14.412	18.210	11.799	11.971	15.130	10.008	10.092	12.838
Energy [μ J/block]	@32.768KHz	7.44	7.31	5.78	8.92	8.79	6.96	10.52	10.43	8.20
	@4MHz	5.83	5.72	4.53	6.99	6.89	5.45	8.24	8.17	6.43
Decryption										
Code Size [bytes]		3,062	2,462	4,152	3,416	2,670	4,542	3,382	2,698	4,825
Cycles		64,636	23,901	18,688	78,315	28,552	22,120	92,378	33,554	26,242
T'put [kbps]	@32.768KHz	0.0317	0.0857	0.1096	0.0262	0.0717	0.0926	0.0222	0.0610	0.0780
	@4MHz	3.868	10.460	13.378	3.192	8.756	11.302	2.706	7.451	9.527
Energy [μ J/block]	@32.768KHz	27.22	10.07	7.87	32.98	12.02	9.32	38.90	14.13	11.05
	@4MHz	21.33	7.89	6.17	25.84	9.42	7.30	30.48	11.07	8.66

describe several tricks that we applied to optimize the implementation regarding to code size and speed.

4.1 Introduction to SHA-1

SHA-1 is one of the Secure Hash Standards published by the National Institute of Standards and Technology (NIST) [36] in 1995. **SHA-1** is an iterative, one-way hash function that can process messages up to a length $< 2^{64}$ bits and produces a 160-bit message digest. According to [36], there are two stages of **SHA-1**, i.e., *preprocessing* and *hash computation*.

Preprocessing stage: It consists of the following 3 steps.

1. *Padding:* The message is padded by a single '1' bit followed by the necessary number ($0 \sim 511$) of '0' bits, and then the bit length of the original message, represented by a 64-bit integer, is appended. After padding, the bit length will be a multiple of 512.
2. *Parsing the padded message:* This step divides the message into N blocks, each block containing 512 bits.
3. *Initialize Hash Value:* In this step the 160-bit starting value is initialized by five 32-bit words as follows: $A = 0x67452301$, $B = 0xEFCDAB89$, $C = 0x98BADCFE$, $D = 0x10325476$ and $E = 0xC3D2E1F0$.

Hash computation: This stage is the heart of **SHA-1** and consists of 80 rounds. Each round (see Fig. 2) uses a function $f_t(B, C, D)$ and processes constants (K_t) as described in Table 3.

After processing all N blocks, the resulting 160-bit message digest is the concatenation of $A\|B\|C\|D\|E$. For further details, please refer to [36].

Fig. 2. One round of SHA-1 computation.

Table 3. SHA-1 function ($f_t(B, C, D)$) and constants (K_t).

Round (t)	$f_t(B, C, D)$	K_t
0 to 19	$(B \wedge C) \oplus (\neg B \wedge D)$	0x5A827999
20 to 39	$B \oplus C \oplus D$	0x6ED9EBA1
40 to 59	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8F1BBCDC
60 to 79	$B \oplus C \oplus D$	0xCA62C1D6

4.2 Implementation of SHA-1

There are two different methods to compute message scheduling as mentioned in [36]. We choose the method for devices with limited memory.

There are five operations required for **SHA-1**: *AND*, *NOT*, *XOR*, *Left Rotation* and *Modulo Addition*. However, as the MCU does not support *NOT*, we use *XOR* with 0xF instead. The 5-bit left rotation is implemented by first swapping 4-bit words within the 32-bit chunk and then left rotating the result by 1-bit. For the 30-bit left rotation, we apply a similar technique, i.e., first swapping 4-bit words within the 32-bit chunk to get a 28-bit left rotation and then doing a 1-bit left rotation twice. The 5-bit left rotation requires 223 bytes of code and 180 clock cycles. The 30-bit left rotation reuses some sub-functions in the 5-bit left rotation, so it requires additional 21 bytes and in total takes 301 cycles. In addition, the 32-bit modulo addition requires 82 bytes of code and 93 cycles.

Memory-Optimized Implementation: As for **AES** we tried to avoid extended jumps and calls by rearranging parts of the code. Furthermore, in the specification of **SHA-1** it can be seen that the round function of rounds 60 to 79 uses the same round function as rounds 20 to 39. Thus, we can use the same function twice and hence, save 13 bytes of program memory at no additional timing cost.

Speed-Optimized Implementation: We have applied 2-bit right rotation instead of 30-bit left rotation. As a consequence, we could speed up this part by 17%, while the code size increased a mere 7% as compared to the memory-optimized version. We also applied loop-unrolling where possible and inline expansion of the function *left rotation*, *right rotation* and *modulo addition*. Additionally, we also applied the trick for minimizing the round function as mentioned in [32] to boost up the speed.

4.3 Implementation Results

Our SHA-1 can be implemented on the smallest Epson S1C63003 MCU running at 3 volts. Detailed results are provided in Table 8 in Sec. 6. As can be seen the speed-optimized version requires 14% more memory, but it can boost the speed by almost 24% in comparison to the memory-optimized implementation.

5 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) [27, 33] is a public-key cryptography with much smaller key sizes than RSA or other public-key systems based on the discrete logarithm problem. Because of its small key sizes, it is very suitable for constrained devices. ECC can be used to implement a wide variety of applications, such as Diffie-Hellman key exchange (e.g. ECDH [41]) or digital signatures (e.g. ECDSA [21]) amongst others.

ECC relies upon group operations in elliptic curve group, and a group $\mathbb{E}$ over field $\mathbb{F}_p$ can be defined by the points (x, y) satisfying the short Weierstrass form:

$$\mathbb{E} : y^2 = x^3 + ax + b, \text{ where } 4a^3 + 27b^2 \neq 0.$$

In “Standards for Efficient Cryptography 2” [7] a curve is specified by a sextuple: (p, a, b, G, n, h) , where p is the prime, a and b are the curve parameters, G is a base point with order n , and h is the cofactor. In this paper, we have implemented ECDH using the 160-bit curve `secp160r1` of SEC 2 on the S1C63016 MCU.

5.1 Efficient Implementation of ECC

The computation in ECC can be divided into three layers: prime field arithmetic at the bottom, point arithmetic in the middle, and protocol on top. The three layers are implemented separately and the implementations at the lowest level greatly impacts the overall run time, hence optimizations at this level are crucial.

Prime Field Arithmetic The curves suggested by SEC 2 employ *pseudo Mersenne primes*, e.g., $p = 2^{160} - 2^{31} - 1$ in `secp160r1`. Modulo operations will be much more efficient than using the Barrett reduction [3] or the Montgomery multiplication [34]. We have implemented the following operations in this layer.

Modular addition/subtraction: The naïve addition/subtraction with reduction takes at most 428 cycles. It might be susceptible to timing attacks because of the unfixed run time. By using some extra data RAM a fixed run time of 292 cycles is achieved. Besides, the code size can be optionally reduced from 352 to 291 bytes along with the data RAM usage increasing from 40 to 60 bytes.

Multiplication (M) and Squaring (S): According to our implementation results (Table 4), row-wise (operand scanning form) multiplication is more suitable for the 4-bit architecture than column-wise (product scanning form) and hybrid [50] algorithms. When using row-wise multiplication, the first operand remains unchanged for the whole row, and only the second operand needs to be updated.

This is why, compared to the column-wise and hybrid method, the row-wise method requires less IO operations and hence resulting in a better performance.

The main bottlenecks on this 4-bit MCU are carry handling and the slow multiplication instruction³. The first one can be solved by using additional 160-bit of memory to accumulate the results of atomic multiplications in one round of the row-wise multiplication instead of adding them to the final result separately. The second one can be solved by interleaving multiplication with other instructions. These two improvements together deliver about twice the performance, i.e., reducing the run time from 28,680 to 16,690 cycles with a code size of 324 bytes and an additional 10-byte data RAM. A separate squaring algorithm [6] further improves the performance, each squaring takes 12,985 cycles at an additional cost of 330 bytes of code size.

Table 4. Implementation results of field arithmetic.

Naïve implementation	No. of cycles	Code size [bytes]
Row-wise [19]	28,680	300
Column-wise[19]	30,953	344
Hybrid [50]	29,485	368

Bisection: For an even number, division by 2 is a right shift of its binary representation. For an odd number, an extra addition of the prime p is required before the right shift.

Inversion (I): We have implemented the binary extended GCD algorithm [26, Ch 4.5.2] which requires multi-precision addition/subtraction and right shift (division by 2).

Reduction: Because of the pseudo Mersenne prime $p = 2^{160} - 2^{31} - 1$ in the curve `secp160r1`, the 320-to-160-bit reduction can be evaluated by shift and add. A naïve implementation of reducing from MSB to LSB takes 1,604 cycles with a code size of 775 bytes, and most of them are spent on the carry handling. In order to avoid this bottleneck, we rearrange the reduction as

$$\begin{aligned} H \times 2^{160} + L &\equiv H \times (2^{31} + 1) + L \\ &\equiv H + L + (H \lll 31) + H_{159 \sim 129} \times 2^{31} \pmod{p}, \end{aligned} \quad (1)$$

where H and L are the most and least significant 160 bits of the 320-bit value respectively, “ $\lll 31$ ” is a 31-bit left rotation, and $H_{159 \sim 129}$ is the most significant 31 bits of H . By accumulating the carries of the three additions and then handling them together, the performance will be significantly improved, and it requires only 679 cycles and 624 bytes.

Point Arithmetic The major computation in ECC is the point multiplication nP which can be evaluated through the combination of point doubling $2P$ and

³ Refer to the specifications in the Epson technical manual [43].

point addition $P_1 + P_2$. Instead of representing points in affine coordinates ($\mathcal{A}$), we employ Jacobian projective coordinates ($\mathcal{J}$) to implement point doubling and addition. A point (x, y) in $\mathcal{A}$ can be represented by $(x, y, 1)$ in $\mathcal{J}$, and a point (X, Y, Z) in $\mathcal{J}$ is identical to the point $(X/Z^2, Y/Z^3)$ in $\mathcal{A}$. The advantage of using projective coordinates is that we only need 1I after finishing the point multiplication instead of 1I for every point doubling and point addition in affine coordinates. The following are the three point operations, and detailed results are summarized in Table 5 and Table 6.

Point doubling (D): We implement point doubling in Jacobian coordinates ($2\mathcal{J} \rightarrow \mathcal{J}$). It requires 4M and 4S as well as some minor field operations, or alternatively 3M and 5S by using the trick $\alpha\beta = \frac{1}{2}((\alpha + \beta)^2 - \alpha^2 - \beta^2)$ [30]. Since in our implementation $1S \approx 0.7M$, this trick achieves 2% speed-up along with 3% increment in code size.

Point addition/subtraction (A): Point addition in mixed coordinates ($\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$) takes 8M and 3S, or 7M and 4S by using the trick described above. Point subtraction is similar to point addition but has an extra subtraction to calculate the y -coordinate.

Point Multiplication (PM): The left-to-right binary PM requires 159D and on average 80A. When recoding the scalar to non-adjacent form (NAF) [37], PM requires 159D and on average 53A, that is, about 14% speed-up compared to the one with a binary scalar. However, the NAF algorithm recodes from right to left and needs more data RAM to separately store the NAF-recoded signed-digit scalar. To overcome this, we have employed the left-to-right analogue of NAF [23] to achieve PM along with on-the-fly scalar recoding.

Table 5. Implementation results of point arithmetic.

Operation	Description	No. of cycles	Code size [bytes]
Point doubling	4M + 4S	128,453	1,038
$2\mathcal{J} \rightarrow \mathcal{J}$	3M + 5S	125,883	1,064
Point addition	8M + 3S	183,698	1,945
$\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$	7M + 4S	181,343	1,994

Protocol Layer Optimizations at this level depend on the protocol being implemented. We focus on the Elliptic Curve Diffie-Hellman key exchange (ECDH) [19]. Each key exchange requires 2PM and 2I.

5.2 Side Channel Resistant Implementations of ECC

The immunity against side-channel attacks is also investigated. All prime field arithmetic in our implementation takes a fixed amount of time and is resistant to timing attacks. We also investigate the performance and side-channel immunity of various PM algorithms in the curve arithmetic.

Table 6. Implementation results of point multiplication and SCA.

Point multiplication algorithm	Cycles [millions]	Code size [bytes]	Side-channel immunity
Left-to-right PM	35.20	7,201	-
Left-to-right PM with left-to-right NAF	30.39	10,387	-
Double-and-add-always	49.48	8,042	SPA
BRIP	50.00	8,080	SPA,RPA,ZPA,DPA
Randomization of scalar (20-bit)	33.21	10,404	DPA
Randomization of scalar (64-bit)	43.04	10,531	DPA
Randomized projective coordinates	31.50	10,282	DPA
Randomization of scalar (20-bit) & Randomized projective coordinates	33.52	10,501	DPA

Double-and-add-always algorithm [10]: It achieves a regular computational sequences by introducing dummy operations. A regular computational sequence can prevent SPA but it is still vulnerable to DPA and other sophisticated attacks, e.g., Refined Power Analysis (RPA) [17] and Zero Power Analysis (ZPA) [1].

Binary expansion with random initial point (BRIP) [31]: In BRIP, the intermediate result of PM is blinded by a random point R . It is also a double-and-add-always algorithm but without dummy operations and thus making it resistant to even DPA, RPA and ZPA.

Randomization of scalar [10]: The scalar d is randomized by $d' = d + n \times \#E$, where $\#E$ is the curve order and n is a random number. There is an inherent security-efficiency trade-off: Using, say, a 20-bit random number already provides a decent level of DPA resistance as it multiplies the effort of an attacker by 6 orders of magnitude while having a mere 10% overhead in execution time.⁴ For a highly secure implementation using a 64-bit random number the timing increases by around 42%.

Randomized projective coordinates [10]: A point $P = (x, y)$ can be represented in Jacobian projective coordinates as (r^2x, r^3y, r) for any $r \in \mathbf{Z}_p^*$. When computing dP by the left-to-right scalar algorithm, the temporary result T is initialized by $T = P$, and in each iteration, updated by $T = 2T$ and optionally $T = T + P$. In our implementation, instead of randomizing P , we only randomize the initial value of T , i.e., P is still of affine coordinates, to avoid the much more expensive point addition in Jacobian coordinates ($\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$, 12M and 4S).

Table 6 summarizes the implementation results of various point scalar multiplication algorithms. All of them are implemented by using Jacobian projective coordinates. In randomization of scalar and randomized projective coordinates, the scalar will be also recoded by the left-to-right NAF method. As one can see, BRIP is the most secure one but also the slowest. For constrained devices, we suggest using 20-bit randomization of scalar along with randomized projective

⁴ Assuming a successful DPA would require 100 measurements, it would take around 54 years to *measure* the power consumption compared to 26 minutes without randomization.

coordinates, as it provides basic resistance against DPA and also increases the difficulty for SPA [8] due to the NAF-recoded scalar.

6 Discussion

In this work we have already shown the feasibility of implementing symmetric and asymmetric cryptography on 4-bit MCUs. What is left to be shown is the practicability and meaning of cryptography on 4-bit MCUs in real-world scenarios as well as the performance in comparison to common 8-bit implementations regarding three important criteria for embedded systems: *energy consumption*, *speed* and *code size*. As already pointed out in Section 2, we use the software simulator and the debugger to obtain the cycle count and the code size. Power figures from the data sheet are used to estimate the energy consumption of our implementations.

6.1 AES

We compare our 4-bit AES implementation with the only other 4-bit AES implementation known so far [24] and with two recent 8-bit AES implementations [5, 38]. The MCUs used in [5, 38] belong to the megaAVR family, which are popular 8-bit processors used in research as well as in embedded devices. The most notable difference to our 4-bit MCU is that the megaAVRs have 32 8-bit registers which can all be used as destination registers of arithmetic operations. In addition, the megaAVRs support the pre-decrement and post-increment functionality, whereas our 4-bit MCU only supports the post-increment functionality.

Table 7 shows the results regarding speed, code size and energy consumption for different optimization strategies. It should be noted that the comparison is not completely fair since the ATmega128 and the AT90USB162 are not the lowest-power devices but still are stated as low-power MCUs by Atmel. Note further that it was not possible to compare the energy consumptions at the same frequency because the manufacturers only provide numbers for one (the most typical) frequency of their MCUs. To estimate the energy consumption of our 4-bit MCU we chose 4 MHz because first, at this frequency our implementations can be directly compared to most of the 8-bit implementations in the following sections, and second, this configuration is also the most energy efficient.

Not surprising, the 8-bit implementations of AES beat the 4-bit implementations by an order of magnitude in terms of speed since the AES operations are predestined to be very efficient on 8-bit architectures. But, as one can see, our implementation beats the implementation on the MARC4 in terms of speed. Hence, we can claim that this implementation is the fastest 4-bit implementation of AES up to now. More importantly, our implementation is the most efficient one in terms of energy consumption as it consumes nearly half the energy per block in comparison to the best 8-bit implementation and it outperforms the other implementations by a factor of approximately three. This can be seen as

Table 7. Comparison of different **AES** implementations.

	MCU	Opt.	Enc. [cycles]	Dec. [cycles]	Code size [bytes]	Energy** [μ J/block] @ 3V
4-bit	Epson S1C63	Speed	13,929	18,688	4,674	4.6 @ 4MHz
		Size	16603	23,901	2,625	5.5 @ 4MHz
	MARC4 [24]	Speed	15,977	20,801	3,679	14.3 @ 1MHz
		Size	24,012	34,432	1,775	21.6 @ 1MHz
8-bit	ATmega128 [38]	-	3,766	4,558	3,410	14.1 @ 4MHz
	AT90USB162 [5]	Speed	2,740	3,648	1,912	7.2 @ 8MHz
		Size	2,942	3,690	1,912*	7.7 @ 8MHz

* uses less RAM. ** energy consumption for encryption.

a strong argument for implementing cryptography on a 4-bit MCU since energy consumption is arguably one of the most important criteria for low-power devices.

6.2 SHA-1

We only found two references for **SHA-1** implementations on an 8-bit MCU in [39], which are compared to our 4-bit implementation in Table 8.

Table 8. Comparison of different **SHA-1** implementations.

	MCU	Opt.	Cycles per block	Code size [bytes]	Energy [μ J/block] @ 3V
4-bit	Epson S1C63	Speed	87,788	2,324	29.0 @ 4 MHz
		Size	108,666	2,038	35.9 @ 4 MHz
8-bit	ATmega128 [16]	Generic	63,000	4,000	236 @ 4 MHz
	megaAVR family [12]		37,030	1,022	139 @ 4 MHz

*Since [12] is a generic implementation, we used the properties of the ATmega128 to estimate the energy consumption.

As can be seen our implementation takes roughly double the clock cycles and double the size compared to the best 8-bit implementation. This is due to the fact that **SHA-1** mostly uses primitive instructions (1 cycle) on 4- or 8-bit of data. Hence, we need roughly twice the amount of instructions resulting in twice the amount of cycles and code size. But again, our implementation on the 4-bit MCU outperforms the other implementation in terms of energy consumption by a factor of almost five.

6.3 ECC

In this section we compare our 4-bit **ECC** implementation regarding speed and code size with three other 8-bit implementations using the same curve, i.e. secp160r1, found in [18] and [9]. [18] describes two implementations on two

different MCUs, namely the Chipcon CC1010 and the Atmel ATmega128. [9] describes a ECC implementation on an Atmel ATmega8 [2] which is very similar to the ATmega128 as both belong to the megaAVR family.

The Chipcon CC1010 is an 8-bit MCU which implements the 8051 instruction set [47]. The most notable properties are that every instruction cycle takes 4 clock cycles, hence quartering the effective frequency. Second, the 8-bit accumulator is the destination register for all arithmetic functions resulting in the same complexity for storing or further processing intermediate results as on our 4-bit MCU, thus making the CC1010 a perfect candidate for the comparison.

As can be seen in Table 9 a field multiplication on the 4-bit MCU takes around 8 times as many instruction cycles as the fastest implementation on an 8-bit controller and twice as many cycles as on the CC1010. This is due to the fact that a multiplication instruction on the Epson S1C63 requires 10 cycles, compared to only 2 cycles and 5 cycles on the ATmega and on the CC1010, respectively. This fact also leads to a higher time share of the field multiplication.

As a consequence, our fastest point multiplication requires roughly 5 times more instruction cycles than the fastest ATmega implementation and twice as many instruction cycles as the CC1010. But, as for AES and SHA-1, the 4-bit implementation beats the 8-bit implementations by a factor of more than two regarding the energy consumption.

Table 9. Comparison of ECC implementations.

	MCU	Point Mul. [cycles in M]	Field Mul. [cycles]	Time share in PM	Code size [bytes]	Energy [mJ/block] @ 3V
4-bit	Epson S1C63	30.39	16,690	88%	10,616	10.02 @ 4 MHz
8-bit	ATmega128 [18]	6.48	1,986	77%	3,682	24.3 @ 4 MHz
	ATmega8 [9]	10.4	n/a	n/a	7,080*	28.08 @ 4 MHz
	CC1010 [18]	16.88	9,321	85%	2,166	253.2 @ 4 MHz

* includes ECDSA.

7 Conclusion

In this work we have studied the feasibility and practicability of standardized cryptography on a 4-bit MCU. We have provided the fastest implementation of AES, and the first implementations of SHA-1 and ECC, this way proving their feasibility on an ultra-constrained 4-bit MCUs. More interesting, however, is that all three primitives require between 39% to 79% less energy than previously published implementations on low-power 8-bit MCUs. Our crypto modules can be combined to enable a wide range of different applications. For example, non-timing critical applications, such as legally binding sensor readings or secure firmware updates can be achieved using ECDSA [21] by combining our SHA-1 and ECC modules.

Though it is not the main focus of this paper, we have investigated the SCA resistance trade-offs for ECC by implementing a variety of countermeasures. Future work includes implementation of SCA countermeasures for AES and evaluation of our implemented SCA countermeasures for ECC and AES by conducting physical experiments, which was beyond the scope of this work. Another interesting topic might be the analysis of the resistance against fault attacks of 4-bit MCUs.

Acknowledgement

The authors were supported in part by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03. The authors would also like to thank Andreas Enge for many fruitful discussions.

References

1. T. Akishita and T. Takagi. Zero-value point attacks on elliptic curve cryptosystem. In C. Boyd and W. Mao, editors, *ISC*, volume 2851 of *LNCS*, pages 218–233. Springer, 2003.
2. Atmel Corporation. ATmega8/ATmega8L datasheet. <http://www.atmel.com/Images/doc2486.pdf>, February 2011.
3. P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *CRYPTO*, volume 263 of *LNCS*, pages 311–323. Springer, 1986.
4. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Viskelsøe. PRESENT: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007. <http://lightweightcrypto.org/present/>.
5. J. W. Bos, D. A. Osvik, and D. Stefan. Fast implementations of AES on various platforms. Cryptology ePrint Archive, Report 2009/501, 2009.
6. M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. In D. Naccache, editor, *CT-RSA*, volume 2020 of *LNCS*, pages 250–265. Springer, 2001.
7. Certicom Research. Standards for efficient cryptography, SEC 2: Recommended elliptic curve domain parameters, 2000.
8. C.-N. Chen, S.-M. Yen, and S.-J. Moon. On the computational sequence of scalar multiplication with left-to-right recoded NAF and sliding window technique. *IEICE Transactions*, 93-A(10):1806–1812, 2010.
9. A. Chmielowiec. Elliptic curve cryptography in small devices. http://students.mimuw.edu.pl/~ac181080/data/ecc_in_small_devices.pdf.
10. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin Kaya Koç and C. Paar, editors, *CHES*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.
11. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
12. Das Labor. Crypto-avr-lib. <http://www.das-labor.org/wiki/Crypto-avr-lib>, January 2008.

13. D. Engels, X. Fan, G. Gong, H. Hu, and E. Smith. Ultra-lightweight cryptography for low-cost RFID tags: Hummingbird algorithm and protocol. Technical report, Centre for Applied Cryptographic Research (CACR), 2009.
14. D. Engels, M. Saarinen, and E. Smith. The Hummingbird-2 lightweight authenticated encryption algorithm. In *proceedings of The 7th Workshop on RFID Security and Privacy-RFIDSec*, 2011.
15. X. Fan, H. Hu, G. Gong, E. Smith, and D. Engels. Lightweight implementation of Hummingbird cryptographic algorithm on 4-bit microcontrollers. In *International Conference for Internet Technology and Secured Transactions, 2009*, pages 1–5, 2009.
16. P. Ganesan, R. Venugopalan, P. Peddabachagari, A. G. Dean, F. Mueller, and M. L. Sichitiu. Analyzing and modeling encryption overhead for sensor network nodes. In C. S. Raghavendra, K. M. Sivalingam, R. Govindan, and P. Ramanathan, editors, *Wireless Sensor Networks and Applications*, pages 151–159. ACM, 2003.
17. L. Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In Y. Desmedt, editor, *Public Key Cryptography*, volume 2567 of *LNCS*, pages 199–210. Springer, 2003.
18. N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In Joye and Quisquater [22], pages 119–132.
19. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
20. ISO/IEC. 29192-2: Information technology – security techniques – lightweight cryptography – part 2: Block ciphers. available via http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=56552.
21. D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.
22. M. Joye and J.-J. Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *LNCS*. Springer, 2004.
23. M. Joye and S.-M. Yen. Optimal left-to-right binary signed-digit recoding. *IEEE Trans. Comput.*, 49:740–748, July 2000.
24. T. Kaufmann and A. Poschmann. Enabling standardized cryptography on ultra-constrained 4-bit microcontrollers. In *International IEEE Conference on RFID*, Orlando, USA, 2012, to appear.
25. L. R. Knudsen, G. Leander, A. Poschmann, and M. J. B. Robshaw. PRINTcipher: A block cipher for IC-printing. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *LNCS*, pages 16–32. Springer, 2010.
26. D. E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, third edition, 1997.
27. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.
28. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
29. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
30. P. Longa and A. Miri. Fast and flexible elliptic curve point arithmetic over prime fields. *IEEE Trans. Comput.*, 57:289–302, March 2008.

31. H. Mamiya, A. Miyaji, and H. Morimoto. Efficient countermeasures against RPA, DPA, and SPA. In Joye and Quisquater [22], pages 343–356.
32. K. McCurley. A fast portable implementation of the secure hash algorithm, III. <http://www.mccurley.org>, July 1994.
33. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *CRYPTO*, volume 218 of *LNCS*, pages 417–426. Springer, 1985.
34. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
35. National Institute of Standards and Technology. FIPS 197: Announcing the advanced encryption standard (AES). <http://csrc.nist.gov>, November 2001.
36. National Institute of Standards and Technology. FIPS 180-3: Secure hash standard. <http://csrc.nist.gov>, October 2008.
37. G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.
38. S. Rinne, T. Eisenbarth, and C. Paar. Performance analysis of contemporary light-weight block ciphers on 8-bit microcontrollers. In *ecrypt workshop SPEED - Software Performance Enhancement for Encryption and Decryption*, 2007.
39. S. Rohde, T. Eisenbarth, E. Dahmen, J. Buchmann, and C. Paar. Fast hash-based signatures on constrained devices. In G. Grimaud and F.-X. Standaert, editors, *CARDIS*, volume 5189 of *LNCS*, pages 104–117. Springer, 2008.
40. M.-J. O. Saarinen. Cryptanalysis of Hummingbird-1. In A. Joux, editor, *FSE*, *LNCS*, pages 328–341. Springer, 2011.
41. R. Schroepel, H. K. Orman, S. W. O’Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *CRYPTO*, volume 963 of *LNCS*, pages 43–56. Springer, 1995.
42. Seiko Epson Corporation. CMOS 4-bit single chip microcomputer S1C63000 core CPU manual. http://www.epson.jp/device/semicon_e/product/index_mcu.htm, 2011.
43. Seiko Epson Corporation. CMOS 4-bit single chip microcontroller S1C63003/004/008/016 technical manual. http://www.epson.jp/device/semicon_e/product/index_mcu.htm, 2011.
44. Seiko Epson Corporation. Microcontrollers 2011. http://www.epsondevice.com/webapp/docs_ic/DownloadServlet?id=ID000463, 2011.
45. Seiko Epson Corporation. Program development process. http://www.epson.jp/device/semicon_e/product/mcu/development/tool.htm, 2011.
46. TEMIC Semiconductors. Automotive safety and convenience data book. <http://pe2bz.philpem.me.uk/pdf%20on%20typenumber/S/SAFE96.pdf>, 1996.
47. Texas Instruments. CC1010 datasheet. <http://www.ti.com/lit/ds/symlink/cc1010.pdf>, September 2009.
48. J. Turley. Microprocessors for consumer electronics, PDAs, and communications. *Embedded Systems Programming*, 10:116–128, 1997.
49. J. Turley. *The Essential guide to semiconductors*. Prentice Hall PTR, 2003.
50. L. Uhsadel, A. Poschmann, and C. Paar. Enabling full-size public-key algorithms on 8-bit sensor nodes. In F. Stajano, C. Meadows, S. Capkun, and T. Moore, editors, *ESAS*, volume 4572 of *LNCS*, pages 73–86. Springer, 2007.
51. M. Vogt, A. Poschmann, and C. Paar. Cryptography is feasible on 4-bit microcontrollers - a proof of concept. In *International IEEE Conference on RFID*, pages 267–274, Orlando, USA, 2009.