Small Scalar Multiplication on Weierstrass Curves using Division Polynomials

Sergey Agievich, Stanislav Poruchnik, Vladislav Semenov

Research Institute for Applied Problems of Mathematics and Informatics

Belarusian State University

agievich@{bsu.by,gmail.com}, semenov.vlad.by@gmail.com, poruchnikstanislav@gmail.com

Abstract

This paper deals with elliptic curves in the short Weierstrass form over large prime fields and presents algorithms for computing small odd multiples of a given point P on a curve. Our algorithms make use of division polynomials and are more efficient than the naive algorithm based on repeated additions with $2P$. We show how to perform scalar multiplication in the variable base settings using the precomputed small multiples. By employing the window method and avoiding conditional branches, we achieve the constant-time property for the final scalar multiplication algorithm. Small multiples are computed in either Jacobian or affine coordinates. To obtain affine coordinates, we use Montgomery's trick of simultaneous multiplicative inversion of several field elements. The conversion to affine coordinates slows down the precomputations but speeds up the main scalar multiplication loop. We show that the conversion makes sense and gives an overall performance boost in practical settings.

Keywords: elliptic curve, short Weierstrass form, division polynomial, scalar multiplication.

1 Preliminaries

Elliptic curves in the short Weierstrass form are historically the first curves of ECC (Elliptic Curve Cryptography, [6]). They appeared in the pioneering papers by V. Miller [12] and N. Koblitz [9], they formed the basis of a dozen cryptography standards.

Comparing the efficiency of arithmetic operations, Weierstrass curves are not the fastest. They are inferior to Montgomery and Edwards curves [13, 2, 3, 4], the current champions. Despite this, Weierstrass curves continue to be widely used. It is not only due to legacy issues. Unlike Montgomery and Edwards curves, Weierstrass curves can possess the cofactor-one property, which makes them very convenient for use in various cryptographic protocols.

Let $\mathbb{F}$ be a large prime finite field. An elliptic curve over $\mathbb{F}$ in the short Weierstrass form is defined by the equation

$$E\colon y^2 = x^3 + ax + b, \quad a, b \in \mathbb{F}, \quad 4a^3 + 27b^2 \neq 0.$$

Affine points of the curve, that is, pairs $(x, y) \in \mathbb{F}^2$ satisfying E, are added using special rules. A resulting sum can be a special point at infinity, denoted by O, and this point can also be a summand. The addition is defined in such a way that affine points complemented by O form an abelian group. In this group, O is zero and $(x, -y)$ is the inverse of (x, y).

Let G be an affine point, $\mathbb{G} = \langle G \rangle$ be the corresponding cyclic group, q be an order of $\mathbb{G}$ and l be the length of q in bits. In cryptography, (E, G) are chosen so that q is a large prime close to $|\mathbb{F}|$. We further assume that this is indeed the case.

The main operation of ECC is scalar multiplication, that is, computing dP given $P \in \mathbb{G} \setminus \{O\}$ and $d \in \{1, 2, \ldots, q-1\}$. We consider the so-called *variable base* settings, when P is volatile and precomputations with it are not possible. These settings cover constructing shared secrets in Diffie-Hellman-type protocols or verifying signatures of ElGamal and Schnorr types. The scalar d is often secret.

Computations with points of an elliptic curve are reduced to computations with their coordinates (elements of $\mathbb{F}$), which are described by arithmetic (over $\mathbb{F}$) circuits. The main contribution to the circuit complexity is made by the multiplicative operations: $\mathtt{I}$ — inversion, $\mathtt{M}$ — multiplication of arbitrary elements, $\mathtt{S}$ — squaring. The notation $i\mathtt{I} + m\mathtt{M} + s\mathtt{S}$ means that a circuit contains i operations $\mathtt{I}$, m operations $\mathtt{M}$ and s operations $\mathtt{S}$. For example, on Weierstrass curves, the addition of affine points and their doubling can be done with the complexity $1\mathtt{I} + 2\mathtt{M} + 1\mathtt{S}$ and $1\mathtt{I} + 2\mathtt{M} + 2\mathtt{S}$ respectively.

The operation $\mathtt{I}$ is the most expensive, for practical dimensions its complexity is 80–100 times higher than that of $\mathtt{M}$. To reduce the use of $\mathtt{I}$, affine points (x, y) are converted into projective points $(X, Y, Z) \in \mathbb{F}^3$. We use Jacobian projective points: $X/Z^2 = x$, $Y/Z^3 = y$. The coordinate Z acts as a normalizing factor that "absorbs" the inconvenient operation $\mathtt{I}$. On a Weierstrass curve with $a = -3$ (this is the optimal choice), the operations $J \leftarrow J + J$ (addition of two non-equivalent Jacobian points providing the Jacobian result), $J \leftarrow 2J$ (doubling of a Jacobian point), $J \leftarrow J + A$ (addition of a Jacobian point with a non-equivalent affine point) can be done with the complexity $11\mathtt{M} + 5\mathtt{S}$, $3\mathtt{M} + 5\mathtt{S}$ and $7\mathtt{M} + 4\mathtt{S}$ respectively.

There are many methods for scalar multiplication $(d, P) \mapsto dP$. In this paper, we use the window method which consists of two stages:

I. For a small *window width* w and a *base* $\mathcal{B} \subseteq \{\pm 1, \pm 2, \ldots, \pm(2^w - 1)\}$, the small multiples $\{nP \colon n \in \mathcal{B}\}$ are computed.

II. The resulting point dP is computed using point doublings and additions with small multiples. A sequence of multiples added at each step is determined using a *recording* algorithm that processes the binary representation of d. The bits of d are processed either from right (less significant) to left (more significant) or from left to right.

We propose algorithms that implement both stages of the window method with $\mathcal{B} = \{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$. This base was introduced in [14] and then used in [7] accompanied by a different recording (*recoding* in the original) algorithm. Both recording algorithms of [14] and [7] are right-to-left. We use the left-to-right direction. This direction is not the best (see discussion in [7]) from the perspective of power analysis, a branch of side-channel attacks, but in this paper, we do not take power analysis into consideration.

Our algorithms of the first stage are based on division polynomials. Their use for scalar multiplication was proposed by V. Miller in the already mentioned paper [12]. It appears that Miller's proposal was first implemented in [8] by adapting an algorithm for computing elliptic nets from [16]. The proposal is not used in practice since the resulting circuits for computing dP have high complexity. On the other hand, the circuits for computing small multiples of P have acceptable complexity and, moreover, these circuits are more efficient than the naive circuit based on repeated additions with $2P$ (see below). Note that we use division polynomials in a rather straightforward manner compared to a more sophisticated approach of [8, 16] where the double-and-add method based on division polynomials is actually designed.

In our algorithm of the second stage, conditional branches are avoided. In cryptography,

algorithms without branches are called *constant-time*. Only constant-time algorithms are considered safe since branches usually induce fluctuations of the runtime with potential leakage of sensitive data (in our case, bits of d).

The small multiples of P that are computed at the first stage and used at the second stage can be either Jacobian or affine points. To compute affine points, we use a circuit for simultaneous inversion of several field elements with only one $\mathtt{I}$ operation and some extra multiplications. This circuit was proposed by P. Montgomery in [13] and is often referred to as Montgomery's trick. When using affine points, the first stage is slower but the second stage is faster.

The following table summarizes the complexity of the proposed algorithms applying for Weierstrass curves in the short form with $a = -3$. In the table, $k = \lceil l/w \rceil$. The second column presents the form of the small multiples: in Jacobian coordinates (J) or in affine (A). The complexity of the first stage actually covers the computation of only $2^{w-1} - 1$ points nP, $n = 3, 5, \ldots, 2^w - 1$, since the remaining points of the form nP, $n \in \mathcal{B}$, can be computed through field negations, that is, without multiplicative operations.

Stage	Small multiples	Complexity
I	J	$(19 \cdot 2^{w-2} - 11)\mathtt{M} + (7 \cdot 2^{w-2} - 1)\mathtt{S}$
	A	$\mathtt{I} + (25 \cdot 2^{w-2} - 13)\mathtt{M} + (5 \cdot 2^{w-1} - 3)\mathtt{S}$
II	J	$\mathtt{I} + (3(k-1)w + 11k - 2)\mathtt{M} + (6(k-1)w + 5k - 8)\mathtt{S}$
	A	$\mathtt{I} + (3(k-1)w + 8k + 1)\mathtt{M} + (6(k-1)w + k - 1)\mathtt{S}$

Let us discuss the first line of the table. It means that the computation of one small multiple takes time $\approx \frac{19}{2}\mathtt{M} + \frac{7}{2}\mathtt{S}$. On the other hand, the usual algorithm for computing small multiples is to compute the Jacobian point $2P$ and then add it with P, $3P$, $5P$, and so on. Each addition, except the first, is performed in time $\approx 11\mathtt{M} + 5\mathtt{S}$. As we can see, using division polynomials to compute small multiples gives improvement.

The rest of the paper is organized as follows. In Section 2 we recall the notion of division polynomials and outline a way to use them for computing small multiples. The exact algorithms and their complexity are provided in Section 3. In Section 4 we present the constant-time window algorithm for scalar multiplication in variable base settings. Using popular heuristics about the complexity ratio between $\mathtt{I}$, $\mathtt{M}$, $\mathtt{S}$ and additive field operations, we estimate the optimal window width for $l = 256, 384, 512$ and different coordinate systems of small multiples (Jacobian or affine). It turns out that affine coordinates are better in all considered cases.

2 Division polynomials

Division polynomials $\psi_n(x, y)$ describe the coordinates of multiples of an elliptic curve point. More precisely, if $P = (x, y)$ is an affine point, $n \geq 2$ and $nP \neq O$, then

$$nP = \left(x - \frac{\psi_{n-1}(x,y)\psi_{n+1}(x,y)}{\psi_n(x,y)^2}, \frac{\psi_{n+2}(x,y)\psi_{n-1}(x,y)^2 - \psi_{n-2}(x,y)\psi_{n+1}(x,y)^2}{4y\psi_n(x,y)^3} \right).$$

The polynomials $\psi_n(x, y)$ are computed recursively. It would be convenient to define the recursion in terms of auxiliary polynomials $W_n(x, y^2)$ such that

$$\psi_n(x,y) = \begin{cases} 2yW_n(x,y^2), & n \text{ is even}, \\ W_n(x,y^2), & n \text{ is odd}. \end{cases}$$

Assuming (x, y) is fixed, denote $W_n = W_n(x, y^2)$. With that,

$$W_n = \begin{cases} -1, & n = -1, \\ 0, & n = 0, \\ 1, & n = 1, \\ 1, & n = 2, \\ 3(x^2+a)^2 + 4(3bx - a^2), & n = 3, \\ 2(x^4(x^2+5a) + bx(20x^2 - 4a) - 5a^2x^2 - 8b^2 - a^3), & n = 4, \\ W_m \left(W_{m+2}W_{m-1}^2 - W_{m-2}W_{m+1}^2\right), & n = 2m, \\ ((2y)^4 W_m W_{m+2})\, W_m^2 - (W_{m-1}W_{m+1})\, W_{m+1}^2, & n = 4k+1, \quad m = 2k, \\ (W_m W_{m+2})\, W_m^2 - ((2y)^4 W_{m-1}W_{m+1})\, W_{m+1}^2, & n = 4k+3, \quad m = 2k+1. \end{cases}$$

For odd n, the affine coordinates of the point nP can be represented as follows:

$$\begin{aligned} nP &= \left(\frac{X_n}{W_n^2}, \frac{Y_n}{W_n^3}\right), \\ X_n &= xW_n^2 - (2y)^2 W_{n-1}W_{n+1}, \\ Y_n &= y\left(W_{n+2}W_{n-1}^2 - W_{n-2}W_{n+1}^2\right). \end{aligned}$$

Besides, we automatically get the Jacobian form of nP: $nP = (X_n, Y_n, W_n)$.

Carrying out the computation for n up to $2^w - 1$, we obtain an arithmetic circuit with two non-constant inputs, that is, the affine coordinates of P, and $3 \cdot (2^{w-1} - 1)$ outputs, that is, the Jacobian coordinates of the points $3P, 5P, \ldots, (2^w - 1)P$.

The resulting circuit can be extended with a circuit taking Jacobian coordinates and returning affine ones. In the additional circuit, we use the following algorithm proposed by P. Montgomery and mentioned in Section 1. The algorithm inverts k nonzero field elements in time $\mathtt{I} + (3k - 3)\mathtt{M}$.

Algorithm `MontInv`

Input: $(u_1, \ldots, u_k)$, $u_i \in \mathbb{F}$, $u_i \neq 0$.
Output: $(u_1^{-1}, \ldots, u_k^{-1})$.
Steps:

1. $v_1 \leftarrow u_1$.
2. For $i = 2, \ldots, k$: $v_i \leftarrow v_{i-1}u_i$.
3. $t \leftarrow v_k^{-1}$.
4. For $i = k, k-1, \ldots, 2$:
   (1) $u_i^{-1} \leftarrow v_{i-1}t$;
   (2) $t \leftarrow tu_i^{-1}$.
5. $u_1^{-1} \leftarrow t$.
6. Return $(u_1^{-1}, \ldots, u_k^{-1})$.

Further, to slightly simplify the transition from Jacobian to affine coordinates, we use the following form of nP:

$$\begin{aligned} nP &= \left(\frac{X_n}{W_n^2}, \frac{Y'_n}{W_n^4}\right), \\ Y'_n &= y\left(W_n W_{n+2}W_{n-1}^2 - W_{n-2}W_n W_{n+1}^2\right). \end{aligned}$$

3 Small scalar multiplication

3.1 Jacobian coordinates

Let us analyze equations for computing the Jacobian points nP, $n = 3, 5, \ldots, 2^w - 1$. We make a list of intermediate and final expressions to be computed, determine which expressions need to be computed before others, count the arithmetic complexity of the computations. We count not only multiplicative operations of the field $\mathbb{F}$ but also simpler ones: $\mathtt{A}$ — addition or subtraction, $\mathtt{m}$ — multiplication by a small (≤ 5) constant, $\mathtt{half}$ — division by 2. Multiplication by the curve coefficient a or b is treated as the operation $\mathtt{M}$. We suppose that the expressions a^2, $a^3 + 8b^2$ are precomputed and can be used along with a and b at no cost.

The results of the analysis are given in Table 1.

Table 1: Computing small multiples in Jacobian coordinates

Expression(s)	n	Require(s)	Complexity
W_n	$1, 2, 3, 4$	a, b, a^2, $a^3 + 8b^2$, x	$4\mathtt{M} + 3\mathtt{S} + 7\mathtt{m} + 10\mathtt{A}$
$y^2, (2y)^2$		y	$1\mathtt{S} + 1\mathtt{m}$
W_{2n}	$3, 4, \ldots, 2^{w-1}$	W_n, $W_{n+2}W_{n-1}^2 - W_{n-2}W_{n+1}^2$	$(2^{w-1} - 2)\mathtt{M}$
W_{2n+1}	$2, 3, \ldots, 2^{w-1}$	$(2y)^4 W_n W_{n+2}$, W_n^2, W_{n+1}^2, $W_{n-1}W_{n+1}$	$(2^{w-1} - 1)(2\mathtt{M} + \mathtt{A}) - \mathtt{M}$
W_n^2	$1, 2, \ldots, 2^w$	W_n	$(2^w - 2)\mathtt{S}$
$W_n W_{n+2}$	$1, 2, \ldots, 2^{w-1}$ $2, 4, \ldots, 2^w - 2$	W_n, W_{n+2}, W_n^2, W_{n+2}^2	$(3 \cdot 2^{w-2} - 3)(\mathtt{S} + 3\mathtt{A} + \mathtt{half})$
$(2y)^2 W_n W_{n+2}$	$2, 4, \ldots, 2^w - 2$	$(2y)^2$, $W_n W_{n+2}$	$(2^{w-1} - 1)\mathtt{M}$
$(2y)^4 W_n W_{n+2}$	$2, 4, \ldots, 2^{w-1}$	$(2y)^2$, $(2y)^2 W_n W_{n+2}$	$2^{w-2}\mathtt{M}$
$W_{n+2}W_{n-1}^2 - W_{n-2}W_{n+1}^2$	$3, 4, \ldots, 2^{w-1}$ $3, 5, \ldots, 2^w - 1$	W_{n-2}, W_{n+2}, W_{n-1}^2, W_{n+1}^2	$(3 \cdot 2^{w-1} - 7)\mathtt{M} +$ $(3 \cdot 2^{w-2} - 2)\mathtt{A}$
X_n	$3, 5, \ldots, 2^w - 1$	x, W_n^2, $(2y)^2 W_{n-1} W_{n+1}$	$(2^{w-1} - 1)(\mathtt{M} + \mathtt{A})$
Y_n	$3, 5, \ldots, 2^w - 1$	y, $W_{n+2}W_{n-1}^2 - W_{n-2}W_{n+1}^2$	$(2^{w-1} - 1)\mathtt{M}$

Calculating the complexity, we take into account the following simplifications:

$$
\begin{aligned}
&W_5 = (2y)^4 W_2 W_4 - W_3 W_3^2, \\
&W_1^2 = W_2^2 = 1, \\
&W_n W_{n+2} = ((W_n + W_{n+2})^2 - W_n^2 - W_{n+2}^2)/2, \\
&W_1 W_3 = W_3, \\
&W_2 W_4 = W_4, \\
&W_5 W_2^2 - W_1 W_4^2 = W_5 - W_4^2, \\
&W_6 W_3^2 - W_2 W_5^2 = W_6 W_3^2 - W_5^2.
\end{aligned}
$$

The total complexity of the circuit for computing small multiples in Jacobian coordinates:

$$(19 \cdot 2^{w-2} - 11)\mathtt{M} + (7 \cdot 2^{w-2} - 1)\mathtt{S} + 8\mathtt{m} + (2^{w+2} - 3)\mathtt{A} + (3 \cdot 2^{w-2} - 3)\mathtt{half}.$$

The circuit is presented in detail in the algorithm `SmallMultJ`. Here we take into account that certain expressions are used multiple times. Such expressions are cached implicitly, that

is, they are stored locally and then reused without recalculation. Cached expressions are enclosed in square brackets. Expressions in the brackets to the left of $\leftarrow$ are inserted into the cache, and expressions to the right are retrieved from it.

Algorithm `SmallMultJ`

Input: $P = (x, y) \in \mathbb{G} \setminus \{O\}$, w $(3 \leq w < \log_2 q)$.

Output: $3P, 5P, \ldots, (2^w - 1)P$ (in Jacobian coordinates).

Steps:

1. $[y^2] \leftarrow y^2$.
2. $[(2y)^2] \leftarrow 4 \cdot [y^2]$.
3. Compute W_3:
   (1) $[x^2] \leftarrow x^2$;
   (2) $[bx] \leftarrow b \cdot x$;
   (3) $[W_3] \leftarrow 3 \cdot ([x^2] + a)^2 - 4 \cdot ([a^2] - 3 \cdot [bx])$.
4. Compute W_4:
   (1) $[ax] \leftarrow a \cdot x$;
   (2) $[x^3] \leftarrow [y^2] - [ax] - b$;
   (3) $[W_4] \leftarrow 2 \cdot ([x^3]^2 + 4 \cdot [bx] \cdot (5 \cdot [x^2] - a) + 5 \cdot [ax] \cdot ([x^3] - [ax]) - [a^3 + 8b^2])$.
5. $[W_2^2] \leftarrow 1$.
6. $[W_3^2] \leftarrow [W_3]^2$.
7. $[W_4^2] \leftarrow [W_4]^2$.
8. $[W_1 W_3] \leftarrow [W_3]$.
9. $[W_2 W_4] \leftarrow [W_4]$.
10. $[(2y)^2 W_2 W_4] \leftarrow [(2y)^2] \cdot [W_2 W_4]$.
11. $[(2y)^4 W_2 W_4] \leftarrow [(2y)^2] \cdot [(2y)^2 W_2 W_4]$.
12. $[W_5] \leftarrow [(2y)^4 W_2 W_4] - [W_1 W_3] \cdot [W_3^2]$.
13. $[W_5^2] \leftarrow ([W_5])^2$.
14. $[W_5 W_2^2 - W_1 W_4^2] \leftarrow [W_5] - [W_4^2]$.
15. $[W_6 W_3^2 - W_2 W_5^2] \leftarrow [W_6] \cdot [W_3^2] - [W_5^2]$.
16. For $n = 3, 4, \ldots, 2^{w-1}$:
   (1) if $n \geq 5$:
   $[W_{n+2} W_{n-1}^2 - W_{n-2} W_{n+1}^2] \leftarrow [W_{n+2}] \cdot [W_{n-1}^2] - [W_{n-2}] \cdot [W_{n+1}^2]$;
   (2) $[W_{2n}] \leftarrow [W_n] \cdot [W_{n+2} W_{n-1}^2 - W_{n-2} W_{n+1}^2]$;
   (3) $[W_{2n}^2] \leftarrow ([W_{2n}])^2$;
   (4) $[W_n W_{n+2}] \leftarrow (([W_n] + [W_{n+2}])^2 - [W_n^2] - [W_{n+2}^2])/2$;
   (5) if n is odd:
   a) $[W_{2n+1}] \leftarrow [W_n W_{n+2}] \cdot [W_n^2] - [(2y)^4 W_{n-1} W_{n+1}] \cdot [W_{n+1}^2]$;
   else:
   a) $[(2y)^2 W_n W_{n+2}] \leftarrow [(2y)^2] \cdot [W_n W_{n+2}]$;
   b) $[(2y)^4 W_n W_{n+2}] \leftarrow [(2y)^2] \cdot [(2y)^2 W_n W_{n+2}]$;
   c) $[W_{2n+1}] \leftarrow [(2y)^4 W_n W_{n+2}] \cdot [W_n^2] - [W_{n-1} W_{n+1}] \cdot [W_{n+1}^2]$;
   (6) if $n \neq 2^{w-1}$:
   $[W_{2n+1}^2] \leftarrow ([W_{2n+1}])^2$.
17. For $n = 3, 5, \ldots, 2^{w-1} + 1$:
   (1) $[X_n] \leftarrow x \cdot [W_n^2] - [(2y)^2 W_{n-1} W_{n+1}]$.
18. For $n = 2^{w-1} + 3, 2^{w-1} + 5, \ldots, 2^w - 1$:
   (1) $t \leftarrow (([W_{n-1}] + [W_{n+1}])^2 - [W_{n-1}^2] - [W_{n+1}^2])/2$;

(2) $[X_n] \leftarrow x \cdot [W_n^2] - [(2y)^2] \cdot t$.

19. For $n = 3, 5, \ldots, 2^{w-1} - 1$:
    (1) $[Y_n] \leftarrow y \cdot [W_{n+2}W_{n-1}^2 - W_{n-2}W_{n+1}^2]$.
20. For $n = 2^{w-1} + 1, 2^{w-1} + 3, \ldots, 2^w - 1$:
    (1) $[Y_n] \leftarrow y \cdot ([W_{n+2}] \cdot [W_{n-1}^2] - [W_{n-2}] \cdot [W_{n+1}^2])$.
21. Return the Jacobian points $nP = ([X_n], [Y_n], [W_n])$, $n = 3, 5, \ldots, 2^w - 1$.

The algorithm `SmallMultJ` is constant-time. Indeed, its branching conditions depend only on the window width w (public parameter), but not on the base point P.

The algorithm uses $18 \cdot 2^{w-2} + 1$ field registers, that is, memory cells for storing elements of $\mathbb{F}$. Some registers can be reused but we do not consider this optimization here.

3.2 Affine coordinates

To compute small multiples nP, $n = 3, 5, \ldots, 2^w - 1$ in affine coordinates, we modify the previous circuit as follows.

1. The expressions W_nW_{n+2} are computed for $n = 1, 2, \ldots, 2^w - 1$. For $n = 1, 2$ the computation again has no cost since $W_1W_3 = W_3$ and $W_2W_4 = W_4$. For $3 \le n \le 2^w - 2$ we again use the scheme $W_nW_{n+2} = ((W_n + W_{n+2})^2 - W_n^2 - W_{n+2}^2)/2$. For $n = 2^w - 1$ we directly multiply W_n by W_{n+2} as $W_{2^w+1}^2$ is not computed. The overall complexity: $(2^w - 4)(\mathtt{S} + 3\mathtt{A} + \mathtt{half}) + \mathtt{M}$.
2. Instead of the expressions $W_{n+2}W_{n-1}^2 - W_{n-2}W_{n+1}^2$ we compute $W_nW_{n+2}W_{n-1}^2$ and $W_{n-2}W_nW_{n+1}^2$. We multiply W_nW_{n+2} by W_{n-1}^2 and $W_{n-2}W_n$ by W_{n+1}^2. Complexity: $(3 \cdot 2^{w-1} - 5)\mathtt{M}$. Here we take into account the simplification: $W_3W_5W_2^2 = W_3W_5$.
3. The expressions W_{2n}, $n = 3, \ldots, 2^{w-1}$, are computed by subtracting $W_{n-2}W_nW_{n+1}^2$ from $W_nW_{n+2}W_{n-1}^2$. Complexity: $(2^{w-1} - 2)\mathtt{A}$.
4. The expressions W_n^{-2}, $n = 3, 5, \ldots, 2^w - 1$, are computed simultaneously using `MontInv`. After that we compute W_n^{-4} by squaring W_n^{-2}. Complexity: $\mathtt{I} + (3 \cdot (2^{w-1} - 1) - 3)\mathtt{M} + (2^{w-1} - 1)\mathtt{S}$.
5. Instead of Y_n we compute Y'_n, $n = 3, 5, \ldots, 2^w - 1$. The computations use y, $W_nW_{n+2}W_{n-1}^2$ and $W_{n-2}W_nW_{n+1}^2$. For each n it takes one multiplication and for $n > 2^{w-1}$ it additionally takes one subtraction. Complexity: $(2^{w-1} - 1)\mathtt{M} + 2^{w-2}\mathtt{A}$.
6. The conversion into affine coordinates consists of multiplying X_n by W_n^{-2} and Y'_n by W_n^{-4}, $n = 3, 5, \ldots, 2^w - 1$. Complexity: $(2^w - 2)\mathtt{M}$.

The total complexity of the modified circuit:

$$\mathtt{I} + (25 \cdot 2^{w-2} - 13)\mathtt{M} + (5 \cdot 2^{w-1} - 3)\mathtt{S} + 8\mathtt{m} + (19 \cdot 2^{w-2} - 6)\mathtt{A} + (2^w - 4)\mathtt{half}.$$

The circuit is detailed in the following algorithm.

Algorithm `SmallMultA`

Input: $P = (x, y) \in \mathbb{G} \setminus \{O\}$, w $(3 \le w < \log_2 q)$.

Output: $3P, 5P, \ldots, (2^w - 1)P$ (in affine coordinates).

Steps:

1. $[y^2] \leftarrow y^2$.
2. $[(2y)^2] \leftarrow 4 \cdot [y^2]$.
3. Compute W_3:
   (1) $[x^2] \leftarrow x^2$;

(2) $[bx] \leftarrow b \cdot x$;
(3) $[(x^2+a)^2] \leftarrow ([x^2]+a)^2$;
(4) $[W_3] \leftarrow 3 \cdot [(x^2+a)^2] - 4 \cdot ([a^2] - 3 \cdot [bx])$.

4. Compute W_4:
(1) $[ax] \leftarrow a \cdot x$;
(2) $[x^3] \leftarrow [y^2] - [ax] - b$;
(3) $[x^6] \leftarrow ([x^3])^2$;
(4) $[W_4] \leftarrow 2 \cdot ([x^6] + 4 \cdot [bx] \cdot (5 \cdot [x^2] - a) + 5 \cdot [ax] \cdot ([x^3] - [ax]) - [a^3 + 8b^2])$.
5. $[W_2^2] \leftarrow 1$.
6. $[W_3^2] \leftarrow [W_3]^2$.
7. $[W_4^2] \leftarrow [W_4]^2$.
8. $[W_1 W_3] \leftarrow [W_3]$.
9. $[W_2 W_4] \leftarrow [W_4]$.
10. $[(2y)^2 W_2 W_4] \leftarrow [(2y)^2] \cdot [W_2 W_4]$.
11. $[(2y)^4 W_2 W_4] \leftarrow [(2y)^2] \cdot [(2y)^2 W_2 W_4]$.
12. $[W_5] \leftarrow [(2y)^4 W_2 W_4] - [W_1 W_3] \cdot [W_3^2]$.
13. $[W_5^2] \leftarrow ([W_5])^2$.
14. For $n = 3, 4, \ldots, 2^{w-1}$:
(1) $[W_n W_{n+2}] \leftarrow (([W_n] + [W_{n+2}])^2 - [W_n^2] - [W_{n+2}^2])/2$;
(2) if $n = 3$:
$[W_{2n}] \leftarrow [W_n W_{n+2}] - [W_{n-2} W_n] \cdot [W_{n+1}^2]$;
else:
$[W_{2n}] \leftarrow [W_n W_{n+2}] \cdot [W_{n-1}^2] - [W_{n-2} W_n] \cdot [W_{n+1}^2]$;
(3) $[W_{2n}^2] \leftarrow ([W_{2n}])^2$.
(4) if n is odd:
a) $[W_{2n+1}] \leftarrow [W_n W_{n+2}] \cdot [W_n^2] - [(2y)^4 W_{n-1} W_{n+1}] \cdot [W_{n+1}^2]$;
else:
a) $[(2y)^2 W_n W_{n+2}] \leftarrow [(2y)^2] \cdot [W_n W_{n+2}]$;
b) $[(2y)^4 W_n W_{n+2}] \leftarrow [(2y)^2] \cdot [(2y)^2 W_n W_{n+2}]$;
c) $[W_{2n+1}] \leftarrow [(2y)^4 W_n W_{n+2}] \cdot [W_n^2] - [W_{n-1} W_{n+1}] \cdot [W_{n+1}^2]$;
(5) if $n \neq 2^{w-1}$:
$[W_{2n+1}^2] \leftarrow ([W_{2n+1}])^2$.
15. $[W_3^{-2}], [W_5^{-2}], \ldots, [W_{2^w-1}^{-2}] \leftarrow \texttt{MontInv}([W_3^2], [W_5^2], \ldots, [W_{2^w-1}^2])$.
16. For $n = 3, 5, \ldots, 2^{w-1} + 1$:
(1) $[X'_n] \leftarrow x - [(2y)^2 W_{n-1} W_{n+1}] \cdot [W_n^{-2}]$.
17. For $n = 2^{w-1} + 3, 2^{w-1} + 5, \ldots, 2^w - 1$:
(1) $t \leftarrow (([W_{n-1}] + [W_{n+1}])^2 - [W_{n-1}^2] - [W_{n+1}^2])/2$;
(2) $[X'_n] \leftarrow x - [(2y)^2] \cdot t \cdot [W_n^{-2}]$.
18. For $n = 3, 5, \ldots, 2^{w-1} - 1$:
(1) $[Y'_n] \leftarrow y \cdot [W_{2n}] \cdot ([W_n^{-2}])^2$.
19. For $n = 2^{w-1} + 1, 2^{w-1} + 3, \ldots, 2^w - 3$:
(1) $[W_n W_{n+2}] \leftarrow (([W_n] + [W_{n+2}])^2 - [W_n^2] - [W_{n+2}^2])/2$;
(2) $[Y'_n] \leftarrow y \cdot ([W_n W_{n+2}] \cdot [W_{n-1}^2] - [W_{n-2} W_n] \cdot [W_{n+1}^2]) \cdot ([W_n^{-2}])^2$.
20. $[Y'_{2^w-1}] \leftarrow y \cdot ([W_{2^w-1}] \cdot [W_{2^w+1}] \cdot [W_{2^w-2}^2] - [W_{2^w-3} W_{2^w-1}] \cdot [W_{2^w}^2]) \cdot ([W_{2^w-1}^{-2}])^2$.
21. Return the affine points $nP = ([X'_n], [Y'_n])$, $n = 3, 5, \ldots, 2^w - 1$.

The algorithm `SmallMultA` is constant-time for the same reasons as for `SmallMultJ`. The algorithm uses $17 \cdot 2^{w-2} + 6$ field registers.

4 Scalar multiplication

Let us proceed with scalar multiplication, that is, computing dP from an affine point $P = (x, y)$ and a scalar $d \in \{1, 2, \ldots, q-1\}$. Recall that q is a large (odd) prime and the length of q in bits equals l.

We start by choosing some window width w and computing the small multiples nP, $n = 3, 5, \ldots, 2^w - 1$, using either the `SmallMultJ` or `SmallMultA` algorithm. Next, we compute the negative small multiples $-nP$, $n = 1, 3, \ldots, 2^w - 1$, with a small overhead ($2^{w-1}\mathtt{A}$).

The computation of dP is performed in Jacobian coordinates. We use point doublings ($J \leftarrow 2J$) and additions with the points $\pm nP$ (either $J \leftarrow J + J$ or $J \leftarrow J + A$ depending on the form of the small multiples). The last addition is followed by converting the sum to affine coordinates: $A \leftarrow J + J$ or $A \leftarrow J + A$. This addition is performed using *complete formulas* which process both distinct (non-equivalent) and equal input points in a unified way. We explain the need for complete formulas a little later.

To determine small multiples added at each step, we record d as follows. First, we write d in base 2^w:

$$d = \sum_{i=0}^{k-1} d_i 2^{wi}. \qquad (\star)$$

Here $d_i \in \{0, 1, \ldots, 2^w - 1\}$ are digits of the representation and $k = \lceil l/w \rceil$ is their number.

Second, for odd d, the digits d_i are adjusted to get into the set $\mathcal{B} = \{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$ such that $(\star)$ is still valid. The adjustment is as follows. For $i = k-1, k-2, \ldots, 1$, the parity of the digit d_i is tested. If the digit is even, it is increased by 1 and the previous digit is simultaneously decreased by 2^w:

$$(d_i, d_{i-1}) \leftarrow (d_i + 1, d_{i-1} - 2^w).$$

This equation can be written in the constant-time (applicable to d_i of any parity) form:

$$(d_i, d_{i-1}) \leftarrow (d_i + \mathtt{even}(d_i), d_{i-1} - \mathtt{even}(d_i)2^w).$$

Here $\mathtt{even}(d_i) = 1 - d_i \bmod 2$.

Third, if the scalar d is even, then it is replaced with the odd scalar $q - d$. We compute the point $(q-d)P = -dP$ and negate it afterwards.

Altogether, we get the following algorithm.

Algorithm `ScalarMult`

Input: $P \in \mathbb{G} \setminus \{O\}$, $d \in \{1, 2, \ldots, q-1\}$.

Output: dP (in affine coordinates).

Steps:

1. $\delta \leftarrow d \bmod 2$, $d \leftarrow (1 - \delta)q + (2\delta - 1)d$.
2. Choose a window width w ($3 \le w < \log_2 q$).
3. $P[1] \leftarrow P$, $(P[3], P[5], \ldots, P[2^w - 1]) \leftarrow \mathtt{alg}(P, w)$, $\mathtt{alg} \in \{\mathtt{SmallMultJ}, \mathtt{SmallMultA}\}$.
4. $(P[-1], P[-3], \ldots, P[-2^w + 1]) \leftarrow (-P[1], -P[3], \ldots, -P[2^w - 1])$.
5. Represent d as $\sum_{i=0}^{k-1} d_i 2^{wi}$, $d_0, d_1, \ldots, d_{k-1} \in \{0, 1, \ldots, 2^w - 1\}$.
6. $(d_{k-1}, d_{k-2}) \leftarrow (d_{k-1} + \mathtt{even}(d_{k-1}), d_{k-2} - \mathtt{even}(d_{k-1})2^w)$.
7. $Q \leftarrow P[d_{k-1}]$.
8. For $i = k-2, k-3, \ldots, 1$:
   (1) $(d_i, d_{i-1}) \leftarrow (d_i + \mathtt{even}(d_i), d_{i-1} - \mathtt{even}(d_i)2^w)$;

(2) $Q \leftarrow 2^w Q$ ($J \leftarrow 2J$, w times);
(3) $Q \leftarrow Q + P[d_i]$ ($J \leftarrow J + J$ or $J \leftarrow J + A$).
9. $Q \leftarrow 2^w Q$.
10. $Q \leftarrow Q + P[d_0]$ ($A \leftarrow J + J$ or $A \leftarrow J + A$).
11. $Q \leftarrow (-1)^\delta Q$.
12. Return Q.

Note that in doublings $J \leftarrow 2J$ and additions $J/A \leftarrow J + J/A$, the exceptional case when some of the operands equals O is not possible. This is because the condition $Q \neq O$ is an invariant of `ScalarMult`. Let us prove this fact.

First, Q is obviously not equal to O after Steps 7, 10, 11. Second, Q cannot become O after doubling since Q belongs to the group $\mathbb{G}$ of odd order q. Third, Q cannot become O after Step 8.3. Indeed, after this step the point Q has the form eP, where $e = \sum_{j=i}^{k-1} d_j 2^{w(j-i)}$. The equality $Q = O$ means that either $e = 0$ or $e = q$. The case $e = 0$ is impossible because e is odd. The case $e = q$ is impossible since $i \geq 1$ and, therefore, $e < 2^{w(k-1)} < q$.

An additional exception can occur during the addition $Q + P[d_i]$ when the terms (even in different coordinates) coincide, that is, when doubling is actually performed. For this, it must hold that

$$\sum_{j=i+1}^{k-1} d_j 2^{w(j-i)} \equiv d_i \pmod{q}.$$

This is only possible if $d_i < 0$ and

$$e = \sum_{j=i}^{k-1} d_j 2^{w(j-i)} = q + 2d_i.$$

From estimates $e \leq d/2^{wi} + 1 < q/2^{wi} + 1$ and $q + 2d_i \geq q - 2(2^w - 1)$ it follows that $i = 0$, that is, the exception can only occur at Step 10. But here we use complete formulas that block it.[1]

Conventional algorithms for doubling and adding points on Weierstrass curves are constant-time provided that exceptional cases are not possible. Using these algorithms in `ScalarMult`, we achieve for it the constant-time property. We take into account that `ScalarMult` does not contain conditional branches and that the nested algorithms `SmallMultJ` and `SmallMultA` are constant-time.

It should be noted that in addition to conditional branches, there is another factor of non-constant running time. Modern microprocessors load data through the cache memory and the loading time may vary depending on the cache state. Since `ScalarMult` does use the array $P[i]$ to store small multiples, additional measures should be taken to prevent running time fluctuations.

One of the natural measures is to avoid storing negative points $P[-n]$ by switching from the operations $J/A \leftarrow J + J/A$ to $J/A \leftarrow J \pm J/A$ while preserving the constant-time property. In result, `ScalarMult` requires either $3 \cdot 2^{w-2}$ or $2 \cdot 2^{w-2}$ field registers to store Jacobian or affine small multiples $P[n]$, $n = 1, 3, \ldots, 2^{w-1}$, respectively.

Let `ScalarMult`[`alg`, w] be the algorithm `ScalarMult` instantiated with `alg` $\in$ {`SmallMultJ`, `SmallMultA`} and a window width w. Let a curve with $a = -3$ be used and

[1] Complete formulas allows to properly manage scalars d of the form $q - 2\delta$, where $\delta \in \{1, 3, \ldots, 2^w - 1\}$ is such that $q - \delta$ is divisible by 2^w and $(q - \delta)/2^w$ is odd. If for a given (q, w) such δ does not exist, then the standard (more efficient) addition formulas can be used.

the operations $J \leftarrow 2J$, $J \leftarrow J + J$ and $J \leftarrow J + A$ be implemented with the complexity $3\mathtt{M} + 6\mathtt{S} + 4\mathtt{m} + 6\mathtt{A} + \mathtt{half}$, $11\mathtt{M} + 5\mathtt{S} + 4\mathtt{m} + 9\mathtt{A}$ and $7\mathtt{M} + 4\mathtt{S} + 4\mathtt{m} + 9\mathtt{A}$ respectively.[2] The choice $a = -3$ made provides the fastest time for the operation $J \leftarrow 2J$ without affecting the complexity of other operations. Moreover, since a is small, multiplication by a in `alg` costs $1\mathtt{m}$, not $1\mathtt{M}$.

Let the complexity of the operations $A \leftarrow J + J$ and $A \leftarrow J + A$ be $\mathtt{I} + 20\mathtt{M} + 2\mathtt{S} + 29\mathtt{A}$ and $\mathtt{I} + 17\mathtt{M} + 1\mathtt{S} + 23\mathtt{A}$ respectively. To achieve this complexity, we use complete formulas from [15] to add points in homogeneous projective coordinates: a homogeneous projective point (X, Y, Z) represents the affine point $(x, y) = (X/Z, Y/Z)$. The complete addition in homogeneous coordinates, denoted as $H \leftarrow H + H$ and $H \leftarrow H + A$, can be done in time $14\mathtt{M} + 29\mathtt{A}$ and $13\mathtt{M} + 23\mathtt{A}$ respectively (see [15, Algorithms 4 and 5]) provided that $a = -3$.[3] The conversion $H \leftarrow J$ takes time $2\mathtt{M} + 1\mathtt{S}$. We need two conversions before $H \leftarrow H + H$ and one conversion before $H \leftarrow H + A$. The additional conversion $A \leftarrow H$ of complexity $1\mathtt{I} + 2\mathtt{M}$ is required in both cases. Summing up the times, we obtain the declared complexity of $A \leftarrow J + J$ and $A \leftarrow J + A$.

For the case $\mathtt{alg} = \mathtt{SmallMultA}$, let the cascade $(J \leftarrow 2J, J \leftarrow J+A)$ at the junction of the steps 8.2, 8.3 and 9, 10 be additionally optimized. The cascade is treated as a special operation $J \leftarrow 2J + A$ which is implemented in time $11\mathtt{M} + 7\mathtt{S} + 27\mathtt{A}$ according to [11, Appendix A.3].

In these settings, the final algorithms have the following complexity:

Operation in $\mathbb{F}$	$\mathtt{ScalarMult}[\mathtt{SmallMultJ}, w]$	$\mathtt{ScalarMult}[\mathtt{SmallMultA}, w]$
$\mathtt{I}$	1	2
$\mathtt{M}$	$\lceil l/w - 1 \rceil (3w + 11) + 19 \cdot 2^{w-2} - 2$	$\lceil l/w - 1 \rceil (3w + 8) + 25 \cdot 2^{w-2} - 4$
$\mathtt{S}$	$\lceil l/w - 1 \rceil (6w + 5) + 7 \cdot 2^{w-2} - 4$	$\lceil l/w - 1 \rceil (6w + 1) + 5 \cdot 2^{w-1} - 3$
$\mathtt{m}$	$\lceil l/w - 1 \rceil (4w + 4) - 4w + 4$	$\lceil l/w - 1 \rceil (4w - 4) + 20$
$\mathtt{A}$	$\lceil l/w - 1 \rceil (6w + 9) + 2 \cdot 2^{w+2} + 17$	$\lceil l/w - 1 \rceil (6w + 21) + 19 \cdot 2^{w-2} - 4$
$\mathtt{half}$	$\lceil l/w - 1 \rceil w + 3 \cdot 2^{w-2} - 3$	$\lceil l/w - 1 \rceil (w - 1) + 2^w - 3$

To simplify the expressions above, let us apply the following heuristic often used in practice:

$$\mathtt{I} = 100\mathtt{M}, \quad \mathtt{S} = 0.8\mathtt{M}, \quad \mathtt{m} = \mathtt{A} = \mathtt{half} = 0\mathtt{M},$$

or $(100, 0.8, 0)$ for short. Let us also consider two additional heuristics: $(100, 0.67, 0)$ and $(100, 0.67, 0.05)$. With a heuristic h, the complexity of an algorithm is expressed as the number of $\mathtt{M}$ operations, *M-complexity*, denoted as $\mathtt{M}(h)$.

Table 2 presents $\mathtt{M}$-complexity of the algorithms $\mathtt{ScalarMult}[\mathtt{alg}, w^*]$ for $l = 256, 384, 512$ and different heuristics listed above. Here w^* is the window width that provides the smallest $\mathtt{M}$-complexity for a given `alg` and l. The optimal width w^* is the same for all three heuristics. The table shows that the choice $\mathtt{alg} = \mathtt{SmallMultA}$ is preferable to $\mathtt{alg} = \mathtt{SmallMultJ}$ providing both smaller $\mathtt{M}$-complexity and smaller optimal window width. Note that the smaller the window width, the less memory is required.

The table additionally covers the algorithm from [10] that performs scalar multiplication on Montgomery curves by the left-to-right Montgomery ladder according to [13]. This algorithm, denoted as $\mathtt{MontLadder}[\mathtt{MontCurve}]$, is considered one of the most efficient in the constant-time class, its complexity is

$$l(5\mathtt{M} + 4\mathtt{S} + \mathtt{m} + 8\mathtt{A}) + \mathtt{I} + \mathtt{M}.$$

[2] See [1] for detailed circuits. They are named `dbl-1998-hnm`, `add-2007-bl` and `madd-2007-bl`.

[3] In [15], multiplications by small constants ($\mathtt{m}$) are converted into additions ($\mathtt{A}$).

Table 2: M-complexity of scalar multiplication algorithms

l	Algorithm	M-complexity (rounded to the nearest integer)		
		$\mathtt{M}(100, 0.8, 0)$	$\mathtt{M}(100, 0.67, 0)$	$\mathtt{M}(100, 0.67, 0.05)$
256	ScalarMult[SmallMultJ, 5]	3046	2807	2989
	ScalarMult[SmallMultA, 4]	2846	2636	2830
	MontLadder[WeierCurve]	2724	2590	2731
	MontLadder[MontCurve]	2200	2067	2182
384	ScalarMult[SmallMultJ, 6]	4382	4032	4297
	ScalarMult[SmallMultA, 5]	4090	3774	4053
	MontLadder[WeierCurve]	4030	3829	4041
	MontLadder[MontCurve]	3250	3050	3223
512	ScalarMult[SmallMultJ, 6]	5741	5274	5626
	ScalarMult[SmallMultA, 5]	5333	4912	5284
	MontLadder[WeierCurve]	5335	5068	5350
	MontLadder[MontCurve]	4299	4033	4264

Here we treat multiplication by a (presumably small) curve coefficient as the operation m and multiplication by a coordinate of P as the operation M (the latter because scalar multiplication is performed in the variable base settings).

The table also covers the algorithm from [5] which we denote as MontLadder[WeierCurve]. This algorithm also uses the Montgomery ladder but over short Weierstrass curves. This is probably the fastest constant-time algorithm for these curves, its complexity is

$$l(7\mathtt{M} + 4\mathtt{S} + 10\mathtt{A} + 1\mathtt{half}) + \mathtt{I} + 8\mathtt{M} + 6\mathtt{S} + 3\mathtt{m} + 6\mathtt{A} + 2\mathtt{half}.$$

We suppose here that the simple finalization technique [5, Section 2.4.1] and the S-M tradeoff [5, Figure 3] are used. We also suppose that the constant 1/3 is precomputed so that division by 3 costs 1M.

The table shows that the algorithm ScalarMult[SmallMultA, w^*] is competitive to MontLadder[WeierCurve] especially for large l. A drawback of ScalarMult[SmallMultA, w^*] is a rather large memory requirements compared to only 6 field registers used in MontLadder[WeierCurve].

Conclusion

Combining division polynomial-driven algorithms for small scalar multiplication on elliptic curves in the short Weierstrass form with several well-known optimization techniques we obtain constant-time algorithms for scalar multiplication that are competitive to the recent developments in the subject based on the Montgomery ladder. The integrated techniques are: the window method, skipping even small multiples, left-to-right scalar recording avoiding exceptional cases, Montgomery's trick for simultaneous inversion of several field elements, the fast point doubling-addition.

References

[1] D. J. Bernstein and T. Lange. *Explicit-formulas database*. 2007. URL: http://hyperelliptic.org/EFD.

[2] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In: *Advances in Cryptology – ASIACRYPT 2007*. Ed. by K. Kurosawa. Vol. 4833. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 29–50.

[3] D. J. Bernstein et al. Twisted Edwards curves. In: *Advances in Cryptology – AFRICACRYPT 2008*. Ed. by S. Vaudenay. Vol. 5023. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 389–405.

[4] H. M. Edwards. A normal form for elliptic curves. *Bulletin (New Series) of the American Mathematical Society* **44** (3) (2007), pp. 393–422.

[5] M. Hamburg. *Faster Montgomery and double-add ladders for short Weierstrass curves*. Cryptology ePrint Archive, Report 2020/437. https://eprint.iacr.org/2020/437. 2020.

[6] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Berlin, Heidelberg: Springer-Verlag, 2003.

[7] M. Joye and M. Tunstall. Exponent recoding and regular exponentiation algorithms. In: *Progress in Cryptology, Second International Conference on Cryptology in Africa — AFRICACRYPT 2009*. Ed. by B. Preneel. Vol. 5580. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 334–349.

[8] N. Kanayama et al. Implementation of an elliptic curve scalar multiplication method using division polynomials. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* **E97.A** (1) (2014), pp. 300–302.

[9] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation* **48** (177) (1987), pp. 203–209.

[10] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. RFC 7748. 2016. URL: https://rfc-editor.org/rfc/rfc7748.txt.

[11] P. Longa and A. Miri. *New Multibase Non-Adjacent Form Scalar Multiplication and its Application to Elliptic Curve Cryptosystems (extended version)*. Cryptology ePrint Archive, Report 2008/052. https://eprint.iacr.org/2008/052. 2008.

[12] V. S. Miller. Use of elliptic curves in cryptography. In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by H. C. Williams. Vol. 218. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1986, pp. 417–426.

[13] P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation* **48** (177) (1987), pp. 243–264.

[14] K. Okeya and T. Takagi. The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In: *Topics in Cryptology, The Cryptographers' Track at the RSA Conference — CT-RSA 2003*. Ed. by M. Joye. Vol. 2612. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 328–342.

[15] J. Renes, C. Costello, and L. Batina. Complete addition formulas for prime order elliptic curves. In: *Advances in Cryptology — EUROCRYPT 2016*. Ed. by M. Fischlin and JS. Coron. Vol. 9665. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 403–428.

[16] K. E. Stange. The Tate pairing via elliptic nets. In: *Pairing-Based Cryptography – Pairing 2007*. Ed. by T. Takagi et al. Vol. 4575. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 329–348.