COMPARISON OF ALGORITHMS FOR ELLIPTIC CURVE CRYPTOGRAPHY OVER FINITE FIELDS OF $GF(2^m)$

Mathias Schmalisch, Dirk Timmermann
University of Rostock, Institute of Applied Microelectronics and Computer Science
Richard-Wagner-Str. 31, 18119 Rostock, Germany
{mathias.schmalisch, dirk.timmermann}@etechnik.uni-rostock.de

Abstract

For elliptic curve cryptosystems does exist many algorithms, that computes the scalar multiplication $k \cdot P$. Some are better for a software solution and others are better for a hardware solution. In this paper we compare algorithms without precomputation for the scalar multiplication on elliptic curves over a finite field of $GF(2^m)$. At the end we show which algorithm is the best for a hardware or software solution.

Key Words

elliptic curve cryptography, scalar multiplication algorithms

1. Introduction

Elliptic curve cryptography (ECC) based on the scalar multiplication $k \cdot P$, where k is an integer and P a point on an elliptic curve. The scalar multiplication can be computed with point addition and point doubling on the elliptic curve. These again are based on computations in finite fields. In the finite field $GF(2^m)$ the arithmetic operations addition, multiplication, squaring and inversion have to be supported.

To compute the scalar multiplications several algorithms with different numbers of finite field operations exist. The algorithms are split in two great groups. One group which compute the coordinates of the points in the affine plane and the other group using the projective plane. If we compute the scalar multiplication in affine coordinates, then we need an inversion for each point addition and point doubling. On the other hand if we compute the scalar multiplication in projective coordinates we need only one inversion but more multiplications and squarings.

The inversion is the operation which is the hardest to compute. So the choice for an algorithm depends on the speed of the inversion. To compute the inversion there exist two methods one is the extended Euclidian Algorithm and the other is Fermat's Theorem. The inversion with Fermat's Theorem can be computed with several multiplications and squarings, so this method is very applicable for software computing. For the extended Euclidian Algorithm is a hardware solution , that computes an inversion as fast as a multiplication. Therefore for hardware solutions are other algorithms better that for software solutions.

2. Elliptic Curves

Elliptic Curve Cryptography (ECC) first suggested by N. Koblitz [1] and V. Miller [2] in 1987 and 1986 is used in the meantime for different kinds of cryptosystems, for instance digital signature algorithm [3] and key exchange protocols [4]. The security of these cryptosystems is based on the problem to compute the discrete logarithm on elliptic curves. This problem is harder to compute than other trapdoor one-way functions that used for public key cryptosystems. Therefore the key length for ECC can be considered shorter in comparison with other well known public key cryptosystems such as RSA with the same level of security. So today RSA needs a key length of 1024 bits where ECC only needs 160 bits.

For the implementation of elliptic curve cryptosystems in hardware are the elliptic curves over finite field $GF(2^m)$ best suitably. In this case the field has the characteristic 2 and the equation for a non-supersingular elliptic curve has the form

$$E / K : y^2 + xy = x^3 + a_2 x^2 + a_6 \ , \tag{1}$$

with $a_6 \neq 0$. To compute the scalar multiplication we have to compute the point addition and point doubling on this curve.

2.1 Affine Coordinates

To compute the point addition of $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ we need the sloop λ of the line through P and Q. If $P = Q$ than we have a point doubling, in this case the line is the tangent to the curve P. Therefore the equation for the slope λ of the line has the following form

$$\lambda = \begin{cases} \dfrac{y_1 + y_2}{x_1 + x_2}, & P \neq Q, \\ x_1 + \dfrac{y_1}{x_1}, & P = Q. \end{cases} \quad (2)$$

The new coordinates for $R = P + Q$ with $R = (x_3, y_3)$ can be computed with the equation

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a_6, \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1. \end{aligned} \quad (3)$$

For the point doubling, this equation can be simplified to

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a_6, \\ y_3 &= \lambda x_3 + x_3 + x_1^2. \end{aligned} \quad (4)$$

With these equations we can summarize the number of operations in the finite field for computation in affine coordinates.

Table 1. Number of Operations in Affine Coordinates

Point Operation	#Add.	#Mult.	#Sqr.	#Inv.
Adding	8	2	1	1
Doubling	5	2	2	1

2.2 Projective Coordinates

Another method to compute the point addition and point doubling is to use projective coordinates $P = (X, Y, Z)$. This method has the advantage to save the inversion which is the hardest to compute operation. So we need only on inversion to back conversion. In order to use this method, the coordinates must be converted. For this there exist several solutions, which differ in the number on field operations for the point operations. The conversion from affine to projektive coordinates is for all the same

$$X = x, Y = y, Z = 1. \quad (5)$$

But the back conversion is different and therefore the equations for the point addition and point doubling. One solution is to find in [4] where

$$x = X / Z^2, y = Y / Z^3 \quad (6)$$

Here a point addition needs 7 additions, 15 multiplications and 5 squarings and the point doubling needs 4 additions, 5 multiplications and 5 squarings.

Other solutions for projective coordinates are to find in [5] and [6]. The number of field operations for the point addition and point doubling for these three cases is summarized in Table 2.

Table 2. Number of Operations in Projective Coordinates

Point Operation	Projective Coordinates	#Add.	#Mult.	#Sqr.
Addition	$(X/Z, Y/Z)$ [6]	7	13	1
	$(X/Z, Y/Z^2)$ [5]	8	9	4
	$(X/Z^2, Y/Z^3)$ [4]	7	15	5
Doubling	$(X/Z, Y/Z)$ [6]	4	7	5
	$(X/Z, Y/Z^2)$ [5]	4	4	5
	$(X/Z^2, Y/Z^3)$ [4]	4	5	5

3. Algorithms for Scalar Multiplications

3.1 Standard Algorithm

The easiest and most naive way to compute the scalar multiplication $k \cdot P$ is the left to right binary algorithm also known as the Double-and-Add (DaA) algorithm. In this algorithm k is in a binary representation with n bits and the most significant bit (MSB) k_{n-1} = “1”. First initialize Q with P ($Q = P$). Then perform a loop, in which the bits from position n-2 to 0 are tested. In each loop run a point doubling ($Q = 2Q$) is performed. If the current bit k_i is a “1”, then also a point addition ($Q = Q + P$) is to calculate. So we can easily compute $Q = k \cdot P$.

```
Algorithm 1: Double and Add
Input: An integer k > 0 and a point P
Output: Q = k·P
1. k := $(k_{n-1}, ..., k_1, k_0)_2$
2. Q := P
3. for i from n-2 downto 0 do
4.   Q := 2Q
5.   if $k_i$ = 1 then
6.     Q := Q + P
7. return Q
```

With this algorithm we need n-1 point doublings and $Hw(k)$-1 point additions and $Hw(k)$ is the Hamming weight of k. The Hamming weight represents the number of non-zeros in k, which is in the average $n/2$ for a binary representation. To reduce the Hamming weight of k we can recode k in a signed digit (SD) representation $k_i \in \{-1, 0, 1\}$. With a suitable recoding algorithm we can reduce the Hamming weight in the average to $n/3$. So we can eliminate $n/6$ point additions, but we need additionally the point subtraction. For a point $P = (x_1, y_1)$ in affine coordinates on an elliptic curve E is $-P = (x_1, y_1 + x_1)$, therefore the point subtraction can be computed with a point addition and an additional finite field addition. With this approach we can build the Double-and-Add/Subtract (DaAS) algorithm.

```
Algorithm 2: Double and Add/Subtract
Input: An integer k > 0 and a point P
Output: Q = k·P
1. k := $(k_{n-1}, ..., k_1, k_0)_{SD}$
2. Q := P
3. for i from n-2 downto 0 do
4.   Q := 2Q
5.   if $k_i$ = 1 then
6.     Q := Q + P
7.   elseif $k_i$ = -1 then
8.     Q := Q - P
9. return Q
```

To recode k in a signed digit representation there exist several algorithms. To get a minimal Hamming weight we can use the canonical recoding algorithm described in [7]. This is a serial algorithm, which works from right-to-left. Another algorithm is the modified Booth algorithm [8]. This algorithm can compute the signed digit representation parallel and also from left-to-right. So for a hardware solution this algorithm is very suitable.

So the time to compute a scalar multiplication t_{k*P} with Algorithm 1 or Algorithm 2 is

$$t_{k*P} = t_{PA} * Hw(k) + t_{PD} * (n-1), \tag{7}$$

where t_{PA} is the time for a point addition and t_{PD} is the time for a point doubling. If we want to know the average time for the scalar multiplication, we have to add the average number of non zeros for the Hamming weight and get the equations for the DaA and DaAS algorithm

$$\begin{aligned} t_{k*P,DaA} &= t_{PA}\frac{n}{2} + t_{PD}\left(n-1\right), \\ t_{k*P,DaAS} &= t_{PA}\frac{n}{3} + t_{PD}\left(n-1\right) + t_A\frac{n}{6}. \end{aligned} \tag{8}$$

Because we need the point subtraction for the DaAS algorithm, we have to add the time for a field addition t_A multiplied with the average number of negative ones in k.

The time for a point addition and a point doubling depends on the coordinate system, on which the points are computed. For the affine coordinates the time is

$$\begin{aligned} t_{PA} &= 8t_A + 2t_M + t_S + t_I, \\ t_{PD} &= 5t_A + 2t_M + 2t_S + t_I. \end{aligned} \tag{9}$$

For projective coordinates you find the number of field operations in Table 2. But additional the time for the back conversion from projective to affine coordinates is to add to the time for a scalar multiplication

$$t_{k*P,projective} = t_{k*P} + t_{back\,conversion}. \tag{10}$$

3.2 Improvement

If a point addition is faster to calculate than a point doubling, then it is possible to speed up the DaA and DaAS algorithm because

$$2Q + P = \left(Q + P\right) + Q \tag{11}$$

In this case we have to compute two successive point additions, where the second point additions includes an addend from the first. The publication [9] shows a method to save the computation of the y-coordinate for the intermediate result in affine coordinates for this case. So it is possible to save a field addition, multiplication and squaring.

3.3 Montgomery Algorithm

A different approach for computing $k{\cdot}P$ was introduced by Montgomery [10]. This approach is based on the DaA algorithm and the observation that the y-coordinates of the sum of two points whose difference is known can be computed in terms of the x-coordinates of the involved points.

```
Algorithm 3: Montgomery
Input: An integer k > 0 and a point P
Output: Q = k·P
1. k := $(k_{n-1}, ..., k_1, k_0)_2$
2. $P_1$ := P, $P_2$ := 2P
3. for i from n-2 downto 0 do
4.   if $k_i$ = 1 then
5.     $P_1$ := $P_1$ + $P_2$, $P_2$ := $2P_2$
6.   else
7.     $P_1$ := $2P_1$, $P_2$ := $P_1$ + $P_2$
8. return (Q = $P_1$)
```

The time to compute the scalar multiplication with this algorithm is

$$t_{k*P,Mon} = \left(t_{PA} + t_{PD}\right)\left(n-1\right) + t_y. \tag{12}$$

But we only need to compute the x-coordinates. In the publication [11] we can find the number o field operations for affine and projective coordinates for the Algorithm 3. These are summarized in the following table.

Table 3. Number of Operations for the Montgomery Algorithm

Operation	Coordinates	Add	Mult	Sqr	Inv
Point Addition and Doubling	Affine	4	2	2	2
	Projective	3	6	5	0
Compute y Coordinate	Affine	6	4	2	1
	Projective	7	10	3	1

4. Comparing the Algorithms

4.1 Software

To find out, which algorithm is the best, we need to know the time for the field operation. For a software solution there exist several implementations. Here we compare the algorithms with the Java implementation Cryptix Elliptix [12] and the C++ implementation LiDIA [13]. The time for the computation of the field operations was measured on an AMD Athlon 1400 MHz PC with 512 MB Ram on Windows 2000. To get a useful result, we run every operation with 100000 runs. In Table 4 and Table 5 we can see the running time of the field operations in milliseconds.

Table 4. Running Time in *ms* using Cryptix Elliptix [12]

Field Width m	Add.	Mult.	Sqr.	Inv.
163	230	2203	541	8582
191	250	2323	561	11005
239	271	3164	621	15942

Table 5. Running Time in *ms* using LiDIA [13]

Field Width m	Add.	Mult.	Sqr.	Inv.
163	10	271	45	2944
191	11	281	50	3164
239	13	360	59	4226

If we use the measured time to compute the time for a scalar multiplication with the different algorithms from section 3, then it is possible to compare the algorithms and use the best for the implementation of the scalar multiplication. In Table 6 and Table 7 are the time for the scalar multiplication with the two software implementations summarized.

With these results it is possible to choose the best algorithm for the software computation. The tables shows, that the best computation time has the Montgomery Algorithm with projective coordinates for both software implementations.

Table 6. Time for a Scalar Multiplication using Cryptix Elliptix

Coordinates	Algorithm	Time in *ms* for *m* =		
		163	191	239
Affine	DaAS	29.20	41.07	70.64
	DaAS [9]	30.90	43.18	74.52
	Montgomery	38.59	55.16	97.15
Projective	DaAS [4]	31.85	39.24	63.23
	DaAS [5]	31.77	39.09	61.61
	DaAS [6]	35.61	43.80	70.01
	Montgomery	24.59	30.30	48.99

Table 7. Time for a Scalar Multiplication using LiDIA

Coordinates	Algorithm	Time in *ms* for *m* =		
		163	191	239
Affine	DaAS	7.35	9.23	15.28
	DaAS [9]	7.60	9.52	15.77
	Montgomery	10.73	13.47	22.38
Projective	DaAS [4]	3.48	4.27	6.73
	DaAS [5]	3.33	4.12	6.44
	DaAS [6]	3.84	4.73	7.43
	Montgomery	2.74	3.37	5.30

4.2 Hardware

For software is the Montgomery Algorithm with projective coordinates the best algorithm. But is this algorithm also the best for a hardware solution? This question will be answered in this section. First we have to know the computation time for the four field operations.

The time depends of the clock frequency and the number of clock cycles for every operation. The addition can be computed in one clock cycle because it is only a XOR combination of every bit. The squaring can be also computed in on clock cycle, a solution for this can be found in [14]. If we compute the multiplication serial we need *m* clock cycles. The speed for this operation can be increased but these results in a great area.

The last and also the operation which is the hardest to compute is the inversion. For the inversion there exist two algorithms one is the Extended Euclidian Algorithm and the other uses Fermat's Theorem. For the Extended Euclidian Algorithm exist a hardware solution which can be found in [15]. This solution computes the inversion in *m* clock cycles, but need a great amount of area. So this solution only should be used with algorithms which need many inversions. Another solution that computes the inversion with Fermat's Theorem is to find in [16]. This is a solution which computes the inversion with multiplications and squarings. With this we can summarize the number of clock cycles for every field operation:

$$\begin{aligned} t_A &= 1, \\ t_M &= m, \\ t_S &= 1, \\ t_{I,[15]} &= m, \\ t_{I,[16]} &= \left(\log_2(m-1) + Hw(m-1)\right)t_M + (m-1)t_S . \end{aligned} \tag{13}$$

Because of the great amount of area for the inversion with [15], only algorithms with affine coordinates should use this hardware solution. For algorithms with projective coordinates which only need on inversion is the solution from [16] a better choice. So we can compute the number of clock cycles for the different algorithms from sections 3, which are summarized in Table 8.

Table 8. Algorithms for Hardware Computation

Coordinates	Algorithm	Clock cycles for m =		
		163	191	239
Affine	DaAS	108266	148060	231156
	DaAS [9]	99246	135772	211956
	Montgomery	108077	148033	231121
Projective	DaAS [4]	276158	377452	588940
	DaAS [5]	195636	267436	416860
	DaAS [6]	306419	419407	654823
	Montgomery	163987	224816	350144

The fastest hardware solution for a scalar multiplication is to find in [17]. This hardware uses the Montgomery Algorithm with projective coordinates. But if we compare the algorithms in Table 8 we see, that the DaAS Algorithm with the improvement from [9] is the fastest algorithm. Therefore a hardware solution that uses this algorithm and the inversion with [15] can be speed up to the computation time up to 40%.

5. Conclusions

This paper compares the different algorithms that compute the scalar multiplication for the elliptic curve cryptography. Especially in section 4.2 we can show, that other algorithms than the general used are faster to compute the scalar multiplication in hardware. Therefore this paper should be known to everyone who wants to build a hardware or software solution which computes the scalar multiplication for the elliptic curve cryptography.

References

[1] N. Koblitz, Elliptic Curve Cryptosystems, *Mathematics of Computation*, Vol. 48, No. 177, pp. 203-209, 1987.

[2] V. Miller, Use of Elliptic Curves in Cryptography, *Advances in Cryptology - CRYPTO '85*, Springer-Verlag, LNCS 218, pp. 417-426, 1986.

[3] ANSI X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.

[4] IEEE Std. 1363-2000, *IEEE Standard Specifications for Public-Key Cryptography*, IEEE Computer Society, 2000.

[5] J. Lopez and R. Dahab, Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$, *Selected Areas in Cryptography - SAC '98*, Springer Verlag, LNCS 1556, pp. 201-212, 1999.

[6] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.

[7] G. W. Reitwiesner, Binary Arithmetic, *Advances in Computers*, Vol. 1, pp. 231-308, 1960

[8] O. L. MacSorley, High-Speed Arithmetic in Binary Computers, *Proceedings of the IRE*, pp. 67-91, 1961.

[9] K. Eisenträger, K. Lauter and P. L. Montgomery, Fast Elliptic Curve Arithmetic and Improved Weil Pairing Evaluation, *The Cryptographers' Track at the RSA Conference 2003 - CT-RSA 2003*, Springer-Verlag, LNCS 2612, pp. 343-354, 2003.

[10] P. L. Montgomery, Speeding the Pollard and Elliptic Curve Methods of Factorization, *Mathematics of Computation*, Vol. 48, No. 177, pp. 243-264, 1987.

[11] J. Lopez and R. Dahab, Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation, *Workshop on Cryptographic Hardware and Embedded Systems - CHES '99*, Springer Verlag, LNCS 1717, pp. 316-327, 1999.

[12] Cryptix Elliptix, Elliptix is intended to be a complete, 100% pure Java implementation of the IEEE P1363, ANSI X9.62 and ANSI X9.63 standards, Cryptix, http://www.cryptix.org/, 1999.

[13] LiDIA Group, LiDIA 2.1pre7, A Library for Computational Number Theory, TU-Darmstadt, http://www.informatik.tu-darmstadt.de/TI/LiDIA/, 2003.

[14] H. Wu, Low Complexity Bit-Parallel Finite Field Arithmetic Using Polynomial Basis, *Workshop on Cryptographic Hardware and Embedded Systems - CHES '99*, Springer Verlag, LNCS 1717, pp. 281-291, 1999.

[15] H. Brunner, A. Curiger and M. Hofstetter, On Computing Multiplicative Inverses in $GF(2^m)$, *IEEE Transactions on Computation*, Vol. 42, pp. 1010-1015, August 1993.

[16] T. Itoh and S. Tsujii, A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases, *Information on Computation*, Vol. 78, pp. 171-177, 1988.

[17] G. Orlando, C. Paar, A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$, *Workshop on Cryptographic Hardware and Embedded Systems - CHES 2000*, Springer-Verlag, LNCS 1965, pp. 41-56, 2000.