Elliptic Curves in restricted computational environment

Martin Beránek

Czech Technical University – Faculty of Information Technologies

January 31, 2017

Contents

List of Figures

List of Tables

Abstract

This paper should provide basic overview of field algorithms which are used for Elliptic Curves computations. It focuses on fields used in small devices – `Smart Cards`, `Arduino`, ... Basic overview is given with comparison of complexity based on common hardware architectures.

1 Introduction

Elliptic Curves are modern technique for cryptography. They are used in large number of operations – from signatures to encryption. Very promising is the size of used keys which have the same security level as its amount of used bits. That could lead to bigger application in devices with smaller memory capacity. Elliptic Curves are currently used in large scale because of preferred higher level of security [3]. This paper provide basic overview of fields which are used for Elliptic Curves operations. Algorithms should give reader basic knowledge how to implement Elliptic Curves in small scale hardware environment (`Smart Cards`, `Arduino`, ...).

2 Elliptic Curves

Elliptic Curves are algebraic structure which creates finite group. Property of and finite order creates a space of points which can be used in manner of discrete operations. Every coordinate of point in space of Elliptic Curve actually provide much shorter representation in compare of still largely used spaces in `RSA`. As it is mentioned in figure 1 the security level of bits which are able to be later used is much higher for Elliptic Curves than for other cryptographic systems.

Algorithm family	Cryptosystem	Security level(bit)			
		80	128	192	256
Integer factorization	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Discrete logarithm	DH, DSA, Elgamal	1024 bit	3072 bit	7680 bit	15360 bit
Elliptic curves	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetric-key	AES, 3DES	80 bit	128 bit	192 bit	256 bit

Figure 1: Comparison of bit utilisation in different schemes [1]

In pure mathematical point of view, real Elliptic Curve in continuous way is defined as followed [4]:

Definition 2.1. Elliptic Curve (EC) over real numbers $\mathbb{R}$ is a set of points satisfying the *Weierstrass* equation:

$$y^2 = x^3 + a \cdot x + b; x, y, a, b \in \mathbb{R}$$

$$4 \cdot a^3 + 27 \cdot b^2 \neq 0$$

along with point in infinity $\mathbb{O}$

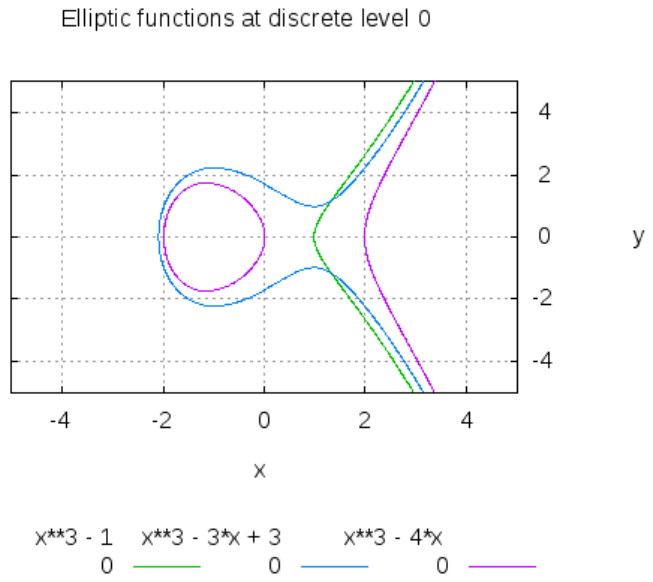

Figure 2: Example of Elliptic Curves

Reason why for Elliptic Curves use just half of its key size is because only one dimension of point (X or Y) is used in real encryption. From 2 there is clear relationship that each point for fixed dimension of X have at the best case two Y possibilities. That's why only one of dimension is used.

In following manner we can derive operations on Elliptic Curve. As it was mentioned, for computations we use only set of points on curve. For this we can define addition and multiplication with constant.

2.1 Operations

Basic operation with points on Elliptic Curve is addition. It's defined in following manners:

Definition 2.2. Point addition of Elliptic Curve is defined as $R = P + Q$ where $R, P, Q \in EC$ as $P = [X_p, Y_p]; Q = [X_q, Y_q]$ with result in $R = [X_r, Y_r]$. Then

$$X_r = \lambda^2 - X_p - X_q; Y_r = (X_q - X_r)\lambda - Y_q$$

where

$$\lambda = \begin{cases} \frac{Y_p - Y_q}{X_p - X_q} \text{ for } P \neq Q (\text{addition}) \\ \\ \frac{3 \cdot X_q^2 + a}{2 \cdot Y_q} \text{ for } P = Q (\text{doubling}) \end{cases}$$

Graphical representation of point addition for $P \neq Q$ would be constructed with intersection of line with curve. From intersection the square line towards Y negative part of curve.

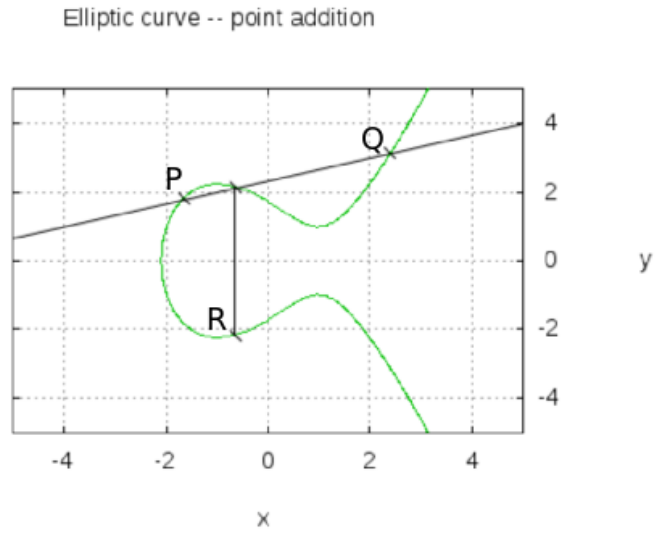

Figure 3: Example of Elliptic Curves point addition

2.2 Definition of infinity

For defining identity in Elliptic Curve space special point in infinity is developed. It have meaning of zero or one in identity kind of way and as well as infinity. In practical point of view, the point of infinity could be defined as something known which is by the definition in the curve space but wouldn't fit *Weierstrass* condition.

`IEEE` standard [4] recommends multiple points which should be considered as point in infinity. For *affine* coordinates point of $\mathbb{O}$ is $[0, 0]$ which have also practical point of view. As well as for curves where $b = 0$ the point is defined as $\mathbb{O} = [0, 1]$. Since in *projective* kind of way the transformation is provided by $X_a \to x_p, Y_a \to y_p, 1$ which give projective point of zero as $\mathbb{O} = [0, 0, 1]$.

Point of infinity have the meaning of providing following operations:

$$P + \mathbb{O} = \mathbb{O} + P = P$$

2.3 Elliptic Curves as Abelian group

We already defined identity. Elliptic space actually follow all the properties of the Abelian group. Axioms are defined as follow:

- Associativity: $(P + Q) + N = P + (Q + N)$ for $\forall P, Q, N$

- Identity: $P + \mathbb{O} = \mathbb{O} + P = P$ for $\forall P, Q$
- Commutativity: $P + Q = Q + P$ for $\forall P, Q$

3 Application

As the Elliptic Curves offer finite space of usable points, via usage of operations numerous use cases can be derived [5]. Application is so wide that Elliptic Curves can provide key exchange, encryption and signature. In this paper, only encryption and key exchange is mentioned.

3.1 Elliptic Curve *Diffie Hellman*

As it was mentioned before the elliptic space provides group. That's why *Diffie Hellman* [6] algorithm can be used over Elliptic Curves. Suppose that Alice and Bob have the same Elliptic Curve and both publicly exchanged same point $P \in EC$. They both create random integer number k which is private and used once (`nonce`).

Alice:

1. Choose random K_a
2. Count $K_a \cdot P = A$
3. Alice send A to Bob
4. Counts $K_a \cdot B = Q$

Bob:

1. Choose random K_b
2. Count $K_b \cdot P = B$
3. Bob send B to Alice
4. Counts $K_b \cdot A = Q$

Where as secret is used only Q_x.

3.2 Elliptic Curve Encryption Scheme

Similar algorithm can be derived for encryption [5]. Suppose that Alice and Bob have the same Elliptic Curve. In this scheme acts like somebody who wants to encrypt the message and Bob as somebody who wants to decrypt it later on:

Alice:
Creating keys:

1. From EC take $ord(p) = n$
2. Choose $d \in \mathbb{Z}_n^*$
3. Compute $Q = d \cdot P$
4. Publish public key: P, Q, n
   and keep private key: d

Encryption:

1. Choose random $k \in \mathbb{Z}_n^*$
2. Count $k \cdot P = (x_1, y_1)$
3. Count $k \cdot Q = (x_2, y_2)$
4. And publicly provides following set
   $C = (x_1, y_1 \mod 2, m \cdot x_2 \mod p)$

Bob:
Decryption:

1. Receive set $C = (x, b, Z)$
2. Create point of $R = (x = x, y = b)$
3. Counts $m = Z \cdot x_0^{-1} \mod p$

4 Double-and-add

In terms of point multiplication by constant, which is used in all mentioned algorithms above, the special algorithm should be used. Of course for $k \cdot P = \sum_{i=1}^{k} P \ldots$ can be used. But complexity of this approach is unreasonable high. Partial speed up of scheme is to use `Double-and-add` algorithm which copy `Multiply-and-square` algorithm but uses operation from Elliptic Curves space.

Algorithms for $N \cdot P$, where $N \in \mathbb{N}$ follow this steps:

```
TMP = P
RESULT := 0
for i from 0 to m do:
  if N_i == 1 then:
     RESULT := add_points(Q, TMP)
  TMP := double_point(TMP)
return Q
```

So at maximum there is $O(\lfloor log_2(d) \rfloor)$ addition and $O(\lfloor log_2(d) \rfloor)$ doubling.

5 Galois finite field

As it was mentioned earlier, operation of Elliptic Curves is actually composite of multiple other operations over different fields. The most natural one seems to be $\mod p$ where $p \in \mathbb{P}$ for having clear definition how to count inversions and other operations. But let's take in count fields as Galois polynomials, which can provide much needed changed perspective in additions [7].

Definition 5.1. Notation $\mathbb{F}_{p^m}$ or $GF(p^m)$ which have finite number of elements p^m

- p – characteristic prime
- m – degree $m \in \mathbb{N}$
- p^m – order

There are special cases as $GF(p)$ where $m = 1$ which is the same as modulo operation.

5.1 EC over $GF(2^n)$

Naturally, implementation of Elliptic Curves as $GF(p)$ would be the most common solution. But for $GF(2^n)$ polynomials we can ease up addition operation. With different *Galois field* we also need to redefine some equations which are now different due to nature of $GF(2^n)$.

Definition 5.2. *Weierstrass* equation over $GF(2^n)$:

$$y^2 + xy = x^3 + a \cdot x^2 + b \mod F(\alpha)$$

Where also points operation is quite different:

Definition 5.3. For $R = P + Q$ where $R, P, Q \in EC$ as $P = [X_p, Y_p]; Q = [X_q, Y_q]$ with result in $R = [X_r, Y_r]$ over $GF(2^n)$ with $\mod F(\alpha)$. Then

$$X_r = a + \lambda^2 + \lambda + X_p + X_q; Y_r = (X_q + X_r)\lambda + X_r + Y_q$$

where

$$\lambda = \begin{cases} \frac{Y_p + Y_q}{X_p + X_q} \text{ for } P \neq Q (\text{addition}) \\ \\ X_q + \frac{Y_q}{X_q} \text{ for } P = Q (\text{doubling}) \end{cases}$$

6 Mathematical operations

6.1 Addition and subtraction

If we take in count just $GF(2^n)$ we actually have addition only over $\mod 2$ which end up only to `XOR` operation. That have complexity over all bits $O(log_2(n))$, which is much bigger estimate since normal processor usually provide single cycle for `XOR` of word size (`32bits`, ...).

Algorithm would go as follow:

```
for i from 0 to m:
   R[i] := P[i] XOR Q[i]
```

Where R is resulting number for input P and Q.

For $GF(p)$ we always have take in account fact of overflowing numbers. If we consider basic addition, we would still need to wait for the last flowing bit. Some solutions are given for this problem (for example parallel adding as `PPS` accumulation [8]), yet speed up is never bigger than just `XOR` of numbers.

6.2 Multiplication

With $GF(2^m)$ multiplication in hardware provide real speed up if used proper `LSFR` [9]. With that we can state algorithm which will reduce every step of `MSB` multipliers:

```
C := 0
for i := m - 1 to 0:
   C := C a mod F(a) + b_i A;
```

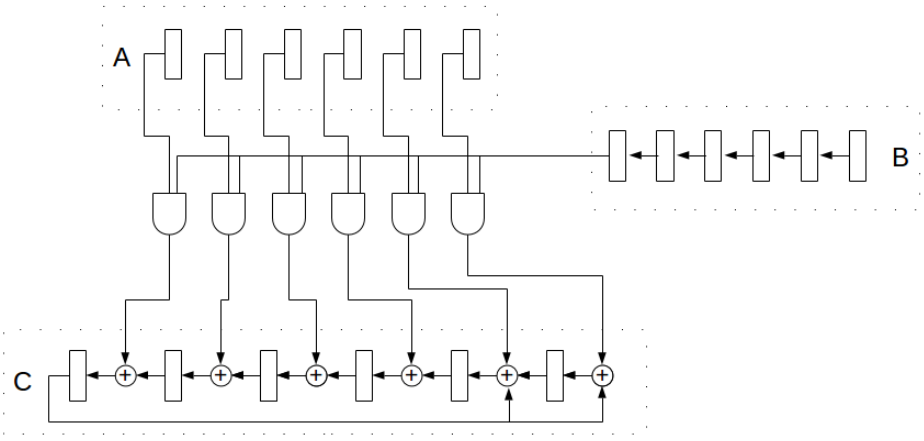

Figure 4: Multiplication modulo $F(a) = a^6 + a + 1$

This approach is not really fast in software. For each step of algorithm modulo operation would take much more time then hardware implemented `LSFR` alternative. That's why with for `LSB` multiplication only the result will be reduced. Algorithm would go as follow:

```
C := 0
for i := 0 to number size:
   j := 1
   if B & j == 1:
      add (C, A)
   shiftLeft A
   j <<= 1
return C
```

This is really complex approach since in every step there is one shift usually over huge number. That could be speed up with shifting wisely. Consider size of word in given processor. For small devices it could be only up to `8` bits. That mean that for example 160 bits would be saved in array of 20 bytes. Shifting in each step over all of them would take impossible amount of time. That's why is shifting done only 8 times and there is no need to count overflows. In each step of algorithm there would be considered just one particular bit of each byte [4]. Algorithm would go as follow:

```
C[20] := 0
j := 1
for k = 0 up to k < 8:
   for i = 0 up to i < 20:
      if B[i] & j
         add C A
   j <<= 1
   shiftLeft A
return C
```

This approach actually works on $GF(p)$ and $GF(2^m)$ with exception of difference in addition operation algorithm. In normal $GF(p)$ there would be much more waiting on overflowing bits. Also, as was mentioned earlier, there is going to be huge delay on reduction after multiplication.

6.3 Squaring

If we consider $GF(2^m)$ we can obtain really fast squaring which is based on a following:

Let's have polynomial x which would give us after square x^2. With that let's investigate polynomial $x+1$ which gives x^2+1 (we are in $GF(2^m)$ where coefficients are modulo 2). This emerging patter actually works based on Binomial distribution [10]. So we can actually state following picture:

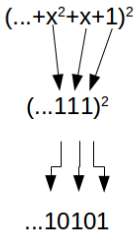

Figure 5: Squaring over $GF(2^m)$

With that we can construct algorithm with lookup table for each square $x, x^2, \ldots$ and latter add them in final result.

```
lookup_table[9] = {
    0     -> 0000 0000 // as well for 3,5,6,7
    1     -> 0000 0001
    10    -> 0000 0100
    100   -> 0001 0000
    1000  -> 0100 0000
}
R := 0
k := 0
for i := 0 up to SIZE of input:
   R[k] XOR = lookup_table[a[i] & 1]
   R[k] XOR = lookup_table[a[i] & 2]
   R[k] XOR = lookup_table[a[i] & 4]
   R[k] XOR = lookup_table[a[i] & 8]
   k += 1
   R[k] XOR = lookup_table[a[i] & 1]
   R[k] XOR = lookup_table[a[i] & 2]
   R[k] XOR = lookup_table[a[i] & 4]
   R[k] XOR = lookup_table[a[i] & 8]
   k += 1
```

Complexity of this algorithm is now linear opposite to quadratic which would be used in normal multiplication.

6.4 Inversion

In affine space for both $GF(2^n)$ and $GF(p)$ inversion is provided as Enhanced Euclidean Algorithm [4]. Operations are edited according to the field on where are used. For simplicity let's use number of steps for inversion as $O(log_{10}(n))$ as upper boundary. Euclidean algorithm goes as follow:

```
t := 0; newt := 0
r := modulo_group_n; newr := input_number
while newr not zero:
   q := r / newr # whole number division
   t := newt
   newt := t - q * newt
   r := newr
   r := r - q * newr
if r > 1:
   "a is not invertible"
if t < 0:
```

```
   t := t + n
return (inverse_a := t)
```

From that we can again estimate number of operation which is used in every step of algorithm:

Table 1: Operations utilization in one step of Enhanced Euclidean Algorithm

Operation	**Count**
Multiplication	2
Addition/substraction	2
Whole number division	1

With exception of last addition. That would seem to state that getting faster multiplications and additions would provide much needed speed up.

7 Special domains

7.1 Projective coordinates

We just stated how many operations are used during affine space inversions. Even though *Enhanced Euclidean Algorithm* is used, opposite to other operations it's the most time demanding. Solution would be to transform affine coordinates to projective in form of $[x, y] \to [X = x, Y = y, Z = 1]$. Conversion back to the affine space is provided as $[X, Y, Z] \to [\frac{x}{Z^2}, \frac{y}{Z^3}]$. Where Z coordinate play a role of common denominator [9]. That means during counting inversions we have constant complexity. Basic overview is provided in `IEEE 1363` with all the operations required for the point operations. In closer look into the description of projective space, we need to consider that with counting Z other computation needed to be done and complexity with that rised for other operations as well.

7.2 Normal base

Normal base have very large potential to speed up some operations. For example squaring is just one shift operation [5]. That's why in this paper is mentioned how to use normal base in Elliptic Curves field.

Definition 7.1. Normal basis is a set of form: $\mathbb{B} = \alpha, \alpha^2, \alpha^{2^2}, \ldots, \alpha^{2^{m-1}}$

Since elements are base, adding up subsets can't create 0 (elements are linear independent). There exist normal basis for every $GF(2^m)$.

The $GF(2^m)$ is represented as in normal basis in following string form:
$(a_0 a_1 a_2 \ldots a_{m-1})$ as element combined out of normal base
$a_0\alpha + a_1\alpha^2 + a_2\alpha^{2^2} + \cdots + a_{m-1}\alpha^{2^{m-1}}$

Normal base works on $GF(2^m)$ field. There is no change in addition over normal base. More interesting operation is squaring which actually rely on following property:

$$A = (a_0 a_1 a_2 \ldots a_{m-1}) = a_0\alpha + a_1\alpha^2 + a_2\alpha^{2^2} + \cdots + a_{m-1}\alpha^{2^{m-1}}$$

$$A^2 = a_0\alpha^{2\cdot 2^0} + a_1\alpha^{2\cdot 2^1} + \cdots + a_{m-1}\alpha^{2^{2\cdot m-2}} + a_{m-1}\alpha^{2\cdot 2^{m-1}}$$

Where for $a_{m-1}\alpha^{2\cdot 2^{m-1}}$ we can use *Little Fermat theorem* and have just $a_{m-1}\alpha^{2^0}$ and put in begging of this additive form:

$$a_{m-1}\alpha^{2^0} + a_0\alpha^{2^1} + a_1\alpha^{2^2} + \cdots + a_{m-2}\alpha^{2^{m-1}}$$

That actually means following:

$$A^2 = (a_{m-1} a_0 a_1 a_2 \ldots a_{m-2})$$

So in normal basis the squaring is just rotational shift with constant complexity. Of course even the bit shift over big amount of bytes could be complex because of overflowing bits. Yet for hardware it's just a matter of right wiring.

Another fast operation would by multiplication for which we need multiplicative matrix $\mathbb{M}$. Algorithm is defined as follow:

```
A := (a0 a1 a2 ... a{m-1})
B := (b0 b1 b2 ... b{m-1})

C := 0
for i := 0 to m - 1:
   ci := A . M . transposed(B)
   A := LeftShift(A)
   B := LeftShift(B)
```

So for input A, B we are getting multiplication of both in form of C. Matrix $\mathbb{M}$ needs to be determined for the basis which is used. Since algorithm uses multiplication of matrices, general rule is to find matrix $\mathbb{M}$ with the smallest number of ones. That in best cases means to find out matrix containing $2m-1$ non zero elements.

Problem with normal basis is no fast inversion calculation. Only known rule is to use *Little Fermat theorem*. So for finding inversion of a the $a^{2^m-2} \equiv a^{-1}$ need to be counted. That is computationally hard problem. One of the solution was provided by *Itoh, Teechai & Tsujii* [11]. It utilize the fact that binary representation of $2^m - 2$ is $1111\ldots11110$.

Algorithm takes as input $pr, pr-1, \ldots, p0$ which is binary representation of $m-1$ and A for which it returns A^{-1}.

```
C := A
k := 1
for i := r down to 1
   B := C
   for j := k down to 1
      B := B*B
   C := B*C
   k := 2k
   if pi-1 == 1:
      C := C * C * A
      k += 1
return(C*C)
```

To compare efficiency of algorithm, we can see how fast would be same operation with `square-and-multiply`.

Table 2: Comparison of `ITT` with `square-and-multiply` [2]

	Clock cycles
`ITT`	$\approx 1.5m\log(m)$
`square-and-multiply`	$\approx 0.5m^2$

7.3 Montgomery multiplication for GF(p)

Another promising field for counting over $GF(p)$ is Montgomery field. It's effectivity is not provided by the number of used operations but with usage of operation which are not that expensive [9].

Definition 7.2. Number $\overline{a} = |aR|_m$ [1] is called m-residuum where numbers $\overline{a}$ are numbers of Montgomery field – Montgomery domain. Where $R > m$ and is the least power of base of positional power system.

To provide example for numbers of R let's stay in numbers of base 10. For example if $m = 78 \rightarrow R = 100$, if we switch to binary form $m = 11 \rightarrow R = 16$ since we are not able to fit 11 under the next lower $R = 8$.

With this we can provide addition which is actually same as it is for basic $GF(p)$ hence it's use current given architecture:

$$\overline{c} = \overline{a} + \overline{b} = ||aR|_m + |bR|_m|_m = |(a+b)R|_m$$

Multiplication is provided as follows:

$$\overline{c} = |\overline{a}\overline{b}R^{-1}|_m$$

From that we can easily provide backwards transformation:

[1] Consider operation $|A|_m$ as $A \mod m$, this notation is used by *Ing. Róbert Lórenz, CSc.* from *CTU*, only convenience for this notation is that it's faster to write

$$a = |\overline{a} \cdot 1 \cdot R^{-1}|_m$$

Algorithm for multiplication can be done in linear complexity in number of bits of base R. It is defined as follow:

Let's have an input $R = 2^n$, $Mont(a) = \overline{(a)}$, $Mont(b) = \overline{(b)}$ and m.

```
s := 0; i := 0
while (i < n)
   x := x + Mont(a_i)Mont(b)
   x := (x + x0 m)/2
   i := i + 1
if x >= m:
   x := x - m
```

Where x is $|\overline{a} \cdot \overline{b}R^{-1}|_m$. That also mean algorithm actually counts out x in decimal base.

Another very promising algorithm is Montgomery inversion:

Input is $a, b \in \mathbb{Z}; a > b > 0$ where a is an odd and n is number of bits of a.

Phase 1:

```
u := a; v := b;
r := 0; s := 1;
k := 0
while v > 0:
   if u even:
      u := u/2
      s := 2*s
   else if v even:
      v := v/2
      r := 2*r
   else if (u > v):
      u := (u - v)/2
      r := r + s
      s := 2*s
   else:
      v := (v - u)/2
      s := r + s
      r := 2*r
   k := k + 1
if u not 1
   return a,b are not relative primes

if r >= a:
   r := r - a
```

Phase 2:

```
while k > n
   if r even:
      r := r / 2
   else:
      r := (r + a)/2
   k := k - 1
return (inv(b) 2^n % n = a - r)
```

Considering both algorithms mentioned above, there is common pattern. Both use operations, which are very hardware efficient. Although we can see multiplication and division by two, it can be interpreted as shift operations. The most complex operations are additions and subtractions.

8 Conclusion and further reading

This paper provided basic overview of methods used for operation with points on Elliptic curves. Main difference in algorithms can be found in usage of Galois fields which can be $GF(p)$ or $GF(2^m)$. Another approach is to work in affine space or projective which speed up usage of inversion. These are all factors to be considered based on hardware platform where the Elliptic curves should be implemented. Some hardware support fast addition with flag on overflow, some support better `XOR`. With information about the platform, reader can find in here which algorithm would be more suitable.

There is plenty of topics which haven't been touched in this paper. One of them is definitely problem of side channels which touches the basic structure of hardware. Algorithm `Double-and-add` is fairly good readable with power analysis. In case of algorithms, there is plenty more solutions for inversions in both $GF(p)$ and $GF(2^m)$ [12]. The way of development in this field is targeting hardware based solution. Therefore more architectural based methods should be covered, since Elliptic curves are easily implemented in hardware.

References

[1] Paar, C.; Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, 2009, ISBN 9783642041013. Available from: `https://books.google.si/books?id=f24wFELSzkoC`

[2] Novotny, M. Arithmetics with Normal Basis. Slides for subject MI-BHW. Available from: `https://edux.fit.cvut.cz/courses/MI-BHW/_media/lectures/elliptic/mi-bhw-09-gfarithmetic.pdf`

[3] OpenSSL. Elliptic Curve Cryptography. Available from: `https://wiki.openssl.org/index.php/Elliptic_Curve_Cryptography`

[4] IEEE Standard Specifications for Public-Key Cryptography. *IEEE Std. 1363-2000*, 2000, doi:10.1109/IEEESTD.2004.94612.

[5] Hankerson, D.; Menezes, A.; Vanstone, S. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing, Springer New York, 2006, ISBN 9780387218465. Available from: `https://books.google.cz/books?id=V5oACAAAQBAJ`

[6] Rescorla, E. Diffie-Hellman Key Agreement Method. RFC 2631, RFC Editor, June 1999. Available from: `https://tools.ietf.org/html/rfc2631`

[7] Benvenuto, C. J. Galois Field in Cryptography. 2012. Available from: `https://www.math.washington.edu/~morrow/336_12/papers/juan.pdf`

[8] Tvrdík, P. *Parallel algorithms and computing*. CTU, 2003, ISBN 80-01-02824-0. Available from: `http://aleph.nkp.cz/F/?func=direct&doc_number=001023210&local_base=SKC`

[9] Guajardo, J.; Güneysu, T.; Kumar, S. S.; et al. Efficient Hardware Implementation of Finite Fields with Applications to Cryptography. *Acta Applicandae Mathematica*, volume 93, no. 1, 2006: pp. 75–118, ISSN 1572-9036, doi:10.1007/s10440-006-9072-z. Available from: `http://dx.doi.org/10.1007/s10440-006-9072-z`

[10] Granville, A. The Arithmetic Properties of Binomial Coefficients I. 1996. Available from: `http://www.cecm.sfu.ca/organics/papers/granville/Binomial/toppage.html`

[11] Itoh, T.; Tsujii, S. A Fast Algorithm for Computing Multiplicative Inverses in GF(2M) Using Normal Bases. *Inf. Comput.*, volume 78, no. 3, Sept. 1988: pp. 171–177, ISSN 0890-5401, doi:10.1016/0890-5401(88)90024-7. Available from: `http://dx.doi.org/10.1016/0890-5401(88)90024-7`

[12] Hlavác, J.; Lórencz, R. Arithmetic Unit for Computations in GF(p) with the Left-Shifting Multiplicative Inverse Algorithm. In *Architecture of Computing Systems - ARCS 2013 - 26th International Conference, Prague, Czech Republic, February 19-22, 2013. Proceedings*, 2013, pp. 268–279, doi:10.1007/978-3-642-36424-2_23. Available from: `http://dx.doi.org/10.1007/978-3-642-36424-2_23`