Shifting Primes: Extension of Pseudo-Mersenne Primes to Optimize ECC for MSP430-Based Future Internet of Things Devices

Leandro Marin, Antonio J. Jara, and Antonio F.G. Skarmeta

Computer Science Faculty,
University of Murcia, Murcia, Spain
{leandro,jara,skarmeta}@um.es

Abstract. Security support for small and smart devices is one of the most important issues in the Future Internet of things, since technologies such as 6LoWPAN are opening the access to the real world through Internet. 6LoWPAN devices are highly constrained in terms of computational capabilities, memory, communication bandwidth, and battery power. Therefore, in order to support security, it is necessary to implement new optimized and scalable cryptographic mechanisms, which provide security, authentication, privacy and integrity to the communications. Our research is focused on the mathematical optimization of cryptographic primitives for Public Key Cryptography (PKC) based on Elliptic Curve Cryptography (ECC) for 6LoWPAN. Specifically, the contribution presented is a set of mathematical optimizations and its implementation for ECC in the 6LoWPAN devices based on the microprocessor Texas Instrument MSP430. The optimizations presented are focused on Montgomery multiplication operation, which has been implemented with bit shifting, and the definition of special pseudo-Mersenne primes, which we have denominated "shifting primes". These optimizations allow to implement the scalar multiplication (operation used for ECC operations) reaching a time of $1,2665$ seconds, which is $42,8\%$ lower of the reached by the state of the art solution TinyECC ($2,217$ seconds).

Keywords: Security, 6LoWPAN, ECC, pseudo-Mersenne primes, shifting prime, Internet of Things.

1 Introduction

Security is one of the major issues for the current digital society. The evolution of hardware technologies with the development of new devices such as wireless personal devices, embedded systems and smart objects, and the evolution of the services with the definition of cloud computing, online services, and ubiquitous access to the information are defining an extension of the capabilities of the current Internet, what make feasible to connect to Internet the objects and devices which are found surround us, it is the so-called Internet of things (IoT). IoT allows that systems can get a total control and access to another systems

A M. Tjoa et al. (Eds.): ARES 2011, LNCS 6908, pp. 205–219, 2011.

for leading to provide ubiquitous communication and computing. Thereby a new generation of smart and small devices, services and applications can be defined.

These small and smart things with connectivity and communication capacity from the IoT are what can be found, since some years ago, in the Low-power Wireless Personal Area Networks (LoWPANs). IETF 6LoWPAN working group has defined, in the RFC4944 [1], the standard to support IPv6 over that LoWPANs (6LoWPAN), in order to provide the technological basis for extending the Internet to small devices. 6LoWPAN offers to the LoWPANs advantages from the Internet Protocol (IP) such as scalability, flexibility, ubiquity, openness, and end-to-end connectivity. It could be considered that 6LoWPAN devices are also empowered with derived IP protocols, i.e., protocols for mobility such as MIPv6, management such as SNMP, and security such as IPSec. However it is not feasible, since 6LoWPAN nodes are highly constrained in terms of computational capabilities, memory, communication bandwidth, and battery power.

Therefore, with the mentioned constrains, it is a challenge to implement and use the cryptographic algorithms and protocols required for the creation of security services. Nowadays, 6LoWPAN security is based on Symmetric Key Cryptography (SKC) which is directly supported by specific hardware in the microprocessor. SKC is suitable to offer local solutions, such as were originally designed these personal area networks and local solutions such as location [2], but for the future internet, it is required a higher scalability. For that reason, Public Key Cryptography (PKC) needs to be supported, Specifically, our research is focused on Elliptic Curve Cryptography for 6LoWPAN devices based on MSP430.

The MSP430 has been chosen since it is one of the most extended microprocessors in the Internet of Things devices and embedded systems. It is used for 6LoWPAN devices such as the Tmote Sky, for active Radio Frequency Identification (RFID) [3], and new hybrid technologies such as DASH7 [4].

The mathematical optimization of cryptographic primitives has been widely mentioned from a general point of view, and also for constrained devices. An overview of the state of the art is carried out in Section 2. From all the related works, TinyECC [9] is one of the most relevant references about ECC implementations for devices such as the based on MSP430. TinyECC chose Barret algorithm for reduction modulo p, but it has been demonstrated, in our previous work [16], that Montgomery multiplication is more suitable for these devices.

In this paper is presented an evolution of the mentioned previous work with the inclusion of special primes, which reduce almost to the half the cycles needed for Montgomery multiplication. The selection of special primes is a very extended technique, for example in FIPS 186-3 are recommended special primes for elliptic curve, called generalized Mersenne numbers, for which modular arithmetic is optimized for processors that use 32 or 64 bits operations, see [18, Section D.2]. Our advance has been to determinate the ideal primes for Montgomery multiplication based on the optimizations defined for 16-bits microprocessors.

In conclusion, this paper proposes an implementation of multiplication operation for ECC based on bit shifting, presented in Section 3, instead of the microprocessor's multiplication operation for microprocessor, which has not hardware

support for multiplication operation such as MSP430. In addition, this has defined specific pseudo-Mersenne primes, which offer a simplification of the Montgomery multiplication implementation based on bit shifting, which have been denominated shifting primes, presented in Section 4. Finally, all the optimizations are implemented in Section 5, and evaluated in Section 6.

2 Related Works

The usual solutions for WSNs are based on Symmetric Key Cryptography (SKC), but it is not suitable for the Future Internet of Things, since it is not scalable. SKC requires that both the origin and destination share the same security credential (i.e. secret key), which is utilized for both encryption and decryption. As a result, any third-party that does not have such secret key cannot access the information exchange. The majority of WSNs, included 6LoWPAN, are based on the IEEE 802.15.4 standard, which offers three levels of security: Hash Functions, Symmetric Key Cryptography and both [8].

This work is focused on Public Key Cryptography (PKC), also known as asymmetric cryptography, which is useful for secure broadcasting and authentication purposes, and this satisfies the scalability requirements from the Future Internet of Things. It requires of two keys: a key called secret key, which has to be kept private, and another key named public key, which is publicly known. Any operation done with the private key can only be reversed with the public key, and vice versa. These primitives provide the confidentiality, integrity, authentication, and non-repudiation properties.

Public Key Cryptography was considered unsuitable for sensor node platforms, but that assumption was a long time ago. The approach that made PKC possible and usable in sensor nodes was Elliptic Curve Cryptography (ECC), which is based on the algebraic structure of elliptic curves over finite fields. Some studies has been carried out about RSA in reduced chips [12], but it was non-viable. Therefore, PKC for small and smart devices is mainly focused on ECC, since ECC presents lower requirements both in computation and memory storage, due to its small key sizes and its simpler primitives [13].

The related works of the implementation of an efficient cryptographic algorithm for constrained devices have been focus on the optimizations of the multiplication operation, since it is the most expensive operation in RSA and ECC algorithms. ECC implementation can be over either $GF(2^m)$ or $GF(p)$, we have focused on modular arithmetic, i.e. $GF(p)$, since this has similar nature to the arithmetic of the micro-controller used in the 6LoWPAN devices. However, some interesting approaches have been defined for ECC implementations over $GF(2^m)$, for example [14] has showed that field multiplication is faster over $GF(2^m)$ than in $GF(p)$ for new suggested hardware implementations, where the instructions set and arithmetic of the micro-controller are closer to $GF(2^m)$.

Some of the most known software implementations over modular arithmetic for ECC are TinyECC [9,10] and NanoECC [11], which implement ECC-based operations. TinyECC adopted several optimization techniques such as optimized

modular reduction using pseudo-Mersenne primes, sliding window method, Jacobian coordinate systems, in-line assembly and hybrid multiplication in order to achieve computational efficiency. Realise, that the computational and memory requirements of these algorithms are not small (e.g. signature requires 19308 bytes ROM and 1510 bytes RAM for the MICAz, generating a signature in 2 seconds. and verifying it in 2.43 seconds), although the implementation of these primitives is constantly evolving and improving.

3 Mathematical Optimization Based on Bit Shifting Instead of Microprocessor's Multiplication Operation

There is an important literature about the advantages of the Montgomery's representation for this calculation [16,5,6]. For that reason, it is used for both solutions i.e. based on bit shifting and multiplication operation. The original details of the Montgomery's representation for can be found in [7], although many other books or papers refer to it. Our solution is based on ECC, which requires an integer size (k) of 160-bits, i.e. $k = 160, R = 2^{160}$).

Montgomery representation has been chosen instead of other solutions, such as the Barrett reduction used in TinyECC [9], since in Montgomery representation, the representation of the numbers a and b, which are going to be multiplied, are aR and bR mod n (n is a prime for ECC). Addition and subtraction operations with these numbers do not cause problems since R is common factor. The problem is coming with the multiplication operation, when aR and bR is multiplied, the result is abR^2, but what is required is abR. Therefore, it needs to be reduced by factor R. The great advantage of Montgomery representation is to carry out the reduction of the factor R during multiplication,

The multiplication operation is what consumes the higher part of the time, since it is repeated thousands of times. For that reason, it is the part optimized and discussed more in detail in the next subsections.

3.1 Bit Shifting

Let a and b two integers in Montgomery representation. Then, aR and bR mod n between 0 and $n-1$. They are stored in binary representation, i.e. $aR = \sum_i a_i 2^i$ and $bR = \sum_i b_i 2^i$.

It is calculated $(aR)(bR)R^{-1} = (ab)R$, therefore it is required to carry out k right bit shifting (with $k = 160$).

Since that, modulus n is odd, because it is a prime in ECC. Thus, when it is divided by 2 mod n, two options are defined: either it is even number and it can be directly shifted, or it is odd number and consequently needs to add n, in order to reach 0 in the least significant bit, in order to be able to shift it.

Multiplication process requires a variable to accumulate the current result, we will call to that variable P, whose digits are $P = \sum_i P_i 2^i$. Each one of the digits B_i is multiplied by $\sum_i A_i 2^i$ and divided by 2. As initially P is 0, if $B_i = 0$

for some initial values, it can be ignored. Therefore, this starts directly by the digit in the position i_0, such that $B_{i_0} = 1$, and copy the value of A_i in P_i.

From the position i_0, we can find in the next steps: $B_i = 0$ or 1. On the one hand, when $B_i = 0$, it divides P by 2, and add n when P is odd. On the other hand, when $B_i = 1$, then it adds the value of aR to P, before it is divided by 2.

To make a first estimation of the time, we consider that the probability to find $B_i = 1$ or 0 is the same, i.e. $0,5$. Therefore, for each k bits of B_i, when it is 1, it needs to carry out an addition of k bits (i.e. addition of a_i) and a division by 2 of P. Otherwise, when it is 0 only one right bit shifting is required. Therefore, k divisions by 2 and $k/2$ additions. Since, each division by 2 is always a shifting and, with probability 0.5, is also an addition of n. Therefore, the total time is:

$k(d + s/2) + (k/2)s = k(d + s)$, where d is the time for k right bit shifting, and s is the time for k bits addition.

Microprocessor MSP430 offers 16-bits operations. Therefore, additions and bits shifting are carried out in blocks of 16 bits. α is the time for 16-bits additions and shifting (usually 1 to 4 CPU cycle for bit shifting and 1 to 6 cycles for additions, depends on access to memory and registers). The final time is:

$$2\alpha k^2/16 = \alpha k^2/8.$$

The program code of the bit shifting algorithm is presented in the Algorithm 1. This has been programmed in assembler code with the other presented optimizations. The assembled code is based on MSPGCC.

3.2 Microprocessor's Multiplication Operation

Let an instruction from the microprocessor's set of instructions to carry out multiplication operation, which operates 2 registers of 16-bits and save the 32-bits of the result in two registers of 16-bits. This instruction is simulated in MSP430 chip, in [17, page. 478-480]. It is called μ to the time spent by that operation, and α for the time of 16-bits additions and 16-bits shifting.

Let the next numbers to apply the multiplication $aR = \sum_j \overline{A}_j 2^{16j}$ and $bR = \sum_j \overline{B}_j 2^{16j}$. In this case, j values are between 0 and $k/16$, instead of between 0 and k. Therefore, for each multiplication of k bits, it needs to carry out $k/16$ 16-bits multiplications and $2k/16$ additions, getting the results in a variable of $k + 16$-bits. Therefore, the time is equal to: $(\mu + 2\alpha)k/16$.

For each step of the multiplication of aR and the digits of $\overline{B}_i$, since multiplication is carried out in blocks of 16-bits, this needs to add the current result with the previous one i.e. an addition of (a sum of two numbers k bits and 16 bits. Thus, $\alpha(k+1)/16$ additions), then it needs to divide it by 2^{16} mod n. It is called δ to the time used for the division by 2^{16}.

The total time for each one of the 16 bits blocks ($k/16$ blocks, $\overline{B}_i$) is $(\mu + 2\alpha)k/16 + \alpha(k+1)/16 = \frac{\mu k + \alpha(3k+1)}{16}$, and addition δ, i.e. the total time is:

$$\frac{\mu k^2 + \alpha(3k+1)k}{256} + \frac{k\delta}{16}.$$

Division of a number of $k+16$ bits by 2^{16} mod n is carried out adding n until that the result is multiple of 2^{16}. If the last digit of n in base 2^{16} is 1 the process is simple, since the number of times to subtract n is indicated by the last digit of the number to be divided by 2^{16} mod n. Therefore the total time is:

$$\delta_t = \frac{(\mu+3\alpha)k+\alpha}{16}.$$

δ_t is the ideal time, when n has been chosen such that its last digit is equal to 1, in order to carry out in a simple way the division. In a general case, it cannot be assumed that the value of the last digit of n is equal to 1, thus this estimation is not realistic. But, Extended Euclidean algorithm can be used, in order to fix the process pre-calculating the modular multiplicative inverse of the last digit of n mod 2^{16} and it can be used with the last digit of p. Therefore, it is reached a more realist time:

$$\delta = \frac{k}{16}(\mu + 3\alpha) + 2\mu + 2\alpha.$$

This can be simplified considering that terms without k are not highly relevant for the total time. Therefore δ_t and δ are very similar, in the order of $\frac{k}{16}(\mu+3\alpha)$. Therefore, based on that expression and reducing terms that do not have k^2, the total time for microprocessor's multiplication operation is:

$$M \simeq \frac{\mu k^2}{256} + \frac{(\mu+3\alpha)k^2}{256} = \frac{(2\mu+3\alpha)k^2}{256}.$$

Algorithm 1. Code based on Bit shifting

```
accumulator = 0
for i = 0 to k do
  if B_i equals 1 then
    accumulator = accumulator + A
  end if
  if accumulator is odd then
    accumulator = (accululator + p)/2
  else
    accumulator = accumulator/2
  end if
end for
```

3.3 Comparative between Bit Shifting and Microprocessor's Multiplication Operation

The comparative between bit shifting and microprocessor's multiplication operation shows us that in a general way bit shifting is better than microprocessor's multiplication operation, when the following equation is true:

$$\frac{\alpha k^2}{8} < \frac{(2\mu+3\alpha)k^2}{256} \Rightarrow 32\alpha < 2\mu + 3\alpha \Rightarrow \frac{29}{2} < \frac{\mu}{\alpha}.$$

In conclusion, when the number of cycles to carry out microprocessor's multiplication operation is more than 15 times the cycles to carry out addition or bit shifting, it is preferable bit shifting solution. Since, MSP430 microprocessor's multiplication operation requires a big amount of clock cycles (150 cycles in the MSP430), while the bit shifting and additions only needs between 1 and 4 cycles for bit shifting and 1 and 6 cycles for addition, this depends on the access to registers and memory, i.e. $rrcR4$, i.e. bit shifting for registers is just 1 cycle, but $rrc0(R1)$, which is bit shifting in the memory address with value $R1$ are 4 cycles. The evaluation has presented that bit shifting is better than microprocessor's multiplication operation with a relation of when its cost is 15 times or less than multiplication i,e, $\mu < 15\alpha$, and MSP430 has a μ/α between 38 and 150.

4 Shifting Primes

Shifting primes are special pseudo-Mersenne primes for bit shifting Montgomery multiplication. They have been defined under this work to optimize them for the bit shifting implementation presented in the Section 3. The shifting primers are formally defined as:

Definition 1. *It is said that p is a shifting prime (of type α and λ), if p is a prime and exists u such that: $p = u \cdot 2^{\lambda-\alpha+1} - 1$ and $2^{\alpha-2} < u < 2^{\alpha-1}$.*

The parameter α denotes the length of the word for addition and λ the length of the prime number. Our work is focused on the case $\alpha = 16$ and $\lambda = 160$, i.e. ECC with 160-bits key length in our MSP430-based 16-bits microprocessor. Notice that if $2^{\alpha-2} < u < 2^{\alpha-1}$ then $2^{\alpha-2+\lambda-\alpha+1} - 1 < p < 2^{\alpha-1+\lambda-\alpha+1} - 1$. Therefore, $2^{\lambda-1} - 1 < p < 2^{\lambda} - 1$ and then p is λ-bits length.

The number of shifting primes depends on λ and α. For example, for $\alpha = 8$ and $\lambda = 160$ there is only one (with $u = 100$). For $\alpha = 16$ and $\lambda = 160$ there are 288 shifting primes.

The basic operations based on these special primes (shifting primes) for the Montgomery multiplication presented in the Section 3 are presented in the next subsection.

4.1 Basic Operations

In order to implement the basic operations, it has been considered, in addition to the mentioned, the next optimizations.

On the one hand, the points and coordinates, for an elliptic curve E over a field is a nonsingular cubic curve, are defined over the projective plane. It has been considered the field $\mathbb{Z}_p$ with p a 160 bits prime, and E in Weierstrass normal form, $E : y^2 = x^3 + ax + b$. It has been considered the special case with $a = -3$, which reduces the amount of operations. There are different coordinate systems that can be used to represent the points. We consider the mixed coordinate system considered in [6] for which the basic time for scalar multiplication is $1610.2M$ with M the time for a basic 160 bits modular multiplication mod p, our focus

in our research is optimize modular multiplication, since scalar multiplication is based on this.

On the other hand, all the operations are implemented in assembler, where one of the main decisions has been to use 10 registers to store a number (called the accumulator) in which we make the basic bit shifting and additions. This decision makes that only 2 registers are available for other operations, but it is worth, since operations with the accumulator are very fast, and when the accumulator is combined with the shifting primes, the result is also very quick.

Following the same methodology defined in the Section 3, there are three main operations in Montgomery multiplication:

Division by 2. The most basic operation for Montgomery multiplication is $x \mapsto x \cdot 2^{-1}$ in $\mathbb{Z}_p$. Suppose $x = x_0 + x_1 \cdot 2^{16} + \cdot + x_9 \cdots 2^{16 \cdot 9} < p$.

The usual algorithm for this operation is as follows:

if x is even **then**
  result is shifting x one position to the left.
else
  result is $x + p$ shifted one position to the left.
end if

Even when is being used the accumulator, it is needed 3 cycles to check if x is odd and jump depending on it. Once we have decided that, it is needed 10 cycles to shift the accumulator in the best case, and 30 cycles to add a general prime p and shift. This makes that this algorithm for a general prime requires between 13 and 33 cycles with an optimal programming.

The algorithm for shifting primes is:

shift x
if no carry (i.e. if x was even) **then**
  jump to (END), because the result is already in x.
end if
ignore the carry and add u to the most significant word of x.
(END) The result is in x

The result is clear when x is even. In case x is odd, if this shifts x, then this gets $(x-1)/2$ and this requires $(x+p)/2$. But, if u is added to the most significant word of x, then the result reached is

$$(x-1)/2 + u \cdot 2^{\lambda-\alpha} = \frac{x - 1 + u \cdot 2^{\lambda-\alpha+1}}{2} = \frac{x+p}{2}$$

It is exactly the result required. This is the advantage from the shifting primes.

The number of cycles for this operation with this optimization is equal to: 10 cycles to shift x, 2 cycles for the jump, and another 2 cycles in case that it is required to add u. Realise, that when x is between 0 and p, then $(x+p)/2$ is also between 0 and p, and consequently it is not required any additional correction. The total number of cycles is 12 in the even case and 14 in the odd case. This reduction is significant because this operation should be done λ times for a

Montgomery multiplication. Since, the probability for odd x is 0.5 the usual algorithm would give $13 \cdot 0.5 + 33 \cdot 0.5 = 23$ cycles and the one with shifting primes 13 cycles. Therefore, a reduction of the $43,47\%$ of the cycles is reached.

Addition and correction modulo p. Montgomery multiplication algorithm requires to add the second operand to the accumulator depending on the value of the bits of the first operand. In order to to keep the result between 0 and $p-1$, when the addition is over this quantity, it is required to make a correction, i.e. subtract p, to offer the result inside the range.

Adding a λ-bits variable to the accumulator requires a lot of cycles, because the variable should be in memory. Specifically, it is required 10 additions `add(c).w mem,reg`, where for each addition are required 3 cycles. Therefore, the whole addition 30 cycles.

The correction for a general prime requires to compare the result with the prime. In order to do it, this compares the significant word of the accumulator with the most significant word of the prime (2 cycles), and then a jump (2 cycles) is carried out to different places, depending on the result. There are two cases in which the problem is clear (if the numbers are not equal). If they are equal, then it needs to check the following values, since it is possible to require a correction or not.

The mentioned correction is simpler in the case of shifting primes since:

Proposition 1. *Let p be a shifting prime $p = u \cdot 2^{\lambda-\alpha+1} - 1$ and $a = \sum_{i=0}^{\lambda/\alpha-1} a_i 2^{\alpha i}$ the accumulator after a partial sum in the Montgomery multiplication of x and y. Then:*

1. *a cannot be exactly p.*
2. *a needs no correction if and only if the most significant word of a is under $2u$.*

Proof.

1. In Montgomery multiplication, the accumulator has partial products $h \cdot y$. If $y \neq 0$ the partial products cannot be 0 (or p, that is the same element in $\mathbb{Z}_p$) and in case y is 0, the partial result would be always 0, not p.
2. The accumulator needs correction if and only if $a \geq p$, that using (1) is equivalent to $a > p$. Let $k = \lambda/\alpha - 1$. Then:

$$a = \sum_{i=0}^{k} a_i 2^{\alpha i} = a_k 2^{\lambda-\alpha} + \sum_{i=0}^{k-1} a_i 2^{\alpha i}$$

$$p = 2u2^{\lambda-\alpha} - 1$$

The number $\sum_{i=0}^{k-1} a_k 2^{\alpha i}$ is between 0 and $2^{\lambda-\alpha} - 1$ because it is written with $k-1$ words. Therefore:

$$a > p \Leftrightarrow a_k 2^{\lambda-\alpha} + \sum_{i=0}^{k-1} a_i 2^{\alpha i} > 2u2^{\lambda-\alpha} - 1$$

$$\Leftrightarrow (a_k - 2u)2^{\lambda-\alpha} > -\left(\sum_{i=0}^{k-1} a_i 2^{\alpha i} + 1\right)$$

The number $-\left(\sum_{i=0}^{k-1} a_i 2^{\alpha i} + 1\right)$ is negative, therefore if $a_k - 2u \geq 0$ we have $a > p$. Conversely, if $a_k - 2u < 0$ then $a_k - 2u \leq -1$. Therefore:

$$(a_k - 2u)2^{\lambda-\alpha} \leq -2^{\lambda-\alpha} \leq -\left(\sum_{i=0}^{k-1} a_i 2^{\alpha i} + 1\right).$$

This has been proved that the result needs correction if $a > p$ and this is equivalent to $a_k \geq 2u$. Then correction is required if and only if $a_k < 2u$.

Addition with shifting and correction modulo *p*. Following the Montgomery multiplication algorithm, this requires after the addition, a shifting for the following loop. It is usually better to consider both operations together because we can reduce the number of cycles avoiding a partial correction.

Suppose a is the accumulator and it required to calculate $(a + w)2^{-160}(p)$. The algorithm is the following:

add w to a from right to left
shift a from left to right with carry without previous correction.
if no carry **then**
  jump to (END)
end if
compare u with the most significant word of a.
if u is less than it **then**
  add u to the most significant word of a and jump to (END)
end if
sub u to the most significant word of a and add 1 to the final result.

4.2 Assembler Implementation and Execution Times

The previous algorithms and optimizations are implemented in the MSP430 with the following conventions: It is used the register R5 for u, R4 for the address of the operand, and 10 registers for the accumulator R6, R7, ..., R15.

DIV2

```
    RRC.w R6
    RRC.w R7
    RRC.w R8
    RRC.w R9
    RRC.w R10
    RRC.w R11
    RRC.w R12
    RRC.w R13
    RRC.w R14
    RRC.w R15
    JNC end
    ADD.w R5,R6
end:
```

In DIV2 it is needed that the carry flag is 0 before executing these instructions. Therefore, an extra instruction CLRC is required to clear the carry bit. The execution time is 12 cycles, when no carry and 13 cycles in the other case (probability 0.5). Therefore, this code needs 12.5 cycles.

In modADD R4, it is needed 30 cycles for the addition to the accumulator, 2 cycles to check if there is carry overflow. The probability of correction is around 0.5, therefore we are going to calculate both cases. If there is no correction, then it compares R6 with $2u$ (2 cycles) and jump to the end (2 cycles). This is equal to $30+2+2+2=36$ cycles. Otherwise, when correction is required, the highest probability if that carry bit is active after the addition, in that case, it is needed $30+2+1+2+\epsilon = 35+\epsilon$ cycles, where ϵ is a part of the code with very low probability. In conclusion, the average cycles needs for this code are 36 cycles.

modADD R4

```
ADD.w 18(R4),R15
ADDC.w 16(R4),R14
ADDC.w 14(R4),R13
ADDC.w 12(R4),R12
ADDC.w 10(R4),R11
ADDC.w 8(R4),R10
ADDC.w 6(R4),R9
ADDC.w 4(R4),R8
ADDC.w 2(R4),R7
ADDC.w 0(R4),R6
JC reqC
CMP.w R6,2u
JL end
```

```
reqC:   SUB.w 2u,R6
    ADD.w #1,R15
    JNC end
    ADD.w #1,R14
    ADDC.w #0,R13
    ADDC.w #0,R12
    ADDC.w #0,R11
    ADDC.w #0,R10
    ADDC.w #0,R9
    ADDC.w #0,R8
    ADDC.w #0,R7
    ADDC.w #0,R6
end:
```

modADD+DIV2 R4

```
ADD.w 18(R4),R15
ADDC.w 16(R4),R14
ADDC.w 14(R4),R13
ADDC.w 12(R4),R12
ADDC.w 10(R4),R11
ADDC.w 8(R4),R10
ADDC.w 6(R4),R9
ADDC.w 4(R4),R8
ADDC.w 2(R4),R7
ADDC.w 0(R4),R6
RRC.w R6
RRC.w R7
RRC.w R8
RRC.w R9
RRC.w R10
RRC.w R11
RRC.w R12
RRC.w R13
RRC.w R14
RRC.w R15
```

```
    JNC end
    CMP.w R5,R6
    JNC pre
    SUB.w R5,R6
    ADD.w #1,R15
    JNC end
    ADD.w #1,R14
    ADDC.w #0,R13
    ADDC.w #0,R12
    ADDC.w #0,R11
    ADDC.w #0,R10
    ADDC.w #0,R9
    ADDC.w #0,R8
    ADDC.w #0,R7
    ADDC.w #0,R6
    JMP end
pre:    ADD.w R5,R6
end:
```

In conclusion, it is required addition and shift. If the accumulator is even (probability is 0.5), then it is required 42 cycles, else it is 4 cycles in case that is required to add p and $7+\epsilon$ in case that it has that subtracts p. Therefore, the final cost is $42+0.5(4+0.5(3+\epsilon)) \equiv 45$ cycles.

5 Bit Shifting and Shifting Primes

The Section 4 has described the advantages for the Montgomery multiplication with the defined shifting primes. This section describes the union of the presented bit shifting implementation in the Section 3 and the mentioned shifting primes. Finally, some additional optimizations have defined for the whole process.

The operations on elliptic curves have been studied extensively and optimized for very different architectures. The basic operation with elliptic curves is the

scalar multiplication $n \times P$, where P is a point on the curve, and n is a number of large size. This operation is deeply analysed in [6], where is defined that the cost of the scalar multiplication for primes of 160-bits requires $1610,12$ modular multiplications of 160-bit numbers. Therefore, modular multiplication is what is being optimized in this work.

Such as mentioned in the related works, Section 2, there are several alternatives for the implementation of the modular multiplication. For example, TinyECC solution is based on Barrett reduction [9]. For our solution, it has been chosen Montgomery representation, since this is more suitable to exploit the advantages from the shifting primes.

Let x and y, which are the multiplication operands and p, a shifting prime for a determined u.

A basic implementation of the Montgomery multiplication with the shifting primes requires the following steps:

1. Traverse bit by bit the operand x.
2. If it is found a bit with value equal to 0, then the accumulator is rotated, i.e. (operation DIV2).
3. Otherwise, if it is found with value equal to 1, then y is added to the accumulator and the accumulator is also rotated (operation modADD+DIV2).

The cost of the operations DIV2 and modADD+DIV2 have been already mentioned in the Section 4.

For traversing x bit by bit has been used the next registers:

- 10 registers to store the accumulator, R6,R7,...,R15.
- 1 register to store u.
- 1 register to read the word of which is being traversed of x. Notice, that x is composed by 10 words of 16 bits.
- In order to access to the right word of x, it is stored in the stack memory the address of the last word which has been access of x increased in 2 memory units, in order that it is pointing to the next word of x.
- In addition, it is also stored in the stack memory the address from the first word of x.
- At the beginning of the loop to traverse the operand x, it is stored in the register R5 the address of the first word of x, and then is used the following code.

```
MOV.w 0(R5),R5
SETC
RRC R5
```

This code introduces a bit of control, which allows us to rotate the register until that the result is equal to 0. When, the result is 0, that bit is ignored, since it was introduced by us, with the presented code. Then, it is read the next word of x. This technique allows to avoid the use of a counter to control when the register has been fully traversed.

For example, an example where is used that method is the next code, which shows how to find the first bit which is equal to 1 of x.

```
      ADD.w #18,R5
      PUSH.w R5
      SUB.w #20,R5
      PUSH.w R5
next0: MOV.w 2(R1),R5
      CMP.w R5,0(R1)
      JZ end0
      SUB.w #2,2(R1)
      MOV.w 0(R5),R5
      SETC
Loop0: RRC R5
      JNC Loop0
      JZ next0
```

The jump to end0 is defined to finish returning the value 0, since it has read all the operand and it has not been found any bit set to 1. The total number of cycles is until 17 cycles for jumping to another word, considering that this operation is required for each new word (i.e. each 16 bits), it can be considered that is introduced 1.7 cycles by each bit of the operand.

Considering that the probability to find a bit set to either 1 or 0 are equal to 0.5, it is obtained the following time:

1. If the bit is 1, then it is required 3 cycles to check it. In addition, it needs to be checked the control bit, i.e. 2 additional cycles, and carry out an addition with rotation, which are 45 cycles. In total 50 cycles with probability 0.5.
2. Otherwise, if the bit is 0, it is also required 3 cycles to check it, and 15.5 cycles to rotate it. In total 18.5 cycles with probability 0.5.
3. Therefore, the mean number of cycles per bit is equal to $25 + 9.25 = 34.25$.
4. To the mentioned mean number of cycles needs to be added a jump, and the checking of end of world. In total is equal to 38.
5. This mean number of cycles per bit needs to be multiplied by 160 bits. Therefore, this results 6080 cycles, in addition this requires some pre-calculus and function callings, that we have estimated in 6293 cycles. This results with a clock speed equal to 8 Mhz from the MSP430,

$$1610 \cdot 6293/8 \cdot 10^6 = 1.2665 \; seconds.$$

This implementation offers better results than other implementations based on other types of primes. For example, notice that this operation with TinyECC has a cost to encrypt or decrypt, where is used the scalar multiplication of $3,271$ seconds and $2,217$ seconds respectively. These operations can be carried out with our implementation in around $1,2665$ seconds with shifting primes.

6 Results and Evaluation

The evaluation of the algorithms optimized has been initially simulated over our own developed simulator, which verifies the results with the cryptographic library LiDIA, and finally evaluated over real motes, specifically over Tmote Sky with the Contiki 2.4 OS, where is defined a set of functions with assembler code inline.

The quickest ECC algorithm is based on *Montgomery + window method*, see [6,16]. This has been optimized for MSP430 with bit shifting in assembler

language for the Montgomery multiplication of 160 bits and for the arithmetic advantages from the defined shifting prime. Modular Montgomery multiplication is carried out in 6293 cycles, and consequently scalar Montgomery in around 1610 times the modular Montgomery multiplication. The time reach, considering the 8 Mhz MSP430 microprocessor found in the Tmote Sky, is:

$$1610 \cdot 6293/8 \cdot 10^6 = 1.2665 \; seconds.$$

In order to reach this solution, we have used 10 microprocessor's registers to keep the 160 bits variable (the accumulator) with the partial multiplication results, with this optimization we have reduced almost the 40% of the total number of cycles, since *rrc* operation for bit shifting, and *add* operation for addition spend 1 cycle and 3 cycles respectively, instead of 4 and 6. In addition, loops have been unrolled in order to optimize more the final assembler code. Finally, such as mentioned special primes have defined in order to optimize the modular Montgomery multiplication, moving from a number of cycles for 12480 following the optimization from the Section 3 to 6293 cycles, i.e. from around $2,5$ seconds similar to TinyECC, which lower time is $2,217$ seconds to $1,2665$ seconds which is a $42,8\%$ lower than TinyEcc and our previous work [16].

7 Conclusions and Future Work

Future Internet of Things is defining a new set of challenges in order to offer security support, since technologies such as 6LoWPAN offers Internet connectivity to small and smart devices with highly constrained resources. Therefore, it is necessary to provide efficient, scalable, and suitable security mechanisms. For that reason, it is required Public Key Cryptography (PKC). This work has evaluated and optimized Elliptic Curve Cryptography (ECC) implementation for Future Internet of Things devices based on the Texas Instrument MSP430 microprocessor, which is used for several Future Internet of Things devices.

The optimizations for ECC are based mainly on bit shifting implementation of the modular Montgomery multiplication, and in a special type of primes (shifting primes) defined under this work, which offer a set of arithmetic advantages for the implementation of the bit shifting based modular Montgomery multiplication.

The result reached with the mentioned optimizations is $1,2665$ seconds for the scalar Montgomery multiplication, which reduces a $42,8\%$, with respect to the TinyECC implementation which offers a result of $2,217$ seconds. Therefore, it can be concluded, that with the reached time, ECC is suitable for the Future Internet of Things.

Finally remark, selection of special primes is a very well-known technique, which does not mean any vulnerability or weakness for our systems, e.g. standards such as FIPS 186-3 recommends specific elliptic curves for which modular arithmetic is simpler for 32 and 64 bits microprocessors. Therefore, our advance has been to determinate the ideal primes for Montgomery multiplication based on bit shifting operations and the 16-bits MSP430 microprocessor.

Ongoing work is focused on carry out additional optimizations based on reduction of the number of additions accessing to blocks of 4 bits in each step, instead of bit by bit, and the use of pre-calculated values.

References

1. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007)
2. Nobles, P., Ali, S., Chivers, H.: Improved Estimation of Trilateration Distances for Indoor Wireless Intrusion Detection. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications 2(1) (2011) ISSN: 2093-5374
3. Zampolli, S., Elmi, I., et al.: Ultra-low-power components for an RFID Tag with physical and chemical sensors. Journal of Microsystem Technologies 14(4), 581–588 (2008)
4. Norair, J.P.: DASH7: ultra-low power wireless data technology (2009)
5. Cohen, H., Miyaji, A., Ono, T.: Efficient Elliptic Curve Exponentiation. In: Han, Y., Quing, S. (eds.) ICICS 1997. LNCS, vol. 1334. Springer, Heidelberg (1997)
6. Cohen, H., Miyaji, A., Ono, T.: Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 51–65. Springer, Heidelberg (1998)
7. Montgomery, P.: Modular Multiplication Without Trial Division. Math. Computation 44, 519–521 (1985)
8. 802.15.4-2003, IEEE Standard, Wireless medium access control and physical layer specifications for low-rate wireless personal area networks (May 2003)
9. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In: 7th International Conference on Information Processing in Sensor Networks, SPOTS Track, USA, pp. 245–256 (2008)
10. Seo, S.C., Han, D.G., et al.: TinyECCK: Efficient Elliptic Curve Cryptography Implementation over GF(2m) on 8-bit MICAz Mote. IEICE Transactions on Info and Systems E91-D(5), 1338–1347 (2008)
11. Szczechowiak, P., Oliveira, L.B., et al.: NanoECC: Testing the Limits of Elliptic Curve Crytography in Sensor Networks. In: UNICAMP, Brasil (2008)
12. Gura, N., Patel, A., et al.: Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In: Workshop on Cryptographic Hardware and Embedded Systems (2004)
13. Hitchcock, Y., Dawson, E., et al.: Implementing an efficient elliptic curve cryptosystem over GF(p) on a smart card. ANZIAM Journal (2003)
14. Uhsadel, L., Poschmann, A., Paar, C.: Enabling Full-Size Public-Key Algorithms on 8-bit Sensor Nodes. In: European Workshop on Security and Privacy in Ad hoc and Sensor Networks (2007)
15. Hodjat, A., Batina, L., et al.: HW/SW Co-Design of a Hyperelliptic Curve Cryptosystem using a Microcode Instruction Set Coprocessor Integration. VLSI Journal 40(1), 45–51 (2007)
16. Ayuso, J., Marin, L., Jara, A., Skarmeta, A.F.G.: Optimization of Public Key Cryptography (RSA and ECC) for 8-bits Devices based on 6LoWPAN. In: 1st International Workshop on the Security of the Internet of Things, Tokyo, Japan (2010)
17. Bierl, L.: MSP430 Family Mixed-Signal Microcontroller Application Reports, pp. 478–480 (2000), `http://focus.ti.com.cn/cn/lit/an/slaa024/slaa024.pdf`
18. Locke, G., Gallagher, P.: FIPS PUB 186-3: Digital Signature Standard (DSS). National Institute of Standards and Technology (2009)