Shifting Primes on OpenRISC Processors with Hardware Multiplier

Leandro Marin, Antonio Jara, Antonio Skarmeta

Shifting Primes on OpenRISC Processors with Hardware Multiplier

Leandro Marin[1], Antonio J. Jara[2], and Antonio Skarmeta[2]

[1] Department of Applied Mathematics
[2] Research Institute for Oriented ICT (INTICO)
Computer Sciences Faculty, University of Murcia
Reg. Campus of Int. Excellence "Campus Mare Nostrum"
Murcia (Spain)
{leandro, jara, skarmeta}@um.es

Abstract. Shifting primes have proved its efficiency in CPUs without hardware multiplier such as the located at the MSP430 from Texas Instruments. This work analyzes and presents the advantages of the shifting primes for CPUs with hardware multiplier such as the JN5139 from NXP/Jennic based on an OpenRISC architecture. This analysis is motivated because Internet of Things is presenting several solutions and use cases where the integrated sensors and actuators are sometimes enabled with higher capabilities. This work has concluded that shifting primes are offering advantages with respect to other kind of primes for both with and without hardware multiplier. Thereby, offering a suitable cryptography primitives based on Elliptic Curve Cryptography (ECC) for the different families of chips used in the Internet of Things solutions. Specifically, this presents the guidelines to optimize the implementation of ECC when it is presented a limited number of registers.

1 Introduction

Internet of Things proposes an ecosystem where all the embedded systems and consumer devices are powered with Internet connectivity, distributed intelligence, higher lifetime and higher autonomy. This evolution of the consumer devices to more connected and intelligent devices is defining the new generation of devices commonly called "smart objects".

Smart objects are enabled with the existing transceivers and CPUs from the Wireless Sensor Networks (WSNs), i.e. CPUs highly constrained of 8 and 16 bits such as ATMega 128, Intel 8051, and MSP430 [7]. But, since the level of intelligence and required functionality is being increased, some vendors are powering the consumer devices with CPUs not so constrained such as ARM 5 used in the SunSpot nodes [8] from Oracle Lab or the NXP/Jennic JN5139 used in the imote and recently in the first smart light from the market based on 6LoWPAN presented by GreenWave [9].

These smart objects require a suitable security primitives to make feasible the usage of scalable security protocols for the application layer such as DTLS, which

has been considered the security to be applied over the Constrained Application Protocol (CoAP) [10] over IPv6 network layer [11].

Specifically, CoAP and the Smart Energy profile for ZigBee alliance (SE 2.0) are considering DTLS 1.2 described in the RFC6347 [12]. This extends the ciphersuites to include the supported by hardware in the majority of the Wireless Sensor Networks transceivers, i.e. AES-128 in CCM mode. In addition, this includes Elliptic Curve Cryptography (ECC) for establishing the session.

Therefore, the challenge is in offering a suitable ECC implementation for the authentication and establishment of the sessions through algorithms such as DTLS.

ECC implementations have been optimized in several works of the state of the art. For example, it has been optimized for constrained devices based on MSP430 in our previous works. But, such as described, the market is not limited to these highly constrained devices therefore it needs to be evaluated how the special primes considered for very specific CPU architectures and conditions are performing for other CPUs.

This work presents the shifting primes and describes how they were used for the MSP430, then it is described the new architecture from the JN5139 CPU, and finally how the shifting primes continue being interesting for the implementation over this higher capabilities, in particular shifting primes offer a feature to carry out the reduction modulo p at the same time that it is carried out the partial multiplication in order to optimize the usage of the registers and consequently reach an implementation which is presenting the best performance from the state of the art for the JN5139 CPU.

2 Shifting Primes

Shifting primes are a family of pseudo-mersenne primes that were designed, in [4], to optimize the ECC cryptography primitives when the CPU is not supporting a hardware multiplication. This type of constrained CPUs is the commonly used in sensors and actuators for home automation and mobile health products. For example, the low category of the MSP430 family from Texas Instrument [7].

Similar primes to the shifting primes have been previously mentioned in [14], but they did not exploited its properties and applications. These new properties which are the used to optimize the implementation for constrained devices without hardware multiplier were described in [4]. In addition, this work presents new features for its optimization in CPUs with hardware multiplication support.

A shifting prime p is a prime number that can be written as follows: $p = u \cdot 2^{\alpha} - 1$, for a small u. In particular we are using for the implementations $p = 200 \cdot 2^{8 \cdot 19} - 1$. There are more than 200 shifting primes that are 160-bit long. The details about this definition can be seen in [4] and [5].

For the implementation of the ECC primitives is used the Montgomery representation for modular numbers. Thereby, computing $x \mapsto x/2(p)$ is very fast even without a hardware multiplier when the shifting primes are used.

Operations using shifting primes can be optimized computing $x \mapsto x/2^{16}(p)$ instead of shifting one by one each step during the multiplication. By using this technique, MSP430 can make a single scalar multiplication within 5.4 million clock cycles in [3].

But, the situation is rather different when the CPU supports hardware multiplication. For this situations, the use of the hardware multiplier through the offered instructions set performs better, since blocks of several bits can be multiplied within a few cycles, for example blocks of 16 bits for a CPU of 32 bits with a 16 bits multiplier such as the located at the JN5139 CPU from Jennic/NXP.

The following sections present as the implementation of the ECC primitives can be optimized for CPUs with hardware multiplier and the advantages that the shifting primes are offering for these high capability CPUs yet.

3 C and Assembler in JN5139

The ECC primitives implementation has been optimized for Jennic/NXP JN5139 microcontroller. The implementation is mainly developed in C, but there are critical parts of the code that require a more precise and low level control, and they have required the use of assembler. In particular, assembler has been used for the basic arithmetic (additions, subtractions and multiplications modulo p).

The target architecture of this chip is based on the OpenRISC 1200 instruction set, and it has been named "Beyond Architecture" or "ba". In particular, the basic instruction set for JN5139 is called "ba1" and the one for JN5148 is called "ba2".

Some of the characteristics that are important in our implementation are:

1. 32 general purpose registers (GPRs) labeled r0-r31. They are 32 bits wide. Some of them are used for specific functions (r0 is constantly 0, r1 is the stack pointer, r3-r8 are used for function parameters and r9 is the link register). See [6, Section 4.4] and [6, Subsection 16.2.1].
2. All arithmetic and logic instructions access only registers. Therefore, they require the use of load and store instructions to access memory. For this purpose, OpenRISC offers the required instructions to load and store in a very flexible way, i.e. this offers operations for bytes, half words and words between registers and memory. The memory operations consume a low number of cycles. Load operations require two clock cycles and store operations require one clock cycle, when there is not cache line miss or DTLB miss. See [1, Page 15].
3. Addition operation offers different instructions; addition with carry and addition with immediate value (`l.add`, `l.addc`, `l.addi`), which consume one clock cycle.
4. Multiplication instruction `l.mul` requires tree clock cycles. See [1, Table 3.2]. This multiplies two registers of 32 bits, but this only stores the result in a single register of 32 bits. Therefore, in order to now loss information, it is only used the least significant part of the registers, i.e. 16 effective bits from the 32 bits, to make sure that the result fit into a single register of 32 bits.

5. The clock frequency of the JN5139 CPU is 16MHz.

Since the RISC principles from the CPU used, the instruction set is reduced, but this offers a very fast multiplication (within 3 clock cycles) with respect to the 16 bits emulated multiplication available in the MSP430 CPUs, which requires more than 150 cycles. Therefore, the support for the multiplication is highly worth for the modular multiplication performance. Such as mentioned, the multiplication offered by the JN5139 CPU is limited to the least significant 32 bits of the result, therefore this requires to limit the multiplication for its usage in the modular multiplications. This is important because the implementation carries out 16x16 multiplications to avoid information loss.

Another important characteristic is that there are available a high number of registers. Therefore, this allows to keep all the information in registers during the multiplication process, and consequently reduce the number of memory operations.

The following sections presents how the multiplication modular is implemented over the JN5139 CPU and how the shifting primes are optimized this implementation thanks to its suitability for the reduction modulo p, in order to reduce the total number of required registers.

4 Multiplication Algorithm

There are different options to compute the product $a \cdot b$ modulo p. The choice of one of them depends on the instruction set and the number of registers available. There are a lot of C implementations that could be optimal for some architectures, but they are rather inconvenient for other architectures.

The decisions considered for this implementation are dependent on this particular architecture, in terms such as the mentioned issues with the multiplication instruction, `l.mul`, which offers a multiplication of two 32-bit numbers, but only offers the least significant 32 bits as a result.

Let x and y, the two operands for the multiplication stored in 16 bits blocks with big endian memory. Then, $x = \sum_{i=0}^{10} x_i 2^{16(10-i)}$ and $y = \sum_{i=0}^{10} y_i 2^{16(10-i)}$. The basic multiplication algorithm requires to multiplicate each $x_i y_j$ and add the partial result from each partial multiplication to the accumulator.

The result from the multiplication of the $x_i y_j$ blocks is a number of 32 bits, which needs to me added to the accumulator. This addition to the accumulator requires previously the shifting of the result to the proper position regarding the index from the $x_i y_j$ blocks, i.e., $m_{ij} = x_i y_j 2^{16(10-i)} 2^{16(10-j)} = x_i y_j 2^{16(10-i-j)}$ requires a shifting of $2^{16(10-i-j)}$ to be added to the accumulator.

Since the result from the multiplication of the $x_i y_j$ blocks is a number of 32 bits, the result can be directly added only when i and j presents the same parity. This is mainly caused because m_{ij} will be divisible by 2^{32}, and consequently it will be aligned to a word (32 bits). Otherwise, when i and j are not presenting the same parity, m_{ij} will be divisible by 2^{16} and not by 2^{32}, consequently the memory is not aligned and it cannot be operated.

A solution for the presented problem with the memory is realign the results m_{ij} when $i + j$ is odd, but this requires to shift operations, and the addition to the accumulator of both results. This means 4 instructions instead of the 1 instruction when the result is . Note, that the 50% of the multiplications will not be aligned.

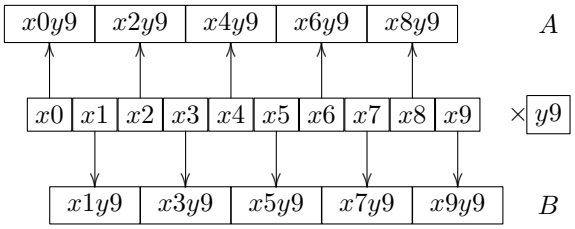
The proposed solution in this work in order to avoid the mentioned extra costs and impact in the performance is to define a second accumulator. Therefore, an accumulator is used for the aligned additions, i.e. $k{\cdot}2^{32}$, and another accumulator is used for the not aligned additions, i.e. $k \cdot 2^{32} + 2^{16}$. Thereby, the realign is only required at the end, when both accumulators are combined.

The proposed solution presents the inconvenient of requiring a high number of registers to store the second accumulator. In the particular case, when the numbers have a length equal to 160 bits, the accumulator needs to be equal to 320 bits, in order to store the result from 160x160. Therefore, it is required 10 registers for a single accumulator, and consequently 20 registers for the two required accumulators. In addition, one of the operands needs to be also stored into the registers, i.e. 160 bits stored in 10 registers, 16 bits per register, note that it is stored only 16 bits per register even when it is feasible to store 32 bits, since the previously described limitations by the hardware multiplication. In summary, it is required 30 registers to keep into the registers the accumulators and one of the operands, but it is not feasible because additional registers for the temporal are required.

But, a solution is feasible to maintain both accumulators into the registers and at the same time the additionally required registers thanks to the features from the shifting primes.

Shifting primes allows to carry out the reduction modulo p, while it is added the partial result to the accumulator. Thereby, the result is 160 bits (when p is a 160 bits number). This allows to keep the two accumulators of 160 bits, instead of the previously mentioned 2 accumulators of 320 bits. Therefore, only 5 registers are required for each accumulator, i.e., 10 registers for both accumulators, what is feasible.

Let two accumulators A and B, $A = \sum_{i=0}^{5} A_i 2^{32 \cdot i}$, $B = \sum_{i=0}^{5} B_i 2^{32 \cdot i}$. A is used for the aligned operations and B for the not aligned operations. Let the previously mentioned operands x and y. The first multiplication iteration is to multiplicate the first operand by $x9$, and after add it to the adequate accumulator such as follows:



Then, it is established the operand x into the registers r13 ,..., r20, the accumulator A in r22 ,..., r26 and the accumulator B in r27 ,..., r31. The assembler

code required to carry out this partial multiplication is composed by 10 l.mul, where each one requires 3 cycles.

```
l.mul   r31,r20,r3          l.mul   r24,r17,r3
l.mul   r26,r21,r3          l.mul   r28,r14,r3
l.mul   r30,r18,r3          l.mul   r23,r15,r3
l.mul   r25,r19,r3          l.mul   r27,r12,r3
l.mul   r29,r16,r3          l.mul   r22,r13,r3
```

The next step is to multiplicate by $x8$, and shift 16 bits the result. Note, that it should be required to shift 16 bits the accumulator, but instead of that operations, the proposed solution stores this partial result in the accumulator B. The global effect is a 16 bits in the accumulator at the end. For the next multiplication, it is required to shift the accumulator A by a block, i.e. 32 bits. For this operations, it is taking into account that $p = \texttt{0xc800} \cdot \texttt{0x10000}^9 - 1$ and consequently $1 \equiv \texttt{0xc800} \cdot \texttt{0x10000}^9$ modulo p, therefore it is equal to:

$$A = A_0 \cdot \texttt{0x10000}^8 + A_1 \cdot \texttt{0x10000}^6 + A_2 \cdot \texttt{0x10000}^4 + A_3 \cdot \texttt{0x10000}^2 + A_4 =$$

$$A_0 \cdot \texttt{0x10000}^8 + A_1 \cdot \texttt{0x10000}^6 + A_2 \cdot \texttt{0x10000}^4 + A_3 \cdot \texttt{0x10000}^2 + A_4 \texttt{0xc800} \cdot \texttt{0x10000}^9$$

It can be splitted into two block of 16 bits, then $A_4 = A_4^H \cdot \texttt{0x10000} + A_4^L$, and it can be considered:

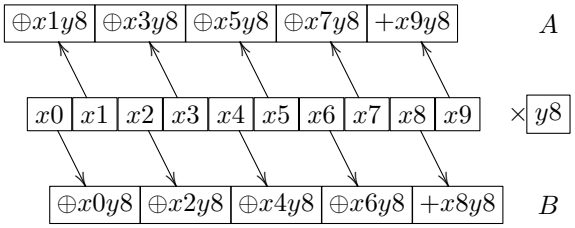
$$A = (\texttt{0xc800} \cdot A_4^H \cdot \texttt{0x10000}^8 + A_0 \cdot \texttt{0x10000}^6 + A_1 \cdot \texttt{0x10000}^4$$
$$+ A_2 \cdot \texttt{0x10000}^2 + A_3) \cdot \texttt{0x10000}^2 + \texttt{0xc800} \cdot A_4^L \cdot \texttt{0x10000}^9$$

The value $\texttt{0xc800} \cdot A_4^L \cdot \texttt{0x10000}^9$ is moved to B, since now it has the adequate alignment to be combined with the other accumulator. Then, the accumulator A can be shifted 32 bits with only change the role of the register. The change of the role for the registers does not require any explicit instruction, it is only a programming issue, which can be adapted when the loop from the multiplication is unrolled. Therefore, it is unrolled the 10 iterations of the loop (one iteration for each 16 bits from the 160 bits of the operand).

Therefore, the registers rotation is directly programmed in the code. Note that the register r6 has the value `0xc800` during all the operation. The code is as follows:

```
l.andi  r8,r31,0xffff       l.add   r26,r26,r8
l.mul   r8,r8,r6            l.addc  r8,r7,r0
l.srli  r31,r31,16          l.slli  r8,r8,16
l.mul   r31,r31,r6          l.add   r31,r31,r8
```

After the change is carried out in the accumulator, then it is carried out the multiplication with the block x_8, and it is added to the appropriated accumulator. The scheme is as follows (where + represent the addition operation, and $\oplus$ the addition with carry):



Finally, in terms of assembly code is:

```
l.mul   r7,r20,r3          l.mul   r8,r21,r3
l.add   r25,r25,r7         l.add   r31,r31,r8
l.mul   r7,r18,r3          l.mul   r8,r19,r3
l.addc  r24,r24,r7         l.addc  r30,r30,r8
l.mul   r7,r16,r3          l.mul   r8,r17,r3
l.addc  r23,r23,r7         l.addc  r29,r29,r8
l.mul   r7,r14,r3          l.mul   r8,r15,r3
l.addc  r22,r22,r7         l.addc  r28,r28,r8
l.mul   r7,r12,r3          l.mul   r8,r13,r3
l.addc  r26,r26,r7         l.addc  r27,r27,r8
l.addc  r7,r0,r0
```

The last carry values from both accumulators needs to be added. The carry of A is shifted 16 bits and added to the most significant part of B. The final carry from this operation is added to the carry of B.

```
l.addc  r8,r0,r0           l.add   r26,r26,r8
l.slli  r8,r8,16           l.addc  r7,r7,r0
```

Now, the role of the accumulators A and B are interchanged, with a shifting of 32 bits in A. The pending carry is added to the most significant part of A.

The presented process needs to be repeated 9 times until that it is completed all the blocks of the operand, i.e. y_i. At the end, all the accumulators are added in order to get the final result.

5 Results and evaluation

The evaluation is focused on the multiplication modulo p, since it is the most critical part in the ECC algorithms. There is a very rich literature about how to implement ECC primitives by using the basic modular arithmetic. For this purpose, it needs to be fixed a curve and implement the point arithmetic. A wide variety of curves, point representations and formulas for point addition and point doubling can be found in [2].

For the prime $p = 200 \cdot 256^{19} - 1$ we have chosen the Weierstrass curve $y^2 = x^3 - 3x - 251$. The number of points of this curve is $p + 1257662074940094999762760$,

that is a prime number. We have chosen the parameter -3 to use the formulas for point addition and point doubling in Jacobian coordinates given in [2].

The time that we have considered as a reference is the time required for a key generation. This requires the selection of a random number s_K (the private key) and the computation of the scalar multiplication $[s_k]G$ where G is a generator of the group of points in the curve. The generator has been set up with the following coordinates: $x_G =$ `0x9866708fe3845ce1d4c1c78e765c4b3ea99538ee` and $y_G =$ `0x58f3926e015460e5c7353e56b03dd17968bfa328`

The time required for the scalar multiplication is usually computed in terms of the time **M** required for a single modular multiplications. A standard reference is [15] that gives 1610**M** for 160-bit scalar multiplication. This result requires some precomputations and considers that computing a square is a bit faster than a standard multiplication, 0.8**M**. We have made a rather optimized multiplication with a code that requires around 2Kb. To have another function for squaring would require more or less the same and all the other precomputations could increase the size of the program too much. We have used an implementation that requires 2100**M** (1245 multiplications and 855 squares).

Standard literature ignores the time required for other operations different from modular multiplication. We have computed in our case that the key generation spends 83.13% of the time doing modular multiplications, this is a big percentage, but not all the time. Of course, our biggest efforts have been done in the optimization of this operation.

The following table presents the real time required for the key generation, one single modular multiplication, and finally for 2100 of them.

Key Generation	140.37 ms
Single multiplication	54.9 μs
2100 multiplications	115.29 ms
rest	25.08 ms

Since the CPU clock from the JN5139 is equal to 16MHz, 54.9 μs are 878 clock cycles in real time. The number of cycles for reading the code is around 750 cycles, this is the theoretical number of cycles. But, the real time is a little bit higher than the theoretical time because the external interruptions, cache failures, and the pipeline could require some extra cycles to execute the code.

6 Conclusions and Future Work

The first conclusion is that the multiplication algorithm presents a high dependence to the CPU architecture where it will be evaluated. From our experience, we have experimented with the MSP430 architecture, which is based on a 16 bits microcontroller, without support for hardware multiplication, 16 registers and a limited set of instructions. For this work, it has been evaluated an architecture

with higher capabilities, specifically, an architecture based on OpenRISC with 32 bits operations, support for hardware multiplication, 32 registers with a very low cost per instruction, and an extended set of instructions. These huge differences make very difficult to compare among different implementations, at least that they have tested over the same architecture and exploiting the same features. For example, it can be found an implementation of ECC for MSP430 and JN5139 in [13] which presents a very low performance, since it is implemented over the WiseBed Operating System. Therefore, even when they are using the same architecture the results are highly different because they are not being able to exploit the main benefits from the architecture.

The second conclusion is that shifting primes have demonstrated be a very useful primes, which are offering a very interesting set of properties in order to optimize the implementations for constrained devices. First, it was optimized for the MSP430 CPU thanks to its low quantity of bits set to 1, which simplifies some iterations from the multiplication when it is implemented through additions and shifts as a consequence that this was not supported supported by the hardware. Second, it has been also presented how to exploit the shifting primes for architectures, such as the JN5139 based on OpenRISC, which are supporting hardware multiplication. The optimization for the JN5139 has been focused on its suitability for the reduction modulo p, while it is added the partial result to the accumulator in order to make feasible the exploitation of the hardware multiplication over the available registers.

Finally, it needs to be considered hybrid scenarios, since in the different use cases from the Internet of Things, it will be common to find a same solution with multiple CPUs in the sensors, actuators and controllers. For example, it could be found a JN5139 module in the controller, since this has a higher memory and processing capabilities to manage multiple nodes requests and maintenance. However, the most common CPU for the sensors and actuators will be the MSP430, since this presents a lower cost but yet enough capabilities for their required functionality. Therefore, it is very relevant to have this kind of primes and implementations such as the presented in this work, which is feasible for devices with different capabilities. For that reason, our future work will be focused on demonstrate a scenario where MSP430 and JN5139 are integrated into the same solution, and both implement high level security algorithms such as DTLS for CoAP, using both of them certificates built with shifting primes-based keys. making thereby them feasible to interoperate and exploit the described optimization.

Acknowledgment

This work has been carried out by the excellence research group "Intelligent Systems and Telematics" granted from the Foundation Seneca (04552/GERM/06). The authors would like also thanks to the Spanish Ministry of Science and Education with the FPU program grant (AP2009-3981), the Ministry of Science and Innovation, through the Walkie-Talkie project (TIN2011-27543-C03-02), the

STREP European Projects "Universal Integration of the Internet of Things through an IPv6-based Service Oriented Architecture enabling heterogeneous components interoperability (IoT6)" from the FP7 with the grant agreement no: 288445, "IPv6 ITS Station Stack (ITSSv6)" from the FP7 with the grant agreement no: 210519, and the GEN6 EU Project.

References

1. *OpenRISC 1200 IP Core Specification (Preliminary Draft)*, 2012. v0.13.
2. D. J. Bernstein and T. Lange. Explicit-formulas database. `http://hyperelliptic.org/EFD`.
3. Leandro Marin, Antonio J. Jara, and Antonio F. Gómez-Skarmeta. Shifting primes: Optimizing elliptic curve cryptography for 16-bit devices without hardware multiplier. *Preprint*.
4. Leandro Marin, Antonio J. Jara, and Antonio F. Gómez-Skarmeta. Shifting primes: Extension of pseudo-mersenne primes to optimize ecc for msp430-based future internet of things devices. In A Min Tjoa, Gerald Quirchmayr, Ilsun You, and Lida Xu, editors, *ARES*, volume 6908 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2011.
5. Leandro Marin, Antonio J. Jara, and Antonio F. Gómez-Skarmeta. Shifting primes: Optimizing elliptic curve cryptography for smart things. In Ilsun You, Leonard Barolli, Antonio Gentile, Hae-Duck Joshua Jeong, Marek R. Ogiela, and Fatos Xhafa, editors, *IMIS*, pages 793–798. IEEE, 2012.
6. opencores.org. *OpenRISC 1000, Architecture Manual*, 2011. Rev. 2011-draft4.
7. Davies, J.H.: MSP430 Microcontroller Basics, 9780080558554, Elsevier Science (2008)
8. Castro, M., Jara, A.J., Skarmeta, A.: Architecture for Improving Terrestrial Logistics Based on the Web of Things. Sensors (2012), 12, 6538–6575.
9. Hoffman, L.: GreenWave Reality Announces Partnership with NXP, GreenWave Reality, http://www.greenwavereality.com/greenwave-reality-announces-partnership-with-nxp-semiconductors/ (2011)
10. Shelby, Z., K. Hartk, C. Borman, B. Fran: Constrained Application Protocol (CoAP), Internet-Draft. Internet Engineering Task Force (IETF) (2012)
11. Jara, A.J., Zamora, M.A., Skarmeta, A.F.: GLoWBAL IPv6: An adaptive and transparent IPv6 integration in the Internet of Things, Mobile Information Systems, Vol. 8, N. 3, (2012) 177–197.
12. Rescorla, E., Modadugu, N.: RFC6347 - Datagram Transport Layer Security Version 1.2, Internet Engineering Task Force (IETF), ISSN: 2070-1721 (2012)
13. Chatzigiannakis, I., Pyrgelis, A., Spirakis, P.G., Stamatiou, Y.C.: Elliptic Curve Based Zero Knowledge Proofs and their Applicability on Resource Constrained Devices, Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on, doi: 10.1109/MASS.2011.77 (2011), 715–720
14. Imai, H., Zheng, Y.: A practical implementation of elliptic curve cryptosystems over GF(p) on a 16-bit microcomputer, Lecture Notes in Computer Science, Springer Berlin Heidelberg, doi: 10.1007/BFb0054024, Vol. 1431 (1998), 182–194
15. H. Cohen, A. Miyaji, T. Ono. *Efficient elliptic curve exponentiation using mixed coordinates*, in Lecture Notes in Computer Science, 1514, Springer-Verlag, Berlin, pp. 5165, 1998.