Elliptic Scalar Multiplication Using Point Halving

Erik Woodward Knudsen

De La Rue Card Systems
erik.knudsen@fr.delarue.com

Abstract. We describe a new method for conducting scalar multiplication on a non-supersingular elliptic curve in characteristic two. The idea is to replace all point doublings in the double-and-add algorithm with a faster operation called point halving.

1 Introduction

The security of cryptosystems like the Diffie-Hellman scheme is based on the intractability of the Discrete Logarithm Problem of the underlying group. For most elliptic curves defined over finite fields the Discrete Logarithm Problem is believed to be hard to solve and for this reason they are interesting in cryptography. The most time consuming part of the Diffie-Hellman key exchange protocol is multiplication of a point on the curve not known in advance by a random scalar. We will only discuss curves defined over fields of characteristic two; a popular choice for implementations since addition in such a field corresponds to the exclusive-or operation. It is known that scalar multiplication can be speeded up on a curve which is defined over a field of small cardinality ([Koblitz],[Meistaff],[Muller1]) using the Frobenius morphism. The curves can be chosen such that no known attack applies to them. However, at least principally, it is of course preferable to be able to choose the curve which one wants to use from as general a class of curves as possible. The method described in this paper applies in its fastest version to half of the elliptic curves. Moreover, from a cryptographic point of view it is the "better" half. Before giving the principle of the method we formulate the basic concepts. See for example [Silverman] for an introduction to the theory of elliptic curves.

Let n be a fixed integer. Let $\mathbf{F}_{2^n}$ denote the field with 2^n elements and let $\overline{\mathbf{F}}_{2^n}$ denote the algebraic closure of $\mathbf{F}_{2^n}$. Let $\mathcal{O}$ denote the point at infinity. By a non-supersingular elliptic curve E defined over $\mathbf{F}_{2^n}$ we mean the set

$$E = \{(x,y) \in \overline{\mathbf{F}}_{2^n} \times \overline{\mathbf{F}}_{2^n} \mid y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\} \qquad a, b \in \mathbf{F}_{2^n},\ b \neq 0$$

It is well known that E can be equipped with an abelian group structure where the point at infinity is the neutral element. It is customary to call the elements

© Springer-Verlag Berlin Heidelberg 1999

of E for points. We will work with the finite subgroup of E

$$E(\mathbf{F}_{2^n}) = \{(x,y) \in \mathbf{F}_{2^n} \times \mathbf{F}_{2^n} \mid y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\}$$

That is the $\mathbf{F}_{2^n}$-rational points in E. For any $m \in \mathbf{N}$ we can define the multiplication-by-m map

$$\begin{aligned} [m] : E &\rightarrow E \\ P &\mapsto \underbrace{P + \cdots + P}_{(m \text{ times})} \end{aligned}$$

and for $m = 0$:

$$\forall P \in E : \ [0]P = \mathcal{O}.$$

The kernel of the multiplication-by-m map is denoted by $E[m]$. The points of the group $E[m]$ are also called the m-torsion points of E. The group structure of the m-torsion points is well known. We will only be interested in the case where m is a power of two. In this case we have:

$$\forall k \in \mathbf{N} : \ E[2^k] \simeq \mathbf{Z}/2^k\mathbf{Z}$$

We will use the notation T_{2^k} for a point of order 2^k. Since T_2 is contained in $E(\mathbf{F}_{2^n})$ and since $E(\mathbf{F}_{2^n})$ is a finite subgroup of E it has the structure:

$$E(\mathbf{F}_{2^n}) = G \times \ E[2^k]$$

where G is a group of odd order and $k \geq 1$. When $k = 1$ we will say that the curve has minimal two-torsion.

After these preliminaries we are ready to explain the aim of the paper. The multiplication-by-two map, denoted by [2], which we will also call the doubling map, is not an injective function when defined on E or $E(\mathbf{F}_{2^n})$ because it has kernel $E[2] = \{\mathcal{O}, T_2\}$. On the other hand, if we restrict the domain of the doubling map to a subgroup $G \subset E(\mathbf{F}_{2^n})$ of odd order the map is a bijection. Consequently, on this subgroup the doubling map has an inverse map which we call the halving map:

$$\begin{aligned} [\frac{1}{2}] : G &\rightarrow G \\ P &\mapsto Q \qquad [2]Q = P \end{aligned}$$

We will write $[\frac{1}{2}]P$ for the point in G which the doubling map sends to P. For all $i \geq 1$ we will write

$$[\frac{1}{2^i}] := [\frac{1}{2}] \circ \cdots \circ [\frac{1}{2}]$$

for the i-fold composition of the halving map. The halving map is interesting in connection with elliptic scalar multiplication for the following reason: it is possible to replace all point doublings used when performing a scalar multiplication by point halvings. As we shall see the halving map is considerably faster to evaluate than the doubling map on a curve with minimal two-torsion when working in affine coordinates. From a cryptographic viewpoint it is good to have as many curves to choose from as possible and it is customary to use a curve for which the two-torsion of $E(\mathbf{F}_{2^n})$ is either minimal or isomorphic to $\mathbf{Z}/4\mathbf{Z}$. We shall see in Appendix A that for a given field $\mathbf{F}_{2^n}$ the curves with minimal two-torsion constitutes exactly half of all the curves defined over $\mathbf{F}_{2^n}$. Therefore, although not completely general, the method described applies in its fastest version to a large class of the curves which are interesting in connection with cryptography. The method can always be implemented if the field elements are represented in a normal basis. If a polynomial basis is used the storage requirements are in the order of magnitude $O(n^2)$ bits.

In section 2 we show how to compute $[\frac{1}{2}]P \in G$ from $P \in G$. In section 3 suggestions are given for fast computations. In section 4 we show how to replace doublings by halvings when performing a scalar multiplication and in section 5 we discuss the expected improvements in running time due to this replacement.

2 Point Halving

Representations of Points

We will use two representations: The usual affine representation of a point

$$P = (x, y)$$

and the representation

$$(x, \lambda_P) \qquad \text{where} \qquad \lambda_P = x + \frac{y}{x}.$$

From the second representation we can evaluate $y = x(x + \lambda_P)$ using one multiplication. The idea is then, when performing a scalar multiplication, to save field multiplications by performing intermediate results using the representation (x, λ_P) and only determining the second coordinate of the affine representation in the very end.

Point Halving

Given a point P in G we want to calculate $[\frac{1}{2}]P$. To do this let $P = (x, y) = (x, x(x+\lambda_P)) \in G$ and $Q = (u, v) = (u, u(u+\lambda_Q)) \in E(\mathbf{F}_{2^n})$ denote points such that $[2]Q = P$. The doubling formulas are given by ([IEEE]):

$$\lambda_Q = u + \frac{v}{u} \tag{1}$$

$$x = \lambda_Q^2 + \lambda_Q + a \tag{2}$$

$$y = (x + u)\lambda_Q + x + v \tag{3}$$

Multiplying (1) by u and substituting the value of v from (1) into (3) this is rewritten to:

$$\begin{aligned} v &= u(u+\lambda_Q) \\ \lambda_Q^2 + \lambda_Q &= a + x \\ y &= (x+u)\lambda_Q + x + u^2 + u\lambda_Q = u^2 + x(\lambda_Q + 1) \end{aligned}$$

Remembering that $y = x(x+\lambda_P)$ we get:

$$\begin{aligned} \lambda_Q^2 + \lambda_Q &= a + x && (i) \\ u^2 &= x(\lambda_Q+1) + y = x(\lambda_Q + \lambda_P + x + 1) && (ii) \\ v &= u(u+\lambda_Q) && (iii) \end{aligned}$$

With input $P = (x, y) = (x, x(x+\lambda_P))$ in either affine coordinates or the representation (x, λ_P) this system of equations determines the two points

$$[\frac{1}{2}]P \in G \qquad \text{and} \qquad [\frac{1}{2}]P + T_2 \in E(\mathbf{F}_{2^n})\backslash G$$

which are mapped to P by the doubling map. We want to be able to distinguish between them. We start by considering curves with minimal two-torsion:

Theorem 1. *Let E be a curve with minimal two-torsion. Let $P \in E(\mathbf{F}_{2^n}) = G \times \{\mathcal{O}, T_2\}$ be an element of odd order. Let Q be a point such that*

$$Q \in \{[\frac{1}{2}]P, [\frac{1}{2}]P + T_2\}$$

and let Q_1 denote either one of the two points in E for which $[2]Q_1 = Q$. We then have the necessary and sufficient condition

$$Q = [\frac{1}{2}]P \Leftrightarrow Q_1 \in E(\mathbf{F}_{2^n})$$

Proof. Q_1 is determined by applying the formulas (i),(ii) and (iii) to Q. Q equals either $[\frac{1}{2}]P$ or $[\frac{1}{2}]P + T_2$. By applying the formulas (i),(ii) and (iii) to $[\frac{1}{2}]P$ we get the two points

$$[\frac{1}{4}]P, [\frac{1}{4}]P + T_2 \in E(\mathbf{F}_{2^n})$$

which are in $E(\mathbf{F}_{2^n})$. Let T_4 and $[3]T_4$ denote the two points of order four in E. Applying the formulas (i),(ii) and (iii) to $[\frac{1}{2}]P + T_2$ we get

$$[\frac{1}{4}]P + T_4, [\frac{1}{4}]P + [3]T_4 \notin E(\mathbf{F}_{2^n})$$

The points are not in $E(\mathbf{F}_{2^n})$ because $T_4 \notin E(\mathbf{F}_{2^n}) = G \times \{\mathcal{O}, T_2\}$ □

Theorem 1 tells us that we on a curve with minimal two-torsion can check whether $Q = [\frac{1}{2}]P$ or $Q = [\frac{1}{2}]P + T_2$ by checking whether the coordinates of Q_1 are in $\mathbf{F}_{2^n}$ or in a field extension. Since Q_1 is determined by the equations

(i),(ii) and (iii) we examine these for operations which are not internal to the field. Solving the second degree equation in (i) is one such operation, and it is in fact the only one: it is true that we also have to calculate a square root to calculate the first coordinate of Q_1, but in characteristic two taking a square root is an operation internal to the field. We thus have:

$$Q = (u, v) = [\frac{1}{2}]P \Leftrightarrow \exists \lambda \in \mathbf{F}_{2^n} : \lambda^2 + \lambda = a + u$$

Since taking a square root is an operation internal to the field we can state this necessary and sufficient condition in another way:

$$Q = (u, v) = [\frac{1}{2}]P \Leftrightarrow \exists \lambda \in \mathbf{F}_{2^n} : \lambda^2 + \lambda = a^2 + u^2$$

Using this last condition will optimize the algorithm given below if the time to compute a square root is non-negligible.

Given $P \in G$, the two solutions to (i) are $\lambda_{[\frac{1}{2}]P}$ and $\lambda_{[\frac{1}{2}]P} + 1$ and we see from (ii) that the first coordinates of the corresponding candidates are u and $u + \sqrt{x}$. We have justified that we can calculate $[\frac{1}{2}]P$ in the following manner on a curve with minimal two-torsion:

Point Halving Algorithm

```
Input: $P = (x, y) = (x, x(x + \lambda_P)) \in G$ represented either as $(x, y)$ or as $(x, \lambda_P)$
Output: $[\frac{1}{2}]P = (u, v) \in G$ represented as $(u, \lambda_{[\frac{1}{2}]P})$
Method:
    1. Compute a solution $\lambda_{[\frac{1}{2}]P}$ from $(i)$.
    2. Compute the corresponding $u^2$ from $(ii)$.
    3. Check if there exists $\lambda \in \mathbf{F}_{2^n}$ such that $\lambda^2 + \lambda = a^2 + u^2$.
    4. If such a $\lambda$ does not exist then compute
         $u^2 := u^2 + x$ and $\lambda_{[\frac{1}{2}]P} := \lambda_{[\frac{1}{2}]P} + 1$
    5. Calculate $u := \sqrt{u^2}$.
    6. Output $(u, \lambda_{[\frac{1}{2}]P})$.
```

If a^2 is precomputed and stored the algorithm requires
solving 1 second degree equation (in 1)
1 multiplication (in 2)
1 check (in 3)
1 square root (in 5)

and if v is to be evaluated using (iii) one extra multiplication is required. We see that if we have to perform k consecutive halvings we can save $k - 1$ field multiplications by keeping the intermediate result in the representation (x, λ_P).

We now turn to the case of an arbitrary curve $E(\mathbf{F}_{2^n}) = G \times E[2^k]$. Let $P \in G$ and $Q \in \{[\frac{1}{2}]P, [\frac{1}{2}]P + T_2\}$ be given. We want to determine whether $Q = [\frac{1}{2}]P$ or $Q = [\frac{1}{2}]P + T_2$ and we can do this by repeating the procedure in the proof of Theorem 1. Apply the formulas (i) and (ii) k times: the first time to Q to get a point Q_1 such that $[2]Q_1 = Q$. In the i'th step apply the formulas to Q_{i-1} to get a point Q_i such that $[2]Q_i = Q_{i-1}$. The resulting point Q_k will be on the form $[\frac{1}{2^{k+1}}]P + T_{2^{k+1}}$ if and only if $Q = [\frac{1}{2}]P + T_2$ and it will be on the form $[\frac{1}{2^{k+1}}]P + T_{2^i}$ where $0 \le i \le k$ if and only if $Q = [\frac{1}{2}]P$. One thus has the necessary and sufficient condition:

$$Q = [\frac{1}{2}]P \Leftrightarrow Q_k \in E(\mathbf{F}_{2^n})$$

With the notation $Q = (u, v) = (u, u(u + \lambda_{[\frac{1}{2}]P}))$ and $Q_{k-1} = (u_{k-1}, v_{k-1})$, $[\frac{1}{2}]P$ is computed in the following manner: compute u_{k-1}^2 by repeated applications of steps 1,2 and 5 of the Point Halving Algorithm. Use u_{k-1}^2 in the check. If the check is negative put $u := u + \sqrt{x}$ and put $\lambda_{[\frac{1}{2}]P} := \lambda_{[\frac{1}{2}]P} + 1$. Finally, output $(u, \lambda_{[\frac{1}{2}]P})$. For a general curve the operations to be performed in a point halving which gives the output in the representation $(u, \lambda_{[\frac{1}{2}]P})$ are:
solving k second degree equations
k multiplications
1 check
k or $k+1$ square roots

3 Computing Efficiently

We show how to perform the check, solve the second degree equation and compute the square root used in the Point Halving Algorithm in an efficient way. By "efficient", we mean time efficient and not necessarily storage efficient. We consider both normal and polynomial bases. In a normal basis everything proceeds smoothly. In a polynomial basis we can likewise perform fast computations, but only if it is possible to store $O(n^2)$ bits.

Normal basis
The results given for the normal basis can be found in [IEEE]. We can view $\mathbf{F}_{2^n}$ as an n-dimensional vectorspace over $\mathbf{F}_2$. In a normal basis a field element is represented as

$$x = \sum_{i=0}^{n-1} x_i \beta^{2^i} \qquad x_i \in \{0, 1\}$$

where $\beta \in \mathbf{F}_{2^n}$ is chosen such that $\{\beta, \beta^2, \cdots, \beta^{2^{n-1}}\}$ is a basis for $\mathbf{F}_{2^n}$. The normal basis has the feature that computing a square root is done by a left cyclic shift and squaring by a right cyclic shift. The time to compute these operations is negligible.

Assume that the second degree equation $\lambda^2 + \lambda = x$ has solutions in $\mathbf{F}_{2^n}$. A solution is then given by

$$\lambda = \sum_{i=1}^{n-1} \lambda_i \beta^{2^i} \qquad \text{where} \qquad \lambda_i = \sum_{k=1}^{i} x_k \qquad \text{for all} \qquad 1 \le i \le n-1$$

We expect the time needed to compute this to be negligible compared to the time needed to compute a field multiplication or an inversion.

Since the time to compute a solution to a second degree equation is negligible we can compute the check in the following way: Compute a candidate λ from x and check if $\lambda^2 + \lambda = x$. If this is not the case then the equation has no solutions in $\mathbf{F}_{2^n}$.

Polynomial Basis
We will use the representation:

$$x = \sum_{i=0}^{n-1} x_i T^i \qquad x_i \in \{0, 1\}$$

The square root of x can be computed with the storage of the element $\sqrt{T}$ after making the following observations:

- in characteristic two the square root map is a field morphism.
- $\sqrt{\sum_{i\ even} x_i T^i} = \sum_{i\ even} x_i T^{\frac{i}{2}}$

Now, splitting x into even and odd powers and taking the square root, we get

$$\sqrt{x} = \sum_{i\ even} x_i T^{\frac{i}{2}} + \sqrt{T} \sum_{i\ odd} x_i T^{\frac{i-1}{2}}$$

so all we have to do to compute a square root is to "shrink" two vectors to half size and then perform a multiplication of a precomputed value with an element of length $\frac{n}{2}$. Therefore we expect the time to compute a square root in a polynomial basis to be equivalent to half the time to compute a field multiplication plus a very small overhead.

To perform the check and to solve the second degree equation we will view $\mathbf{F}_{2^n}$ as an n-dimensional vectorspace over $\mathbf{F}_2$. The map

$$\begin{aligned} F : \mathbf{F}_{2^n} &\rightarrow \mathbf{F}_{2^n} \\ \lambda &\mapsto \lambda^2 + \lambda \end{aligned}$$

is then a linear operator with kernel $\{0, 1\}$.

For a given x, the equation $\lambda^2 + \lambda = x$ has solutions in $\mathbf{F}_{2^n}$ if and only if the vector x is in the image of F. $Im(F)$ is an $n - 1$ dimensional subspace of $\mathbf{F}_{2^n}$. For a given basis of $\mathbf{F}_{2^n}$ with corresponding dot product there is a unique non-zero vector which is orthogonal to all vectors in $Im(F)$. Denote this vector w. We then have:

$$\exists \lambda \in \mathbf{F}_{2^n} : \ \lambda^2 + \lambda = x \ \Leftrightarrow \ x \bullet w = 0$$

so the check can be performed by adding up the entries of x for which the corresponding entries of w hold a 1. We expect the time to perform the check to be negligible.

To solve the second degree equation $F(\lambda) = \lambda^2 + \lambda = x$ in a polynomial basis we propose a straightforward method which requires capability to store an $n \times n$ matrix. We are looking for a linear operator G such that

$$\forall\, x \in Im(F):\ F(G(x)) = (G(x))^2 + G(x) = x$$

Let $\alpha \in \mathbf{F}_{2^n}$ be any vector such that $\alpha \notin Im(F)$ and define G by

$$G := \widetilde{F}^{-1} \qquad \text{where} \qquad \widetilde{F}(T^i) = \begin{cases} \alpha & \text{when } i = 0 \\ F(T^i) & \text{when } 1 \le i \le n-1 \end{cases}$$

With $x = \sum_{i=1}^{n-1} x_i F(T^i) \in Im(F)$ given it is left to the reader to verify that $G(x)$ solves the second degree equation. In an implementation one precomputes the matrix representation for G in the basis $\{1, T, \cdots, T^{n-1}\}$. In characteristic two, multiplication of a matrix by a vector is just adding up the columns of the matrix for which the corresponding entries of the vector hold a 1. Therefore, this method for solving a second degree equation on average requires $\frac{n}{2}$ field additions.

The drawback of the method is the storage needed. In appendix B, an algorithm is given which reduces the storage needed to $\frac{n^2}{2}$ bits. It is even faster requiring on average $\frac{n}{4}$ field additions and a small overhead.

We have one further remark on the storage before ending the section. We do not need to store the vector w needed for the check seperately, since it is the first row of the matrix representation of G. It follows from the fact that G is invertible and $G(F(T^i)) = T^i$ for $1 \le i \le n-1$. This implies that the first row of the matrix representation of G is non-zero and orthogonal to all column vectors of the matrix representation of F.

4 Applications for Scalar Multiplication

Let a point $P \in E(\mathbf{F}_{2^n})$ of odd order r and an integer c be given. Let m denote the integer part of $\log_2(r)$. We want to compute the scalar multiple $[c]P$ employing the halving map. For this purpose, we prove the easy:

Lemma 1. *For every integer c, there is a rational number of the form*

$$\sum_{i=0}^{m} \frac{c_i}{2^i} \qquad c_i \in \{0,1\}$$

such that

$$c \equiv \sum_{i=0}^{m} \frac{c_i}{2^i} \pmod{r}$$

Proof. Calculate the remainder of $2^m c$ after division by r and write the result as a binary number:

$$2^m c \pmod r = \sum_{i=0}^{m} \widehat{c}_i 2^i \qquad \widehat{c}_i \in \{0,1\}$$

Dividing by 2^m and putting $c_i := \widehat{c}_{m-i}$ gives the result:

$$\sum_{i=0}^{m} \frac{c_i}{2^i} := \sum_{i=0}^{m} \widehat{c}_i 2^{i-m} \qquad c_i \in \{0,1\}$$

□

Let $< P >$ denote the cyclic group generated by P. Since we have the isomorphism of rings:

$$\begin{array}{ccc} < P > & \simeq & \mathbf{Z}/r\mathbf{Z} \\ [k]\,P & \mapsto & k \end{array}$$

we can compute the scalar multiple by

$$[c]P = \sum_{i=0}^{m} [\frac{c_i}{2^i}]P$$

using point halvings and point additions. The well known double-and-add algorithm can be used for the computations. We only have to replace doublings by halvings in this algorithm. One has to perform $\log_2(r)$ halvings and on average $\frac{1}{2}\log_2(r)$ additions. There are improvements to the double-and-add algorithm which require only $\frac{1}{3}\log_2(r)$ additions in the average case. In appendix C, we give an automaton suitable for the halve-and-add algorithm. In general, any method which is based on manipulating an integer represented by its binary expansion should be easy to modify so as to make the same manipulations on the rational number from Lemma 1 represented by its $\frac{1}{2}$-adic expansion. We bear in mind here in particular the sliding window method.

For the addition of our original point P and the intermediate result Q, we use the following algorithm which is a small modification of the usual addition algorithm given in for example [IEEE]:

Addition Algorithm

Input: $P = (x, y)$ in affine coordinates and $Q = (u, u(u + \lambda_Q))$ represented as (u, λ_Q)

Output:$P + Q = (s, t)$ in affine coordinates

Method:

```
    1 Compute $\lambda := \frac{y+u(u+\lambda_Q)}{x+u}$
    2 Compute $s := \lambda^2 + \lambda + a + x + u$
    3 Compute $t := (s + x)\lambda + s + y$
    4 Output $(s, t)$
```

The algorithm requires 1 inversion,3 multiplications and 1 squaring.

5 Expected Performance

We will only consider curves with minimal two-torsion in this section. The time saved by using halvings instead of doublings is significant. In affine coordinates, both elliptic doubling and addition require 1 inversion, 2 multiplications and 1 squaring. If the scalar for the scalar multiplication is represented by a bitvector of length m with k non-zero entries the operations needed for the scalar multiplication are:

operation	double-and-add	halve-and-add
inversions	$m+k$	k
multiplications	$2m+2k$	$m+3k$
squarings	$m+k$	k
solving $\lambda^2+\lambda=a+x$	0	m
square roots	0	m
checks	0	m

Thus, by using halvings one saves m inversions, $m-k$ multiplications and m squarings at the cost of solving m second degree equations, calculating m square roots and performing m checks. We have shown how to compute the "new" operations fast. In the average case of the optimized version of the double-and-add algorithm and halve-and-add algorithm given in Appendix C we have $k=\frac{m}{3}$. In a polynomial basis, it is difficult to give a general estimate of the improvement in running time because of the many different operations involved. Based on [SOOS] we will put the time to compute an inversion equivalent to the time to compute 3 multiplications. A field multiplication in the average case requires $\frac{n}{2}$ field additions and afterwards a reduction by the reduction polynomial defining the field. With the following assumptions on equivalence of timings:

1 inversion $\sim$ 3 multiplications
1 multiplication $\sim$ 10 squarings
$1(\lambda^2+\lambda=a+x)$ + 1 check + 1 square root $\sim$ 1 multiplication + 1 squaring

we get an improvement in the running time on 39%. In a normal basis, as mentioned in Section 3, we assume that the time needed to calculate the square root, the check and the second degree equation is negligible compared to the time needed to compute a multiplication or an inversion. An inversion can be computed using $[\log_2(n-1)]+\omega(n-1)-1$ multiplications ([Menezes]), where n is the degree of the field extension and ω is the number of $1's$ in the binary expansion of $n-1$. As an example, with $n=155$ the number of multiplications needed for one inversion is 10. The improvement in running time is then 67%. Even with the time to compute an inversion being equivalent to the time to calculate 3 multiplications we get a 55% improvement of the running time.

6 Conclusion

A fast method for elliptic scalar multiplication has been introduced. In its fastest version it applies to half the curves: the ones with minimal two-torsion. For a polynomial basis, the disadvantage is the amount of storage needed. For a normal basis, there are no disadvantages. The algorithm is clearly superior to any double-and-add algorithm when this is implemented using affine coordinates. In [CLNZ] it is investigated how to reduce the amount of curve additions by representing the scalar by a bit vector which is longer, but has a lower Hamming-weight. This is of particular interest in this context since point halving is much faster than point addition. The current limitations of the method give rise to the challenges: Find a fast check for curves with higher two-torsion. Derive an efficient halving algorithm for projective coordinates. Reduce the storage needed in a polynomial basis.

7 Acknowledgements

The author would like to thank Kristian Pedersen, Jean-Bernard Fischer and Jacques Stern for fruitful comments.

References

MorOli. F. Morain and J. Olivos: Speeding up computations on an elliptic curve using addition-subtraction chains ln Theoretical Informatics and Applications 24, No. 6, 1990 pp.531-544 149

Zhang. C.N.Zhang: An improved binary algorithm for RSA ln Computers and Mathematics with Applications, vol. 25, 1993, pp.15-24 149

IEEE. *Standard Specifications for Public Key Cryptography, Annex A. Number Theoretic Background.* IEEE Standards Department, August 20, 1998. 137, 140, 143

Koblitz. N.Koblitz: CM-Curves with Good Cryptographic Properties, *Advances in Cryptology-CRYPTO 91,* Lecture Notes in Computer Science, No. 576, Springer-Verlag, Berlin, 1992, pp. 279-287. 135

Meistaff. W.Meier,O.Staffelbach: Efficient Multiplication on Certain Nonsupersingular Elliptic Curves, *Advances in Cryptology-CRYPTO 92,* Lecture Notes in Computer Science,No. 740, Springer-Verlag, Berlin, 1992, pp. 333-344. 135

Muller1. Volker Muller: Fast Multiplication on Elliptic Curves over Small Fields of Characteristic Two, *Journal of Cryptology 1998,* pp. 219-234. 135

Silverman. J.Silverman: *The arithmetic of Elliptic Curves,* Graduate Texts in Mathematics **106**, Springer-Verlag, Berlin Heidelberg New York 1986. 135

CLNZ. G.Cohen,A.Lobstein,D.Naccache,G.Zemor: *How to Improve an Exponetiation Black-box,* Technical Report AP03-1998, Gemplus' Corporate Product R&D Division 145

Muller2. Volker Muller: *Efficient Algorithms for Multiplication on Elliptic Curves* TI-9/97,1997, Institut fur theoretische Informatik 149

Menezes. Alfred J. Menezes: *Elliptic Curve Public Key Cryptosystems* Kluwer Acedemic Publishers 144

SOOS. R. Schroeppel, H. Orman, S. O'Malley, O. Spatscheck: Fast Key Exchange with Elliptic Curve Systems, *Advances in cryptology - CRYPTO '95, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed., Springer-Verlag, 1995* 144

A Half the Curves Have Minimal Two-Torsion

Theorem 2. *Let a field* $\mathbf{F}_{2^n}$ *be given. Half the curves defined on* $\mathbf{F}_{2^n}$ *have minimal two-torsion.*

Proof. As mentioned in the introduction, a non-supersingular curve E is defined by an equation

$$y^2 + xy = x^3 + ax^2 + b \qquad a, b \in \mathbf{F}_{2^n} \qquad b \neq 0$$

That is, it is defined by the pair $(a, b) \in \mathbf{F}_{2^n} \times \mathbf{F}_{2^n}$. The unique point of order two is given by $T_2 = (0, \sqrt{b})$. With T_2 as input, we can calculate the two points of order four by the equations $(i), (ii), (iii)$ in Section 2. Therefore, by repeating the arguments of Theorem 1 and the analysis afterwards leading to the check in the Point Halving Algorithm, we have with T_2 as input in equation (i):

$$T_4, [3]T_4 \in E(\mathbf{F}_{2^n}) \Leftrightarrow \exists \lambda \in \mathbf{F}_{2^n} : \lambda^2 + \lambda = a$$

Let F denote the linear operator $F(\lambda) = \lambda^2 + \lambda$ with domain $\mathbf{F}_{2^n}$. We negate the necessary and sufficient condition and get:

$$E \text{ has minimal two-torsion} \Leftrightarrow a \notin Im(F)$$

Since F has kernel $\{0, 1\}$, this condition holds for 2^{n-1} values of a, thus half the curves. □

B Reducing Storage in a Polynomial Basis

Assume that the equation $F(\lambda) = \lambda^2 + \lambda = x$ has solutions in $\mathbf{F}_{2^n}$. As explained in Chapter 3, the solutions can be computed with the storage of an $n \times n$ matrix representation of a linear operator G, where $G(x)$ is a solution to the second degree equation. In this Appendix we show how to reduce the storage needed. The idea is to write x as

$$x = F(y) + z$$

where $y = \sum_{i=1}^{n-1} y_i T^i$ and where z is an element of the subspace of $\mathbf{F}_{2^n}$ generated by the vector 1 and the vectors $\{T^i\}$ where i is odd. Define $\widetilde{G}$ by

$$\widetilde{G}(T^i) = \begin{cases} G(1) & \text{if } i = 0 \\ G(T^i) & \text{if } i = 1, 3, 5, \ldots \\ 0 & \text{if } i = 2, 4, 6, \ldots \end{cases}$$

Then $\widetilde{G}(z) = G(z)$. It follows from the definition of G that $G(F(T^i)) = T^i$ for all $1 \leq i \leq n - 1$. Therefore a solution to the second degree equation is given by

$$G(x) = G(F(y) + z) = G(\sum_{i=1}^{n-1} y_i F(T^i)) + G(z) = y + G(z) = y + \widetilde{G}(z)$$

So, using this approach, we only have to store the $[\frac{n}{2}] + 1$ nontrivial vectors of the matrix representation of $\widetilde{G}$. Let x be represented as

$$x = \sum_{i=0}^{n-1} x_i T^i$$

and assume for simplicity that $n-1$ is a power of two. The following algorithm calculates a solution to the second degree equation using $\log_2(n-1)$ iterations.

```
Input: $x \in Im(F)$
Output: $\lambda$ such that $\lambda^2 + \lambda = x$
Method:
    $y := 0$
    $m := n - 1$
    while $m > 2$ do
        $u := \sum_{i=\frac{m}{4}+1}^{\frac{m}{2}} x_{2i} T^i$
        $x := x + F(u)$
        $y := y + u$
        $m := \frac{m}{2}$
    od
    $u := x_2 T$
    $x := x + F(u)$
    $y := y + u$
    $\lambda := y + \widetilde{G}(x)$
```

Theorem 3. *The algorithm works.*

Proof. Define values

$$a_k := \begin{cases} \frac{n-1}{2^k} & \text{when } 0 \leq k \leq \log_2(n-1) \\ 0 & \text{when } k = 1 + \log_2(n-1) \end{cases}$$

$$x^{(0)} := x$$

$$y^{(0)} := 0$$

and recursively for $1 \leq k \leq \log_2(n-1)$:

$$x^{(k)} = x^{(k-1)} + F(\sum_{i=1+a_{k+1}}^{a_k} x_{2i}^{(k-1)} T^i)$$

$$y^{(k)} = y^{(k-1)} + \sum_{i=1+a_{k+1}}^{a_k} x_{2i}^{(k-1)} T^i$$

corresponding to the operations performed in the algorithm. Using the fact that F is a linear operator, it is easily seen that we have for all $0 \leq k \leq \log_2(n-1)$:

$$x = F(y^{(k)}) + x^{(k)}$$

In particular, this is true for $k := \log_2(n-1)$. It is immediate from the recursion formula defining $y^{(k)}$ that the constant term of $y^{(log_2(n-1))}$ is zero. It remains to show that all even coefficients of $x^{(\log_2(n-1))}$ of index greater than or equal to two are zero. Then, a solution can be calculated by $y^{(\log_2(n-1))} + \widetilde{G}(x^{(\log_2(n-1))})$ which is the final step of the algorithm. With $2+a_k \leq n-1$ we show by induction on k that

$$x^{(k)}_{2+a_k} = x^{(k)}_{4+a_k} = \cdots = x^{(k)}_{n-1} = 0 \qquad \text{for all} \qquad 0 \leq k < \log_2(n-1)$$

For $k = 0$ this is trivially true. Assume that the statement holds for $k-1$. We then have:

$$\begin{aligned} x^{(k)} &= x^{(k-1)} + \sum_{i=1+a_{k+1}}^{a_k} x^{(k-1)}_{2i} T^{2i} + \sum_{i=1+a_{k+1}}^{a_k} x^{(k-1)}_{2i} T^{i} \\ &= \left(x^{(k-1)} + \sum_{i\ even,\ i=2+a_k}^{a_{k-1}} x^{(k-1)}_{i} T^{i} \right) + \sum_{i=1+a_{k+1}}^{a_k} x^{(k-1)}_{2i} T^{i} \end{aligned}$$

It is clear that the coefficients of the expression in the parenthesis are zero for even indices $2 + a_k, \cdots, a_{k-1}$. By the induction assumption, the same is true for the even indices $2 + a_{k-1}, \cdots, n-1$. Adding the term outside the parenthesis does not affect basis vectors of index greater than a_k and this completes the induction. Finally, for the last iteration in the algorithm with $k = \log_2(n-1)$, we get:

$$x^{(k)} = (x^{(k-1)} + x^{(k-1)}_2 T^2) + x^{(k-1)}_2 T$$

from which we see that $x^{(log_2(n-1))}_2 = 0$ and we are done. □

Each iteration in the algorithm is fast. The kth iteration consists in removing the relevant coefficients of even index from x, "squeezing" them to a vector of length $\frac{n-1}{2^{k+1}}$ and then add this vector to x and y. It is left to the reader to see that the addition in the reassigning of y is not really an addition but a concatenation. Therefore the total amount of field additions in the loop corresponds to the addition of a vector of length $\frac{n}{2}$. We can thus expect the running time of the algorithm to be a small overhead plus, on average, $\frac{n}{4}$ field additions from the final multiplication of the matrix representation of $\widetilde{G}$ with x.

Using the same idea, one can hope to get the amount of storage even further down by exploiting the specific properties of the reduction polynomial. More precisely, the idea is that one can avoid to store a column vector of high index k if the degree of T^{2k} after reduction by the reduction polynomial is less than k. We can then once more write $x = (x + F(x_k T^k)) + F(x_k T^k)$ and calculate a solution by $G(x) = G(x + F(x_k T^k)) + x_k T^k$ which does not involve computing $G(T^k)$. We will not pursue this further.

C An Optimized Version of the Halving-and-add Algorithm

We give below an automaton which takes as input a point P of odd order and a bitvector $(c_0, \cdots, c_m)$ and outputs $\sum_{i=0}^{m}[\frac{c_i}{2^i}]P$. The basic idea, which we have from [MorOli], is to minimize the number of curve additions by applying the following identities of strings to the bitvector:

$$\underbrace{1\cdots1}_{k} = 1\underbrace{0\cdots0}_{k-1}{-1}$$

and

$$\underbrace{1\cdots1}_{k_1}\,0\,\underbrace{1\cdots1}_{k_2} = \underbrace{1\cdots1}_{k_1+1}\,\underbrace{0\cdots0}_{k_2-1}\,{-1} = 1\,\underbrace{0\cdots0}_{k_1}{-1}\,\underbrace{0\cdots0}_{k_2-1}\,{-1}$$

In this way one can always obtain an identity of rational numbers:

$$\sum_{i=1}^{m}\frac{c_i}{2^i} = \sum_{i=0}^{m}\frac{d_i}{2^i}, \qquad d_i \in \{0, \pm 1\}$$

where the coefficients d_i have the further feature that $\forall i \in \{0, \cdots, m-1\} : \ d_i \neq 0 \Rightarrow d_{i+1} = 0$. It is proven in for example [Zhang] that the amount of non-zero coefficients d_i on average will be one third of m. Since the time to calculate $-P$ from P is negligible, we can now calculate

$$\sum_{i=0}^{m}[\frac{c_i}{2^i}]P = [c_0]P + \sum_{i=1}^{m}[\frac{c_i}{2^i}]P = [c_0]P + \sum_{i=0}^{m}[\frac{d_i}{2^i}]P$$

using fewer curve additions and still m halvings. In [Muller2] is given an automaton to be used in an optimization of the Double-and-add Algorithm. It is identical to the automaton given below except for the last bit. For the correctness of the automaton given here we therefore refer to [Muller2]. The automaton is to be used on the bits $c_m, \cdots, c_0$ in descending order of indices. The arrows pointing out are for the calculations regarding the final bit c_0.