IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 1, 45 pages.

https://doi.org/10.62056/akmp-4c2h

A Prime-Order Group with Complete Formulas from Even-Order Elliptic Curves

Thomas Pornin

NCC Group, Canada

Abstract. This paper describes a generic methodology for obtaining unified, and then complete formulas for a prime-order group abstraction homomorphic to a subgroup of an elliptic curve with even order. The method is applicable to any curve with even order, in finite fields of both even and odd characteristic; it is most efficient on curves with order equal to 2 modulo 4, dubbed "double-odd curves". In large characteristic fields, we obtain doubling formulas with cost as low as $1M + 5S$, and the resulting group allows building schemes such as signatures that outperform existing fast solutions, e.g. Ed25519. In binary fields, the obtained formulas are not only complete but also faster than previously known incomplete formulas; we can sign and verify in as low as 18k and 27k cycles on x86 CPUs, respectively.

Keywords: elliptic curves · complete formulas · double-odd curves

1 Introduction

Many cryptographic protocols, starting with the classic Diffie-Hellman key exchange [DH76] and Schnorr signatures [Sch90], can be expressed over a *prime-order group* abstraction. Elliptic curves defined over finite fields have a group structure which provides the required properties:

- Elements can be serialized into a canonical representation, and deserialized efficiently.
- The group law (hereafter denoted additively) can be efficiently computed between any two elements. Similarly, the opposite of an element is easily obtained.
- Curves with prime order (thus being cyclic groups) can be found.
- Solving discrete logarithm in that group can be made computationally infeasible with curves of moderate size.

An elliptic curve over a finite field can always be described as a set of points (x, y) that are solution to a degree-3 polynomial equation known as the Weierstraß equation. The group law can be expressed as a pair of rational functions over the coordinates of the operands, and yielding the two coordinates of the result. Unfortunately, these expressions work only for most curve points; there are a few *exceptional inputs* for which they lead to divisions by zero:

- The neutral element of the group is the "point-at-infinity" which does not have defined affine coordinates x and y. Any application of the group law that involves this point as input (adding the point-at-infinity to another point) or as output (adding a point to its opposite) is thus an exceptional case.

E-mail: thomas.pornin@nccgroup.com (Thomas Pornin)

Received: 2024-01-05 Accepted: 2024-03-05

- Adding a point to itself is also an exceptional situation: the normal law expression involves computing the slope of the line joining the two points, which must be replaced with the tangent to the curve when the two points are equal to each other.

Such exceptional points are problematic for cryptographic implementations, because their proper handling requires testing for the exceptional cases and using alternate formulas, which would naturally lead to data-dependent variations in execution time and other similar side-channel leaks. Alternatively, an implementation would run the normal *and* exceptional formulas systematically, and perform a side-channel-free selection of the appropriate result, but that would imply a substantial increase in the computing cost. A better solution would be to find an alternate representation of points, and formulas that do not have exceptional inputs; following the terminology in [BL07], we will say that formulas with no exceptional case are *complete*, while formulas for which only the point-at-infinity is exceptional are *unified* (i.e. unified formulas at least properly handle the addition of a point with itself)[1].

Projective coordinates $(X{:}Y{:}Z)$, such that $x = X/Z$ and $y = Y/Z$, allow a representation of the point-at-infinity with a triplet of coordinates such that $Z = 0$. It has been shown by Arène, Kohel and Ritzenthaler [AKR12] that this is not, in all generality, sufficient to achieve formula completeness: *any* set of formulas will have exceptional inputs when considered over the algebraic closure of the field over which the curve is defined. However, for practical usages, we only use finite fields, and complete elliptic curve point formulas are then feasible, even if the same formulas would fail if the curve were lifted into an extension of the base field. Renes, Costello and Batina [RCB16] have previously published formulas for short Weierstraß curves over fields of caracteristic 5 or more, using projective coordinates; the formulas are complete as long as the difference between the two input operands is not a point of order two. Classic standard elliptic curves such as NIST's P-256 have odd order, and thus do not have any point of order two.

The Renes-Costello-Batina formulas have a non-negligible runtime cost (12M in general). Faster operations can be achieved with twisted Edwards curves [BBJ+08, HWCD08], an alternate normal form of curves with formulas that lower the cost of point addition to 8M, and, more importantly, can be complete (depending on the exact curve definition). The well-known Curve25519, as used in Ed25519 signatures [BDL+11], leverages this curve representation. The order of a twisted Edwards curve, however, is necessarily a multiple of four; such a curve cannot be a prime-order group. At best, we can have a curve of order fr for a (large) prime r and a (small) cofactor f, and try to work in the cyclic subgroup of points of r-torsion. Input points might however lie outside of that subgroup, leading to potential trouble in some protocols, because that allows the existence of distinct curve points which are still "equivalent" in the sense that they differ only by a small order point [CJ19]. Even in the simple case of signatures, the non-trivial cofactor f can lead to different implementations disagreeing on whether a given signature (maliciously crafted by the signer) is valid or not, which can break consensus protocols [CGN20].

A prime-order group abstraction, avoiding cofactor issues, can be built on top of a (twisted) Edwards curve with cofactor $f = 4$ with the Decaf construction [Ham15]. An extension of that method called Ristretto is applicable to cofactor $f = 8$, which covers the case of Curve25519 [ALdV, dVGH+23]. Decaf and Ristretto offer all the features expected from a prime-order group abstraction, albeit with relatively complex encoding and decoding procedures.

In this paper, we describe a general methodology to obtain a prime-order group abstraction out of elliptic curves with even order, with efficient formulas. The core idea consists in representing an r-torsion point P (for a prime r) by the point $P + N$, with N being a point of order two on the curve. The idea maps especially well to curves with order equal to 2 modulo 4, dubbed double-odd curves; however, it can be applied to other

[1]There are other definitions of "unified"; their common practical aspect is that they can be applied in general, and in particular when adding a point to itself, with only a few easily handled special cases.

elliptic curves with an even order. With suitable choices of coordinate systems, coupled with a validation process for incoming points to verify that they have the expected $P + N$ format, we obtain complete formulas that provide performance on par with or even faster than existing fast curves such as Curve25519, with canonical encoding into and decoding from a short, compressed format. We cover all finite field characteristics (2, 3, 5 and more). We thus obtain appropriate prime-order groups on which complex protocols can be built. As an illustration, we implemented both key exchange and signatures; with binary curve GLS254, we obtain short signatures (48 bytes at the 128-bit security level), with signature generation and verification in as little as 18k and 27k cycles on an Intel x86 CPU, respectively, which is 3 to 4 times faster than the best reported Ed25519 timings on the same platform.

2 Notations

In all of the following, we work in a finite field $\mathbb{F}_q$, with $q = p^m$ for a prime p (the field characteristic) and an integer $m \geq 1$. The case $p = 2$, dubbed "binary field", is handled in section 5; the rest of this paper uses $p \geq 3$. When $m = 1$, and thus $q = p$ is prime, this is called a "prime field", which is canonically defined as integers taken modulo q.

We denote by $z \in QR$ the fact that z is a square in $\mathbb{F}_q$, i.e. there exists some element $s \in \mathbb{F}_q$ such that $s^2 = z$. s is a square root of z, denoted $\sqrt{z}$. A non-zero square has two square roots, and the notation $\sqrt{z}$ does not distinguish between the two; square root computations appear only in contexts where an extra normalization step is applied to choose a specific root. Non-square z are denoted $z \notin QR$. Note that zero is a square. In binary fields, every element is a square and has a single square root, and the QR notation does not apply.

Elliptic curves are defined through equations that link point coordinates with some constant values, usually denoted a and b. These constants are assumed to be chosen such that multiplication by a or b is inexpensive. Practical implementations are expected to use some kind of fractional representation (e.g. projective coordinates) in which all divisions are mutualized into a single inversion that is typically performed only when serializing a point into bits; thus, most of the cost of individual operations (point additions, doublings...) can be approximated by the number of multiplications (M) and squarings (S) that they involve, while neglecting extra "inexpensive" operations such as additions, subtractions, and multiplications by constants derived from a and b. Squarings are typically faster than multiplications, though the ratio between the two depends on the used hardware platform, and the notion of the individual cost of an elementary operation is not well-defined in modern pipelined processors, especially superscalar CPUs that support out-of-order execution. As a rule of thumb, a squaring cost can be estimated to be about 80% the cost of a multiplication.

In the case of binary curves (section 5), the method is applicable to existing standard curves, and some standard NIST curves (the B-* non-Koblitz curves) use pseudorandom values for the b constant, not chosen to make multiplications fast. To cover that extra cost, we denote by $\mathrm{m_b}$ the cost of multiplying by b (or a constant derived from b) in estimates for binary curves.

3 Generic Methodology

3.1 Geometrical Description

The core idea is to use a point of order two as pivot. Let $\mathbb{F}_q$ be a finite field, and $\mathcal{E}$ an elliptic curve defined over that field with a degree-3 equation (i.e. a Weierstraß curve). Let n be the order of $\mathcal{E}$; by Hasse's theorem, we know that n is close to q (specifically,

$|\sqrt{q} - \sqrt{n}| \leq 1$). We suppose that n is even, and r is a large prime divisor of n. The points of r-torsion in $\mathcal{E}$ are:

$$\mathcal{E}[r] = \{P \in \mathcal{E} \mid rP = \mathbb{O}\}$$

where $\mathbb{O}$ is the point-at-infinity (the neutral element of the group). $\mathcal{E}[r]$ is a subgroup with order r or r^2; the latter case may happen only if r^2 divides n, and, in general, we prefer to avoid it, except in the context of pairing-friendly curves. We will hereafter suppose that r^2 does not divide n. The subgroup $\mathcal{E}[r]$ is the prime-order group structure over which we wish to build our abstraction.

The group law over the curve is defined as follows:

- The point-at-infinity $\mathbb{O}$ is the neutral element.
- For any given point $P = (x, y)$ on the curve, there is a single other point P' with the same x coordinate; we define that other point to be the opposite of P, denoted $-P$.
- For two points P and Q on the curve, the line (PQ) intersects the curve in a third point R; we define $P + Q = -R$. The point R is obtained from $-R$ by mirroring it against the x axis.

This definition highlights the exceptional points, where the formulation above does not work properly: the point-at-infinity has no defined coordinates, making the line $(P\mathbb{O})$ ill-defined; and if P and Q are the same point, the line (PQ) is similarly underspecified (we use the tangent to the curve in that case).

Since we supposed that n is even, we can write $n = fr$ for some even cofactor f. There must be at least one point of order two, which we call N (since $\mathcal{E}[2]$ over the algebraic closure of $\mathbb{F}_q$ is homomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$, there are either one or three points of order two). By definition, N is not part of $\mathcal{E}[r]$. The main idea is to *represent* a point $P \in \mathcal{E}[r]$ by the point $P + N$ (which is not in $\mathcal{E}[r]$). This has the following advantages:

- The neutral $\mathbb{O}$ is now represented by the point $\mathbb{O} + N = N$, which has defined coordinates.
- The sum of P and Q, represented by $P + N$ and $Q + N$, respectively, will be itself represented by $(P + Q) + N$. The latter point can be computed as:

  $$(P + Q) + N = P + (Q + N)$$

  with the important remark that if P and Q are in $\mathcal{E}[r]$, then $Q + N \notin \mathcal{E}[r]$, and, in particular, $Q + N$ cannot be the same point as P.

In this way, we can get at least unified formulas: when adding two points P and Q, known through their representants $P + N$ and $Q + N$, we first apply the group law on $P + N$ and N, to yield the point P, then we apply the law again, between P and $Q + N$; in neither case, the two operands can be the same point, which fully avoids the exceptional case of adding a point to itself.

A remaining potential issue is when the first operand is $\mathbb{O}$; if $P = \mathbb{O}$ and is represented by $P + N = N$, then the addition of N to N is an exceptional case by itself, both because it adds a point to itself, and because the output is the point-at-infinity. However, as will be detailed in the next section, that case can be eliminated in the formulas themselves.

3.2 Analytical Description

With appropriate changes of variables that do not modify the curve's algebraic shape (i.e. reversible affine transforms in the plane), the elliptic curve $\mathcal{E}$ can always be described by a *Weierstraß equation*; the elements of $\mathcal{E}$ are the point-at-infinity, and the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ such that:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

for some constants a_1, a_2, a_3, a_4 and a_6. For a point $P = (x, y) \in \mathcal{E}$, its opposite is $-P = (x, -y - a_1x - a_3)$. The geometrical definition translates into the following, when adding points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ together:

- The slope of the line (P_1P_2) is computed as:

  $$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

  If P_1 and P_2 have the same x coordinate, then the formula above fails. In that case, then either $y_2 = -y_1 - a_1x - a_3$, which implies that $P_2 = -P_1$, and their sum is then the point-at-infinity $\mathbb{O}$; or $P_1 = P_2 \neq -P_1$, and we replace the line (P_1P_2) with the tangent to the curve at that point, and its slope is:

  $$\lambda = \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3}$$

- The coordinates of $P_3 = (x_3, y_3) = P_1 + P_2$ are then obtained as:

  $$\begin{aligned} x_3 &= \lambda^2 + a_1\lambda - a_2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 - a_1x_3 - a_3 \end{aligned}$$

These equations are the “classic affine addition formulas”, which are detailed in many textbooks (e.g. [Was08]).

From that point, we consider that $\mathbb{F}_q$ has characteristic 3 or more; the case of fields with characteristic 2 will be covered in section 5. We can therefore divide values by 2.

Since we assumed that the curve $\mathcal{E}$ has even order, and chose a point $N = (x_N, y_N)$ of order two, we can do a further change of variable $(x, y) \mapsto (x + x_N, y + y_N)$, which corresponds to moving the coordinate reference point to N itself. This does not change the general shape of the equation, even though the constants are modified; we can now assume, without loss of generality, that $N = (0, 0)$. Substituting these coordinates into the Weierstraß equation yields $a_6 = 0$. Similarly, N being of order two, it must be that $-N = N$, and this implies that $a_3 = 0$. Since we consider here a field of characteristic distinct from 2, we can apply a further change of variable $y \mapsto y + (a_1x)/2$, which modifies the equation into the following:

$$y^2 = x^3 + ax^2 + bx$$

for two constants $a = a_2 + a_1^2/4$ and $b = a_4$. The following notes apply:

- This shape is *not* the classical “short Weierstraß” equation. It can be obtained under the hypothesis that there is a point of order two, i.e. that the curve order is even.

- In descriptions of elliptic curves, the case of a field of characteristic 3 is normally distinguished, due to the inability to divide by 3 in such a field. As a consequence of the assumption of an even curve order, we do not need to make that distinction here; in this paper, fields of characteristic 3 can use the same treatment and formulas as fields of characteristic 5 and more.

- The group law definition requires the existence of a well-defined tangent on each point, i.e. that the two partial derivatives $2y$ and $3x^2 + 2ax + b$ do not vanish simultaneously on any curve point (if such a case happens, the curve is said to be *singular*). With the curve shape above, the curve is non-singular if and only if $b \neq 0$ and $a^2 - 4b \neq 0$.

- The j-invariant of the curve is:

$$j = \frac{256(a^2 - 3b)^3}{b^2(a^2 - 4b)}$$

- N is the only curve point such that $x = 0$.

For a curve point $P = (x, y) \neq N, \mathbb{O}$, the point $P + N$ is obtained as: $P + N = (b/x, -by/x^2)$. Applying this formula into the group law definition, a straightforward calculation yields generic addition formulas. Given points $P_1, P_2 \in \mathcal{E}[r]$, represented by points $P_1 + N = (x_1, y_1)$ and $P_2 + N = (x_2, y_2)$, we want to obtain the representant of their sum $P_3 + N = (P_1 + P_2) + N = (x_3, y_3)$. We first suppose that P_1, P_2 and P_3 are all distinct from $\mathbb{O}$, which implies that x_1 and x_2 are non-zero. As explained above, under these conditions, no exceptional case occurs in the addition formulas, and we get the following expressions:

$$x_3 = \frac{b((x_1 + x_2)(x_1x_2 + b) + 2ax_1x_2 + 2y_1y_2)}{(x_1x_2 - b)^2}$$
$$y_3 = \frac{b(2a(x_1y_2 + x_2y_1)(x_1x_2 + b) + (x_1^2y_2 + x_2^2y_1)(x_1x_2 + 3b) + (y_1 + y_2)(3bx_1x_2 + b^2))}{-(x_1x_2 - b)^3}$$

During the calculation, the expressions were simplified by removing a factor x_1x_2 common to the numerator and denominator; this removal happens to also make the formulas compatible with the cases involving $\mathbb{O}$. Indeed, it can be verified that if $x_1 = 0$ or $x_2 = 0$, then the correct result is obtained; similarly, if $x_1 = x_2$ and $y_1 = -y_2$, i.e. adding a point to its opposite, the point N is correctly obtained as the representant of the sum $\mathbb{O}$. These formulas are thus *complete* – provided that the two inputs are indeed the correct representants of two elements of $\mathcal{E}[r]$. The problem of verifying that an input point is a correct representant of an element of $\mathcal{E}[r]$ is covered in the section 3.3.

Some sub-cases of the curve equation $y^2 = x^3 + ax^2 + bx$ have already been studied in previous works (e.g. Doche-Icart-Kohel double-oriented curves use $b = 16a$ [DIK06]; Castryck and Decru suggest $b = -1$ in the context of isogeny-based cryptography [CD20]). To our knowledge, none envisioned using point N as pivot to avoid exceptional cases in formulas.

3.3 Point Validation and Double-Odd Curves

In order to provide the expected prime-order group abstraction, it is necessary to make sure that externally received inputs (e.g. public keys) are correct representants of elements of $\mathcal{E}[r]$. We suppose here that we have received a point $Q = (x, y)$ and that we have verified in some way that x and y fulfill the curve equation $y^2 = x^3 + ax^2 + bx$ (the point is on the curve).

Torsion Check. A conceptually simple method to validate the point Q is to multiply it by the scalar r; for points of r-torsion, this must yield the neutral $\mathbb{O}$ (represented by N). A potential difficulty is that the formulas shown in section 3.2 are guaranteed complete only as long as the operands are correct representants of r-torsion points, which is not yet known to be the case for the point Q. Analysis of the formulas shows that they fail in a single case: when inputs (x_1, y_1) and (x_2, y_2) are such that $(x_1, y_1) = (x_2, y_2) + N$. In such a case, all numerators and denominators vanish in the expressions of x_3 and y_3. A consequence is that in actual implementations of the formulas, using any fractional representation (e.g. projective coordinates), an exceptional case will yield all-zero coordinates, and that situation will be "sticky": subsequent point additions using the formulas will remain all-zero values. The exact details vary depending on the concrete choice of coordinate system, but in general one can expect the following validation process to work:

1. Verify that the provided x and y coordinates fulfill the curve equation.

2. Assuming that $(x, y) = P + N$ for some curve point P, compute $rP + N$ with any double-and-add variant leveraging the formulas from section 3.2 with a fractional/projective concrete representation of coordinates.

3. If an invalid (all-zero) representation is obtained, then an exceptional case was hit and the point (x, y) is not valid. Otherwise, if no such case was obtained but $rP + N$ is not the point N itself, then (x, y) is not valid. If $rP + N$ yields a proper representation of N, then the input (x, y) is a valid representant of a point $P \in \mathcal{E}[r]$.

This process can work for any elliptic curve with an even order. It is unfortunately relatively expensive (the computation of rP can use the fact that r is not secret and is known in advance, thus allowing the precomputation of an efficient addition chain, but the cost will still be commensurate with that of a generic multiplication of a point by a scalar). However, for some curves, a substantially faster process can be used, as detailed below.

Point Halving and Double-Odd Curves Suppose that the cofactor is a power of two, i.e. that the whole curve order is equal to $2^t r$ for an odd prime r and an integer $t \geq 1$. Any curve point can be uniquely written as the sum of a point of r-torsion and a point of 2^t-torsion. Let 2^h be the maximum order of all 2^t-torsion points (we have $h \leq t$, but not necessarily $h = t$). Suppose that a given point Q can be halved h times, i.e. that $Q = 2^h R$ for some other curve point R (not necessarily unique). We can write R as $R = P + T$, with $P \in \mathcal{E}[r]$ and $T \in \mathcal{E}[2^t]$; thus:

$$Q = 2^h R = 2^h P + 2^h T = 2^h P$$

i.e. Q is then a point of r-torsion. Conversely, if Q is a point of r-torsion, then it can be halved indefinitely, by multiplying it with the inverse of 2 modulo r. We therefore obtain a test for membership to the subgroup of r-torsion points: these are exactly the points that can be halved at least h times.

Point halving, in general, is a somewhat expensive operation, since it involves two square roots, and each square root in the field is (roughly) equivalent to about 1/10th of the cost of a point multiplication by a scalar. Moreover, each halving yields at least two solutions, and one must select the "right one", which is complicated if $\mathcal{E}[2^t]$ is not cyclic. However, there are curves for which the validation process is efficient, namely if $t = 1$. These are curves of order $2r$, i.e. twice an odd integer; we call such curves *double-odd*:

Definition 1. A *double-odd elliptic curve* is a non-singular elliptic curve defined over a given finite field, such that its order is equal to 2 modulo 4.

An elliptic curve of equation $y^2 = x^3 + ax^2 + bx$ in a field of characteristic 3 or more is double-odd if and only if both b and $a^2 - 4b$ are non-squares in the field. Indeed:

- If $a^2 - 4b \in QR$ then the polynomial $X^3 + aX^2 + bX$ has three roots, leading to three points of order two in the curve. These three points, together with $\mathbb{O}$, make a subgroup homomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$, whose order is 4 and must then divide the curve order.

- If $a^2 - 4b \notin QR$ but $b = c^2$ for some constant c, then $a^2 - 4b = (a + 2c)(a - 2c)$, which implies that one of $a + 2c$ and $a - 2c$ must be a square. We can assume without loss of generality that $a + 2c \in QR$ (otherwise, we just replace c with the other root $-c$). Then it can be easily verified that $T = (c, c\sqrt{a + 2c})$ is a valid curve point whose double is N; this is therefore a point of order 4, and the curve order is a multiple of 4.

- Conversely, if $b \notin QR$ and $a^2 - 4b \notin QR$, then N is the only point of order two (the polynomial $X^3 + aX^2 + bX$ has a single root, which is zero); $\mathcal{E}$ not being double-odd would then imply the existence of a point $M = (x_m, y_m)$ such that $2M = N$, which would imply that $M + N = -M$. As was pointed out previously, the x coordinate of $-M$ is the same as the x coordinate of M (i.e. x_m), and the x coordinate of $M + N$ is b/x_m, leading to $x_m = b/x_m$, which is not possible when $b \notin QR$.

If $P = (x, y) \in \mathcal{E}$ (with $P \neq N, \mathbb{O}$), then $2P = (x', y')$ with:

$$x' = \left(\frac{x^2 - b}{2y}\right)^2$$

This is straightforwardly derived from the classic doubling formulas; one may also apply the formulas from section 3.2 and obtain $2P + N = (b/x', -by'/x'^2)$. The x coordinate of the double of any point is therefore a quadratic residue. Conversely, on a double-odd curve (thus with $b \notin QR$), if $P = (x, y)$ then the x coordinate of $P + N$ is b/x, which is a square if and only if x is not a square. This leads to the following property: in a double-odd curve with equation $y^2 = x^3 + ax^2 + bx$, the curve points segregate into:

- the point-at-infinity $\mathbb{O}$;
- the unique point N of order two, whose x coordinate is zero;
- the non-infinity points of r-torsion, whose x coordinates are non-zero squares in the field;
- the non-infinity points of order exactly $2r$, whose x coordinates are non-squares in the field.

This finally yields an efficient point validation test for double-odd curves: a point (x, y) on the curve is the correct representant $P + N$ of a point $P \in \mathcal{E}[r]$ if and only if either $x = 0$ (in which case $P = \mathbb{O}$), or $x \notin QR$. This validation can be implemented with a single Legendre symbol computation, which can be done efficiently, e.g. with a binary GCD variant [Por20a] or a similar plus/minus algorithm [AHST23].

The fact that $x \in QR$ if and only if (x, y) is the double of another rational point can also be deduced by noticing that x is equal to the unreduced 2-Tate pairing $\langle N, (x, y)\rangle_2$. The reduced 2-Tate pairing (after raising to exponent $(q - 1)/2$) is thus the Legendre symbol of x. The bilinearity of the pairing then implies the result.

3.4 Compression and Canonical Decoding

Using the pivot point N allows for a straightforward encoding of a subgroup element into a compressed format consisting of a single field element. Consider a point $P \in \mathcal{E}[r]$, represented by $P + N = (x, y)$. The line that joins N and $P + N$ intersects the curve in a third point, which is $-(P + N + N) = -P$. Note that $-P \in \mathcal{E}[r]$ while $P + N \notin \mathcal{E}[r]$. Therefore, for a given slope $w \in \mathbb{F}_q$, the line with slope w and passing through N can intersect the curve in at most one correct representant $P + N$ of a point $P \in \mathcal{E}[r]$. We can thus encode a point $P + N = (x, y)$ by mapping it to the line slope $w = y/x$ (for the point N itself, we can use the value $w_N = 0$, since the other points with $y = 0$, if any, have order two and thus are not correct representants $P + N$ for any $P \in \mathcal{E}[r]$).

Decoding w into the matching point representant $P + N$ involves the following:

1. If $w = 0$ then the matching point is N. We can now assume that $w \neq 0$, and the solution (x, y), if any, must have $x \neq 0$ and $y \neq 0$.

2. Since $w = y/x$, we can rewrite the curve equation into $x^2 + (a - w^2)x + b = 0$, a degree-2 equation in x that can be solved by computing its discriminant $\Delta = (a - w^2)^2 - 4b$ and extracting its square root. If the discriminant is not a square then there is no solution and the input w is invalid.

3. Once x is obtained, the corresponding y is computed with $y = wx$. The quadratic equation above, in general, has two solutions, only one of which (at most) being a correct representant of an r-torsion point; thus, point validation should be performed to identify the correct solution.

In the case of a double-odd curve, the process in steps 2 and 3 simplifies into the following: the two solutions to the quadratic equation, if they exist, are such that their product is b, which is not a square for a double-odd curve. Thus, exactly one of the two solutions for x will be a non-square; this solution should be selected, and no further point validation is necessary. The cost of point decoding from the compressed format for a double-odd curve is mostly that of a square root and an additional Legendre symbol to select the correct solution. This process ensures a canonical encoding: a group element (represented by a point $P + N$) can be encoded into a unique field element w, and the decoding process unambiguously reconstructs the corresponding source element, rejecting invalid encodings. Once decoded, the element is guaranteed to be a correct representant of an r-torsion point, and the formulas from section 3.2 can be applied without any exceptional case, thanks to their completeness.

3.5 Point Doubling and Isogenies

For this section, we denote the curve parameters explicitly: $\mathcal{E}(a, b)$ is the curve of equation $y^2 = x^3 + ax^2 + bx$. We use (x, w) coordinates, with $w = y/x$; the w value is the slope of the line from N to (x, y), and we already encountered it in section 3.4. Since $\{\mathbb{O}, N\}$ is a subgroup of $\mathcal{E}(a, b)$, we can use Vélu's formulas [V71] to obtain an isogeny from $\mathcal{E}(a, b)$ into a dual curve, whose kernel is exactly $\{\mathbb{O}, N\}$. Doing it twice, we obtain two isogenies:

$$\begin{array}{rcl}
\psi_1 : \mathcal{E}(a,b) & \longrightarrow & \mathcal{E}(-2a, a^2 - 4b) \\
(x, w) & \longmapsto & \left(w^2, -\dfrac{(x - b/x)}{w} \right) \\
\psi'_{1/2} : \mathcal{E}(-2a, a^2 - 4b) & \longrightarrow & \mathcal{E}(a, b) \\
(x, w) & \longmapsto & \left(w^2/4, -\dfrac{(x - (a^2 - 4b)/x)}{2w} \right)
\end{array}$$

(We also applied an extra scaling by 1/4 in the second isogeny; Vélu's formulas would otherwise lead us to $\mathcal{E}(4a, 16b)$, which is isomorphic to $\mathcal{E}(a, b)$.)

If $\mathcal{E}(a, b)$ is double-odd, then $w \neq 0$ for all points other than N and $\mathbb{O}$, and $\mathcal{E}(-2a, a^2 - 4b)$ is also double-odd. Moreover, for any point $P \in \mathcal{E}(a, b)$, then $\psi'_{1/2}(\psi(P)) = 2P$. These isogenies can be translated into efficient point-doubling formulas with the following characteristics:

- For the purposes of point doublings, we can represent a curve point P in Jacobian (x, w) coordinates with a triplet $(X{:}W{:}J)$ such that $x = X/J^2$ and $w = W/J$, with $J \neq 0$. The special points N and $\mathbb{O}$ are represented with $(0{:}W{:}0)$ and $(W^2{:}W{:}0)$ for any $W \neq 0$, respectively.

- Since we want to work with representants $P + N$, we need to include the overhead of converting the source to Jacobian (x, w) coordinates from whatever coordinate system we are otherwise using for the curve, and converting back the result with the extra addition of N after the doubling. In a sequence of doublings, "internal" doublings in Jacobian (x, w) coordinates avoid these conversion costs, so that the overhead is only a one-time cost for any sequence of successive doublings. Conversions and addition of N can be merged with the first and last doublings in a sequence.

- In Jacobian (x, w) coordinates, a point doubling and addition of N can be computed in cost 2M + 5S generically. If $q = 3 \bmod 4$, the cost can be 1M + 6S. If $q = 3 \bmod 4$, $a = -1$ and $b = 1/2$, the cost is down to 2M + 4S. The best case is when $a = 0$, for which a doubling (without the addition of N) can be done in cost 1M + 5S. Moreover, all these formulas are complete. Details on the formulas are included in appendix B.

3.6 Fractional (x,u) Coordinates

The formulas in section 3.2 use the classic affine (x, y) coordinates. We can obtain other formulas amenable to faster implementations by replacing the y coordinate with another value, namely $u = x/y$. We choose this value u instead of its inverse $w = 1/u$, used above for compression, because we can naturally extend u to the value zero for the point N, where both x and y are zero: indeed, w is the slope of the line from N to a point P, and for N itself, that line should be the tangent to the curve on N, which is vertical; thus, its inverse should intuitively go to zero when reaching N. Replacing y with x/u in the previous formulas and simplifying leads to the following formulas. For two points P_1 and P_2 in $\mathcal{E}[r]$, represented by $P_1 + N = (x_1, u_1)$ and $P_2 + N = (x_2, u_2)$, respectively, their sum is represented by $P_3 + N = P_1 + P_2 + N = (x_3, u_3)$ with:

$$x_3 = \frac{b((x_1 + x_2)(1 + au_1u_2) + 2u_1u_2(x_1x_2 + b))}{(x_1x_2 + b)(1 - au_1u_2) - 2bu_1u_2(x_1 + x_2)}$$
$$u_3 = \frac{-(u_1 + u_2)(x_1x_2 - b)}{(x_1x_2 + b)(1 + au_1u_2) + 2bu_1u_2(x_1 + x_2)}$$

These formulas are complete, as long as the two inputs are correct, i.e. have been validated to be proper representants of r-torsion points. In fact, in all generality, the formulas above have been derived under the assumption that $P_1 + N \neq \pm P_2$, and P_1 and P_2 are distinct from N and $\mathbb{O}$. It can be verified that they still work when $P_1 + N$ and/or $P_2 + N$ is equal to N, and the point validation process from section 3.3 ensures that the other cases cannot happen.

In a practical implementation, we can use a fractional representation: a point (x, u) is represented by a quadruplet $(X{:}Z{:}U{:}T)$ such that $Z \neq 0$, $T \neq 0$, $x = X/Z$ and $u = U/T$. This leads to general point addition formulas with cost 10M; sequences of n successive point doublings can be computed with cost as low as $3\text{M} + n(1\text{M} + 5\text{S})$ for some specific curves. These processes are detailed in appendix C.

4 Double-Odd Jacobi Quartic

In section 3, we described a generic process by which a prime-order group abstraction could be extracted from an elliptic curve with an even order. In case the curve is double-odd, the point decompression and validation is reasonably efficient, though somewhat slower than the equivalent process in, for instance, Ristretto. The derived formulas in fractional (x, u) coordinates are also slightly less efficient than for twisted Edwards curves, except for long sequences of successive doublings. We can, however, do better, with another change of coordinates.

4.1 Map to a Jacobi Quartic

As previously, we consider a finite field $\mathbb{F}_q$ of characteristic 3 or more, and an elliptic curve $\mathcal{E}$ of even order, expressed with the non-short Weierstraß equation $y^2 = x^3 + ax^2 + bx$

for two constants a and b. We now focus on a double-odd curve, i.e. its order is $2r$ for some odd integer r; as explained in section 3.3, this means that both b and $a^2 - 4b$ are non-squares. Point-at-infinity is denoted $\mathbb{O}$; the point $N = (0, 0)$ is the only point of order two on the curve. For all other curve points, both x and y coordinates are well-defined and non-zero.

Given a point (x, y) on the curve (other than N and $\mathbb{O}$), we define the (e, u) coordinates as follows:

$$\begin{aligned} u &= \frac{x}{y} \\ e &= u^2 \left(x - \frac{b}{x} \right) \end{aligned}$$

Applying the curve equation, the coordinate e can also be written as:

$$e = \frac{x^2 - b}{x^2 + ax + b}$$

This last formula can also be computed for point N, yielding $e = -1$. We formally define the (e, u) coordinates of N and $\mathbb{O}$ to be $(-1, 0)$ and $(1, 0)$, respectively.

The mapping $(x, y) \mapsto (e, u)$ is reversible, by noticing that:

$$x = \frac{1}{2u^2} \left(u^2 \left(x - \frac{b}{x} \right) + u^2 \left(x + \frac{b}{x} \right) \right) = \frac{1}{2u^2} \left(e + 1 - au^2 \right)$$

We can thus use (e, u) coordinates to represent all curve points without any loss of information.

In (e, u) coordinates, the curve equation becomes:

$$e^2 = (a^2 - 4b)u^4 - 2au^2 + 1$$

which is a form known as the (extended) *Jacobi quartic*, first studied by Jacobi in the 19th century [Jac29]. In the usual formulation, the equation is denoted $Y^2 = DX^4 + 2AX^2 + 1$, with subvariants depending on whether D is a square or not. In our case, e and u coordinates play the role of Y and X, respectively, and the D constant is $a^2 - 4b$, which is a non-square for all double-odd curves.

The mapping goes in both directions: a Jacobi quartic $Y^2 = DX^4 + 2AX^2 + 1$ is turned into a curve of equation $y^2 = x(x^2 + ax + b)$ with $a = -A$ and $b = (A^2 - D)/4$ by setting $x = (Y + 1 + AX^2)$ and $y = x/X$. Thus, the Jacobi quartic $Y^2 = DX^4 + 2AX^2 + 1$ is a double-odd curve if and only if $D \notin QR$ and $A^2 - D \notin QR$.

There exists some extensive literature on Jacobi quartics and their formulas [BJ03, CC86, Jac29, WW27]. The following classic point addition formulas can also be derived straightforwardly from the (x, u) formulas shown in section 3.6; for points $P_1 = (e_1, u_1)$ and $P_2 = (e_2, u_2)$, their sum $P_3 = (e_3, u_3) = P_1 + P_2$ can be computed as:

$$\begin{aligned} e_3 &= \frac{(1 + (a^2 - 4b)u_1^2u_2^2)(e_1e_2 - 2au_1u_2) + 2(a^2 - 4b)u_1u_2(u_1^2 + u_2^2)}{(1 - (a^2 - 4b)u_1^2u_2^2)^2} \\ u_3 &= \frac{e_1u_2 + e_2u_1}{1 - (a^2 - 4b)u_1^2u_2^2} \end{aligned}$$

These formulas are complete *for all curve points*: it can be easily verified that they also work for the cases on which the (x, u) formulas would have failed (note that since $a^2 - 4b \notin QR$, the denominator $1 - (a^2 - 4b)u_1^2u_2^2$ cannot be zero).

In (e, u) coordinates, if $P = (e, u)$ then $P + N = (-e, -u)$ and $-P = (e, -u)$ (this explains why the coordinates of $\mathbb{O}$ were chosen to be $(1, 0)$, since N has coordinates $(-1, 0)$).

It should be noted that the above formulas were considered in the case of double-odd curves, but they are still complete for curves whose order is a multiple of 4, as long as $a^2 - 4b \notin QR$, i.e. that N is the only point of order 2. If $b \in QR$ then there are points of order 4 (see section 3.3) whose x coordinate is a square root of b; for such points, the e coordinate is equal to zero. On double-odd curves, $e \neq 0$ for all curve points.

4.2 Prime-Order Group and Point Compression

Since the formulas over the Jacobi quartic are complete on a double-odd curve, we can redefine the prime-order group abstraction with a different description that leads to a more efficient point compression and decompression process. Namely, we consider the *quotient group* $\mathcal{E}/\{\mathbb{O}, N\}$, which is homomorphic to $\mathcal{E}[r]$; elements of the quotient group are pairs $\{P, P+N\}$, and any two such pairs are either identical or disjoint. Equivalently, we are still working with $\mathcal{E}[r]$ but for each point $P \in \mathcal{E}[r]$ we allow two possible representants, which are P and $P+N$ (whereas in section 3 we enforced the use of only $P+N$ as a valid representant). We do not really care which representant we are using, since both work; we only need a normalizing rule for canonical encoding, which we will now define.

Let sign be an arbitrary *sign function* over $\mathbb{F}_q$, i.e. a function such that:

- For any $z \in \mathbb{F}_q$, $\mathsf{sign}(z) \in \{0, 1\}$.
- $\mathsf{sign}(0) = 0$.
- For any $z \in \mathbb{F}_q$ such that $z \neq 0$, $\mathsf{sign}(-z) = 1 - \mathsf{sign}(z)$.

There are many such functions; a convenient one for prime fields (fields of integers modulo a given prime q) is to represent the value z as an integer $i \in [0, q-1]$ range, and use $\mathsf{sign}(z) = i \bmod 2$ (the “least significant bit” of the value). A value z is said to be “negative” if $\mathsf{sign}(z) = 1$, or “non-negative” if $\mathsf{sign}(z) = 0$.

Encoding of a point $P = (e, u)$ into an element of $\mathbb{F}_q$ uses the following process:

1. If $\mathsf{sign}(e) = 1$, then return $-u$, otherwise return u.

Since $P + N = (-e, -u)$, P and $P+N$ encode into the same output value. An equivalent description of encoding is that for a given element $\{P, P+N\}$ of the quotient group, the two points P and $P+N$ are the two possible representants, and exactly one of them has a non-negative e coordinate; we use the u coordinate of that specific representant. When encoding the group neutral $\{\mathbb{O}, N\}$, we get zero.

Decoding of a given value u into a point P is performed by recomputing a matching e coordinate, as follows:

1. Compute $\Delta = (a^2 - 4b)u^4 - 2au^2 + 1$.
2. If $\Delta \notin QR$, then there is no solution and u is invalid. Otherwise, let $e = \sqrt{\Delta}$.
3. If $\mathsf{sign}(e) = 1$ then replace e with $-e$.
4. Return (e, u).

This process illustrates that decoding must work for any validly encoded point, but also that it is unambiguous: if there is a matching e coordinate, then there can be at most two solutions for e, but they are opposite to each other and thus cannot be both non-negative, and the output of the decoding process is necessarily a representant of the same quotient group element as was used as input to the encoding process.

Starting from (e, u) affine coordinates, the encoding process cost is trivial; realistically, an implementation will use a fractional representation of the coordinates, and the cost of the encoding is the cost of normalizing that representation to affine coordinates, i.e. mostly one inversion in the field. This is faster than the encoding process in Decaf or Ristretto, where a combined square root and inversion must be performed: when working in a prime field, that operation requires an exponentiation with an exponent of about the same size as the modulus, whereas a single inversion, as needed for double-odd Jacobi quartics, can be performed more efficiently with a binary GCD derivative [BY19, Por20a]. For decoding, a square root computation is needed, and should offer performance similar to that of Decaf or Ristretto, albeit with a somewhat simpler process since there is no combined square root/inversion operation; the decoding process inherently outputs affine (e, u) coordinates.

It may be noted that all of the above was described in the context of double-odd curves, but can be potentially applied to curves whose order is a multiple of 4, in which case either or both of the following apply:

- There are two other points of order two on the curve, with $y = 0$ but $x \neq 0$ (in (x, y) coordinates). For these two points, the u coordinate is not well-defined; moreover, the addition formulas are incomplete, since they fail for such points as input or as output. This case corresponds to $a^2 - 4b \in QR$.
- There are two points of order four; if one of them is T then the other is $T + N$. These two points have the same x coordinate, and for both $e = 0$. They thus have two distinct coordinates u, but both have a non-negative e, and the process described above is ambiguous. This case corresponds to $b \in QR$.

The second situation can be solved with an extra normalization rule. The problem, in that case, is that the encoding process cannot choose which of T or $T + N$ to use, and the encoding is not canonical; the decoding process still works properly, in that it will return either T or $T + N$ but both represent the same quotient group element. In order to obtain a canonical encoding, we only need to apply an extra normalization rule, e.g.: if $e = 0$ and $\mathsf{sign}(u) = -1$, then replace u with $-u$. With this extra rule, we obtain a canonical encoding of the quotient group for all even-ordered curves, as long as they contain a single point of order two (this would apply to both Curve448 and Curve25519, for instance). Of course, if the curve is not double-odd, but its order is a multiple of 4, then the quotient group has even order and we will not obtain a prime-order group abstraction that way.

4.3 Isogenies and Point Equality

In section 3.5, we defined the isogenies ψ_1 and $\psi'_{1/2}$, over the curve points in (x, w) coordinates. The formulas can be straightforwardly transformed into (e, u) coordinates:

$$\begin{aligned}
\psi_1 : \mathcal{E}(a, b) &\longrightarrow \mathcal{E}(-2a, a^2 - 4b) \\
(e, u) &\longmapsto \left(\frac{1 - (a^2 - 4b)u^4}{e^2}, -\frac{u}{e}\right) \\
\psi'_{1/2} : \mathcal{E}(-2a, a^2 - 4b) &\longrightarrow \mathcal{E}(a, b) \\
(e, u) &\longmapsto \left(\frac{1 - 16bu^4}{e^2}, -\frac{2u}{e}\right)
\end{aligned}$$

For a double-odd curve, $e \neq 0$ for all curve points, and these expressions are well-defined without any exceptional case.

If a canonical encoding is defined, as in section 4.2, then two points can be compared with each other by encoding both and checking whether the outputs are identical. This can be inefficient, though, as encoding typically involves normalization from a fractional representation to affine coordinates, hence at least one field inversion. A faster method can be obtained by using the ψ_1 isogeny, whose kernel is $\{\mathbb{O}, N\}$; for any points P_1 and P_2 on the curve, the two points represent the same quotient group element if and only if $\psi_1(P_1) = \psi_1(P_2)$. Moreover, $\psi_1(P_1)$ is necessarily an r-torsion point in the destination curve ($\mathcal{E}(-2a, a^2 - 4b)$), and no two distinct elements of the subgroup of r-torsion points may share the same u coordinate.

We thus only need to compute the u coordinates of $\psi_1(P_1)$ and $\psi_1(P_2)$ and compare the two values together, which can be done while staying in the fractional domain. As expressed above, the two values are $-u_1/e_1$ and $-u_2/e_2$, respectively; therefore, the points P_1 and P_2 represent the same quotient group element if and only if $u_1e_2 = u_2e_1$.

4.4 Extended (e,u) Coordinates

The formulas described above can be transformed into an implementation by using several possible representations. Since the addition formulas shown in section 4.1 were already known, several previous papers explored them; the most efficient were found by Duquesne [Duq07], and also rediscovered by Hisil, Wong, Carter and Dawson [HWCD09] under different notations that ultimately resolve to the same elementary operations. We use notations close to the ones used by Hisil *et al*: a point (e, u) is represented by a quadruplet of values $(E{:}Z{:}U{:}T)$ such that $Z \neq 0$, $e = E/Z$, $u = U/Z$ and $U^2 = TZ$ (which implies that $u^2 = T/Z$). The cost is 8M + 3S; this is similar to the 10M cost for (x, u) coordinates (section 3.6). On the other hand, we can apply to double-odd curves in Jacobi quartic form the same improvements on sequences of doublings as in (x, u) coordinates, with a lower per-sequence overhead:

- A single doubling can be done in cost 2M + 5S for all curves. If $q = 3 \bmod 4$, or if $a = 0$, then the cost can be lowered to 1M + 6S.
- The single doubling formulas internally use Jacobian (x, w) coordinates, and additional doublings can be performed with the costs expressed in section 3.5, down to 1M + 5S per doubling on curves with $a = 0$.

All the relevant formulas are described in appendix D.

Compared with twisted Edwards curves, the double-odd Jacobi quartics offer slower generic addition but faster doublings: on twisted Edwards curves, generic point addition has cost 8M with extended coordinates [HWCD08], but single doublings in the same coordinate system have cost 4M + 4S (additional doublings can be performed at cost 3M + 4S). The overall gain or loss of Jacobi quartics depends on how many operations of each type are used in a given protocol. The multiplication of a group element by a scalar is a very common primitive, and it is typically implemented with a window-optimized double-and-add algorithm; for such an implementation, there will be four or five point doublings for every generic addition, and the faster doubling formulas make Jacobi quartics faster than twisted Edwards curves for this task.

4.5 j255e and jq255s

As a concrete example of double-odd Jacobi quartics amenable to fast software implementations, we define two curves at the usual "128-bit" security level. In both cases we use fields of integers modulo the prime $q = 2^{255} - m$ for a small integer m, for which fast implementations are feasible, especially on low-power microcontrollers, as long as m is small enough (roughly speaking, best performance is achieved with $m < 2^{15}$).

The jq255e curve uses $a = 0$, so as to leverage the fastest doublings. Such a curve has j-invariant equal to 1728. We need $q = 1 \bmod 4$ to avoid the curve being supersingular; we prefer to set $q = 5 \bmod 8$ so that square roots can still be computed with relative ease (using Atkin's formulas [Atk92]). For such curves, trying $b = \pm 2$ exhausts all possibilities; we thus look for the lowest $m = 3 \bmod 8$ such that $q = 2^{255} - m$ is prime and $y^2 = x^3 + bx$ defines over $\mathbb{F}_q$ an elliptic curve of order $2r$ for a prime r (which will be close to 2^{254}). The first hit is for $m = 18651$ and $b = -2$. Using the methodology described here, this yields a group of prime order r (which is very slightly lower than 2^{254}). As a nice bonus, such a curve is a GLV curve [GLV01], with a very efficient endomorphism that can be used to achieve substantial speed-ups for some operations, in particular multiplication of a point by a scalar (the endomorphism is simply $(e, u) \mapsto (e, u\sqrt{-1})$).

The jq255s curve is an alternate choice such that $q = 3 \bmod 4$, $a = -1$ and $b = 1/2$: for such curves, doublings in Jacobian (x, w) coordinates can be done in cost 2M+4S. Again, we select the lowest m such that $q = 2^{255} - m$ is prime and the equation $y^2 = x^3 - x^2 + 1/2$ defines a curve over $\mathbb{F}_q$ of order $2r$ for a prime r; the first match is $m = 3957$. The point of jq255s is to demonstrate that double-odd Jacobi quartics can achieve performance at least on par with twisted Edwards curves without requiring the extra structure of a GLV curve; in practical situations, though, jq255e should be preferred.

For both curves, we define and use the prime-order quotient groups, as explained in the previous sections. Over these groups, we can define efficient key exchange (Diffie-Hellman [DH76]) and signature (Schnorr [Sch90]) protocols; for the latter, we leverage the curiously neglected remark from Schnorr that the "challenge" part only needs to be half the size of the group, thus yielding signatures of size 48 bytes at the 128-bit security level, which is shorter than the commonly encountered 64 bytes from Ed25519 [BDL+11] or standard ECDSA on the P-256 curve [ECD23][2]. Half-size challenges still ensure the expected security [NSW09]. A full specification of the jq255e and jq255s groups, along with the key exchange, signature, and hash-to-curve protocols, is available at:

https://c2sp.org/jq255

Open-source implementations and a summary of the formulas are available from the dedicated Web site at:

https://doubleodd.group/

As an example of the achieved performance, on an Intel Core i5-8259U x86 processor ("Coffee Lake" core, in 64-bit mode), we achieve, with an optimized (and fully constant-time) implementation in Rust, a general jq255e point multiplication by a scalar in about 69k clock cycles, and a signature verification in 82k cycles. For jq255s, which does not benefit from a GLV endomorphism, the values are 103k and 83k, respectively. For Curve25519, the corresponding values are 103k and 108k[3]. The faster signature verifications for jq255e and jq255s come mostly from the use of shorter challenge values (which also make signatures shorter); the compared Ed25519 implementation leverages the Antipa *et al* optimization [ABG+06, Por20b] but is still more expensive because of its longer challenge values.

5 Binary Curves

So far we have considered the case of finite fields with characteristic 3 or more. We now explore how our generic methodology can be applied to elliptic curves defined over fields of characteristic two; such fields and curves are often said to be "binary".

[2]In the case of Ed25519, the longer signature is explicitly meant to support batch verification; the lack of use of compressed challenges is still a common feature of most Schnorr-derived schemes, and rarely explained.

[3]An x-only point multiplication routine is often used with Curve25519, under the name "X25519"; it is sufficient for some protocols, e.g. most usages of Diffie-Hellman key exchange. X25519 can be faster than generic point multiplication; 88k cycles have been reported on a CPU type equivalent to our test system [x2523]. However, X25519 does not validate that incoming points are part of the expected subgroup, admits multiple equivalent inputs, and does not yield the complete output point to support further arbitrary group operations; it thus departs from the prime-order group abstraction. We consider here a full point multiplication, which is somewhat more expensive.

5.1 Binary Fields

Let $q = 2^m$. We can define the field $\mathbb{F}_q$ as the ring of polynomials with binary coefficients ($\mathbb{F}_2[z]$) with all computations performed modulo a given polynomial M of degree m; any M can be used as long as it is irreducible. All finite fields with the same cardinal are isomorphic to each other, and the isomorphisms can be computed in practice; therefore, no choice of M is better or worse than any other from a security point of view. We can thus choose a modulus M that favours performance (e.g. with a very low Hamming weight). We recall here some relevant properties of binary fields:

- Squaring is a field automorphism: for any field elements x and y, $(xy)^2 = x^2y^2$ and $(x+y)^2 = x^2 + y^2$. The same applies to raising to power 2^n for any integer n, including negative values (e.g. $\sqrt{x+y} = \sqrt{x} + \sqrt{y}$). Every field element is a square and has a unique square root. Raising to the power 2^n is a linear operation (when considering $\mathbb{F}_q$ as a vector space of dimension m over $\mathbb{F}_2$) and can be implemented efficiently.

- Inversion can be performed with a variant of Fermat's little theorem described by Itoh and Tsujii [IT88]. This does not make inversions fast enough to contemplate using them for each curve point addition, but it still makes them fast enough to justify some practices, e.g. normalizing to affine coordinates a "window" of values for an optimized double-and-add algorithm.

- The *trace* of a field element x is:

$$\mathrm{Tr}(x) = \sum_{i=0}^{m-1} x^{2^i}$$

  The trace is linear; it is always equal to 0 or 1; the trace of 0 is 0. For any x, $\mathrm{Tr}(x^2) = \mathrm{Tr}(x)^2 = \mathrm{Tr}(x)$. If m is odd then $\mathrm{Tr}(1) = 1$. The trace of an element x can always be efficiently computed because it is equal to the sum of a few specific coefficients of x as a binary polynomial in z (which coefficients are part of the trace expression depends on the choice of the field modulus M). Correspondingly, there is always at least one field element of trace 1 with minimal Hamming weight (i.e. a single bit).

- Quadratic equations can be reduced to either a square root computation, or to the equation $x^2 + x = d$ for some field element d. If $\mathrm{Tr}(d) = 1$ then the equation has no solution; if $\mathrm{Tr}(d) = 0$ then there are two solutions; if x is a solution then $x + 1$ is the other solution. When m is odd, a solution can be obtained with the halftrace:

$$H(d) = \sum_{i=0}^{(m-1)/2} d^{2^{2i}}$$

  In fact, for an odd m, we always have $H(d)^2 + H(d) = d + \mathrm{Tr}(d)$. Since the halftrace is linear, it can be computed with a simple bit-by-bit process (or even faster with look-up tables to process bits by chunks).

5.2 Structure of Binary Curves

Over a binary field, elliptic curves with an odd order are supersingular, which is usually a problem for security, because discrete logarithm over supersingular curves can normally be reduced to the much easier problem of discrete logarithm in a finite field through pairings. Practical binary elliptic curves thus have even order.

A non-supersingular binary curve can always be represented, through changes of variables, with a short Weierstraß equation:

$$y^2 + xy = x^3 + Ax^2 + B$$

for two constants A and B. Moreover, the change of variable $y \mapsto y + Cx$, for any constant C, modifies the equation by replacing A with $A + C^2 + C$, but keeping B unchanged. Since $C^2 + C$ can be any field element of trace zero, we can always replace A with any other constant with the same trace. In particular, when $\mathrm{Tr}(A) = 0$, we can replace A with zero, and otherwise we can arrange for A to have minimal Hamming weight. The ten standard NIST binary curves (B-* and K-* curves with field degrees m ranging from 163 to 571 [NIS23]) systematically use an odd m, and constants A equal to zero or one.

The curve order is $n = 2^t r$ for some integer $t \geq 1$, and odd integer r. Moreover, the subgroup of 2^t-torsion points is always cyclic, and there are 2^{t-1} generators; we call T one of these generators (arbitrarily chosen). There is a single point of order two, which is $N = 2^{t-1}T = (0, \sqrt{B})$. When $\mathrm{Tr}(A) = 1$, $t = 1$, i.e. the curve has order $2r$ (this is then a double-odd curve).

We can apply the methodology exposed in section 3. The following notes apply:

- A change of variable is applied, to move N to $(0, 0)$. Namely, we add $\sqrt{B}$ to the y coordinate, and we get the equation:

  $$y^2 + xy = x^3 + ax^2 + bx$$

  for constants $a = A$ and $b = \sqrt{B}$. In this new equation, adding N to point (x, y) yields:

  $$(x, y) + N = \left(\frac{b}{x}, \frac{b(y + x)}{x^2}\right)$$

- Point validation involves, as in section 3.3, checking whether a given point can be halved t times. A point (x, y) can be halved if and only if $\mathrm{Tr}(x) = \mathrm{Tr}(A)$; computing the half-point involves solving a quadratic equation, which is feasible at moderate cost (e.g. with the halftrace when m is odd); see [Knu99] for details. In general, given a point $P = (x, y)$, we can not only check whether it is an r-torsion point, but we can even compute the unique point Q and integer k such that $Q \in \mathcal{E}[r]$, $0 \leq k < 2^t$, and $P = Q + kT$. When $\mathrm{Tr}(A) = 1$, this is especially easy since $T = N$ and $k = 1 - \mathrm{Tr}(x)$. Point validation can therefore be extended into full-curve computations by splitting points into their r-torsion and 2^t-torsion parts, and performing computations over the latter with simple integer additions modulo 2^t. For a prime-order group abstraction this is not needed; we only need to ensure that a given point is equal to $P + N$ for P an r-torsion point.

- As in section 3.4, we can compress a point $P + N = (x, y)$ into the value $w = y/x$; decompression can be coupled with point validation, and requires one inversion and solving t quadratic equations.

- The use of $P + N$ instead of P immediately yields unified formulas for point additions, but they are not especially fast, and some exceptional cases remain when the neutral point appears as operand or output. To solve both issues, we change the coordinate system; this will be described in the next section.

5.3 (x,s) Coordinates

We consider the field $\mathbb{F}_q$ with $q = 2^m$, and the elliptic curve of equation $y^2 + xy = x^3 + ax^2 + bx$ for two constants a and b (with $b \neq 0$). The curve has order $2^t r$ for an odd integer r (the curve is usually chosen so that r is prime, but this is not strictly necessary here). An r-torsion point P is represented by the point $P + N$, with $N = (0, 0)$ being the unique point of order two on the curve. For such a point $P + N = (x, y)$, we define the extra coordinate s:

$$s = y + x^2 + ax + b$$

Given x, s can always be computed from y and vice versa; we can thus use (x, s) coordinates without any loss of information. When $x \neq 0$, we have $s = y^2/x$, and $y/x = \sqrt{s/x}$ (which can be used for point compression).

Applying the s coordinates to the unified formulas obtained by using our methodology yields the following formulas. Given $P_1 + N = (x_1, s_1)$ and $P_2 + N = (x_2, s_2)$, which represent r-torsion points P_1 and P_2, respectively, the representant $P_3+N = (P_1+P_2)+N = (x_3, s_3)$ is such that:

$$\begin{aligned} x_3 &= \frac{b(x_1x_2 + s_1x_2 + s_2x_1)}{(x_1x_2 + b)^2} \\ s_3 &= \frac{b(b^2(s_1s_2 + a^2x_1x_2) + x_1^2x_2^2((a^2 + 1)x_1x_2 + s_1x_2 + s_2x_1 + s_1s_2))}{(x_1x_2 + b)^4} \end{aligned}$$

These formulas were derived from unified formulas, but it can be verified that they also work for the neutral N, both as operand and as output. These formulas are thus complete (for correct representants of r-torsion points).

In (x, s) coordinates, the neutral is $N = (0, b)$; the opposite of $P + N = (x, s)$ is $-P + N = (x, s + x)$. A practical implementation may represent a point $P + N = (x, s)$ with a quadruplet $(X{:}S{:}Z{:}T)$, such that $Z \neq 0$, $x = \sqrt{b}X/Z$, $s = \sqrt{b}S/Z^2$ and $T = XZ$ (the factors "$\sqrt{b}$" are used here to reduce the number of multiplications by $\sqrt{b}$ in the formulas).

Since our methodology applies to pre-existing standard curves, and the NIST B-* curves use pseudorandom B constants that were *not* chosen so that multiplication by B is inexpensive, we also include such multiplications in cost estimates; specifically, we denote by $\mathrm{m_b}$ the cost of multiplying by $\sqrt{b} = \sqrt[4]{B}$. With our chosen representation of points as quadruplets $(X{:}S{:}Z{:}T)$, we obtain generic point addition complete formulas with cost $8\mathrm{M} + 2\mathrm{S} + 2\mathrm{m_b}$ (when $a = 0$, this is reduced to $7\mathrm{M} + 2\mathrm{S} + 2\mathrm{m_b}$); this is better than the previously best known formulas, which were furthermore incomplete (using (x, λ) coordinates, with cost $11\mathrm{M} + 2\mathrm{S}$ [OLAR13]). We can also compute a sequence of n point doublings in cost $n(2\mathrm{M} + 4\mathrm{S} + 2\mathrm{m_b}) + 2\mathrm{S} + 6\mathrm{m_b}$, or in cost $n(3\mathrm{M} + 4\mathrm{S} + \mathrm{m_b}) + 3\mathrm{m_b}$ (which one is faster depends on the implementation platform, the constant b, and the number of successive doublings n). All formulas are detailed in appendix E.

5.4 Application to GLS254

We applied the (x, s) formulas to the GLS254 curve. That curve was defined by Aardal and Aranha [AA22], following up on previous work by Oliveira *et al* [OLAR14], and leveraging an extensive security analysis by Hankerson, Karabina and Menezes [HKM09], showing that the chosen parameters are not susceptible to the GHS attack [GHS02] even though the curve uses a composite extension degree $m = 254$. The GLS254 curve uses the GLS structure [GLS09], which offers an efficient endomorphism that can be used to optimize some operations such as multiplication of a point by a scalar.

The base field $\mathbb{F}_q$, with $q = 2^{254}$, is defined as a tower of field extensions:

- A field of order 2^{127} is obtained by considering binary polynomials in z, modulo $M = z^{127} + z^{63} + 1$.
- Elements of $\mathbb{F}_{2^{254}}$ are $x = x_0 + ux_1$, where x_0 and x_1 are elements of $\mathbb{F}_{2^{127}}$, and u is a formal value such that $u^2 + u = 1$.

The trace (in $\mathbb{F}_{2^{254}}$) of $x_0 + ux_1$ is equal to the trace of x_1 (in $\mathbb{F}_{2^{127}}$), which is itself equal to the least significant bit of x_1 (its degree-zero coefficient as a binary polynomial of degree at most 126). Inversions in $\mathbb{F}_{2^{254}}$ can be reduced to inversions in $\mathbb{F}_{2^{127}}$ by noticing that $(x_0 + ux_1)(x_0 + x_1 + ux_1) = (x_0^2 + x_0x_1 + x_1^2)$, which is an element of $\mathbb{F}_{2^{127}}$. A quadratic equation in $\mathbb{F}_{2^{254}}$ can similarly be solved by computing two halftraces in $\mathbb{F}_{2^{127}}$.

The curve equation parameters are $A = u$ and $B = 1 + z^{27}$. The value B was chosen so that multiplications by B are especially efficient. The curve order is $2r$, with $r \approx 2^{253}$. It should be noted that $\mathrm{Tr}(A) = 1$; point validation is thus especially simple since it suffices to check that x has trace 0 (to ensure that a point is a $P + N$ representant of an r-torsion point P), and no actual point halving is required.

For better performance, when converting to (x, s) coordinates, we first raise everything to the power 4. Raising to power 4 is a field automorphism; the curve algebraic structure is thus preserved. With this step, we define $b = B^2$, and $\sqrt{b} = B = 1 + z^{27}$; this operation thus ensures that multiplications by $\sqrt{b}$ are fast.

The curve endomorphism ζ is the following. For point P represented by $P + N = (x, s)$, the endomorphism output is point P' represented by $P' + N = (x', s')$ with:

$$\begin{aligned} x' &= \phi(x) && = (x_0 + x_1) + ux_1 \\ s' &= \phi(s) + (u+1)\phi(x) && = (s_0 + s_1 + x_0) + u(s_1 + x_0 + x_1) \end{aligned}$$

where $\phi(v) = v^{2^{127}}$ (Frobenius endomorphism on the field as a degree-2 extension of $\mathbb{F}_{2^{127}}$). In the quadruplet representation $(X{:}S{:}Z{:}T)$, this yields:

$$\begin{aligned} X' &= \phi(X) && = (X_0 + X_1) + uX_1 \\ S' &= \phi(S) + (u+1)\phi(T) && = (S_0 + S_1 + T_0) + u(S_1 + T_0 + T_1) \\ Z' &= \phi(Z) && = (Z_0 + Z_1) + uZ_1 \\ T' &= \phi(T) && = (T_0 + T_1) + uT_1 \end{aligned}$$

which can be computed with only a few field additions, which are just bitwise XORs.

Using these formulas, our implementation (fully constant-time, in Rust) achieves signature generation and verification in 18k and 27k cycles on our test Intel Core i5-8259U x86 processor, respectively (again using 48-byte Schnorr signatures). This is 3 to 4 times faster than the fastest Ed25519 implementations on that same platform. This code leverages the carryless multiplication opcode (`pclmulqdq`) that originally appeared on x86 CPUs along with the AES hardware implementation (because it helps with the GCM authenticated encryption mode). On platforms that do *not* have such an opcode, we can use integer multiplications and masking, as originally suggested by Knuth [Knu69] (exercise 4 of section 4.6, page 363, solution on page 536). On a RISC-V SiFive-U74 core, with fast 64-bit multiplications but no carryless multiplication, we sign and verify in 378k and 636k cycles with GLS254, compared to 158k and 304k for Ed25519 on the same platform; the GLS254 performance is still sufficient for most uses of signatures, but not a record breaker.

6 Conclusion

We presented a generic method to avoid most or all of exceptional cases in point addition formulas on large classes of elliptic curves. The method can be applied to any curve with even order, on any kind of finite field (including binary fields); it offers secure prime-order group abstractions free of cofactor issues and amenable to constant-time implementations. The formulas are also as fast as, or faster than, existing fast curves, so that the security of formula completeness and removal of cofactors does not come at the price of reduced performance.

Existing standard binary curves (e.g. the NIST B-* and K-* curves) can immediately benefit from our formulas. For curves in odd characteristic fields, many standard curves have a prime order (e.g. P-256 or secp256k1) and our methods cannot be used with them; curves meant to be used in Montgomery or Edwards form (e.g. Curve25519) are compatible with our methodology, though they are not optimal in the sense that point validation can be expensive. In odd characteristic fields, the optimal case is double-odd curves, which is why we defined two such curves (jq255e and jq255s).

All our treatment considered the case of a curve subgroup of prime order r; however, our formulas also work with any odd composite r. A potential application is CSIDH, which, as defined in [CLM+18], works in a finite field $\mathbb{F}_q$ with $q = 3 \bmod 8$, over which (supersingular) curves of order $q+1$ are defined. For such curves, $r = (q+1)/4$ is a product of many small primes. Moreover, the suggested implementation of CSIDH already uses a curve equation with form $y^2 = x^3 + ax^2 + x$, which matches our equation from section 3.2 (using $b = 1$). Whether our methodology can help with speed and/or convenience of CSIDH implementations remains to be explored.

Acknowledgments

We thank the anonymous reviewers for their useful comments and suggestions.

References

[AA22] Marius A. Aardal and Diego F. Aranha. 2D-GLS: Faster and exception-free scalar multiplication in the GLS254 binary curve. Cryptology ePrint Archive, Report 2022/748, 2022. https://eprint.iacr.org/2022/748.

[ABG+06] Adrian Antipa, Daniel R. L. Brown, Robert Gallant, Rob Lambert, René Struik, and Scott A. Vanstone. Accelerated verification of ECDSA signatures. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 307–318. Springer, Heidelberg, August 2006. doi:10.1007/11693383_21.

[AHST23] Diego F. Aranha, Benjamin Salling Hvass, Bas Spitters, and Mehdi Tibouchi. Faster constant-time evaluation of the Kronecker symbol with application to elliptic curve hashing. In *ACM CCS 2023*, pages 3228–3238. ACM Press, November 2023. doi:10.1145/3576915.3616597.

[AKR12] Christophe Arene, David Kohel, and Christophe Ritzenthaler. Complete addition laws on abelian varieties. *LMS Journal of Computation and Mathematics*, 15:308–316, 2012. doi:10.1112/S1461157012001027.

[ALdV] Tony Arcieri, Isis Lovecruft, and Henry de Valence. The Ristretto group. URL: https://ristretto.group/.

[Atk92] A. Oliver L. Atkin. Probabilistic primality testing (summary by F. Morain). Technical Report 1779, INRIA, 1992. URL: http://algo.inria.fr/seminars/sem91-92/atkin.pdf.

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In Serge Vaudenay, editor, *AFRICACRYPT 08*, volume 5023 of *LNCS*, pages 389–405. Springer, Heidelberg, June 2008. doi:10.1007/978-3-540-68164-9_26.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, Heidelberg, September / October 2011. doi:10.1007/978-3-642-23951-9_9.

[BJ03] Olivier Billet and Marc Joye. The Jacobi model of an elliptic curve and side-channel analysis. In Marc Fossorier, Tom Høholdt, and Alain Poli, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 2643 of *LNCS*, pages 34–42. Springer, Heidelberg, April 2003. doi:10.1007/3-540-44828-4_5.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 29–50. Springer, Heidelberg, December 2007. doi:10.1007/978-3-540-76900-2_3.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019(3):340–398, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8298. doi:10.13154/tches.v2019.i3.340-398.

[CC86] David V. Chudnovsky and Gregory V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986. doi:10.1016/0196-8858(86)90023-0.

[CD20] Wouter Castryck and Thomas Decru. CSIDH on the surface. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 111–129. Springer, Heidelberg, June 2020. doi:10.1007/978-3-030-44223-1_7.

[CGN20] Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many EdDSAs. Cryptology ePrint Archive, Report 2020/1244, 2020. https://eprint.iacr.org/2020/1244.

[CJ19] Cas Cremers and Dennis Jackson. Prime, order please! Revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman. In Stephanie Delaune and Limin Jia, editors, *CSF 2019 Computer Security Foundations Symposium*, pages 78–93. IEEE Computer Society Press, June 2019. doi:10.1109/CSF.2019.00013.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 395–427. Springer, Heidelberg, December 2018. doi:10.1007/978-3-030-03332-3_15.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.

[DIK06] Christophe Doche, Thomas Icart, and David R. Kohel. Efficient scalar multiplication by isogeny decompositions. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 191–206. Springer, Heidelberg, April 2006. doi:10.1007/11745853_13.

[Duq07] Sylvain Duquesne. Improving the arithmetic of elliptic curves in the Jacobi model. *Information Processing Letters*, 104(3):101–105, 2007. doi:10.1016/j.ipl.2007.05.012.

[dVGH$^+$23] Henry de Valence, Jack Grigg, Mike Hamburg, Isis Lovecruft, George Tankersley, and Filippo Valsorda. The ristretto255 and decaf448 Groups. IETF RFC 9496 (Informational), 2023.

[ECD23] Digital Signature Standard (DSS). National Institute of Standards and Technology, NIST FIPS PUB 186-5, U.S. Department of Commerce, February 2023. doi:10.6028/NIST.FIPS.186-5.

[GHS02] Pierrick Gaudry, Florian Hess, and Nigel P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *Journal of Cryptology*, 15(1):19–46, January 2002. doi:10.1007/s00145-001-0011-x.

[GLS09] Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 518–535. Springer, Heidelberg, April 2009. doi:10.1007/978-3-642-01001-9_30.

[GLV01] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 190–200. Springer, Heidelberg, August 2001. doi:10.1007/3-540-44647-8_11.

[Ham15] Mike Hamburg. Decaf: Eliminating cofactors through point compression. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 705–723. Springer, Heidelberg, August 2015. doi:10.1007/978-3-662-47989-6_34.

[HKM09] Darrel Hankerson, Koray Karabina, and Alfred Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. *IEEE Transactions on Computers*, 58(10):1411–1420, 2009. doi:10.1109/TC.2009.61.

[HWCD08] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 326–343. Springer, Heidelberg, December 2008. doi:10.1007/978-3-540-89255-7_20.

[HWCD09] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Jacobi quartic curves revisited. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP 09*, volume 5594 of *LNCS*, pages 452–468. Springer, Heidelberg, July 2009. doi:10.1007/978-3-642-02620-1_31.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Inf. Comput.*, 78(3):171–177, 1988. doi:10.1016/0890-5401(88)90024-7.

[Jac29] Carl Gustav Jacob Jacobi. Fundamenta nova theoriae functionum ellipticarum. Sumtibus Fratrum, 1829.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley, first edition, 1969.

[Knu99] Erik Woodward Knudsen. Elliptic scalar multiplication using point halving. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *ASIACRYPT'99*, volume 1716 of *LNCS*, pages 135–149. Springer, Heidelberg, November 1999. doi:10.1007/978-3-540-48000-6_12.

[NIS23] Recommendations for discrete-logarithm based cryptography: Elliptic curve domain parameters. National Institute of Standards and Technology, NIST Special Publication (SP) 800-186, U.S. Department of Commerce, February 2023. doi:10.6028/NIST.SP.800-186.

[NSW09] Gregory Neven, Nigel P. Smart, and Bogdan Warinschi. Hash function requirements for Schnorr signatures. *J. Math. Cryptol.*, 3(1):69–87, 2009. doi:10.1515/JMC.2009.004.

[OLAR13] Thomaz Oliveira, Julio Cesar López-Hernández, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 311–330. Springer, Heidelberg, August 2013. doi:10.1007/978-3-642-40349-1_18.

[OLAR14] Thomaz Oliveira, Julio Cesar López-Hernández, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *Journal of Cryptographic Engineering*, 4(1):3–17, April 2014. doi:10.1007/s13389-013-0069-z.

[Por20a] Thomas Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Report 2020/972, 2020. https://eprint.iacr.org/2020/972.

[Por20b] Thomas Pornin. Optimized lattice basis reduction in dimension 2, and fast schnorr and EdDSA signature verification. Cryptology ePrint Archive, Report 2020/454, 2020. https://eprint.iacr.org/2020/454.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 403–428. Springer, Heidelberg, May 2016. doi:10.1007/978-3-662-49890-3_16.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990. doi:10.1007/0-387-34805-0_22.

[V́71] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences, Série A*, 273(4):238–241, 1971.

[Was08] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography.* Chapman & Hall, second edition, 2008.

[WW27] Edmund T. Whittaker and George N. Watson. *A Course of Modern Analysis.* Cambridge University Press, fourth edition, 1927.

[x2523] libx25519, version 2023.06.30, 2023. URL: https://lib25519.cr.yp.to/.

A Conventions for Formulas

In subsequent sections, detailed formulas are presented for operations on elliptic curves. The formulas use pseudocode in Python/Sage syntax; the code can actually run, provided that an appropriate finite field implementation is included under the name `Fq`:

- The function `Fq`, when applied to an integer, must return an object representing that integer reduced modulo the field characteristic. Field elements must implement the usual arithmetic operators (addition `+`, subtraction `-`, multiplication `*`, division `/`, exponentiation `**`, and comparison operators `==` and `!=`). Field elements must also provide the methods `is_zero()` (which returns `True` if and only if the element is zero), `is_square()` (which returns `True` if and only if the element is a square in the field), and `sqrt()` (which returns a square root for the element, assuming one exists).
- Division by 2 (halving) is assumed to be fast, and denoted with the division operator. Practical implementations will use a dedicated function (for prime fields, this is typically done with a right-shift, followed by adding $(q+1)/2$ if the dropped bit had value 1).
- The curve parameters are defined by the field elements `a` and `b`, corresponding to constants a and b, respectively. Some functions use extra constants derived from a and b, e.g. $b' = a^2 - 4b$, under the name `bp`. Each function header comment recalls the needed constants (if any).
- The `sign()` function must return the conventional "sign" (as defined in section 4.2) for a field element, as an integer of value 0 or 1.

For a prime field of modulus q, the Sage `Zmod` primitive is appropriate. For instance, for the curve jq255s, the following definitions provide the field and constants:

```
Fq = Zmod(2**255 - 3957)
a = Fq(-1)
b = Fq(1)/2
bp = a**2 - 4*b
alpha = (4*b - a**2)/(2*b - a)
beta = (a - 2)/(2*b - a)
gamma = (4*b - a**2).sqrt()/2
def sign(x):
    return int(x) & 1
```

(Constants b', α, β and γ are all used by some of the functions listed in the following sections; for this curve, they all exist and the relevant functions can be used.)

Curve points are represented by tuples of field elements, for parameters to function calls, and values returned by functions. The pseudocode aims at clarity of exposition and does not try to achieve constant-time operations; however, conditional jumps are used only in compression-related functions, and can be implemented with constant-time conditional setters at negligible computational overhead.

Appendices B, C and D use these conventions. They do not necessarily require the curve to be double-odd, unless explicitly specified. Appendix E covers the case of binary curves and uses its own conventions.

B Formulas for Jacobian (x,w) Coordinates

We use the conventions expressed in appendix A; this section is for a field of characteristic 3 or more.

In (x,w) coordinates, the y coordinate of a point (x,y) is replaced with $w = y/x$. We use a three-value representation $(X{:}W{:}J)$ with $J \neq 0$, $x = X/J^2$ and $w = W/J$ (it is called "Jacobian" because the J factor is equivalent to curve isomorphisms, as in classic Jacobian coordinates). The point-at-infinity $\mathbb{O}$ and the point of order two N do not have a well-defined w coordinate, but they can be represented as triplets $(W^2{:}W{:}0)$ and $(0{:}W{:}0)$ for any $W \neq 0$, respectively. We present here four doubling formulas, presented as pseudocode (Python/Sage syntax); one computes $2P$ for an input point P, while the three others compute $2P+N$ for the input P.

The formulas are built on the ψ_1 and $\psi'_{1/2}$ isogenies presented in section 3.5. We can also define the functions θ_1 and $\theta'_{1/2}$ such that $\theta_1(P) = \psi_1(P) + N$ and $\theta'_{1/2}(P) = \psi'_{1/2}(P) + N$, i.e. the isogeny followed by addition of N in their respective output curves. Table 1 shows expressions of all four functions on input $(X{:}W{:}J)$.

Table 1: Isogenies and associate functions in Jacobian (x,w) coordinates, on input $(X{:}W{:}J)$.

out	ψ_1	θ_1	$\psi'_{1/2}$	$\theta'_{1/2}$
X'	W^4	$(a^2-4b)J^4$	W^4	$16bJ^4$
W'	$W^2-2X-aJ^2$	$2X+aJ^2-W^2$	$W^2-2X+2aJ^2$	$2X-2aJ^2-W^2$
J'	WJ	WJ	$2WJ$	$2WJ$

The expressions given in table 1 work for all points P and $P+N$ for any $P \in \mathcal{E}[r]$; they can fail in the presence of a point of order two which is not N. If there is only one point of order two in the curve (i.e. $a^2-4b \notin QR$), then the formulas are complete; this includes the case of double-odd curves. The derived formulas will be used only on points on which exceptional cases are not encountered. For any such point P:

$$\begin{array}{rcccl} \psi'_{1/2}(\psi_1(P)) & = & \psi'_{1/2}(\theta_1(P)) & = & 2P \\ \theta'_{1/2}(\psi_1(P)) & = & \theta'_{1/2}(\theta_1(P)) & = & 2P+N \end{array}$$

Combining the expressions and seeking some computational shortcuts yields the functions presented thereafter. The function `double_addN_xwj_generic()` computes $2P+N$ for an input point P, on all curves (figure 1).

```
# Input: P in XWJ coordinates
# Output: 2*P+N in XWJ coordinates
# Constants: bp = a**2 - 4*b
# Cost: 2M + 5S
def double_addN_xwj_generic(P):
    (X, W, J) = P
    C = J**2
    D = C**2                    # D = J**4
    E = W**2
    F = E**2                    # F = W**4
    G = (W + J)**2 - C - E      # G = 2*W*J
    X2 = 16*b*D*F               # X2 = 16*b*(W**4)*(J**4)
    W2 = bp*D - F               # W2 = -(W**4 + (4*b - a**2)*(J**4))
    J2 = G*(2*X + a*C  - E)     # J2 = 2*W*J*(2*X + a*(J**2) - W**2)
    return (X2, W2, J2)
```

Figure 1: Point doubling in Jacobian (x, w) coordinates (all curves).

If $4b - a^2$ is a square, then we can define the constant $\gamma = (\sqrt{4b - a^2})/2$ and use it in a slightly faster formula, with cost 1M + 6S (instead of 2M + 5S). This is exposed in function `double_addN_xwj_gamma()` (figure 2). In the expected case of a double-odd curve, for which $a^2 - 4b \notin QR$, γ exists if and only if $q = 3 \bmod 4$.

```
# Input: P in XWJ coordinates
# Output: 2*P+N in XWJ coordinates
# Constants: gamma = sqrt(4*b - a**2) / 2
# Constraints: 4*b - a**2 is a square
# Cost: 1M + 6S
def double_addN_xwj_gamma(P):
    (X, W, J) = P
    C = J**2
    E = W**2
    G = (W + J)**2 - C - E          # G = 2*W*J
    H = G**2
    I = (E + 2*gamma*C)**2
    X2 = b*(H**2)                   # X2 = 16*b*(W**4)*(J**4)
    W2 = gamma*H - I                # W2 = -(W**4 + (4*b -
       a**2)*(J**4))
    J2 = G*(2*X + a*C  - E)         # J2 = 2*W*J*(2*X + a*(J**2) -
       W**2)
    return (X2, W2, J2)
```

Figure 2: Point doubling in Jacobian (x, w) coordinates ($4b - a^2 \in QR$).

When $a = -1$ and $b = 1/2$, some computational shortcuts can apply, resulting in a doubling process with cost 2M + 4S. This is illustrated in function `double_addN_xwj_s()` (figure 3). This case applies to the double-odd curve jq255s (defined in section 4.5).

```
# Input: P in XWJ coordinates
# Output: 2*P+N in XWJ coordinates
# Constraints: (a, b) = (-1, 1/2)
# Cost: 2M + 4S
def double_addN_xwj_s(P):
    (X, W, J) = P
    C = W*J
    D = C**2
    E = (W + J)**2 - 2*C        # E = W**2 + J**2
    X2 = 8*(D**2)               # X2 = 8*(W**4)*(J**4)
    W2 = 2*D - E**2             # W2 = -(W**4 + J**4)
    J2 = 2*C*(2*X - E)          # J2 = 2*W*J*(2*X - J**2 - W**2)
    return (X2, W2, J2)
```

Figure 3: Point doubling in Jacobian (x, w) coordinates ($a = -1$ and $b = 1/2$).

When $a = 0$, the curve has equation $y^2 = x^3 + bx$ and doubling formulas with cost 1M+5S apply; this is function `double_xwj_e()` (figure 4). Take care that for this function, the point $2P$ is returned, *not* $2P + N$.

```
# Input: P in XWJ coordinates
# Output: 2*P in XWJ coordinates
# Constraints: a = 0
# Cost: 1M + 5S
def double_xwj_e(P):
    (X, W, J) = P
    C = W**2
    X1 = C**2
    W1 = C - 2*X                  # psi_1(P) = (X1,W1,J1) with J1 =
        W*J
    D = W1**2
    X2 = D**2
    W2 = D - 2*X1
    J2 = J*((W + W1)**2 - C - D) # J2 = 2*W*W1*J = 2*W1*J1
    return (X2, W2, J2)
```

Figure 4: Point doubling in Jacobian (x, w) coordinates ($a = 0$).

C Formulas for Fractional (x,u) Coordinates

In fractional (x, u) coordinates, an r-torsion point P is represented by the point $P + N$, whose coordinates (x, u) are themselves represented by the quadruplet $(X{:}Z{:}U{:}T)$ for $Z \neq 0$, $T \neq 0$, $x = X/Z$ and $u = U/T$. The neutral element uses $X = U = 0$. Points are serialized by compressing them into $w = 1/u = T/U$ (if $U = 0$ then w is conventionally set to 0). The decompression process rebuilds a matching $(X{:}Z{:}U{:}T)$ representation, taking care to verify that the obtained point is indeed a correct $P + N$ representant of an r-torsion point P. Functions `compress_xu()` and `decompress_doubleodd_xu()` (figure 5) implement this process in the specific case of a double-odd curve, where point validation is easy.

```
# Input: P+N in XZUT coordinates
# Output: encoding of P+N as w
def compress_xu(P):
    (X, Z, U, T) = P
    if U.is_zero():
        return Fq(0)
    else:
        return T/U

# Input: w
# Output: the decoded P+N in XZUT coordinates, or False if invalid
def decompress_doubleodd_xu(w):
    if w.is_zero():
        return (Fq(0), Fq(1), Fq(0), Fq(1))
    C = w**2 - a
    D = C**2 - 4*b
    if not D.is_square():
        return False
    E = D.sqrt()
    x = (C + E)/2
    if x.is_square():
        x = x - E
    return (x, Fq(1), Fq(1), w)
```

Figure 5: Point compression and decompression in fractional (x, u) coordinates for a double-odd curve.

Since many quadruplets can match the same point, function `equals_xu()` (figure 6) should be used to check whether two quadruplets represent the same point.

```
# Input: P1+N and P2+N in XZUT coordinates
# Output: P1 == P2
def equals_xu(P1, P2):
    (X1, Z1, U1, T1) = P1
    (X2, Z2, U2, T2) = P2
    return U1*T2 == U2*T1
```

Figure 6: Point comparison in fractional (x, u) coordinates.

For general point addition, two extra constants $\alpha = (4b - a^2)/(2b - a)$ and $\beta = (a - 2)/(2b - a)$ are used. If the curve is such that $a = 2b$, then α and β are not defined; a workaround is to move to the isomorphic curve $y^2 = x^3 + (a\epsilon^2)x^2 + (b\epsilon^4)x$ by multiplying x and u by ϵ^2 and $1/\epsilon$, respectively, for any $\epsilon \neq 0$ (e.g. $\epsilon = 2$). The function `add_xu()` (figure 7) computes $(P_1 + P_2) + N$, given $P_1 + N$ and $P_2 + N$, assuming that both inputs are correct representants of r-torsion points.

```
# Input: P1+N and P2+N in XZUT coordinates
# Output: (P1+P2)+N in XZUT coordinates
# Constants: alpha = (4*b - a**2)/(2*b - a), beta = (a - 2)/(2*b -
    a)
# Cost: 10M
def add_xu(P1, P2):
    (X1, Z1, U1, T1) = P1
    (X2, Z2, U2, T2) = P2
    X1X2 = X1*X2
    Z1Z2 = Z1*Z2
    U1U2 = U1*U2
    T1T2 = T1*T2
    C = (X1 + Z1)*(X2 + Z2) - X1X2 - Z1Z2    # C = X1*Z2 + X2*Z1
    D = (U1 + T1)*(U2 + T2) - U1U2 - T1T2    # D = U1*T2 + U2*T1
    E = X1X2 + b*Z1Z2
    F = E*T1T2
    G = U1U2*(2*b*C + a*E)
    H = (T1T2 + alpha*U1U2)*(C + E)
    X3 = b*(H - F + beta*G)
    Z3 = F - G
    U3 = -D*(X1X2 - b*Z1Z2)
    T3 = F + G
    return (X3, Z3, U3, T3)
```

Figure 7: Point addition in fractional (x, u) coordinates.

Negation is obtained by negating the u coordinate (figure 8). Subtraction is obtained by combining addition with negation of the second operand.

```
# Input: P+N in XZUT coordinates
# Output: -P+N in XZUT coordinates
def neg_xu(P):
    (X, Z, U, T) = P
    return (X, Z, -U, T)
```

Figure 8: Point negation in fractional (x, u) coordinates.

Point doubling takes $P + N = (x, u)$ as parameter, and returns $2P + N$. Function `double_xu()` (figure 9) implements this operation, using the extra constant $b' = a^2 - 4b$ (under the name `bp`).

```
# Input: P+N in XZUT coordinates
# Output: 2*P+N in XZUT coordinates
# Constants: bp = a**2 - 4*b
# Cost: 3M + 6S
def double_xu(P):
    (X, Z, U, T) = P
    C = X**2
    D = Z**2
    E = (X + Z)**2 - C - D        # E = 2*X*Z
    X1 = bp*E
    Z1 = 2*(C + b*D) + a*E
    F = X1**2
    G = Z1**2
    H = (X1 + Z1)**2 - F - G      # H = 2*X1*Z1
    X2 = 2*b*H
    Z2 = F + bp*G - a*H
    U2 = 4*bp*(C - b*D)*Z1*U
    T2 = (F - bp*G)*T
    return (X2, Z2, U2, T2)
```

Figure 9: Point doubling in fractional (x, u) coordinates.

A sequence of n point doublings can always be computed by calling function `double_xu` repeatedly, but a faster method, especially for long sequences, is to arrange for the first doubling to produce an output in Jacobian (x, w) coordinates, then performing $(n - 2)$ doublings in that coordinate system (using the functions described in appendix B, which are faster), and making the last doubling with formulas that output the final result in fractional (x, u) coordinates. Function `xdouble_xu` (figure 10) implements this process. It internally uses a doubling function in Jacobian (x, w) coordinates, which can be any of the four functions from appendix B that applies to the used curve (in particular, both functions that return $2P$ and functions that return $2P + N$ can be used, since this is for the "internal" doublings).

```
# Input: P+N in XZUT coordinates, n >= 1
# Output: (2**n)*P+N in XZUT coordinates
# Constants: bp = a**2 - 4*b
# Cost: n*(2M + 5S) + 1M + 2S     all curves
#       n*(1M + 6S) + 3M          4*b - a**2 is a square
#       n*(2M + 4S) + 1M + 4S     a = -1, b = 1/2
#       n*(1M + 5S) + 3M + 2S     a = 0
def xdouble_xu(P, n):
    # For n == 1, use the single-doubling function.
    if n == 1:
        return double_xu(P)
    (X, Z, U, T) = P

    # Compute 2*P+N in XWJ coordinates (cost: 4M+6S)
    C = Z*T                        # Start of psi_1
    D = C*T
    X1 = D**2                      # X1 = (Z**2)*(T**4)
    J1 = C*U                       # J1 = Z*U*T
    E = U**2
    W1 = D - (2*X + a*Z)*E         # W1 = Z*(T**2) - (2*X + a*Z)*(U**2)
    F = W1**2                      # Start of theta_{1/2}
    G = J1**2
    X = 16*b*(G**2)
    W = 2*X1 - 2*a*G - F
    J = (W1 + J1)**2 - F - G       # (X:W:J) = 2*P+N

    # Perform n-2 times P <- 2*P+N in XWJ coordinates.
    # We can use any of the double_addN_xwj_*() or double_xwj_*()
    # functions that apply to the current curve.
    # cost: (n-2)*cost(double_xwj)
    for i in range(0, n-2):
        (X, W, J) = double_addN_xwj_generic((X, W, J))

    # Final doubling, with output in XZUT coordinates (cost: 1M +
       6S)
    C = W**2
    D = J**2
    E = (W + J)**2 - C - D
    F = C - 2*X - a*D
    G = E**2
    X2 = b*G
    Z2 = F**2
    U2 = (E + F)**2 - G - Z2
    T2 = 4*C*(C - a*D) - 2*Z2
    return (X2, Z2, U2, T2)
```

Figure 10: Sequence of point doublings in fractional (x, u) coordinates.

For some curves, the per-sequence overhead can be reduced with some curve-specific optimizations. If $a = -1$ and $b = 1/2$ (as in the jq255s curve), function `xdouble_xu_s()` (figure 11) saves one squaring (the saving is in the final doubling). If $a = 0$ (e.g. for the jq255e curve), then two squarings can be avoided with a slightly different process: only the ψ_1 isogeny is applied (i.e. "half" of the first doubling), then $n-1$ doublings in the dual curve $\mathcal{E}(-2a, a^2-4b)$ (for which $-2a=0$), and finally $\theta'_{1/2}$ is applied with conversion back to fractional (x,u) coordinates. This is shown in function `xdouble_xu_e()` (figure 12).

```
# Input: P+N in XZUT coordinates, n >= 1
# Output: (2**n)*P+N in XZUT coordinates
# Constraints: a = -1, b = 1/2
# Cost: n*(2M + 4S) + 1M + 3S
def xdouble_xu_s(P, n):
    # For n == 1, use the single-doubling function.
    if n == 1:
        return double_xu(P)
    (X, Z, U, T) = P

    # Compute 2*P+N in XWJ coordinates (cost: 4M+6S)
    C = Z*T                       # Start of psi_1
    D = C*T
    X1 = D**2                     # X1 = (Z**2)*(T**4)
    J1 = C*U                      # J1 = Z*U*T
    E = U**2
    W1 = D - (2*X - Z)*E          # W1 = Z*(T**2) - (2*X + a*Z)*(U**2)
    F = W1**2                     # Start of theta_{1/2}
    G = J1**2
    X = 8*(G**2)
    W = 2*X1 + 2*G - F
    J = (W1 + J1)**2 - F - G      # (X:W:J) = 2*P+N

    # Perform n-2 times P <- 2*P+N in XWJ coordinates.
    # cost: (n-2)*(2M + 4S)
    for i in range(0, n-2):
        (X, W, J) = double_addN_xwj_s((X, W, J))

    # Final doubling, with output in XZUT coordinates (cost: 1M +
       5S)
    C = W*J
    D = C**2
    E = (W + J)**2 - 2*C          # E = W**2 - a*J**2
    F = 2*X - E
    X2 = 2*D                      # X2 = 4*b*(W**2)*(J**2)
    Z2 = F**2                     # Z2 = (2*X + a*J**2 - W**2)**2
    U2 = (C + F)**2 - Z2 - D      # U2 = 2*W*J*(2*X + a*J**2 - W**2)
    T2 = 2*D - E**2               # T2 = -(W**4 - (a**2 - 4*b)*J**4)
    return (X2, Z2, U2, T2)
```

Figure 11: Sequence of point doublings in fractional (x,u) coordinates ($a=-1$ and $b=1/2$).

```
# Input: P+N in XZUT coordinates, n >= 1
# Output: (2**n)*P+N in XZUT coordinates
# Constraints: a = 0
# Cost: n*(1M + 5S) + 3M
def xdouble_xu_e(P, n):
    # For n == 1, use the single-doubling function.
    if n == 1:
        return double_xu(P)
    (X, Z, U, T) = P

    # psi_1(P), into XWJ coordinates (cost: 4M + 2S)
    C = Z*T
    D = C*T
    E = U**2
    W = D - 2*X*E
    J = C*U
    X = D**2

    # Perform n-1 doublings over EC(-2*a,a**2-4*b) (dual curve)
    # (cost: (n-1)*(1M + 5S))
    for i in range(0, n - 1):
        C = W**2
        E = C - 2*X
        X = C**2
        D = E**2
        J = J*((W + E)**2 - C - D)
        W = D - 2*X
        X = D**2

    # theta'_{1/2}, output in XZUT coordinates (cost: 3S)
    C = J**2
    D = W**2
    X2 = 4*b*C
    Z2 = D
    U2 = (W + J)**2 - C - D
    T2 = 2*X - D
    return (X2, Z2, U2, T2)
```

Figure 12: Sequence of point doublings in fractional (x, u) coordinates $(a = 0)$.

D Formulas for Extended (e,u) Coordinates

Extended (e,u) coordinates use the Jacobi quartic form. As explained in section 4, this really applies only to double-odd curves, for which the efficient compression and decompression process yields a proper prime-order group abstraction. We therefore only consider double-odd curves in this section.

In (e,u) coordinates, an r-torsion point P can be represented by either P or $P+N$, and both representants work equally well. The coordinates are represented by a quadruplet $(E{:}Z{:}U{:}T)$ such that $Z \neq 0$, $e = E/Z$, $u = U/Z$ and $U^2 = TZ$. For all points, $E \neq 0$. The neutral uses $U = T = 0$ and $E = \pm Z$. Compression and decompression use functions `compress_doubleodd_eu()` and `decompress_doubleodd_eu()`, respectively (figure 13), using the method exposed in section 4.2.

```
# Input: P in EZUT coordinates
# Output: encoding of P as u
def compress_doubleodd_eu(P):
    (E, Z, U, T) = P
    C = 1/Z
    u = C*U
    if sign(C*E) == 1:
        u = -u
    return u

# Input: u
# Output: the decoded P in EZUT coordinates, or False if invalid
# Constants: bp = a**2 - 4*b
def decompress_doubleodd_eu(u):
    T = u**2
    D = bp*(T**2) - 2*a*T + 1
    if not D.is_square():
        return False
    E = D.sqrt()
    if sign(E) == 1:
        E = -E
    return (E, Fq(1), u, T)
```

Figure 13: Point compression and decompression in extended (e,u) coordinates for a double-odd curve.

The function `equals_eu()` (figure 14) checks whether two given quadruplets represent the same group element.

```
# Input: P1 and P2 in EZUT coordinates
# Output: P1 == P2
def equals_eu(P1, P2):
    (E1, Z1, U1, T1) = P1
    (E2, Z2, U2, T2) = P2
    return U1*E2 == U2*E1
```

Figure 14: Point comparison in extended (e,u) coordinates for a double-odd curve.

Addition of two group elements uses the function `add_eu()` (figure 15). The constant `bp` is $b' = a^2 - 4b$.

```
# Input: P1 and P2 in EZUT coordinates
# Output: P1+P2 in EZUT coordinates
# Constants: bp = a**2 - 4*b
# Cost: 8M + 3S
def add_eu(P1, P2):
    (E1, Z1, U1, T1) = P1
    (E2, Z2, U2, T2) = P2
    E1E2 = E1*E2
    Z1Z2 = Z1*Z2
    U1U2 = U1*U2
    T1T2 = T1*T2
    tz = (Z1 + T1)*(Z2 + T2) - Z1Z2 - T1T2    # tz = Z1*T2 + Z2*T1
    eu = (E1 + U1)*(E2 + U2) - E1E2 - U1U2    # eu = E1*U2 + E2*U1
    hd = Z1Z2 - bp*T1T2
    E3 = (Z1Z2 + bp*T1T2)*(E1E2 - 2*a*U1U2) + 2*bp*U1U2*tz
    Z3 = hd**2
    T3 = eu**2
    U3 = ((hd + eu)**2 - Z3 - T3)/2           # U3 = hd*eu
    return (E3, Z3, U3, T3)
```

Figure 15: Point addition in extended (e, u) coordinates for a double-odd curve.

Negation is done by function `neg_eu()` (figure 16); a subtraction can then be computed as the combination of a negation and an addition.

```
# Input: P in EZUT coordinates
# Output: -P in EZUT coordinates
def neg_eu(P):
    (E, Z, U, T) = P
    return (E, Z, -U, T)
```

Figure 16: Point negation in extended (e, u) coordinates for a double-odd curve.

Function `xdouble_eu_generic()` (figure 17) performs a sequence of n successive point doublings, and works on all double-odd curves; by setting $n = 1$, a single doubling is obtained. This process is simpler than in (x, u) coordinates because each point has two representants and we can use either, including for the output, which avoids the need for a special process for the last doubling. This is also why a dedicated function for single doublings is not needed.

```
# Input: P in EZUT coordinates, n >= 1
# Output: (2**n)*P in EZUT coordinates
# Constants: bp = a**2 - 4*b
# Cost: n*(2M + 5S)
def xdouble_eu_generic(P, n):
    (E, Z, U, T) = P

    # First doubling, output in XWJ coordinates (cost: 2M+2S)
    C = U**2
    X = 16*b*(C**2)
    W = (bp*T + Z)*(T - Z) + (bp - 1)*C
    J = 2*E*U

    # (n-1) doublings in XWJ coordinates. This can use any of
    # the double_addN_xwj_*() or double_xwj_*() functions that
    # apply to the current curve.
    for i in range(0, n - 1):
        (X, W, J) = double_addN_xwj_generic((X, W, J))

    # Final conversion to EZUT coordinates (cost: 3S)
    Z = W**2
    T = J**2
    U = ((W + J)**2 - Z - T)/2
    E = 2*X - Z + a*T
    return (E, Z, U, T)
```

Figure 17: Sequence of point doublings in extended (e, u) coordinates for a double-odd curve.

Function `xdouble_eu_generic()` performs the first doubling with a special sequence that also converts its output to Jacobian (x, w) coordinates, and the remaining $n - 1$ doublings use the functions specified in appendix B. For many curves, doublings in Jacobian (x, w) coordinates can be performed more efficiently than the generic 2M+5S procedure, and similar optimizations are applicable to the first doubling. Function `xdouble_eu_gamma()` (figure 18) handles curves such that $-b' = 4b - a^2 \in QR$ (for a double-odd curve, $b' \notin QR$, hence this condition is equivalent to working in field $\mathbb{F}_q$ with $q = 3 \bmod 4$); function `xdouble_eu_s()` (figure 19) handles the specific case of $(a, b) = (-1, 1/2)$, which applies to curve jq255s; function `xdouble_eu_e()` (figure 20) is for curves with $a = 0$, e.g. jq255e.

```
# Input: P in EZUT coordinates, n >= 1
# Output: (2**n)*P in EZUT coordinates
# Constants: gamma = sqrt(4*b - a**2) / 2
# Constraints: 4*b - a**2 is a square
# Cost: n*(1M + 6S)
def xdouble_eu_gamma(P, n):
    (E, Z, U, T) = P

    # First doubling, output in XWJ coordinates (cost: 1M+3S)
    C = U**2
    X = 16*b*(C**2)
    W = 4*gamma*C - (2*gamma*T + Z)**2
    J = 2*E*U

    # (n-1) doublings in XWJ coordinates.
    for i in range(0, n - 1):
        (X, W, J) = double_addN_xwj_gamma((X, W, J))

    # Final conversion to EZUT coordinates (cost: 3S)
    Z = W**2
    T = J**2
    U = ((W + J)**2 - Z - T)/2
    E = 2*X - Z + a*T
    return (E, Z, U, T)
```

Figure 18: Sequence of point doublings in extended (e, u) coordinates for a double-odd curve ($4b - a^2 \in QR$).

```
# Input: P in EZUT coordinates, n >= 1
# Output: (2**n)*P in EZUT coordinates
# Constraints: a = -1, b = 1/2
# Cost: n*(2M + 4S) - 1M + 2S
def xdouble_eu_s(P, n):
    (E, Z, U, T) = P

    # First doubling, output in XWJ coordinates (cost: 1M+3S)
    C = U**2
    X = 8*(C**2)
    W = 2*C - (T + Z)**2
    J = 2*E*U

    # (n-1) doublings in XWJ coordinates.
    for i in range(0, n - 1):
        (X, W, J) = double_addN_xwj_s((X, W, J))

    # Final conversion to EZUT coordinates (cost: 3S)
    Z = W**2
    T = J**2
    U = ((W + J)**2 - Z - T)/2
    E = 2*X - Z - T
    return (E, Z, U, T)
```

Figure 19: Sequence of point doublings in extended (e, u) coordinates for a double-odd curve ($a = -1$ and $b = 1/2$).

```
# Input: P in EZUT coordinates, n >= 1
# Output: (2**n)*P in EZUT coordinates
# Constraints: a = 0
# Cost: n*(1M + 5S) + 1S
def xdouble_eu_e(P, n):
    (E, Z, U, T) = P

    # First doubling, output in XWJ coordinates (cost: 1M+3S)
    C = E**2
    X = C**2
    W = 2*(Z**2) - C
    J = 2*E*U

    # (n-1) doublings in XWJ coordinates.
    for i in range(0, n - 1):
        (X, W, J) = double_xwj_e((X, W, J))

    # Final conversion to EZUT coordinates (cost: 3S)
    Z = W**2
    T = J**2
    U = ((W + J)**2 - Z - T)/2
    E = 2*X - Z
    return (E, Z, U, T)
```

Figure 20: Sequence of point doublings in extended (e, u) coordinates for a double-odd curve $(a = 0)$.

E Formulas for Extended (x,s) Coordinates

We here consider a binary curve. Finite field is $\mathbb{F}_{2^m}$ for some integer $m \geq 1$. We assume in this section that an implementation of elementary operations in this field is provided. In particular:

- Operators for addition (`+`), multiplication (`*`), division (`/`) and exponentiation (`**`) should be supported. Exponentiation is used here only with exponent 2, to compute squarings; this is expected to be a fast operation (faster or much faster than a multiplication). In a binary field, subtraction is the same thing as addition, and thus not needed.
- Field elements should support the `sqrt()` method to return their square root. In a binary field, every element is a square and has a single square root; hence, this method should never fail and is unambiguous.
- The `trace()` method should return the trace, as an integer (or a field element) of value 0 or 1.
- The `is_zero()` method should return `True` if and only if its operand is zero.
- The `qsolve()` function, applied to an input d of trace zero, should return x such that $x^2 + x = d$. There are two solutions; it does not matter which of the two solutions is returned by `qsolve()`.

The curve has equation $y^2 + xy = x^3 + ax^2 + bx$ and has order $2^t r$ for some $t \geq 1$ and odd integer r. The constants a, b and t are provided to the code under the names `a`, `b` and `t`, respectively. Some additional derived constants are also provided:

`aa`	Field element a^2.
`bb`	Field element b^2.
`sb`	Field element $\sqrt{b}$.
`w0`	A conventional value for compressing the point N; it can be any value w_0 such that either $w_0^2 + w_0 + a = 0$ or $\mathrm{Tr}(b/(w_0^2 + w_0 + a)) = 1$. For about half of all binary curves, the value $w_0 = 0$ is appropriate.

Following the description in section 5, we represent the point of r-torsion P with the curve point $P + N$, with N being the unique point of order two on the curve. The y coordinate of $P + N$ is replaced with $s = y + x^2 + ax + b$. The (x, s) coordinates are represented by a quadruplet $(X{:}S{:}Z{:}T)$ such that $Z \neq 0$, $x = \sqrt{b}X/Z$, $s = \sqrt{b}S/Z^2$ and $T = XZ$. The point N (which represents the neutral point $\mathbb{O}$) uses $(0{:}\sqrt{b}Z^2{:}Z{:}0)$ for any $Z \neq 0$. Many of the formulas presented thereafter admit straightforwardly optimized "mixed" and "affine" variants when $Z = 1$ for some or all of the operands; when $Z = 1$, we have $X = T = x/\sqrt{b}$, and $S = s/\sqrt{b}$, which is similar to affine (x, s) coordinates, save for the extra $1/\sqrt{b}$ factor.

Functions `compress_xs()` and `decompress_xs()` (figure 21) implement compression and decompression, respectively. The compressed format is the single field element $\sqrt{s/x} + w_0$ (this is the slope of the line from N to the point $P + N$, shifted by the constant w_0 which has been chosen as the conventional encoding of N itself). Upon decompression, the correct solution is recomputed and validated: a valid representant $P + N$ for an r-torsion point P is a point that can be halved exactly $t - 1$ times.

```
# Input: P+N in XSZT coordinates
# Output: encoding of P+N as c
# Constants: w0
def compress_xs(P):
    (X, S, Z, T) = P
    if X == 0:
        return w0
    else:
        return (S/T).sqrt() + w0

# Input: c
# Output: the decoded P+N in XSZT coordinates, or False if invalid
# Constants: w0, sb = sqrt(b)
def decompress_xs(c):
    if c.is_zero():
        return N
    w = c + w0
    D = w**2 + w + a
    if D.is_zero():
        return False
    E = b/(D**2)
    if E.trace() == 1:
        return False
    F = qsolve(E)
    x = D*F

    # We have x matching the w coordinate. Curve order is (2**t)*r;
    # we try to halve it t-1 times. If curve is double-odd (t = 1),
    # then the loop disappears.
    xp = x
    yp = x*w
    for i in range(0, t-1):
        if (xp + a).trace() != 0:
            return False
        L = qsolve(xp + a)
        xp = (yp + L*xp + xp + b).sqrt()
        yp = L*xp + xp**2 + b

    # Adjust the point if we got an r-torsion point.
    if (xp + a).trace() == 0:
        x = x + D
    s = x*(w**2)

    # (x,s) is in affine coordinates (unscaled by sqrt(b)). We
        convert
    # to XSZT.
    return (x, s*sb, sb, x*sb)
```

Figure 21: Point compression and decompression in extended (x, s) coordinates.

Function `equals_xs()` (figure 22) compares two points together; it uses the fact that

$\sqrt{s/x}$ is the slope of the line from N to $P+N$ and thus uniquely identifies a correct representant of an r-torsion point. The formula completeness can be verified by noticing that there is a single valid point $P+N$ such that $x=0$ (this is N itself), and also at most one valid point $P+N$ such that $s=0$ (if $P+N$ is such a point, then the other one is $-P$, which is not a valid representant of an r-torsion point).

```
# Input: P1+N and P2+N in XSZT coordinates
# Output: P1 == P2
def equals_xs(P1, P2):
    (X1, S1, Z1, T1) = P1
    (X2, S2, Z2, T2) = P2
    return S1*T2 == S2*T1
```

Figure 22: Point comparison in extended (x,s) coordinates.

Point addition is performed by function `add_xs()` (figure 23). The cost is $8\mathrm{M}+2\mathrm{S}+2\mathrm{m_b}$. If the curve constant a is zero, then one multiplication is saved (the value `T1T2` does not need to be computed) and the cost is lowered to $7\mathrm{M}+2\mathrm{S}+2\mathrm{m_b}$. As was pointed out in section 5.2, if $\mathrm{Tr}(a)=0$, then we can always apply a curve isomorphism that sets a to zero while leaving b unchanged.

```
# Input: P1+N and P2+N in XSZT coordinates
# Output: (P1+P2)+N in XSZT coordinates
# Constants: aa = a**2, sb = sqrt(b)
# Cost: 8M + 2S + 2mb  (7M + 2S + 2mb if a = 0)
def add_xs(P1, P2):
    (X1, S1, Z1, T1) = P1
    (X2, S2, Z2, T2) = P2
    X1X2 = X1*X2
    S1S2 = S1*S2
    Z1Z2 = Z1*Z2
    T1T2 = T1*T2              # not needed if a = 0
    D = (S1 + T1)*(S2 + T2)
    E = aa*T1T2               # this step disappears if a = 0
    F = X1X2**2
    G = Z1Z2**2
    X3 = D + S1S2
    S3 = sb*(G*(S1S2 + E) + F*(D + E))
    Z3 = sb*(F + G)
    T3 = X3*Z3
    return (X3, S3, Z3, T3)
```

Figure 23: Point addition in extended (x,s) coordinates.

Function `neg_xs()` (figure 24) negates a point. A subtraction can be computed by combining negation and addition.

```
# Input: P+N in XSZT coordinates
# Output: -P+N in XSZT coordinates
def neg_xs(P):
    (X, S, Z, T) = P
    return (X, S + T, Z, T)
```

Figure 24: Point addition in extended (x, s) coordinates.

A sequence of n doublings is computed by function `xdouble_xs()` (figure 25); with $n = 1$, a single doubling is performed. This function temporarily converts back the input to the short Weierstraß curve equation $y^2 + xy = x^3 + ax^2 + b^2$ (by adding b to y), with extended coordinates $(X{:}Y{:}Z{:}T)$ such that $x = X/Z$, $y = Y/Z^2$, and $T = XZ$ (note the absence of $\sqrt{b}$ factors). In this coordinate system, efficient formulas with cost $2\mathrm{M}+4\mathrm{S}+2\mathrm{m_b}$ are applied. Since these formulas are applicable to any non-supersingular binary curve in short Weierstraß format, they could be used outside of the (x, s) formalism; they are complete on the whole curve, provided that the point-at-infinity $(\mathbb{O})$ is represented as $(X{:}0{:}0{:}0)$ for some $X \neq 0$. Conversions to the short Weierstraß curve and back add some per-sequence overhead, for a total cost of $n(2\mathrm{M} + 4\mathrm{S} + 2\mathrm{m_b}) + 2\mathrm{S} + 6\mathrm{m_b}$.

```
# Input: P+N in XSZT coordinates, n >= 1
# Output: (2**n)*P+N in XSZT coordinates
# Constants: sb = sqrt(b)
# Cost: n*(2M + 4S + 2mb) + 2S + 6mb
def xdouble_xs(P, n):
    (X, S, Z, T) = P

    # Conversion to extended (x,y) coordinates (cost: 1S + 3mb).
    # (this moves to curve y**2 + x*y = x**3 + a*(x**2) + (b**2))
    X = sb*X
    T = sb*T
    Y = sb*S + X**2 + a*T

    # n doublings in extended (x,y) coordinates
    # (cost: n*(2M + 4S + 2mb)).
    for i in range(0, n):
        D = (X + sb*Z)**2
        Z = T**2
        X = D**2
        E = D + T
        T = X*Z
        Y = (Y*(Y + E) + (a + b)*Z)**2 + (a + 1)*T

    # Conversion back to XSZT (with addition of N) (cost: 1S +
       3mb).
    X2 = sb*Z
    S2 = sb*(Y + (a + 1)*T + X**2)
    Z2 = X
    T2 = sb*T
    return (X2, S2, Z2, T2)
```

Figure 25: Sequence of point doublings in extended (x, s) coordinates.

The function `xdouble_xs_alt()` (figure 26) is an alternative implementation of a sequence of n point doublings. It starts by converting the point $P + N$ into the point P in (x, λ) coordinates (as previously published in [OLAR13]), then computing $2P + N$ back to (x, s) representation. If $n \geq 2$, then the first $n - 1$ doublings can then be performed in (x, λ) coordinates with the formulas with cost $3\text{M} + 4\text{S} + 1\text{m}_\text{b}$ from [OLAR13]. The overall cost is $n(3\text{M} + 4\text{S} + 1\text{m}_\text{b}) + 3\text{m}_\text{b}$: this function performs more generic multiplications than `double_xs()`, but fewer squarings and especially fewer multiplications by b, which may yield some performance gains when applied to a standard curve with a pseudorandom constant b.

```
# Input: P+N in XSZT coordinates, n >= 1
# Output: (2**n)*P+N in XSZT coordinates
# Constants: aa = a**2, sb = sqrt(b), bb = b**2
# Cost: n*(3M + 4S + mb) + 3mb
def xdouble_xs_alt(P, n):
    (X, S, Z, T) = P

    # P+N (x,s) -> P (x,lambda)
    X = sb*Z**2
    Z = T
    L = S + a*T + T

    # n-1 doublings in (x,lambda) coordinates
    for i in range(0, n - 1):
        D = Z**2
        E = (L + X)**2
        F = L*(L + Z)
        U = F + a*D
        X = U**2
        Z = D*U
        L = E*(E + U + D) + (aa + bb)*D**2 + X + a*Z + Z

    # P (x,lambda) -> 2*P+N (x,s)
    D = Z**2
    E = (L + X)**2
    F = L*(L + Z)
    X = sb*D
    Z = F + a*D
    S = sb*(E*(E + Z + D) + (F + sb*X)**2)
    T = X*Z
    return (X, S, Z, T)
```

Figure 26: Sequence of point doublings in extended (x, s) coordinates (alternate implementation).