

# Implementation of Finite Field Arithmetic Operations for Polynomial and Normal Basis Representations

Mirza Maulana<sup>1</sup>, Wenny Franciska Senjaya<sup>1,3</sup>, Budi Rahardjo<sup>1</sup>, Intan Muchtadi-Alamsyah<sup>2\*</sup>, Marisa W. Paryasto<sup>1,4</sup>

<sup>1</sup>*School of Electrical Engineering and Informatics, Bandung Institute of Technology, Jalan Ganesha 10, Bandung 40132, Indonesia.*

<sup>2</sup>*Faculty of Mathematics and Natural Sciences, Bandung Institute of Technology, Jalan Ganesha 10, Bandung 40132, Indonesia.*

<sup>3</sup>*Faculty of Information Technology, Maranatha Christian University, Jl. Prof. drg. Suria Sumantri No.65 Bandung 40164, Indonesia.*

<sup>4</sup>*School of Economics and Business, Telkom University, Jalan Telekomunikasi Terusan Buah Batu, Bandung 40237, Indonesia.*

Received: 21 October 2014 / Accepted: 30 November 2014

## Abstract:

Elliptic Curve Cryptography is generally implemented over prime fields or binary fields. Arithmetic in binary fields can be classified according to the basis representation being used. Two of the most common basis used in binary fields are polynomial basis and normal basis. The optimal normal basis is especially known to be more efficient than polynomial basis because the inversion can be achieved by performing repeated multiplication using the method of Itoh and Tsujii, and squaring can be executed by performing only one cyclic shift operation. In previous research we have built algorithms and implementations on basis conversion between polynomial basis and normal basis. In this paper we will present implementation of arithmetic operation algorithms on both polynomial basis and optimal normal basis representation and the conversion method between them. We will also give comparison of time and space between implementation by using optimal basis representation with conversion and without conversion.

**Key words:** *Basis conversion, cryptography, elliptic curve cryptography, finite field, normal basis, optimal normal basis*

## Introduction

Elliptic Curve Cryptography (ECC) was first introduced by Neal Koblitz and Victor Miller [4,12]. ECC has several advantages compared to other cryptography method [5,13]: its arithmetic operations are specific and cannot be predicted, it offers smaller key length for the same security level compared to other method and its operations have many layers and combinations.

In ECC we use the set of points in some elliptic curve, and that set has a unique mathematical structure that allows us to define addition of two points and the result will be another point on the same curve. This gives advantage in ECC, since it would be difficult to find an origin point that is used to generate other new points [13].

Finite field has an important role in cryptographic algorithms [21,22]. ECC is one of the cryptosystem which based on finite fields. ECC in general is implemented over prime fields or binary fields. Arithmetic in binary fields can be classified according to basis used. Two of the most common basis used in binary fields are polynomial basis and normal basis. Some special cases such as trinomial basis, pentanomial basis and optimal normal basis are, in practice, used for the purpose of efficient operations. The optimal normal basis especially are known to be more efficient for hardware implementation than polynomial basis because the multiplication operation can be performed very

efficiently and inversion can be achieved by repeated multiplication typically using the method of Itoh and Tsujii, and squaring can be executed by only one cyclic shift operation. In previous research [8,9,10,14] we have built algorithms and implementations on basis conversion between polynomial basis and normal basis. We have also built composite field multiplier for Elliptic Curve Cryptography Implementation [15,16]. In this paper we will present implementation of arithmetic operation algorithms on both polynomial basis and optimal normal basis representation and the conversion method between them. We will also give comparison of time and space between implementation by using optimal basis representation with conversion and without conversion.

## Experimental

In this paper, two methods of arithmetic operations are used. The first method is the arithmetic operations purely in the optimal normal basis representation. The second method is the arithmetic operations in the polynomial basis representation. All methods are implemented in Python.

All arithmetic operations are adopted from Rosing [18]. We implement the basis conversion algorithms from Muchlis et. al [7]. The representation of polynomials is implemented as an integer whose binary digits are representing the coefficients of the polynomials.

\*Corresponding author: Intan Muchtadi-Alamsyah,  
E-mail: ntan@math.itb.ac.id

The objectives of these experiments are (1) to validate the implementations of both the first and the second method, and (2) to compare the efficiency between the first and the second method. The validation of the method is adopted from the basic properties of arithmetic operations: commutative, associative, and distributive, and also from the basic software test [1]. The efficiency of the algorithm is measured from the running time and the usage of the memory using the library provided by Python community.

#### *Optimal Normal Basis Arithmetics*

In this experiment, each type of Optimal Normal Basis (ONB) is implemented as an instance of ONB Class. Everytime the instance is initiated, a multiplication matrix is created based on the key length and the type. The matrix will be read when the multiplication operation is called.

Addition in ONB is implemented by bitwise XOR operation of all bits of the two operands. This operation is identical to Polynomial Basis addition.

Squaring of ONB is implemented by bitwise rotation to the most significant bit (MSB) side. Since the polynomial is implemented as an integer, the rotation will be to the left side and the left-most side will be the right-most side.

The multiplication operation is implemented by using the multiplication matrix which is created at the initiation of the instance. Both the matrix and the multiplication method are adopted from Rosing's method [18].

The inverse operation is adopted from the Itoh-Tsujii's method. The method is found based on the definition of Fermat's Theorem [3].

#### *Polynomial Basis Arithmetics*

In this experiment, the Polynomial Basis (PB) is implemented as an instance of PB Class. Everytime the instance is initiated, an irreducible polynomial is defined. In order to match it with the ONB, the irreducible polynomial is defined from certain formula and it depends on the type of ONB.

The most important operation in PB arithmetics is the reduction operation. In this experiment, the NIST fast reduction algorithm is used [11].

The squaring operation on PB representation uses the inserting 0 bit between each binary digits method. Since it doubles the degree of the polynomial, reduction is needed.

Multiplication in PB is implemented by using the Right-to-Left shift-and-add multiplication method [2]. The result of multiplication operation can have a greater

degree so a reduction process is also required. Inverse operation is implemented by using the Extended Euclidean Algorithm [2].

#### *Basis Conversion*

In this experiment, basis conversion is designed as a class and implemented as an instance of the class based on the key length and the type of ONB. Before a polynomial in ONB representation is converted into PB representation, the ONB representation must be converted into other representation PONB which has the same elements as PB. Therefore, there are two steps in converting ONB into PB and vice versa.

In the implementation, we use algorithms that are introduced by Muchlis et. al. [7].

## **Results and Discussion**

The results in this research are the implementations of arithmetic operation in finite fields, which are represented in polynomial basis (PB) and Optimal Normal Basis (ONB) and also PB-ONB Conversion Algorithms. The arithmetic operations which are implemented are squaring, multiplication, and inverse. The implementation uses programming language Python version 2.7.6. Each operation is implemented in the form of a class method. The maximum key length is 299 bits. List of irreducible binary polynomial taken from G. Seroussi [20].

#### *Implementation in ONB*

The same squaring operation is used for ONB Type I and II. We shift the bits of a polynomial representation to the left (MSB) and change the leftmost bit into the rightmost bits. The following is the implementation of squaring operation ONB in Python.

For multiplication, the construction requires 2 parameters: key length and type of ONB. Every time we construct an object in ONB representation, a multiplication matrix will be constructed in accordance with the input parameters. The ONB multiplication operations for Type I and Type II are the same. They differ only by the multiplication matrices.

```
def squ(self, a) :
    return self.rotl(a)
def rotl(self, n) :
    return ((n << 1) & ((1 << self.__numbits) - 1)) ^ (n >> (self.__numbits - 1))
```

**Figure 1.** Squaring in ONB.

The following is the implementation of multiplication operation ONB in Python.

```
def mul(self, a, b) :
    copya = a & ((1 << self.__numbits) - 1)
    copyb = b & ((1 << self.__numbits) - 1)
```

```

c = 0
zero_index = self.__lamda[0][0]
c=copyb&self.rotrn(copya, zero_index)
for j in range(1, self.__numbits):
    copyb = self.rotr(copyb)
    zero_index = self.__lamda[0][j]
    one_index = self.__lamda[1][j]
    c^=copyb&(self.rotrn(copya,zero_index)
        ^self.rotrn(copya, one_index))
    return c

```

**Figure 2.** Multiplication in ONB.

The following is the implementation of Itoh-Tsujii's method [3] for inverse in ONB by using Python.

```

def inv(self, a) :
    s = self.__lg2_m -1
    result = a & ((1 << self.__numbits) - 1)
    m = self.__numbits - 1
    while (s >= 0) :
        r = m >> s
        shift = result
        for rsht in range((r >> 1)) :
            shift = self.rotl(shift)
        temp = self.mul(result, shift)
        if (r & 1) :
            temp = self.rotl(temp)
        result = self.mul(temp, a)
        else :
            result = temp
        s -= 1
    result = self.rotl(result)
    return result

```

**Figure 3** Itoh-Tsujii's method.

#### *Implementation in PB*

Implementation of squaring operations is based on  $(a+b)^2 = a^2+b^2$ . Therefore its implementation is by shifting each bit with a nonzero coefficient to the left as much  $2i$  where  $i$  is the degree of the polynomial. The following the implementation of PB squaring operation:

```

def squ(self, A):
    B = 0
    i = 0
    while A != 0:
        if (A & 1) != 0:
            B = B ^ (1 << (2*i))
        A = A >> 1
        i = i+1
    return self.red(B)

```

**Figure 4.** Squaring in PB.

The following is the implementation of Right-to-Left shift-and-add method for multiplication in PB by using Python.

```

def righth_to_left(self, A, B):
    bitA=bin(A)[2:]
    if bitA[(self.countDegree(A))]=='1':
        C=B
    else:
        C=0
    i=self.countDegree(A)-1
    while i>=0:
        B=B<<1
        if self.countDegree(B)>=self.degree:
            B=self.binRed(B)
        if bitA[i]=='1':
            C^=B
        i-=1
    return C

```

**Figure 5.** Right-to-Left shift-and-add method.

The following is the implementation of extended euclidean algorithm for inverse operation in PB by using Python.

```

def extEuclidean(self, A):
    if A==0:
        print "Angka tidak boleh nol"
        return 0
    temp=0
    u=A
    v=int(self.bitIrredPol,2)
    g1=1
    g2=0
    while u!=1 :
        j=self.countDegree(u)-self.countDegree(v)
        if j<0 :
            temp=u
            u=v
            v=temp
            temp=g1
            g1=g2
            g2=temp
            j= j*(-1)
        u=self.add(u, (v<<j))
        g1=self.add(g1, (g2<<j))
    return g1

```

**Figure 6.** Extended Euclidean Algorithm.

#### *Implementation of Basis Conversions*

Implementation of the conversions requires digit integers as input parameters that represent polynomial coefficients. Output of the ONB-PONB method is the digit integer coefficients represent a PONB with maximum degree is  $m$  or key length for which the implementation is represented by numbits variables.

```

def onb2ponb(self, a) :
    ponb = 0
    if self.__type == 1:
        for i in range(self.__numbits):
            ponb ^= ((a >> i) & 1) << ((1 << i) %
self.__field_prime - 1)
        elif self.__type == 2:
            for i in range(self.__numbits):
                j = (1 << i) % self.__field_prime
                if j > self.__numbits :
                    ponb ^= ((a >> i) & 1) << ((-j %
self.__field_prime) - 1)
                else :
                    ponb ^= ((a >> i) & 1) << ((j %
self.__field_prime) - 1)
            return ponb

```

**Figure 7.** ONB to PONB conversion.

Implementation of the PONB-PB conversion requires digit integers as input parameters that represent PONB polynomial coefficients. Output of the PONB-ONB method is the digit integer coefficients represent a PB with maximum degree is  $m-1$ .

```

def ponb2pb(self, a) :
    kopia = a
    hasil = 0
    if self.__type == 1 : kopia = (kopia <<
1) & ((1 << self.__numbits) - 1)
    if (a >> (self.__numbits - 1)) == 1 :
        hasil = kopia ^ ((1 << self.__numbits) -
1)
    else :
        hasil = kopia
    elif self.__type == 2 :
        w, v, t = 2, 4, 0
        hasil = (kopia >> (self.__numbits - 1)) &
1 kopia^=(hasil & 1)*((1 << self.__numbits)-
1)
        for i in range(self.__numbits - 1) :
            hasil ^= ((kopia >> i) & 1) * w
            t = v
            v = ((v << 1) & ((1 << self.__numbits) -
1)) ^ w
            w = t
        return hasil

```

**Figure 8.** PONB to PB conversion.

Implementation of the PB-PONB conversion requires digit integers as input parameters that represent PB polynomial coefficients. This implementation requires a process of shifting bits to the right (LSB) with no change in the leftmost bit (MSB). The shifting process is implemented with a method srsh( ). Output of the PB-PONB method is the digit integer coefficients represent a PONB.

```

def pb2ponb(self, a) :
    hasil = 0
    kopia = a
    if self.__type == 1 :
        hasil = (kopia >> 1) ^ (a & 1) * ((1 <<
self.__numbits) - 1)
    elif self.__type == 2 :
        w = (1 << self.__numbits) - 1
        for i in range(self.__numbits) :
            hasil ^= ((kopia >> i) & 1) * w
            w = (w << 1) & ((1 << self.__numbits) -
1) ^ self.srsh(w)
        return hasil

def srsh(self, n) :return (n >> 1) ^ ((n
>> (self.__numbits - 1)) & 1) <<
(self.__numbits - 1))

```

**Figure 9.** PB to PONB conversion.

Implementation of the PONB-ONB conversion requires digit integers as input parameters that represent PONB polynomial coefficients. Output of the PONB-ONB method is the digit integer coefficients represent an ONB.

```

def ponb2onb(self, a) :
    onb = 0
    if self.__type == 1:
        for i in range(self.__numbits):
            idxPonb= (1 << i) % self.__field_prime -1
            if ((a >> idxPonb) & 1) :
                onb ^= 1 << i
        elif self.__type == 2:
            for i in range(self.__numbits):
                j = (1 << i) % self.__field_prime
                if j > self.__numbits :
                    idxPonb = (-j % self.__field_prime) - 1
                else :
                    idxPonb = (j % self.__field_prime) - 1
                if ((a >> idxPonb) & 1) :
                    onb ^= 1 << i

```

**Figure 10.** PONB to ONB conversion.

### Implementation Testing

In order to test the implementation, we use blackbox testing. The test is divided into two major parts, functional testing and performance testing. Functional testing is divided into three parts, testing for correct data, abuse data, and extreme data [17]. Several performance tests were conducted to measure the processing time and memory usage that are needed for arithmetic operations.

### Optimal Normal Basis Testing

Functional testing performed on all arithmetic operations. We compare the results between manual calculation, ONB with conversion, and ONB without conversion (see also [6]).

Time measurements performed with comparing operation time. We compare the operating time between

ONB with conversion and ONB without conversion. Time processing calculated in milliseconds. We compare memory usage by running memory compiler in command prompt. The measurement used 210 bits key length.

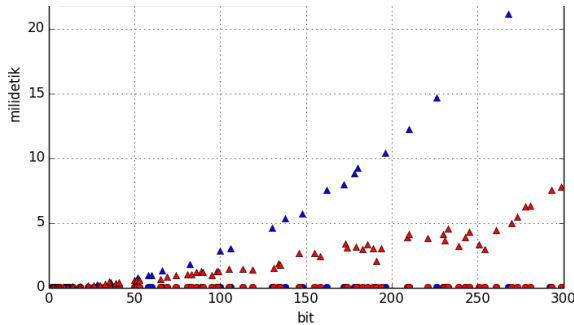
#### Polynomial Basis Testing

Functional testing performed on all arithmetic operations. The tests use the following key length: 4 bits, 8 bits, 16 bits, 32 bits, 64 bits, 128 bits, 256 bits, and 299 bits (see also [19]).

Performance testing performed by executing every method one thousand times before calculating the average value. Library time in Python is used for time measurement and the result of time is milliseconds. Whereas, library psutil in Python is used for memory usage and the result of memory usage is megabyte (MB).

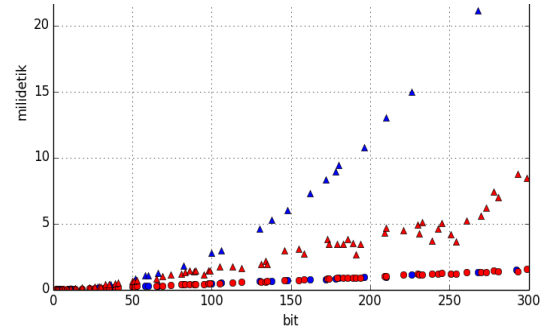
#### Comparison

Here are the results of the time measurement of arithmetic operations in ONB Type I and Type II with and without conversion. The x-axis shows the key length in bits while the y-axis shows the time in milliseconds. The Type I ONB arithmetic operation without conversion is indicated by blue dots, the Type II ONB arithmetic operation without conversion is indicated by red dots, while the blue triangles and red triangles respectively indicate arithmetic operations ONB Type I and Type II with conversion to PB.



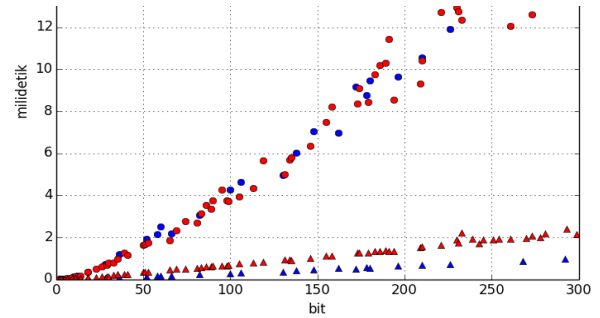
**Figure 11.** Comparison of time for squaring operation.

For squaring operation, ONB without conversion surpasses ONB operation with conversion. This happens because the conversion process is time consuming. ONB squaring operation with conversion requires 2 times conversions, i.e. for input and for output. ONB squaring without conversion produces a graph that is relatively constant. Conversion in Type I ONB requires more time than the conversion in Type II ONB because the algorithms depend on the irreducible polynomials. Irreducible polynomial in Type I ONB is an AOP that means all the irreducible polynomial coefficients are not zero. ONB squaring operation with the conversion produces a graph with a tendency to increase exponentially.



**Figure 12.** Comparison of time for multiplication operation.

The ONB multiplication operation without conversion requires less time than the ONB multiplication operation with conversion. The reason is that the conversion process should be done and the reduction process is performed on PB multiplication operation. The Type I ONB multiplication operation with the conversion increases exponentially. Meanwhile the operation time charts Type II ONB multiplication with conversion and the ONB multiplication operations without conversion tends to increase linearly.



**Figure 13.** Comparison of time for inverse operation.

The ONB inverse operation without conversion requires much longer time than the inverse operation ONB with conversion. The reason is that multiplication is repeated and repeated more as the key length increases. The operating time of ONB inverse operation without conversion tends to increase exponentially. Moreover the reason that the ONB inverse operation with conversion requires less time is because the Extended Euclidean Algorithm method is simpler and we do not need reduction process when performing the inverse operation on the PB.

#### Conclusions

We have implemented finite field arithmetic operations, i.e. squaring, multiplication and inverse algorithms, on both polynomial basis and optimal normal basis representation and the conversion method between them. ONB arithmetic operations with the conversion process is faster than the inverse operation ONB arithmetic operation without conversion, whereas for the other arithmetic operations are faster without conversion.

## Acknowledgement

This research is supported by The Asahi Glass Foundation.

## References

- [1] Google Code Project, PSUTIL, <https://code.google.com/p/psutil/>, 2014.
- [2] D. Hankerson, A. Menezes, and S. Vanstone, Guide to Elliptic Curve Cryptography, New York: Springer, 2004.
- [3] T. Itoh and S. Tsujii, A fast algorithm for computing multiplicative inverses in GF(2<sup>m</sup>) using normal bases, *Information and Computation*, **78**, 1988, 171-177.
- [4] V. Kapoor, V.S. Abraham and R. Singh, "Elliptic Curve Cryptography," *ACM Abiquity*, **9**(20), 2008, 1-8.
- [5] Z. Li, J. Higgins, and M. Clement, Performance of Finite Field Arithmetic in an Elliptic Curve Cryptosystem, Proc. 9th Modeling, *Analysis and Simulation of Computer and Telecommunication Systems*, 2001, 249-256.
- [6] M. Maulana, Desain dan Implementasi Modul Operasi Aritmetika Bilangan pada Finite Field dengan Representasi Optimal Normal Basis, Master Thesis, 2014.
- [7] A. Muchlis, M.H. Khusyairi, and F. Yuliawan, Storage-Efficient Finite Field Basis Conversion via Efficient Multiplication, *Proceeding International Symposium on Computational Sciences*, Bali October 2009.
- [8] I. Muchtadi-Alamsyah, M.W. Paryasto, and M.H. Khusyairi, Finite Field Basis Conversion, *Proc. Intl. Conf. on Math., Statistics and Its Applications*, 2009, 15-18.
- [9] I. Muchtadi-Alamsyah, F. Yuliawan, and A. Muchlis, Finite Field Basis Conversion and Normal Basis in Characteristic Three, *Advances in Algebraic Structures*, *World Scientific*, 2012, 439-447.
- [10] I. Muchtadi-Alamsyah and F. Yuliawan, Basis conversion in composite field, *Int. J. Math Comp.*, **16**(2), 2013, 11-17.
- [11] NIST. Recommended elliptic curves for federal government use. <http://csrc.nist.gov/csrc/fedstandards.html>, Juli 1999.
- [12] C. Paar and J. Pelzl, Understanding Cryptography, New York: Springer, 2010.
- [13] M.W. Paryasto, K. S. Sutikno and A. Sasongko, Issues in elliptic curve cryptography implementation, *Internetworking Indonesia Journal*, **1**(1), 2009, 29-33.
- [14] M.W. Paryasto, B. Rahardjo, I. Muchtadi-Alamsyah, and M.H. Khusyairi, Implementation of polynomial basis-optimal normal basis I basis conversion, *J. Ilmiah Teknik Komputer*, **1**(2), 2010, 148-160.
- [15] M.W. Paryasto, B. Rahardjo, I. Muchtadi-Alamsyah, and Kuspriyanto, Implementasi composite field pada elliptic curve cryptography, *J. Ilmiah Ilmu Komputer*, **7**(2), 2011.
- [16] M.W. Paryasto, B. Rahardjo, I. Muchtadi-Alamsyah, F. Yuliawan, and Kuspriyanto, Composite field multiplier based on look-up table for elliptic curve cryptography implementation, *ITB J. Information and Communication Technology*, **6**(1), 2010, 63-81.
- [17] B. Rahardjo, Keamanan Perangkat Lunak, PT Insan Infonesia, 2014.
- [18] M. Rosing, Implementing Elliptic Curve Cryptography, Manning Publications Co., 1999.
- [19] W.F. Senjaya, Desain dan Implementasi Modul Operasi Aritmetika pada Finite Field Berbasis Polinomial Biner, Master Thesis, 2014.
- [20] G. Seroussi, Table of Low-Weight Binary Irreducible Polynomials, Hewlett Packard Company, 1998.
- [21] S.M. Shohdy, A.B. El-Sisi and N. Ismail, Hardware Implementation of Efficient Modified Karatsuba Multiplier Used in Elliptic Curves, *International Journal of Network Security*, **11**(3), 2010, 155-162.
- [22] W. Stallings, Cryptography and Network Security 3rd ed, New York: McGraw-Hill, 2003.