Lightweight Structures of Big Numbers for Cryptographic Primitives in Limited Devices

Radek Fujdiak, Petr Mlynek
Peoples Friendship University of Russia (RUDN University)
6 Miklukho-Maklaya St, Moscow, 117198, Russian Federation
Brno University of Technology (BUT University)
Technicka 12, Brno, 61600, Czech Republic
{fujdiak, mlynek}@feec.vutbr.cz

Sergey Bezzateev
ITMO University
Kronverkskiy pr. 49, Saint Petersburg, 197101, Russia
bsv@aanet.ru

Romina Muka
Norwegian University of Science and Technology
A Building A108 Teknologivegen 22, Gjovik, 2815, Norway
rominamu@stud.ntnu.no

Jan Slacik, Jiri Misurec, Ondrej Raso
Brno University of Technology
Tecnicka 12, Brno, 61600, Czech Republic
{misurec,slacik,raso}@feec.vutbr.cz

***Abstract*—The new technological approaches bring us into the digital era, where data security is a part of our everyday lives. Nowadays cryptographic algorithms, which are also recommended by international security standards, are often developed for non-limited devices and they are not suitable for limited environments. This paper deals with a lightweight solution for the structure of big numbers, which should help with ordinary recommended cryptographic algorithms in limited devices. We introduce our lightweight structure and elementary algebra for cryptographic primitives based on non-limited OpenSSL library. Last but not least, we provide experimental measurements and verification on the real scenarios.**

***Index Terms*—Big Numbers, Cryptographic Primitives, Limited Devices, Elementary Algebra.**

I. Introduction

We are living in the digital era [1], where the security and cryptography is a part of our everyday lives. The cryptosystems are developed to ensure basic cryptographic function as i.e. Identification, Authentication, Authorization, Confidentiality, Integrity, Non-repudiation, Availability. Nowadays, most of the current cryptosystems are using the cryptographic primitives as i.e. cryptographic algorithms, hash functions, random generators and others. The mentioned keys are one of the most important parameters as they are determining the functional output of cryptographic algorithms. There are many international standards for these cryptographic primitives such as ENCRYPT, NIST, ANSSI, IAD-NSA, RFC, BSI and others [2]. These standards recommend among others the key length for several cryptographic algorithms (see Tab I, SM - Symmetric Cipher, FM - Factoring Modulus, DL - Discrete Logarithm, EC - Elliptic Curve).

The publication was prepared with the support of the RUDN University Program 5-100. The research described in this article was financed by National Sustainability Program under grant LO1401 and the Ministry of the Interior of the Czech Republic under grant no. VI20172019057. For the research, the infrastructure of the SIX Centre was used.

TABLE I
Recommendation for key lengths in cryptographic algorithms for 2020 [2].

SM [b]	**FM** [b]	**DL** [b]	**EC** [b]
82–256	1472–3072	151–250	161–384

In the case of cryptographic algorithms, there are needs for computing over a finite set of integers $\mathbb{Z}_m$, where m is a big integer representing the set size. The final set of integers $\mathbb{Z}_m$ forms from a set of integers $\mathbb{Z}$ by cumulating numbers with same remainder m. The remainder after division a by m (a/m) is referred as $a \bmod m$. Arithmetic operations made on the final set $\mathbb{Z}_m$ are referred as a modular arithmetic. An example of such an operation i.e. addition of two numbers a and b in the set $\mathbb{Z}_m$, is $(a+b) \bmod m$. As mentioned, m will be considered to be size of 128 or 256 bits.

Considering the implementation of a cryptosystem on the computational system platforms with limited physical resources (limited from the point of memory, performance, etc.), there are no possibilities for using the elementary mathematical operations directly [3]. It is always necessary to divide these large numbers m into smaller blocks with a size of 16 or 32 bits and process them separately. Elementary operations such as addition or subtraction are required for solving a single instruction, but for big numbers, a more efficient algorithm is needed.

This article deals with real implementation of described complex algorithms on a limited microcontroller MSP430. The main aim of this paper is to present a possibility for implementation of a known, but computationally difficult methods, on a limited device. We introduce effective implementation for big number structures and elementary algebraic operations (Sec. II), followed by a description of our open lightweight library derived from OpenSSL functions (Sec. III). Further, we provide experimental measurements of this implementation (Sec. IV) with verification in the real environment and discussion

of results (Sec. V). Last but not least, we summarised our approach and contribution (Sec. VI).

II. Effective structures for Big Numbers

This section describes the representation of big numbers, elementary algebraic operations over big numbers with a clear introduction to the implementation process. The presented algorithms were used for the implementation and they were derived from algorithms mentioned in [4]. The bold style is used for the big numbers.

A. Big Number Representation

The position notation is one of the possible ways how to deal with the representation of big numbers. This system is characterised by its base. This base will be referred as b (base/radix). The most common position notation is decimal ($b = 10$). So, if we have $p = 1234_{b=10}$ then we can refer p as:

$$p = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0. \quad (1)$$

Very similar it will be also for binary number $p = 1011_{b=2}$:

$$p = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0. \quad (2)$$

If we generalized equations 1 and 2 then we can refer p as:

$$p = \sum_{i=0}^{n-1} a_i b^i = a_{n-1} b^{n-1} + a_{n-2} b^{n-2} + ... + a_0 b^0, \quad (3)$$

where b is radix ($b \in \mathbb{N}$), a_n are digits ($a \in \{0, 1, ..., b-1\}$), and n is length of number p. Further, we will define the least significant digit a_0 and most significant digit a_{n-1}. If $b = 2$ then a_0 is referred as least significant bit (LSB) and a_{n-1} most significant bit (MSB).

B. Elementary arithmetic with Big Numbers

1) Addition and Subtraction Function: Addition and subtraction are the very elementary operations. We have two big numbers x and y with same length n, their addition $w = x + y$ is described below by Algorithm 1 (addition) and their subtraction $w = x - y$ in Algorithm 2. The is a carry bit in case of register overflow. In the case of $n_x \neq n_y$, we need to complete the smaller one with a corresponding number of zeros so the length would be equal ($n_x = n_y$). Last but not least, each element of the big number is marked by index (0 refers the LSB).

Algorithm 1 Big number addition

Require: positive integers $\mathbf{x}$, $\mathbf{y}$ of n length with base b
Ensure: $\mathbf{w} = \mathbf{x} + \mathbf{y}$, where $\mathbf{w} = (w_n, w_{n-1}, ..., w_0)$
1: $c \leftarrow 0$
2: **for** $(i \leftarrow 0)$ **to** $n-1$ **do**
3: $w_i \leftarrow (x_i + y_i + c) \bmod b$
4: **if** $((x_i + y_i + c) < b)$ **then**
5: $c \leftarrow 0$
6: **else**
7: $c \leftarrow 1$
8: **end if**
9: $w_n \leftarrow c$
10: **end for**
11: **return w**

Algorithm 2 Big number subtraction

Require: positive integers $\mathbf{x}$, $\mathbf{y}$ of n length with base b, $\mathbf{x} \geq \mathbf{y}$
Ensure: $\mathbf{w} = \mathbf{x} - \mathbf{y}$, where $\mathbf{w} = (w_n, w_{n-1}, ..., w_0)$
1: $c \leftarrow 0$
2: **for** $(i \leftarrow 0)$ **to** $n-1$ **do**
3: $w_i \leftarrow (x_i - y_i + c) \bmod b$
4: **if** $((x_i - y_i + c) \geq b)$ **then**
5: $c \leftarrow 0$
6: **else**
7: $c \leftarrow -1$
8: **end if**
9: $w_n \leftarrow c$
10: **end for**
11: **return w**

2) Multiplication Function: Following algorithm 3 describes the ordinary method for big number multiplication $w = x \cdot y$. There are many types of this algorithm, but we used for our purpose (limited device application) the simplest one. This algorithm describes multiplication of two numbers of base b: x with length n and y with length t.

Algorithm 3 Big number multiplication

Require: positive integers $\mathbf{x}$ of n length and $\mathbf{y}$ of t length, $\mathbf{x}$, $\mathbf{y}$ of b base
Ensure: $\mathbf{w} = \mathbf{x} \cdot \mathbf{y}$, where $\mathbf{w} = (w_{n+t+1}, w_{n+t}, ..., w_0)$
1: **for** $(i \leftarrow 0)$ **to** $(n+t+1)$ **do**
2: $w_i \leftarrow 0$
3: **end for**
4: **for** $(i \leftarrow 0)$ **to** $(n-1)$ **do**
5: $c \leftarrow 0$
6: **for** $(j \leftarrow 0)$ **to** $(n-1)$ **do**
7: $(u, v) = w_{i+j} + x_j \cdot y_i + c$, for $w_{i+j} \leftarrow v$ and $c \leftarrow u$
8: **end for**
9: $w_{i+n+1} \leftarrow u$
10: **end for**
11: **return w**

3) Divide Function: Division is the most demanding mathematical operation in case of microcontrollers. The algorithm 4 describes division $q = \lfloor x/y \rfloor$ with remainder r and base b.

Algorithm 4 Big number division

Require: positive integers $\mathbf{x} = (x_n, x_{n-1}, ..., x_0)$, $\mathbf{y} = (y_t, y_{t-1}, ..., y_0)$ with b base, for $n \geq t \geq 1$ and $y_t \neq 0$
Ensure: $\mathbf{q} = \lfloor \mathbf{x}/\mathbf{y} \rfloor$ and remainder $\mathbf{r}$, where $\mathbf{q} = (q_{n-t}, q_{n-t-1}, ..., q_0)_b$, $\mathbf{r} = (r_t, r_{t-1}, ..., r_0)_b$, $\mathbf{x} = \mathbf{q}\mathbf{y} + \mathbf{r}$ and $0 \leq \mathbf{r} \leq \mathbf{y}$
1: **for** $\{i \leftarrow 0)$ **to** $(n-t)$ **do**
2: $q_i \leftarrow 0$
3: **end for**
4: **while** $(\mathbf{x} \geq \mathbf{y} b^{n-t})$ **do**
5: $q_{n-t} \leftarrow q_{n-t} + 1$ and $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{y} b^{n-t}$
6: **end while**
7: **for** $(i \leftarrow n)$ **to** $(t+1)$ **do**
8: **if** $(x_i = y_t)$ **then**
9: $q_{i-t-1} \leftarrow b - 1$
10: **else**
11: $q_{i-t-1} \leftarrow \lfloor (x_i b + x_{i-1}) / y_t \rfloor$
12: **end if**
13: **while** $\left(q_{i-t-1}(y_t b + y_{t-1}) > \left(x_i b^2 + x_{i-1} b x_{i-2}\right)\right)$ **do**
14: $q_{i-t-1} \leftarrow q_{i-t-1} - 1$
15: **end while**
16: $\mathbf{x} \leftarrow \mathbf{x} - q_{i-t-1} \mathbf{y} b^{i-t-1}$
17: **if** $(\mathbf{x} < 0)$ **then**
18: $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{y} b^{i-t-1}$ and $q_{i-t-1} \leftarrow q_{i-t-1} - 1$
19: **end if**
20: **end for**
21: $\mathbf{r} \leftarrow \mathbf{x}$
22: **return** $(\mathbf{q}, \mathbf{r})$

C. Modular Arithmetic

Section 2.2 describes the elementary algebraic algorithms as addition, subtraction, multiplication and division with big numbers. Section 2.3 continues with describing more complex algorithms with big numbers, known as modular algebra for set Z_m. For the elementary addition and subtraction, we can use algorithms 1 and 2 with adding the step of subtracting m with $(x+y) \neq m$ and $x \neq y$. Further, we will describe in this section also more complex methods of modular algebra.

1) Basic Modular Multiplication: The modular multiplication might be also derived from the previous algorithm 3 for elementary multiplication in algorithm 5.

Algorithm 5 Basic modular multiplication for big numbers

Require: positive integers **x**, **y** and modulo **m**
Ensure: $\mathbf{x} \cdot \mathbf{y} \bmod \mathbf{m}$
1: $\mathbf{i} = \mathbf{x} \cdot \mathbf{y}$ by algorithm 3
2: $\mathbf{r} = \mathbf{i} / \mathbf{m}$ by algorithm 4
3: **return r**

2) Montgomery Reduction: Montgomery reduction is a technique allowing effective implementation of modular multiplication. There are big positive integers m, R and T, where $R > m$, $f_{gcd}(m,R) = 1$ (greatest common divisor function) and $0 \leq T < mR$. The main point is to exchange the divide function (used for obtaining remainder) with simple shifting as we can see also in algorithm 6 below. Bit shifting are the fastest operation on most hardware platforms and are much less demanding than divide function.

Algorithm 6 Montgomery reduction for big numbers

Require: positive integers $\mathbf{m} = (m_{n-1}, m_n, ..., m_0)$ with b base, where $gcd(\mathbf{m}, b) = 1$, $\mathbf{R} = b^n$, $\mathbf{m'} = -\mathbf{m}^{-1} \bmod b$ and $\mathbf{T} = (t_{2n-1}, t_{2n}, ..., t_0) < \mathbf{mR}$
Ensure: $\mathbf{A} = \mathbf{TR}^{-1} \bmod \mathbf{m}$
1: $\mathbf{A} \leftarrow \mathbf{T}$, for $\mathbf{A} = (a_{2n-1}, a_{2n}, ..., a_0)$
2: **for** $(i \leftarrow 0)$ **to** $(n-1)$ **do**
3: $\quad u_i \leftarrow a_1 \mathbf{m'} \bmod b$
4: $\quad \mathbf{A} \leftarrow \mathbf{A} + u_i \mathbf{m} b^i$
5: **end for**
6: $\mathbf{A} \leftarrow \mathbf{A} / b^n$
7: **if** $(\mathbf{A} \geq \mathbf{m})$ **then**
8: $\quad \mathbf{A} \leftarrow \mathbf{A} - \mathbf{m}$
9: **end if**
10: **return A**

3) Montgomery Multiplication: Montgomery multiplication is bonded with Montgomery reduction. It is also necessary to choose the right R that we can use the binary shifting operations. Further, this algorithm is designed for cross-phase multiplication and reduction, which lead us to much better performance and memory usage. This algorithm needs two multiplication operations of big number multiplication and two $2n$ operations. These steps are repeated n-times. This algorithm needs $2n(n+1)$ simple (basic) multiplication operations. However, if the R is well chosen then all division operations can be exchanged by simple shifting, which gives us much of performance. Last but not least, thanks to the nature of the Montgomery multiplication, it is much more performance demanding than simple multiplication. The main advantages and performance speed are when we are using Montgomery multiplication over modulo (see Algorithm 7).

Algorithm 7 Montgomery multiplication for big numbers

Require: positive integers $\mathbf{m} = (m_{n-1}, m_n, ..., m_0)$, $\mathbf{x} = (x_{n-1}, x_n, ..., x_0)$, $\mathbf{y} = (y_{n-1}, y_n, ..., y_0)$ with base b, for $0 \leq \mathbf{x}, \mathbf{y} < \mathbf{m}$, $\mathbf{R} = b^n$, $gcd(\mathbf{m}, b) = 1$ and $\mathbf{m'} = \mathbf{m}^{-1} \bmod b$
Ensure: $\mathbf{A} = \mathbf{xyR}^{-1} \bmod \mathbf{m}$
1: $\mathbf{A} \leftarrow 0$, for $\mathbf{A} = (a_n, a_{n-1}, ..., a_0)$
2: **for** $(i \leftarrow 0)$ **to** $(n-1)$ **do**
3: $\quad u_i \leftarrow (a_0 + x_i x_0) \mathbf{m'} \bmod b$
4: $\quad A \leftarrow \mathbf{A} + x_i \mathbf{y} + u_i \mathbf{m} / b$
5: **end for**
6: **if** $(\mathbf{A} \geq \mathbf{m})$ **then**
7: $\quad \mathbf{A} \leftarrow \mathbf{A} - m$
8: **end if**
9: **return A**

III. Lightweight OpenSSL Structures

On the side of the computational unit, the position notation from 3 is used for the big number representation. Further, our lightweight solution is derived from OpenSSL library [5] and algorithms described in Section 2. Significant part of the OpenSSL code was optimized and modified for better suitability for low-power microcontrollers: the data types were standardized, dynamical allocated variables were modified, memory management was changed to the more suitable form for low-power microcontrollers, error statements were added, hardware depended variables were excluded and more[1]. We keep the names of the variables, structured and functions same as in OpenSSL library for better clarity of the solution. Our structure for the big numbers is followed:

```
 Struct bignum_st {
BN_ULONG *d, int top; int dmax; int neg;
int flags; } BIGNUM;
```

Variable d is a pointer for the specific position in memory, where the big number is saved; the top is a position of MSB of the field d; and the $dmax$ is a maximum field size of d. Further, the neg is a flag whether the number is negative and the $flags$ are other auxiliary flags. There are basic addition (*BN_add*, #1) and subtraction function (*BN_sub*, #2) described by algorithmus 1 and 2. Further, we implemented also simple multiplication (*BN_mul*, #8) and division methods (*BN_mul*, #7) solved by simple binary shifting and described by algorithm 3 and 4. However, we modified these function for their modular variant. For the modular operations were created BN_mod function ensuring $r = a \pmod m$ (#6):

```
 int BN_mod {
BIGNUM *r, const BIGNUM *a,
const BIGNUM *m, BN_CTX *ctx };
```

The modular addition function *BN_mod_add* (#3) compute $a+b \pmod m$. The result r must fulfil $(r \neq a) \wedge (r \neq b) \wedge (r \neq m)$. It is a simple modular addition function, derived from algorithm 1 (2) and modified by methods from Section 2.3 (if error occurs, function returns 0).

[1]Our library might be downloaded from the URL source page: http://www.utko.feec.vutbr.cz/raso/libMSP430_aritmetika.html

```
 int BN_mod_add {
BIGNUM *r, const BIGNUM *a, const BIGNUM *b,
const BIGNUM *m, BN_CTX *ctx };
```

Next function is a simple modular subtraction (#4) for operation $r = a - b \pmod{m}$), where r is a result (if error occurs, function returns 0). The a and b might have same values, but the object must differs. Further, the result r must fulfil $(r \neq a) \wedge (r \neq b) \wedge (r \neq m)$. The subtraction function has following syntax:

```
 int BN_mod_sub {
BIGNUM *r, const BIGNUM *a, const BIGNUM *b,
const BIGNUM *m, BN_CTX *ctx };
```

Following function *BN_mod_mul* (#5) serve for simple modular multiplication $r = a \cdot b \pmod{m}$ (Algorithm 3) and the result r must fulfil $(r \neq a) \wedge (r \neq b) \wedge (r \neq m)$.

```
 int BN_mod_mul {
BIGNUM *r, const BIGNUM *a, const BIGNUM *b,
const BIGNUM *m, BN_CTX *ctx };
```

However, there are also Montgomery functions provided - *BN_mod_mul_montgomery* (Montgomery multiplication #10, algorithm 5) and *BN_from_montgomery* (Montgomery reduction, algorithm 6). Montgomery modular multiplication with following syntax:

```
 int BN_mod_mul_montgomery {
BIGNUM *r, const BIGNUM *a, const BIGNUM *b,
BN_MONT_CTX *mont, BN_CTX *ctx };
```

Montgomery reduction with following syntax:

```
 int BN_from_montgomery {
BIGNUM *ret, const BIGNUM *a,
BN_MONT_CTX *mont, BN_CTX *ctx };
```

IV. Experimental Measurements

The limiting factor for real-implementation of cryptographic and communication algorithm is memory and performance. Software implementation must be always lightweight and sufficiently efficient. In this section, the experimental results from the measurements of chosen functions from designed library are presented. Continuous measurements with three big numbers a, b and m, where $m > a > b$ with the length of 128 and 256 bits were made. The variables a and b represents operands of tested functions and m is the size of modulo. The table II shows the summarized experimental results for memory and time consumption of measured algorithms. These algorithms are represented by implemented functions. The tested device was low-power microcontroller MSP430f5438A with 20 MHz CPU frequency and 256 kB memory.

The comparison of these results is in Fig. 1. The results clearly show growing requirements between 128 and 256 b. This growth is significant mostly for the mathematical operation of modular multiplication, function *BN_mod_mul* (more than 300 %). On the other hand, function *BN_mod_sub* shows fast computation and low requirements. This function is fast thanks to the fact that the result is smaller than the modulo (there is no division, with the division this function will be similar fast as *BN_mod_add*). Thanks to the binary shifting, the functions *BN_mod* and *BN_div* are also significantly faster than other functions. Function *BN_MONT_CTX_set* (it is a supplementary function for multiplicative inversion modulo to module m) is most demanding operation. However, there are needs for this function only if we are using *BN_mod_mul_montgomery*. Moreover, function *BN_mod_mul_montgomery* is very fast, but we need to consider the calling of slow supplementary function *BN_MONT_CTX_set*. This function is called always only once. Further, if we are using more than five multiplication then it is more efficient to use *BN_mod_mul_montgomery* (125 b numbers) else it is faster to use simpler functions as *BN_mod_mul*. The whole library takes only 12/256 kB of memory.

The presented results show that the library is sufficiently small and leaves a significant part of the memory for communication protocols and other necessary functions. The speed of designed functions (algorithms) is sufficient mostly for 128 b. However, the growing length is directly proportional to the speed performance, which mostly impacts the multiplication functions.

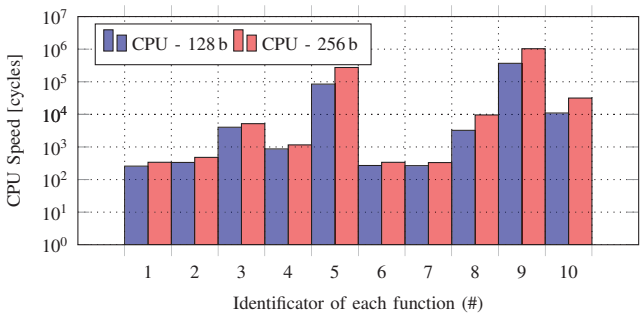

Fig. 1. Speed comparison of each big number primitive.

V. Final Verification of Designed Methods

Fig. 2 shows our real implementation of different cryptographic primitives. We used our lightweight structure to implement random number generator, elliptic curve cryptography (ECDH) algorithms and symmetric cypher AES (AES-128) [6]. These implementations were implemented in one functional solution and used for securing the communication in the real application of part of Smart Grid network, where the limited devices are used.

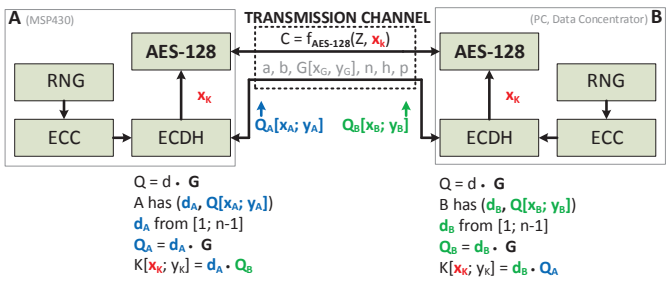

Fig. 2. Block diagram of implemented cryptographic protocol (source [6]).

TABLE II
THE TIME AND MEMORY REQUIREMENTS FOR EACH ALGEBRAIC FUNCTION OF OUR IMPLEMENTATION (# IS AN IDENTIFICATION OF EACH FUNCTION).

#	Function name	Memory [B]	CPU Cycles - 128 b [-]	CPU Cycles - 256 b [-]
Functions without any sub-structures				
1	BN_add	5 686	258	339
2	BN_sub	5 736	335	479
	All	5 856	-	-
Functions using BN_CTX sub-structure				
3	BN_mod_add	8 310	4 028	5 183
4	BN_mod_sub	8 310	869	1 157
5	BN_mod_mul	9 552	84 730	274 088
6	BN_mod	4 176	270	338
7	BN_div	4 204	264	332
8	BN_mul	6 748	3 237	9 603
	All	9 816	-	-
Functions using BN_CTX and BN_MONT_CTX sub-structure				
9	BN_MONT_CTX_set	11 104	367 994	1 033 268
10	BN_mod_mul_montgomery	8 948	11 000	31 641
	All	12 010	-	-

The design of introduced secure communication protocol was implemented into the GPRS communication unit MEg202.2 designed by company MEgA, plc. [7]. This device is using the TI ultra-low-power microcontroller MSP430F5438A. The secure protocol was also used in the Data Concentrator side. We verified our designed secure communication protocol in an real environment of CEZ Distribuce, a.s [8]. The communication chain is shown in Fig. 3. The MEg40+ Universal energy meter was installed in the Noviny transformer station, Velky Grunov area, the Czech Republic. The Data Concentrator was located in Brno, the Czech Republic. The communication distance was approximately 240 km. Detailed description of the results and validation in [9].

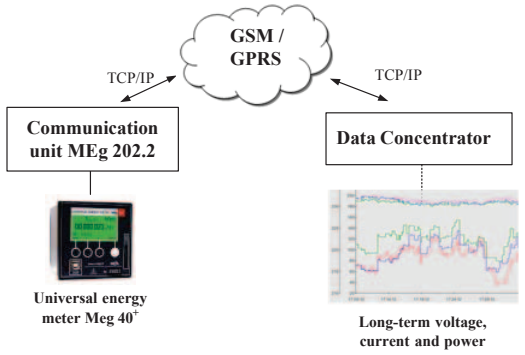

Fig. 3. The schematic of our validation measurement (source [9]).

VI. CONCLUSIONS

This paper presents the fundamental basis for the most cryptographic applications. We introduce the real implementation of main cryptographic primitives, basic algebraic structures and mathematical functions used for big number operations. Further, the algorithmization of these functions was shown together with a clear explanation of its needs. These algorithms were also implemented in our complex lightweight library, which was based on the common knowledge and derived from OpenSSL functions. For the experimental validation of this library, we provide tests of main algebraic functions used for mathematical operations with big numbers. These results show a significant dependence on the size of the big number mostly for the multiplication operations.

The main contribution of our implementation lies in the derivation of the OpenSSL function, which brings a possibility for dealing with big numbers on limited devices such as MSP430 microcontrollers. Moreover, the solution is based on the low-level functions, which provide an option for developers and scientist to build up their own cryptosystems (most of the current lightweight cryptographic libraries provide only the final cryptographic functions).

Last but not least, we show our robust implementation of several cryptographic primitives and algorithms together with our secure communication solution based on our big number lightweight library. This protocol together with the presented library was tested in the real environment (smart grid network) and shows the real usage and impact of our solution. Finally, the main topicality of this paper lies in the whole verified cryptographic chain, from the algebraic basis to the real application and verification.

REFERENCES

[1] N. Al-Falahy and O. Y. Alani, "Technologies for 5g networks: challenges and opportunities," *IT Professional*, vol. 19, no. 1, pp. 12–20, 2017.

[2] D. Giry, "Bluecrypt: Cryptographic key length recommendation," 2017.

[3] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *Privacy, Security and Trust (PST), 2015 13th Annual Conference on*, pp. 145–152, IEEE, 2015.

[4] M. T. Goodrich and R. Tamassia, *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2006.

[5] M. J. Cox, R. S. Engelschall, S. Henson, and B. Laurie, "The openssl cryptography and ssl/tls toolkit," 2017.

[6] R. Fujdiak *et al.*, "Efficiency evaluation of different types of cryptography curves on low-power devices," in *Ultra Modern Telecommunications and Control Systems and Workshops, 2015 7th International Congress on*, pp. 269–274, IEEE, 2015.

[7] MEgA, "Mega measuring power apparatus, plc.," 2017.

[8] CEZ, "Distribution power company," 2017.

[9] P. Mlynek *et al.*, "Design of secure communication in network with limited resources," in *Innovative Smart Grid Technologies Europe, 2013 4th IEEE/PES*, pp. 1–5, IEEE, 2013.