Optimization of Public Key Cryptography (RSA and ECC) for 16-bits Devices based on 6LoWPAN

Jesús Ayuso, Leandro Marin, Antonio Jara and Antonio F. G. Skarmeta

University of Murcia (Spain)

secIoT10

1ST INTERNATIONAL WORKSHOP ON THE SECURITY OF THE INTERNET OF THINGS

Tokyo (Japan), November 29, 2010

HELD IN CONJUNCTION WITH INTERNET OF THINGS 2010

Outline

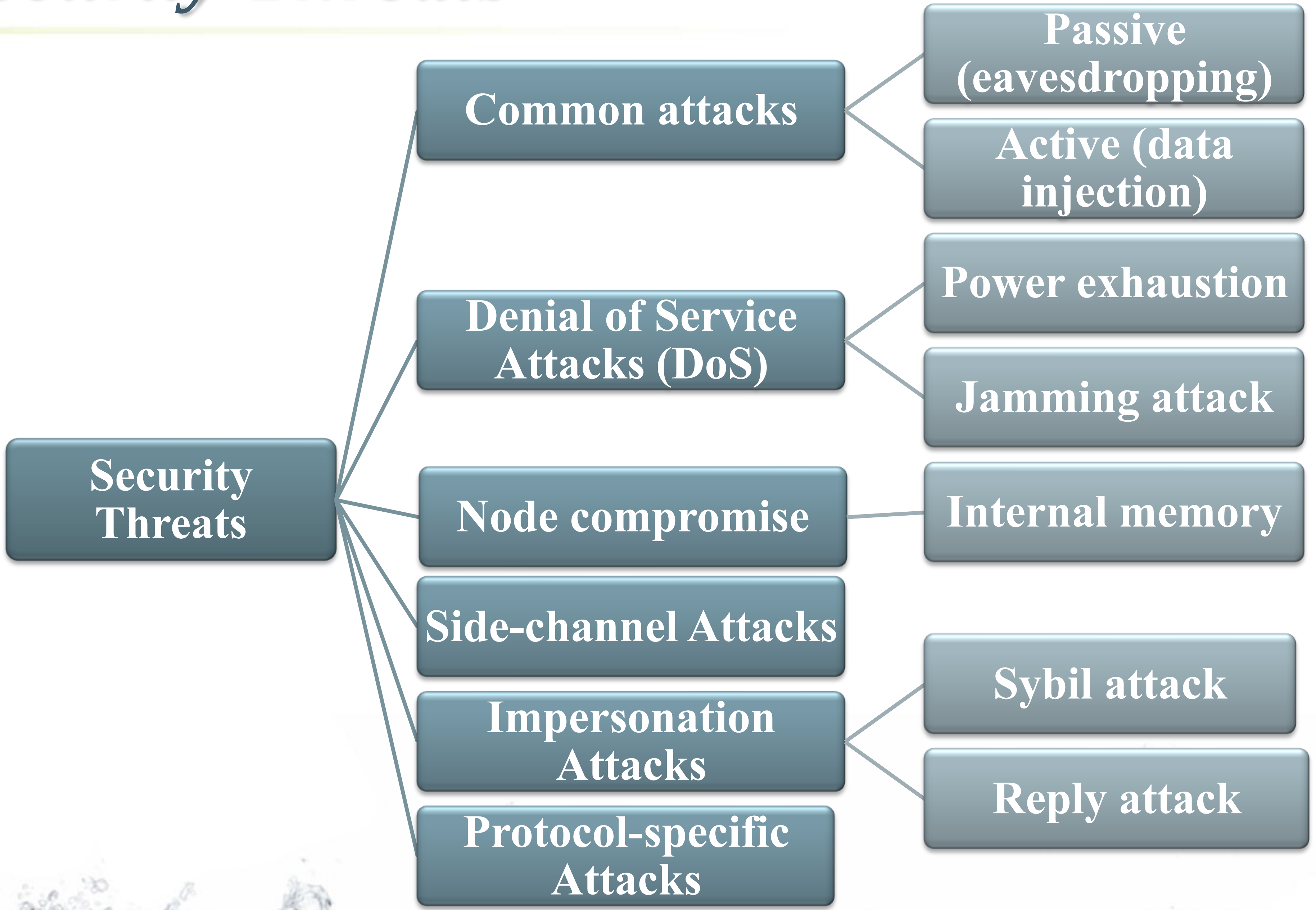
1. Introduction

Internet of things, and particularly 6LoWPAN is defining a new challenge for security, since wireless sensor networks are being connected to the Internet.

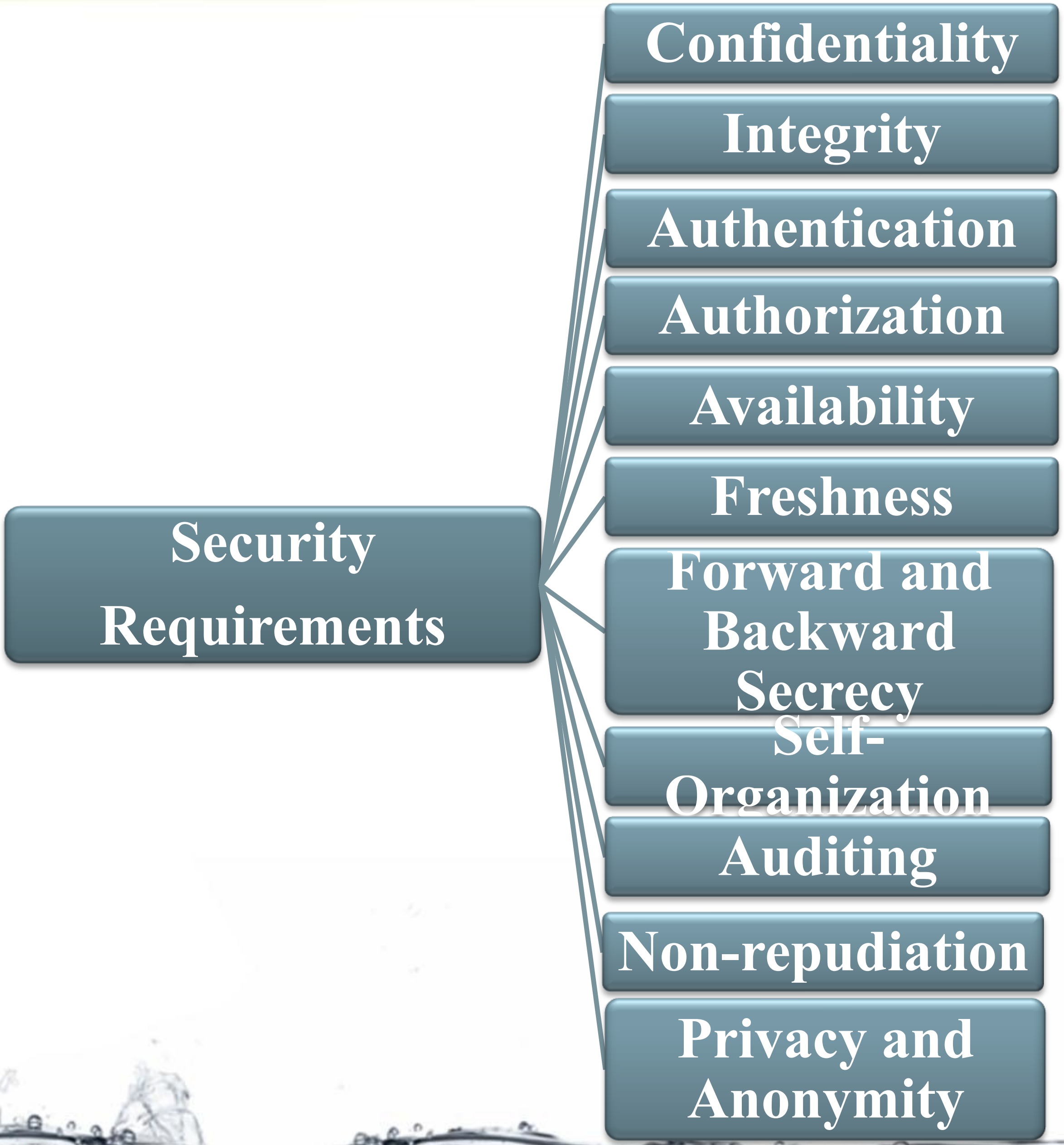
it is necessary to provide efficient and usable security mechanisms that could protect the WSN against attacks.

Healthcare based on Internet of Things present a new set of challenges in security, since this application domain is one of the most restrictive.

2. Security Threats



2. Security Requirements



3. Security primitives for 6LoWPAN devices

Symmetric Key Cryptography (SKC) provides confidentiality and integrity to the communication channel, and requires that both the origin and destination share the same security credential (i.e. secret key), which is utilized for both encryption and decryption.

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 reduce chips but mainly is going to be focused on ECC.

New scalability level, which can not be satisfied with Symmetric Key Cryptography (SKC). For that reason, it is required Public Key Cryptography (PKC), it is evaluated and optimized RSA and ECC algorithms.

4. Mathematical Optimization

RSA and ECC require two different integer sizes (k). Specifically, it is considered 1024-bit for RSA and 160-bit for ECC

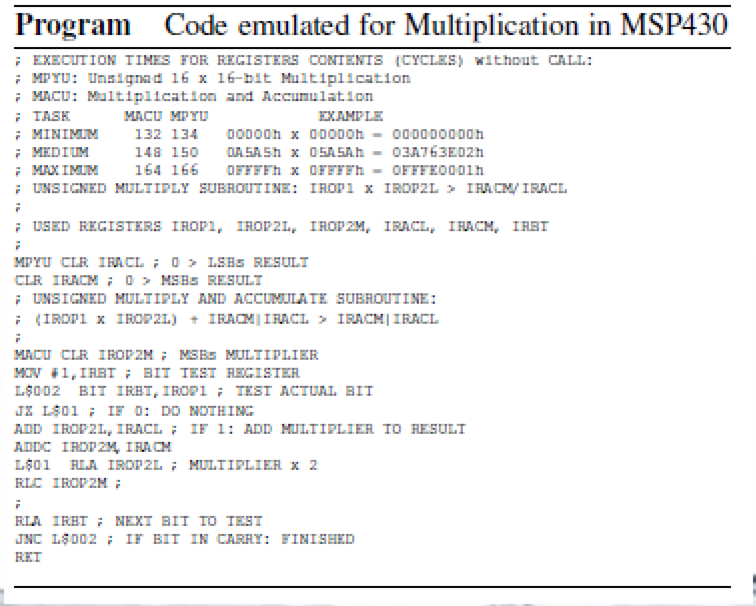
Montgomery's representation

In Montgomery representation for representing the numbers a and b, which are going to be multiplied, we have aR and bR mod n. Problem is coming with the multiplication operation, we get abR^2, but what we need is abR. The great advantage of Montgomery is just to avoid that, since Montgomery offers the reduction of the factor R.

Multiplication operation is what consumes the higher part of the time, since it is repeated thousands of times. For that reason, multiplication operation is what we will go to optimize and discuss more in detail

4. Mathematical Optimization (Multiplication)

Let an instruction from the microprocessor's set of instructions to carry out multiplication operation, which operated 2 registers of 16 bits and save the 32 bits of the result in two registers of 16 bits. For example, this instruction is simulated in MSP430 chip, which is the microprocessor used by Tmote Sky for our evaluation – around 150 cycles

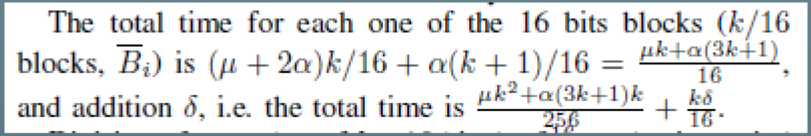


Program Code emulated for Multiplication in MSP430

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; MPYU: Unsigned 16 x 16-bit Multiplication
; MACU: Multiplication and Accumulation
; TASK       MACU MPYU              EXAMPLE
; MINIMUM     132 134    00000h x 00000h = 000000000h
; MEDIUM      148 150    0A5A5h x 05A5Ah = 03A763E02h
; MAXIMUM     164 166    0FFFFh x 0FFFFh = 0FFFE0001h
; UNSIGNED MULTIPLY SUBROUTINE: IROP1 x IROP2L > IRACM/IRACL
;
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM, IRBT
;
MPYU CLR IRACL ; 0 > LSBs RESULT
CLR IRACM ; 0 > MSBs RESULT
; UNSIGNED MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 x IROP2L) + IRACM|IRACL > IRACM|IRACL
;
MACU CLR IROP2M ; MSBs MULTIPLIER
MOV #1,IRBT ; BIT TEST REGISTER
L$002  BIT IRBT,IROP1 ; TEST ACTUAL BIT
JZ L$01 ; IF 0: DO NOTHING
ADD IROP2L,IRACL ; IF 1: ADD MULTIPLIER TO RESULT
ADDC IROP2M,IRACM
L$01  RLA IROP2L ; MULTIPLIER x 2
RLC IROP2M ;
;
RLA IRBT ; NEXT BIT TO TEST
JNC L$002 ; IF BIT IN CARRY: FINISHED
RET
```

4. Mathematical Optimization (Multiplication)

We have aR and bR mod n. For each step of the multiplication of aR by the digits of Bi, since multiplication is carried out in blocks of 16 bits, this needs to add the result with the previous result i.e. an addition of (a sum of two numbers k bits and 16 bits, so alpha (k +1) /16 additions), then this needs to divide it by 2^16 mod n, we call delta for 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 $\frac{\mu k^2 + \alpha(3k+1)k}{256} + \frac{k\delta}{16}$.

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

Based on Extended Euclidean algorithm

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

4. Mathematical Optimization (bit shifting)

We have aR and bR mod n are stored in binary representation, so

$aR = \sum_i a_i 2^i$ and $bR = \sum_i b_i 2^i$

We calculate (aR)(bR)R^-1 = (ab)R, therefore we need to carry out k right bit shiftings (with k = 160 or 1024 according)

Since modulus n is odd, 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 and be able to shift it.

For the multiplication process is needed a variable to accumulate the current result, we will call to that variable P, Each one of the digits from P is going to be multiplied by the digits from A and divided by 2.

4. Mathematical Optimization (bit shifting)

To estimate the time, we will consider that the probability of finding Bi = 1 or 0 is the same, i.e. 1=2. Therefore, for each k bits of Bi, when it is 1, it needs to carry out an addition of k bits (i.e. addition of ai) and a division by 2 of P. And when it is 0 just only one right bit shifting. Therefore, k divisions by 2 and k=2 additions. Since, each division by 2 is, such as mentioned, when it is a shifting and 1/2 times also an addition of n.

The total time is: $k(d + s/2) + (k/2)s = k(d + s)$

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

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

4. Mathematical Optimization (bit shifting)

Code based on Bit shifting

```
for(i=0;i<k;i++)
   P[i]=0;

for(i=0;i<k-1;i++){
   for(shift = 0x01; shift!=0; shift<<=1){
     if(b[i] & shift) add(P,a);
     if(P[0] & 0x01) add(P,n);

     for(j=0; j<k-1; j++){ /*Right bit shifting*/
       P[j] >>= 1;
       if(P[j+1] & 0x01) P[j] |= 0x80;
     }
     P[k-1] >>= 1;
   }
}

if(isGreaterEqual(P,n))
   sub(P,n);
```

Program Sketch of assembler code for Montgomery's multiplication based on bit shifting

```
//res, where is stored the partial result, is kept in the register since %0 to %9
" mov.w #0,%0 \n\t mov.w #0,%1 \n\t mov.w #0,%2 \n\t mov.w #0,%3 \n\t mov.w #0,%4
mov.w #0,%5 \n\t mov.w #0,%6 \n\t mov.w #0,%7 \n\t mov.w #0,%8 \n\t mov.w #0,%9 \n\t"

//keep in the memory stack the fist operand and the second one divided by 2
" rrc.w 18(%10) \n\t push.w 18(%10) \n\t"

//The same between 18 and 0 in block of 2 bytes (16 bits)
" rrc.w 0(%10) \n\t push.w 0(%10) \n\t"

//We carry out two versions depending on the carry flag (oddity of the second operand).
//Depending on the bits of the first operand we have to repeat several additions
//and bit shifting of the registers where is stored the partial result.

//ADDITION OF 160 BITS IS CARRIED OUT IN 30 CYCLES
".LsA2: add.w 2(r1),%0 \n\t"
" addc.w 4(r1),%1 \n\t"
" addc.w 6(r1),%2 \n\t"
" addc.w 8(r1),%3 \n\t"
" addc.w 10(r1),%4 \n\t"
" addc.w 12(r1),%5 \n\t"
" addc.w 14(r1),%6 \n\t"
" addc.w 16(r1),%7 \n\t"
" addc.w 18(r1),%8 \n\t"
" addc.w 20(r1),%9 \n\t"

//BIT SHIFTING OF 160 BITS IS CARRIED OUT IN 10 CYCLES
" rrc %9 \n\t rrc %8 \n\t rrc %7 \n\t rrc %6 \n\t rrc %5 \n\t rrc %4 \n\t rrc %3 \n\t
rrc %2 \n\t rrc %1 \n\t rrc %0 \n\t"
```

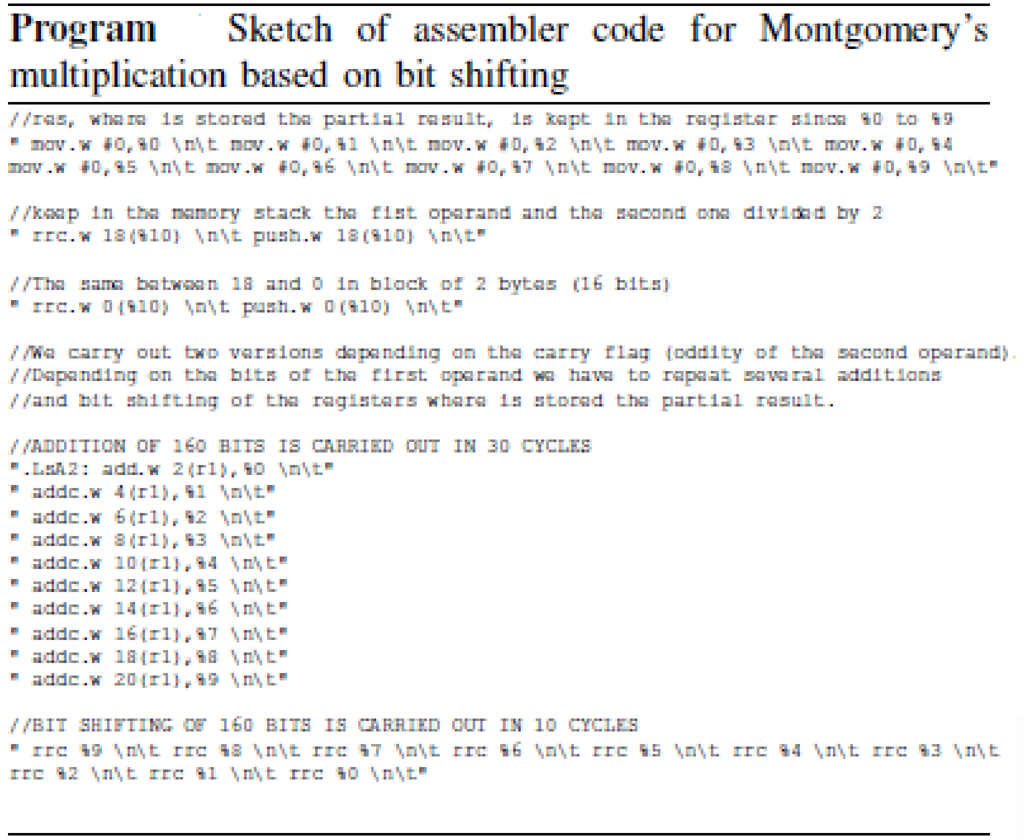
5. Results - Comparative

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

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), while the bit shifting and additions are supported and it just needs between 1 n 4 cycles for bit shifting and 1 and 6 cycles for addition, Therefore, 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. It is between 38 and 150.

5. Results - Comparative

Multiplication is carried out in12480 cycles (1,5625 milliseconds in the 8 Mhz MSP430 from Tmote Sky). In order to reach this solution, we have used 10 microprocessor’s registers to keep the 160 bits variable 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. The Program shows as bit shifting of the 160 bits is carried out in just 10 cycles, and addition in 30 cycles with this optimization. Finally, we have unrolled loops in order to optimize more the final assembler code.



Program Sketch of assembler code for Montgomery’s multiplication based on bit shifting

```
//res, where is stored the partial result, is kept in the register since %0 to %9
" mov.w #0,%0 \n\t mov.w #0,%1 \n\t mov.w #0,%2 \n\t mov.w #0,%3 \n\t mov.w #0,%4
mov.w #0,%5 \n\t mov.w #0,%6 \n\t mov.w #0,%7 \n\t mov.w #0,%8 \n\t mov.w #0,%9 \n\t"

//keep in the memory stack the fist operand and the second one divided by 2
" rrc.w 18(%10) \n\t push.w 18(%10) \n\t"

//The same between 18 and 0 in block of 2 bytes (16 bits)
" rrc.w 0(%10) \n\t push.w 0(%10) \n\t"

//We carry out two versions depending on the carry flag (oddity of the second operand).
//Depending on the bits of the first operand we have to repeat several additions
//and bit shifting of the registers where is stored the partial result.

//ADDITION OF 160 BITS IS CARRIED OUT IN 30 CYCLES
".LsA2: add.w 2(r1),%0 \n\t"
" addc.w 4(r1),%1 \n\t"
" addc.w 6(r1),%2 \n\t"
" addc.w 8(r1),%3 \n\t"
" addc.w 10(r1),%4 \n\t"
" addc.w 12(r1),%5 \n\t"
" addc.w 14(r1),%6 \n\t"
" addc.w 16(r1),%7 \n\t"
" addc.w 18(r1),%8 \n\t"
" addc.w 20(r1),%9 \n\t"

//BIT SHIFTING OF 160 BITS IS CARRIED OUT IN 10 CYCLES
" rrc %9 \n\t rrc %8 \n\t rrc %7 \n\t rrc %6 \n\t rrc %5 \n\t rrc %4 \n\t rrc %3 \n\t
rrc %2 \n\t rrc %1 \n\t rrc %0 \n\t"
```

5. Conclusions and Future Work

Internet of things, and particularly 6LoWPAN is defining a new challenge for security, since wireless sensor networks are connected to the Internet

Healthcare domain and specifically mHealth requires scalable security based on PKC

The evaluation has concluded that ECC is a suitable solution for Future Internet devices, such as 6LoWPANnodes, since time spent for Montgomery multiplication is just 1,5625 milliseconds.

Ongoing work is focused on assembler implementation of the modular inverses, in order to define a full optimized exponentiation to implement ECDSA (for digital signature), ECIES (for data encryption), and ECDH (for key establishment). In order to compare our solution with respect to other current implementations such as TinyECC.

Questions?

Thanks for your attention

Questions?

Do you want to get more information?
Then, get in contact with
Leandro Marin - University of Murcia (Spain) – leandro@um.es

Extra

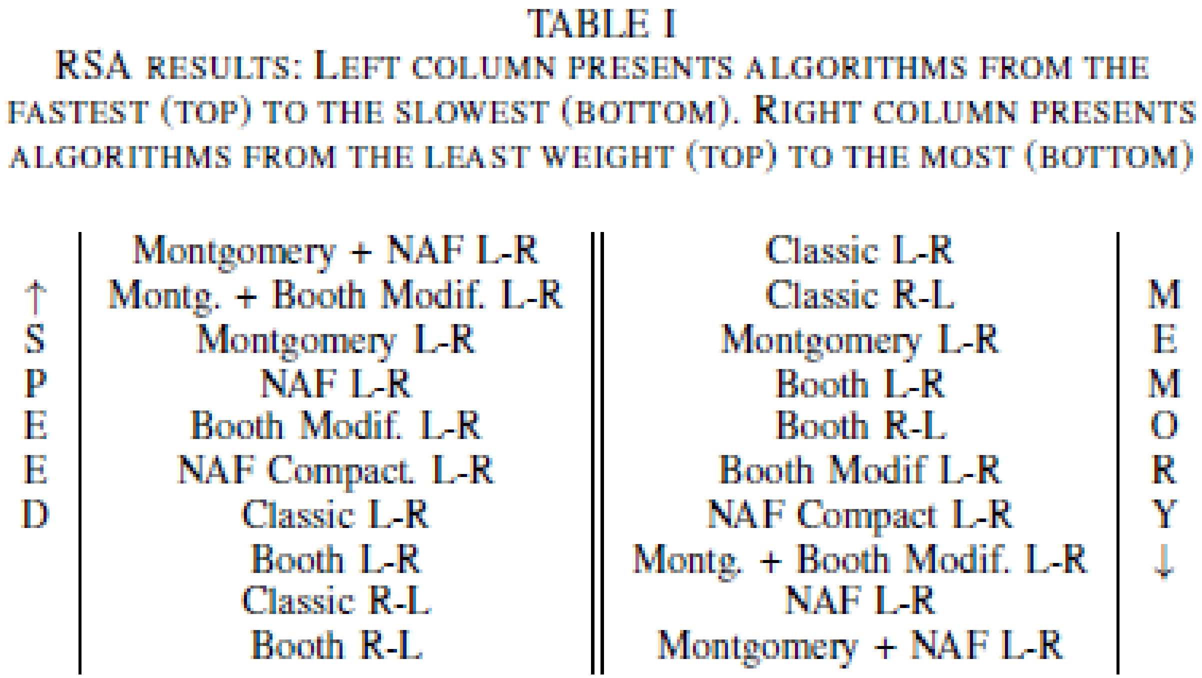


TABLE I
RSA RESULTS: LEFT COLUMN PRESENTS ALGORITHMS FROM THE FASTEST (TOP) TO THE SLOWEST (BOTTOM). RIGHT COLUMN PRESENTS ALGORITHMS FROM THE LEAST WEIGHT (TOP) TO THE MOST (BOTTOM)

	Montgomery + NAF L-R	Classic L-R	
↑	Montg. + Booth Modif. L-R	Classic R-L	M
S	Montgomery L-R	Montgomery L-R	E
P	NAF L-R	Booth L-R	M
E	Booth Modif. L-R	Booth R-L	O
E	NAF Compact. L-R	Booth Modif L-R	R
D	Classic L-R	NAF Compact L-R	Y
	Booth L-R	Montg. + Booth Modif. L-R	↓
	Classic R-L	NAF L-R	
	Booth R-L	Montgomery + NAF L-R	

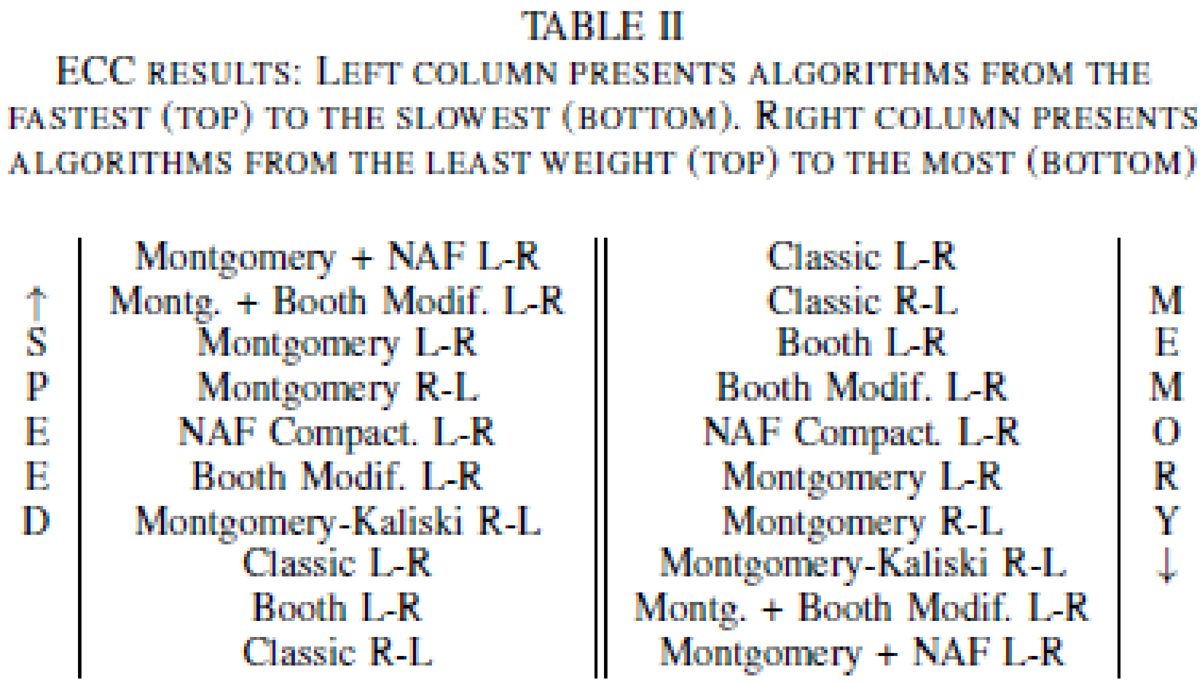


TABLE II
ECC RESULTS: LEFT COLUMN PRESENTS ALGORITHMS FROM THE FASTEST (TOP) TO THE SLOWEST (BOTTOM). RIGHT COLUMN PRESENTS ALGORITHMS FROM THE LEAST WEIGHT (TOP) TO THE MOST (BOTTOM)

	Montgomery + NAF L-R	Classic L-R	
↑	Montg. + Booth Modif. L-R	Classic R-L	M
S	Montgomery L-R	Booth L-R	E
P	Montgomery R-L	Booth Modif. L-R	M
E	NAF Compact. L-R	NAF Compact. L-R	O
E	Booth Modif. L-R	Montgomery L-R	R
D	Montgomery-Kaliski R-L	Montgomery R-L	Y
	Classic L-R	Montgomery-Kaliski R-L	↓
	Booth L-R	Montg. + Booth Modif. L-R	
	Classic R-L	Montgomery + NAF L-R	