Double-Odd Jacobi Quartic

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

13 August, 2022

Abstract. Double-odd curves are curves with order equal to 2 modulo 4. A prime order group with complete formulas and a canonical encoding/decoding process could previously be built over a double-odd curve. In this paper, we reformulate such curves as a specific case of the Jacobi quartic. This allows using slightly faster formulas for point operations, as well as defining a more efficient encoding format, so that decoding and encoding have the same cost as classic point compression (decoding is one square root, encoding is one inversion). We define the prime-order groups jq255e and jq255s as the application of that modified encoding to the do255e and do255s groups. We furthermore define an optimized signature mechanism on these groups, that offers shorter signatures (48 bytes instead of the usual 64 bytes, for 128-bit security) and makes signature verification faster (down to less than 83000 cycles on an Intel x86 Coffee Lake core).

1 Introduction

Elliptic curves are a common way of defining finite groups on which the discrete logarithm problem is believed to be hard; on top of such groups, various protocols can be built. For secure protocol designs, a *prime-order group* is often needed. Elliptic curves with a non-prime order, such as the well-known twisted Edwards curve Curve25519, have led to some serious issues when used in some protocols[13]. The order of a curve meant for cryptographic protocols is often written as the product hr, with r being the prime of interest, and h the *cofactor*. Twisted Edwards curves allow for very efficient computations, but their cofactor h is always a multiple of 4. A prime order group abstraction can be built on top of some twisted Edwards curves with cofactor 4 or 8, using the Decaf/Ristretto technique[3,17]. Applied to Curve25519, the resulting group is called ristretto255 and is currently undergoing standardization as a RFC[35].

In [31], we investigated a different solution, focusing on curves with order $2r$ (i.e. cofactor 2), which we called *double-odd curves*. On top of such curves, we could build prime order group abstractions, with appropriate complete formulas for applying the group law to group elements, as well as canonical encoding and decoding rules. Generally speaking, the defined double-odd groups do255e and do255s offer the usual 128-bit security level with performance roughly similar to that of ristretto255 (general point addition is a bit slower at 10M instead of 8M, but sequences of doublings are faster, with a per-doubling cost down to 1M+5S for do255e). In this paper, we improve on double-odd curves, in three main steps:

1. Through a change of variable, we reformulate double-odd curves as a specific case of the extended Jacobi quartic, which gives access to more already known formulas[19]. We

thus obtain complete formulas on point addition with about the same cost as previously (8M+3S instead of 10M), but a lower overhead for doublings, so that a single doubling costs only 1M+6S, and subsequent doublings in a sequence cost only 1M+5S each. The overall performance is thus improved.

2. The new formulas are complete on the whole curve, not just on the subset we used in the previous double-odd curves. This allows a change of the encoding format, so that decoding a group element from its compressed format now uses only a square root operation, with no additional Legendre symbol computation. We thus define new groups called jq255e and jq255s, which are the same groups as do255e and do255s but with a different encoding format. Decoding and encoding operations are now as fast as in Curve25519; encoding is faster than in ristretto255.
3. Since the encoding rules changed, we take the opportunity to redefine the ECDH and Schnorr signature schemes on these groups (e.g. to replace SHAKE with the definitely faster BLAKE2s). Using an idea which has already been proposed several times (including by Schnorr himself), we define a signature format with a reduced size: we can encode signatures in only 48 bytes, quite smaller than the 64 bytes of Ed25519, while offering the same 128-bit security level. The reduction in size can be an important advantage, especially in embedded systems with severe constraints on communication channels. As a nice side effect, the size reduction also makes signature verification substantially faster; our implementation (in Rust) achieves signature verification in less than 83000 cycles on an Intel x86 Coffee Lake core (less than 93000 cycles if we include the cost of decoding the public key).

In total, the two newly defined groups jq255e and jq255s achieve performance on par with, or better than, Ed25519 and ristretto255, with shorter signatures and faster verification, while providing the clean prime order group abstraction with a compact, canonical and verified encoding format, that is convenient for building secure cryptographic protocols.

In this paper, we first recall some background on double-odd curves (section 2), then show the conversion to a Jacobi quartic (section 3). The new jq255e and jq255s groups are formally defined in section 5. Complete formulas working over extended coordinates are detailed in section 6. Section 7 discusses the new signature format, and includes explicit performance benchmarks. In appendix A, a specification of the ECDH and signature algorithms is included.

Our implementation is available as part of the `crrl` library:

`https://github.com/pornin/crrl`

This paper, as well as the original whitepaper[31], some graphical overview of double-odd curves, and links to various other implementations (notably in C and in Python), are available on the double-odd curves Web site:

`https://doubleodd.group/`

2 Background on Double-Odd Curves

Double-odd curves were formally defined and described in [31]; we recall here the principal properties of such curves.

Curve Equation and Order. We work in a finite field $\mathbb{F}_q$ of order q, with a characteristic $p \geq 5$. For a field element $x \in \mathbb{F}_q$, we write that $x \in QR$ if x is a square in the field, and $x \notin QR$ otherwise. For any two constants $(a, b) \in \mathbb{F}_q \times \mathbb{F}_q$ such that $a \notin QR$ and $a^2 - 4b \notin QR$, the curve $\mathcal{C}(a, b)$ is the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ such that $y^2 = x(x^2 + ax + b)$; a formal "point-at-infinity", denoted $\mathbb{O}$ and with no defined coordinates x or y, is adjoined to the curve. This curve is double-odd, i.e. has order $2r$ for some odd integer r. It can be shown that the two conditions on a and b imply that the curve is well-defined and non-singular, and that any double-odd curve on $\mathbb{F}_q$ can be converted to such an equation with an adequate change of variable.

We denote $\mathcal{C}(a, b)[r]$ the subgroup of r-torsion points, i.e. the points P on the curve such that $rP = \mathbb{O}$. A double-odd curve contains a single point of order 2, which we denote N; its coordinates are $(0, 0)$. Every point $P \in \mathcal{C}(a, b)$ can be decomposed uniquely into $P = Q + R$ with Q an r-torsion point, and $R \in \{N, \mathbb{O}\}$.

Coordinates. For a point $P = (x, y)$, we also define the coordinates $w = y/x$ and $u = 1/w = x/y$. For all points other than N and $\mathbb{O}$, the x, y, w and u coordinates are well-defined and non-zero; we formally set $w = 0$ and $u = 0$ for both N and $\mathbb{O}$. For any point $P = (x, y, w, u)$ (with $P \neq N, \mathbb{O}$), the following properties hold:

- Point $-P$ has coordinates $(x, -y, -w, -u)$.
- Point $P + N$ has coordinates $(b/x, -by/x^2, -w, -u)$.
- Exactly one of P and $P + N$ is an r-torsion point.
- Coordinate $x \in QR$ if and only if $P \in \mathcal{C}(a, b)[r]$.
- Point $2P \in \mathcal{C}(a, b)[r]$, and point $2P + N \notin \mathcal{C}(a, b)[r]$.
- The point $-P+N$ is the only other curve point with the same w coordinate (equivalently, the same u coordinate) as P.

Prime-Order Group. From $\mathcal{C}(a, b)$, we can define the group $\mathcal{G}(a, b)$ as the quotient group of $\mathcal{C}(a, b)$ by $\{N, \mathbb{O}\}$. In other words, each element of $\mathcal{G}(a, b)$ is a pair of points $\{P, P+N\}$, where P is an r-torsion point and $P + N$ is not; the points P and $P + N$ are called the *representants* of the group element. $\mathcal{G}(a, b)$ has order r; if we choose the curve such that r is prime, then we obtain a prime-order group which is convenient for building cryptographic protocols. The original double-odd curves whitepaper[31] leveraged the properties described above in order to efficiently implement $\mathcal{G}(a, b)$:

- Each group element is systematically represented by its non-r-torsion point $P+N$. Thus, the group neutral is represented by N, with defined coordinates, thereby removing the troublesome "point-at-infinity".
- Addition of $P + N$ and $Q + N$ can be done by using P and $Q + N$, one being an r-torsion point and the other a non-r-torsion point, thus always different points (even if representing the same group element), which avoids the known special cases of the usual addition law on elliptic curves.
- A group element can be encoded as its w coordinate; upon decoding, two matching curve points are rebuilt, but only one of them is a non-r-torsion point, and it can be identified by computing a Legendre symbol on its x coordinate.

Using fractional (x, u) coordinates, generic point addition can then be computed with cost 10M, and point doubling in cost 3M+6S.

Isogenies. Some useful isogenies also exist on double-odd curves. The $\mathcal{C}(a, b)$ double-odd curve is isogenous to another double-odd curve $\mathcal{C}(-2a, a^2 - 4b)$. Two isogenies that map between these two curves are the following:

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

These isogenies are in fact exactly the ones that can be obtained from Vélu's formulas[36], and have kernel $\{N, \mathbb{O}\}$ in their respective definition curves. Moreover, for any $P \in \mathcal{C}(a, b)$, we have $\psi'_{1/2}(\psi_1(P)) = 2P$. Using these isogenies, efficient point doubling formulas on $\mathcal{G}(a, b)$ can be defined, allowing per-doubling overhead in long sequences of successive doublings to be as low as 1M+5S or 2M+4S for some double-odd curves.

3 Double-Odd Curves as Jacobi Quartics

For a point $P = (x, u) \in \mathcal{C}(a, b)$, with $P \neq N, \mathbb{O}$, we define an additional coordinate:

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

This value is well-defined, since curve points other than N and $\mathbb{O}$ have a non-zero x coordinate. Using the curve equation, we can also derive another expression:

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

Since $x^2 + ax + b \neq 0$ for all x (otherwise, the curve would have additional points of order 2 and would not be double-odd), this expression is also well-defined, and can be applied to N as well, leading to $e = -1$ for that point. For reasons explained later on, we formally define $e = 1$ for the point-at-infinity $\mathbb{O}$. Since $b \notin QR$, $x^2 - b$ can never be zero, and thus $e \neq 0$ for all curve points.

Given e and u for a point, it is possible to recompute the x coordinate 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 for representing points. With our conventions, all points have such coordinates, including $N = (-1, 0)$ and $\mathbb{O} = (1, 0)$.

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[20]. 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[8,12,20,37]. The following classic point addition formulas can also be derived straightforwardly from the (x, u) formulas in [31]; 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:

$$e_3 = \frac{(1 + (a^2 - 4b)u_1^2u_2^2)(e_1e_2 - 2au_1u_2) + 2(a^2 - 4b)u_1u_2(u_1^2 + u_2^2)}{(1 - (a^2 - 4b)u_1^2u_2^2)^2}$$

$$u_3 = \frac{e_1u_2 + e_2u_1}{1 - (a^2 - 4b)u_1^2u_2^2}$$

These formulas are complete; they work on all curve points, including the point of order two N, and the point-at-infinity $\mathbb{O}$, thanks to the formal definitions of their coordinates as $(-1, 0)$ and $(1, 0)$, respectively. We also note that when $P = (e, u)$, then $P + N = (-e, -u)$, which further validates the choice of $(1, 0)$ for $\mathbb{O}$.

4 Group Element Encoding and Decoding

The main virtue of the representation of curve points in the Jacobi quartic is that the addition formulas work for all points *on the curve*. In the original definition of $\mathcal{C}(a, b)$ in [31], the (x, u) formulas were complete *on the group* under the assumption that both input points were the non-r-torsion representants of their respective group elements (if the formulas were to be used on P and $-P + N$ for some point P, then the x formula would fail). This required a point encoding format and decoding process that reliably return a non-r-torsion point. In that original description, this was done in the following way:

1. From the input field element w, get the two matching x coordinates by solving the equation $x^2 - (w^2 - a)x + b = 0$. This involves computing the discriminant $\Delta = (w^2 - a)^2 - 4b$, then extracing its square root $\sqrt{\Delta}$. The two candidates are $x = ((w^2 - a) \pm \sqrt{\Delta})/2$.
2. Choose the solution which is not a square. This requires computing a Legendre symbol.

The Legendre symbol computation is normally less expensive than the square root, but it is not negligible either, depending on the involved architecture; for a field of about 256 bits and a large 64-bit CPU, the cost of the Legendre symbol can be up to about 70% that of a square root, with optimized constant-time implementations of both operations. Removing the need of the additional Legendre symbol would make the decoding process noticeably faster. This can be done when using the Jacobi quartic, though it requires changing the encoding rule, so that the new encoding is not backward compatible with the previous one.

Sign. Let sign be an arbitrary "sign function" over the elements of $\mathbb{F}_q$, with the following characteristics:

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

A value z such that $\mathsf{sign}(z) = 1$ is said to be *negative*, while other values are *non-negative*. Any convention that fulfills these rules is usable; in practice, when $\mathbb{F}_q$ is the field of integers modulo a prime q, we may use the least significant bit of z when represented by an integer in the 0 to $q - 1$ range. For usual implementations of such finite fields, this sign convention can be evaluated inexpensively, with a cost lower than that of a multiplication in the field, i.e. much lower than that of a Legendre symbol, and negligible with regard to the cost of a square root extraction.

Encoding. A group element represented by the point $P = (e, u)$ is encoded into a field element u with the following process:

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

In other words, the group element has two representants, points $P = (e, u)$ and $P + N = (-e, -u)$. Exactly one of these two points has a non-negative e coordinate; we return the u coordinate of that point.

Decoding. Given an input field element u, the decoding process works as follows:

1. Compute $\Delta = (a^2 - 4b)u^4 - 2au^2 + 1$.
2. Compute $e = \sqrt{\Delta}$; if Δ has no square root, then the input u is invalid.
3. If $\mathsf{sign}(e) = 1$ then replace e with $-e$.
4. Return point (e, u).

This process simply computes e^2 from the curve equation, then extracts e as a square root, choosing the non-negative root, since that is the root that the encoding process used. Note that a single square root computation is needed, with no extra expensive operation. Moreover, the obtained point is in affine coordinates; there is no need to merge that root extraction with an inversion into an aggregate "square root of a ratio" operation, as is commonly used in the decoding of Ed25519 public keys[21].

Contrary to the original encoding/decoding rules from [31], this process may return any curve point, not necessarily the non-r-torsion points only. This is not a problem if all further operations use formulas that are complete on the curve, such as the ones detailed thereafter. If the original (x, u) formulas must be used, then an extra Legendre symbol will be needed to select the "right" representant (i.e. the point whose x coordinate is not a square).

Comparison with Decaf/Ristretto. The encoding/decoding process described above can be interpreted as a reduced version of the Decaf[17] process. Decaf works on a twisted Edwards curve with cofactor 4; the cofactor is eliminated by quotienting the curve with the

subgroup of 4-torsion points. The encoding and decoding rules then select a canonical representant through the use of an appropriate sign convention. Ristretto[3] is an adaptation of that process for twisted Edwards curves with cofactor 8, such as the well-known Curve25519.

For both Decaf and Ristretto, the decoding involves mainly a square root computation; another square root is needed when encoding. In the case of double-odd Jacobi quartics, we also need a square root for decoding, but the simpler situation (cofactor is only 2) allows for a direct encoding with only the inversion stemming from normalization (since, in practice, points are internally represented with some sort of fractional coordinate system). In general, inversions are more efficient than square roots, for two reasons:

- In small embedded CPUs (microcontrollers), inversion can be performed with an optimized binary GCD[30] or a similar quadratic algorithm[6] with a cost much lower than the modular exponentiation used at the core of a square root algorithm (on an ARM Cortex M0+ with the field of integers modulo $2^{255} - 19$, inversion can be done in about 20% of the cost of a square root).
- On large systems, the speed difference is less dramatic, but inversions are amenable to batching: using a trick due to Montgomery, a batch of n field elements can be inverted with a single inversion in the field, and an extra $3(n-1)$ multiplications, i.e. an asymptotic overhead of just 3 multiplications per value to invert. No such batching optimization is known for square roots[1].

Our new encoding/decoding rules provide the same performance as the usual point compression/decompression process that can be used on any elliptic curve.

5 The jq255e and jq255s Groups

We formally define the jq255e and jq255s groups. They are the same groups as the do255e and do255s groups defined in [31], except for the encoding and decoding rules, which use the process described above (in section 4). These groups offer the traditional "128-bit" security level, with a prime order (no non-trivial cofactor to deal with) and canonical encoding/decoding rules. We recall here the group parameters; see [31] for details on how they were chosen.

jq255e

- Field $\mathbb{F}_q$ with $q = 2^{255} - 18651$
- Curve equation parameters: $(a, b) = (0, -2)$

$$y^2 = x(x^2 - 2)$$
$$e^2 = 8u^4 + 1$$

- Curve order: $2r$, with $r = 2^{254} - 13152828129176421300604241380250168393 1$

[1] As was noted in [17], Decaf encoding can be optimized in that way if combined with point doubling, i.e. encoding the double of each element instead of the element itself. This requires backporting that extra doubling into the protocol that uses Decaf, and thus is somewhat at odds with the goal of providing a clean prime order group abstraction.

– Conventional generator G:

$$G_e = -3$$
$$G_u = -1$$

This curve's specific parameters allow for the most efficient known point doubling formulas among double-odd curves, with a per-doubling cost of 1M+5S only when used in sequences of doublings.

This curve is also a GLV curve[16], for which an efficient non-trivial endomorphism is known. Using the notations of [31] (section 6.2), the function $\delta(e, u) = (e, \eta u)$ for a given $\eta = \sqrt{-1}$ in $\mathbb{F}_q$ computes in a single field multiplication the product of a source point by a given scalar μ (which is a square root of -1 modulo r). This provides a non-negligible speed-up for some protocols that multiply a dynamically obtained point by a scalar, e.g. in a Diffie-Hellman key exchange.

jq255s

– Field $\mathbb{F}_q$ with $q = 2^{255} - 3957$
– Curve equation parameters: $(a, b) = (-1, 1/2)$

$$y^2 = x(x^2 - x + 1/2)$$
$$e^2 = -u^4 + 2u^2 + 1$$

– Curve order: $2r$, with $r = 2^{254} + 56904135270672826811114353017034461895$
– Conventional generator G:

$$G_e = 692965085280583754648534883375157967083785062147916414370316472331356868302 4$$
$$G_u = 3$$

This curve allows per-doubling costs of 2M+4S (within sequences of doublings). It does not otherwise have noticeable structure; e.g. its complex multiplication discriminant is very large, as would be expected from any randomly chosen curve. Contrary to jq255e, it does not offer an efficient endomorphism, but it is still quite efficient as a general purpose curve. This curve is meant as an alternative to jq255e in case it is feared that the low CM discriminant of jq255e may lead to exploitable weaknesses (no such attack is currently known, even though GLV curves have been proposed and used for more than 20 years now).

Other parameters. The double-odd curves underlying jq255e and jq255s were chosen so as to allow use of the most optimized point doubling formulas; this required varying the field modulus, as described in [31] (section 5). For implementations of operations on field elements that internally use 32-bit or 64-bit limbs, all moduli $2^{255} - m$ for $m < 2^{15}$ yield the same performance, and that kind of internal representation is what achieves the best performance *in general* on both small embedded CPUs (e.g. ARM Cortex M0+ and M4) and large CPUs (e.g. Intel x86 with Skylake or more recent cores). However, if m is very small (i.e. $m = 19$, the smallest m such that $2^{255} - m$ is prime), then that can lead to some minor speed-ups in

representations with more limbs, specifically 9 or 10 limbs, that themselves can better support implementations that leverage SIMD opcodes (e.g. AVX2 or NEON), in conjunction with specific curve point addition formulas or with higher-level batching of operations.

With modulus $q = 2^{255} - 19$, then point doubings will have cost 2M+5S (which is still quite efficient). It is still interesting to choose constants a and b such that a, b and $a^2 - 4b$ are as small as possible. Enumeration of all combinations with $|a| \leq 10$ and $|b| \leq 10$ (as plain integers) finds exactly two double-odd curves of order $2r$ with r prime: $\mathcal{C}(7, 8)$ and $\mathcal{C}(-7, 8)$ (these two curves are isomorphic to each other, since $-1 \in QR$ in that field). Extending the range of a and b to $[-20 ... +20]$ yields the curve $\mathcal{C}(14, 17)$ (and the isomorphic $\mathcal{C}(-14, 17)$), which is isogenous to $\mathcal{C}(7, 8)$ (ψ_1 maps from $\mathcal{C}(-7, 8)$ to $\mathcal{C}(14, 17)$). If experiments with $q = 2^{255} - 19$ are attempted, then it is suggested to use $\mathcal{C}(7, 8)$.

6 Formulas

For an efficient implementation, we use the extended coordinates proposed by Hisil, Wong, Carter and Dawson[19] (the same coordinate system had in fact already been described by Duquesne[15], albeit with different notations; we use here the notations of Hisil *et al*, which are more convenient). A point $P = (e, u)$ is represented as the quadruplet $(E{:}Z{:}U{:}T)$ such that $Z \neq 0$, $e = E/Z$, $u = U/Z$ and $u^2 = T/Z$ (this implies that $U^2 = TZ$). There are $q - 1$ possible quadruplets for a given curve point (for all possible non-zero values of Z).

Coordinates of $-P$ are $(E{:}Z{:}-U{:}T)$. Coordinates of $P + N$ are $(-E{:}Z{:}-U{:}T)$. Since P and $P + N$ represent the same group elements, we may freely move between P and $P + N$ as is convenient. The neutral element in the group $\mathcal{G}(a, b)$ can be represented by $N = (-Z{:}Z{:}0{:}0)$ or by $\mathbb{O} = (Z{:}Z{:}0{:}0)$, for any $Z \neq 0$.

In the rest of this section, extended coordinates are designed as EZUT. For implementing efficient doublings, we also make use of Jacobian (x, w) coordinates (denoted XWJ), in which a point $P = (x, w)$ is represented as a triplet $(X{:}W{:}J)$, such that $x = X/J^2$ and $w = W/J$; in XWJ coordinates, $N = (0{:}W{:}0)$ for any $W \neq 0$, and $\mathbb{O} = (W^2{:}W{:}0)$ for any $W \neq 0$. Following these conventions for N and $\mathbb{O}$ makes all formulas described thereafter complete.

6.1 Decoding

Algorithm 1 decodes an input field element into a point that represents an element of the group $\mathcal{G}(a, b)$. In practical situations, a binary format is needed (i.e. we encode into *bytes*); we assume here that a proper encoding of field elements into bytes has been defined (see appendix A.1).

Algorithm 1 Decoding from a field element

Require: $u \in \mathbb{F}_q$
Ensure: $P = (E{:}Z{:}U{:}T)$, or INVALID
1: $t \leftarrow u^2$
2: $\Delta \leftarrow (a^2 - 4b)t^2 - 2at + 1$
3: $E \leftarrow \sqrt{\Delta}$ ▷ The $\sqrt{\ }$ operator returns INVALID for non-squares.
4: **if** $E =$ INVALID **then return** INVALID
5: **if** $\mathrm{sign}(E) = 1$ **then**
6: $\quad E \leftarrow -E$
7: $Z \leftarrow 1$
8: $U \leftarrow u$
9: $T \leftarrow t$

6.2 Encoding

Algorithm 2 performs the reverse of algorithm 1: it turns a group element (represented by a point P) into a field element u. The same u is obtained for both possible representant points of a given element of $\mathcal{G}(a, b)$.

Algorithm 2 Encoding into a field element

Require: $P = (E{:}Z{:}U{:}T)$
Ensure: $u \in \mathbb{F}_q$
1: $K \leftarrow 1/Z$
2: $u \leftarrow KU$
3: **if** $\mathrm{sign}(KE) = 1$ **then**
4: $\quad u \leftarrow -u$

6.3 General Point Addition

Given points $P_1 = (E_1{:}Z_1{:}U_1{:}T_1)$ and $P_2 = (E_2{:}Z_2{:}U_2{:}T_2)$, their sum $P_3 = P_1 + P_2 = (E_3{:}Z_3{:}U_3{:}T_3)$ can be computed as:

$$E_3 = (Z_1Z_2 + (a^2 - 4b)T_1T_2)(E_1E_2 - 2aU_1U_2) + 2(a^2 - 4b)U_1U_2(Z_1T_2 + Z_2T_1)$$
$$Z_3 = (Z_1Z_2 - (a^2 - 4b)T_1T_2)^2$$
$$U_3 = (E_1U_2 + E_2U_1)(Z_1Z_2 - (a^2 - 4b)T_1T_2)$$
$$T_3 = (E_1U_2 + E_2U_1)^2$$

Algorithm 3 implements these formulas.

Algorithm 3 Point addition (cost: 8M+3S)

Require: $P_1 = (E_1{:}Z_1{:}U_1{:}T_1)$ and $P_2 = (E_2{:}Z_2{:}U_2{:}T_2)$
Ensure: $P_3 = P_1 + P_2 = (E_3{:}Z_3{:}U_3{:}T_3)$

1: $n_1 \leftarrow E_1E_2$
2: $n_2 \leftarrow Z_1Z_2$
3: $n_3 \leftarrow U_1U_2$
4: $n_4 \leftarrow T_1T_2$
5: $n_5 \leftarrow (Z_1 + T_1)(Z_2 + T_2) - n_2 - n_4$ ▷ $n_5 = Z_1T_2 + Z_2T_1$
6: $n_6 \leftarrow (E_1 + U_1)(E_2 + U_2) - n_1 - n_3$ ▷ $n_6 = E_1U_2 + E_2U_1$
7: $n_7 \leftarrow n_2 - (a^2 - 4b)n_4$ ▷ $n_7 = Z_1Z_2 - (a^2 - 4b)T_1T_2$
8: $E_3 \leftarrow (n_2 + (a^2 - 4b)n_4)(n_1 - 2an_3) + 2(a^2 - 4b)n_3n_5$
9: $Z_3 \leftarrow n_7^2$
10: $T_3 \leftarrow n_6^2$
11: $Z_3 \leftarrow ((n_6 + n_7)^2 - n_6 - n_7)/2$ ▷ $Z_3 = n_6n_7$

The cost estimate of 8M+3S assumes that multiplication by the constants $a^2 - 4b$ and $-2a$ are fast (there are three such multiplications in the formulas above). On the jq255e group, $a^2 - 4b = 8$ and $-2a = 0$; on jq255s, $a^2 - 4b = -1$ and $-2a = 2$. The computation of Z_3 can be replaced with the simple product n_6n_7; depending on the field implementation, target system, compiler version, and the usage context, this alternate computation may or may not provide slightly better or slightly worse performance.

When point P_2 is in affine coordinates (i.e. $Z_2 = 1$), then the computation of $n_2 = Z_1Z_2$ vanishes, and the computation of n_5 simplifies into $n_5 = Z_1T_2 + T_1$. The total cost is then slightly lower, at 7M+3S. If both P_1 and P_2 are in affine coordinates, then $n_2 = 1$ and $n_5 = T_1 + T_2$, for a cost of 6M+3S.

6.4 Negation and Subtraction

A point P is negated by negating its U coordinate; the other coordinate values are unmodified. Subtraction is performed by negation of the second operand, followed by addition. Negation in the field is inexpensive; therefore, point subtraction has about the same cost as point addition.

6.5 Doubling to Jacobian (*x*,*w*)

While point doublings can be performed correctly with the general addition formulas, a substantially faster way involves temporarily switching to xwJ coordinates. The following formulas compute the point $2P = (X'{:}W'{:}J')$ from $P = (E{:}Z{:}U{:}T)$ (with three possible choices for the computation of W'):

$$\begin{aligned} X' &= E^4 \\ W' &= 2Z^2 - 2aU^2 - E^2 = Z^2 - (a^2 - 4b)T^2 = E^2 + 2aU^2 - 2(a^2 - 4b)T^2 \\ J' &= 2EU \end{aligned}$$

Since we can use both P and $P + N$ as representants for group elements, we can also use the following formulas that output $2P + N = (X' : W' : J')$ from $P = (E : Z : U : T)$:

$$\begin{aligned} X' &= 16bU^4 &&= 16b(TZ)^2 \\ W' &= E^2 + 2aU^2 - 2Z^2 &&= (a^2 - 4b)T^2 - Z^2 = 2(a^2 - 4b)T^2 - E^2 - 2aU^2 \\ J' &= 2EU \end{aligned}$$

It can be verified that if the source is $\mathbb{O} = (Z : Z : 0 : 0)$ or $N = (-Z : Z : 0 : 0)$ for some $Z \neq 0$, then these formulas indeed yield $(Z^4 : Z^2 : 0)$ and $(0 : -Z^2 : 0)$, respectively, i.e. correct XWJ representations of $\mathbb{O}$ and N.

Algorithm 4 is applicable to all double-odd curves and computes $2P + N$ from P in cost 2M+2S.

Algorithm 4 Doubling EZUT to XWJ (cost: 2M+2S)

Require: $P = (E : Z : U : T)$
Ensure: $P' = 2P + N = (X' : W' : J')$
1: $n \leftarrow U^2$ ▷ $n = U^2 = TZ$
2: $X' \leftarrow 16bn^2$
3: $W' \leftarrow ((a^2 - 4b)T + Z)(T - Z) + (a^2 - 4b - 1)n$ ▷ $W' = (a^2 - 4b)T^2 - Z^2$
4: $J' \leftarrow 2EU$

When working in a field $\mathbb{F}_q$ where $q = 3 \bmod 4$, $-1 \notin QR$, and therefore $4b - a^2 \in QR$, and we can use $\sqrt{4b - a^2}$ to change the computation of W' into:

$$W' = 2(\sqrt{4b - a^2})n - ((\sqrt{4b - a^2})T + Z)^2$$

This lowers the cost down to 1M+3S, provided that multiplications by the constant value $\sqrt{4b - a^2}$ are fast; this is assuredly the case for jq255s, where $4b - a^2 = 1$. Doubling to XWJ for jq255s, using that trick, is shown in algorithm 5.

Algorithm 5 Doubling EZUT to XWJ on jq255s (cost: 1M+3S)

Require: $P = (E : Z : U : T) \in \mathcal{C}(-1, 1/2)$
Ensure: $P' = 2P + N = (X' : W' : J')$
1: $n \leftarrow U^2$ ▷ $n = U^2 = TZ$
2: $X' \leftarrow 8n^2$
3: $W' \leftarrow 2n - (T + Z)^2$ ▷ $W' = -T^2 - Z^2$
4: $J' \leftarrow 2EU$

On jq255e, and more generally on double-odd curves with $a = 0$, the generic formulas lead to a cost 2M+2S (the requirement that both b and $a^2 - 4b$ are non-squares, combined with $a = 0$, implies that $-4 \in QR$, which is not possible in a field $\mathbb{F}_q$ with $q = 3 \bmod 4$).

However, if computing $2P$ instead of $2P + N$, then it is possible to obtain 1M+3S formulas applicable to such curves; this is shown in algorithm 6.

Algorithm 6 Doubling EZUT to XWJ on $\mathcal{C}(0, b)$ (e.g. jq255e) (cost: 1M+3S)

Require: $P = (E{:}Z{:}U{:}T) \in \mathcal{C}(-1, 1/2)$
Ensure: $P' = 2P = (X'{:}W'{:}J')$
1: $n \leftarrow E^2$
2: $X' \leftarrow n^2$
3: $W' \leftarrow 2Z^2 - n$ ▷ $W' = 2Z^2 - 2aU^2 - E^2$
4: $J' \leftarrow 2EU$

6.6 Conversion from XWJ to EZUT

A point $(X{:}W{:}J)$ in XWJ coordinates can be converted back to EZUT coordinates $(E{:}Z{:}U{:}T)$ with the following:

$$\begin{aligned} Z &= W^2 \\ T &= J^2 \\ U &= JW \\ E &= 2X - Z + aT \end{aligned}$$

Since W^2 and J^2 are also computed, the product JW can be done with a squaring operation instead of a multiplication. This is illustrated in algorithm 7.

Algorithm 7 Conversion from XWJ to EZUT (cost: 3S)

Require: $P = (X{:}W{:}J)$
Ensure: $P = (E{:}Z{:}U{:}T)$
1: $Z \leftarrow W^2$
2: $T \leftarrow J^2$
3: $U \leftarrow ((W + J)^2 - Z - T)/2$ ▷ $U = JW$
4: $E \leftarrow 2X - Z + aT$

6.7 Doublings and Sequences of Doublings

A doubling in EZUT coordinates is done trivially by doing the doubling to XWJ (with algorithm 4, 5 or 6, depending on the curve type), then converting back the result to EZUT with algorithm 7. This leads to a cost of 2M+5S generically (which matches the best generic formulas from [19]), but only 1M+6S when $q = 3 \bmod 4$ (a category that includes jq255s), or when $a = 0$ (which is the case of jq255e).

To compute a sequence of n doublings, it is of course possible to invoke the doubling process n times. However, for some curves (curves $\mathcal{C}(0, b)$, such as jq255e, and also specifically curve jq255s), one can do better by noticing that there exist faster doubling formulas for these curves when in XWJ coordinates. This leads to the following process:

1. Compute the first doubling with output in xwJ coordinates (algorithm 5 or 6, cost 1M+3S).
2. Compute the next $n-1$ doublings over xwJ coordinates (algorithm 8 or 9, with cost 2M+4S or 1M+5S per doubling, respectively).
3. Convert the result back to ezut (algorithm 7, cost 3S).

In total, the first doubling costs exactly as much as in the single doubling case (1M+6S) but extra doublings are cheaper (2M+4S or 1M+5S each). This is beneficial to algorithms that evaluate long sequences of doublings, in particular multiplication of a point by a scalar, with classic window optimizations, or wNAF scalar representation.

Algorithms 8 and 9, shown below, detail how fast doublings can be computed on jq255s and jq255e in xwJ coordinates, respectively. These algorithms were already described in the original double-odd whitepaper[31] and are recalled here for completeness. Like all other algorithms in this paper, they are complete.

Algorithm 8 Doubling on jq255s (xwJ) (cost: 2M+4S)

Require: $P = (X{:}W{:}J) \in \mathcal{C}(-1, 1/2)$
Ensure: $P' = 2P + N = (X'{:}W'{:}J')$
1: $n_1 \leftarrow WJ$
2: $n_2 \leftarrow n_1^2$
3: $n_3 \leftarrow (W+J)^2 - 2n_1$ ▷ $n_3 = W^2 + J^2$
4: $X' \leftarrow 8n_2^2$
5: $W' \leftarrow 2n_2 - n_3^2$
6: $J' \leftarrow 2n_1(2X - n_3)$

Algorithm 9 Doubling on $\mathcal{C}(0, b)$ (e.g. jq255e) (xwJ) (cost: 1M+5S)

Require: $P = (X{:}W{:}J) \in \mathcal{C}(0, -2)$
Ensure: $P' = 2P = (X'{:}W'{:}J')$
1: $n_1 \leftarrow W^2$
2: $n_2 \leftarrow n_1 - 2X$
3: $n_3 \leftarrow n_2^2$
4: $X' \leftarrow n_3^2$
5: $W' \leftarrow n_3 - 2n_1^2$
6: $J' \leftarrow J((W+n_2)^2 - n_1 - n_3)$ ▷ $J' = 2JWn_2$

6.8 Equality Tests

Two points $P_1 = (E_1{:}Z_1{:}U_1{:}T_1)$ and $P_2 = (E_2{:}Z_2{:}U_2{:}T_2)$ can be compared with each other (as representants of group elements) by noticing that P_1 and P_2 represent the same group element if and only if $\psi_1(P_1)$ and $\psi_1(P_2)$ are the same point on curve $\mathcal{C}(-2a, a^2 - 4b)$, since $\{\mathbb{O}, N\}$ is the kernel of the ψ_1 isogeny. Moreover, for any source point $P \in \mathcal{C}(a, b)$, $\psi_1(P) \in$

$\mathcal{C}(-2a, a^2 - 4b)[r]$ (ψ_1 only outputs r-torsion points), and the u coordinate uniquely determines a point in the subgroup of r-torsion points. Thus, we only have to compare the u coordinates of $\psi_1(P_1)$ and $\psi_1(P_2)$, which are equal to $-U_1/E_1$ and $-U_2/E_2$, respectively.

This leads us to the following efficient test: P_1 and P_2 represent the same element in $\mathcal{G}(a, b)$ if and only if $U_1E_2 = U_2E_1$. This test is complete (it also works for N and $\mathbb{O}$) since the E coordinate is never zero.

A special case is when comparing a point to the group neutral: a point $P = (E{:}Z{:}U{:}T)$ represents the neutral of $\mathcal{G}(a, b)$ if and only if $U = 0$.

6.9 Field to Group Map

Mapping a field element to a group element is an important support functionality for defining a hash-to-curve process. Two such maps were previously specified in [31] (sections 3.7 and 6.1.6): the general case, applicable to all double-odd curves with $a \neq 0$, is the Elligator2[7] map, while a special-case map is defined for curves with $a = 0$, to which Elligator2 does not apply. Since jq255e and jq255s use the same underlying curves as do255e and do255s, we can use the same maps, with only an extra final conversion to obtain the (e, u) coordinates for the obtained points. The map-to-curve processes for jq255e and jq255s are specified in algorithms 10 and 11, respectively. In these algorithms, all square root computations return the non-negative root.

Algorithm 10 Mapping a field element to $\mathcal{C}(0, -2)$ (jq255e)

Require: $f \in \mathbb{F}_q$
Ensure: $P = (E{:}Z{:}U{:}T) \in \mathcal{C}(0, -2)$

if $f = 0$ **then**
  return $\mathbb{O} = (1{:}1{:}0{:}0)$
$\dot{x}_1 \leftarrow 4f^2 - 7$
$\dot{x}_2 \leftarrow (4f^2 + 7)\sqrt{-1}$ ▷ Use the non-negative square root of –1.
$\ddot{x}_0 \leftarrow 4f$
$z_1 \leftarrow 64f^7 + 176f^5 - 308f^3 - 343f$
$z_2 \leftarrow -(64f^7 - 176f^5 - 308f^3 + 343f)\sqrt{-1}$
$\ddot{y}_0 \leftarrow 8f^2$
if $z_1 \in QR$ **then**
  $(\dot{x}, \ddot{x}, \dot{y}, \ddot{y}) \leftarrow (\dot{x}_1, \ddot{x}_0, \sqrt{z_1}, \ddot{y}_0)$
else if $z_{n,2} \in QR$ **then**
  $(\dot{x}, \ddot{x}, \dot{y}, \ddot{y}) \leftarrow (\dot{x}_2, \ddot{x}_0, \sqrt{z_2}, \ddot{y}_0)$
else
  $(\dot{x}, \ddot{x}, \dot{y}, \ddot{y}) \leftarrow (\dot{x}_1\dot{x}_2, \ddot{x}_0^2, \sqrt{z_1 z_2}, \ddot{y}_0^2)$
$(\dot{u}, \ddot{u}) \leftarrow (\dot{x}\ddot{y}, \ddot{x}\dot{y})$ ▷ $(\dot{x}/\ddot{x}, \dot{y}/\ddot{y}, \dot{u}/\ddot{u})$ is a point on the dual curve $\mathcal{C}(0, 8)$.
$(\dot{X}, \ddot{X}) \leftarrow (-8\dot{u}^2, \ddot{u}^2)$
$(\dot{U}, \ddot{U}) \leftarrow (2\dot{x}\ddot{x}\dot{u}, \dot{u}(\dot{x}^2 - 8\ddot{x}^2))$
$(\dot{E}, \ddot{E}) \leftarrow (\dot{X}^2 + 2\ddot{X}^2, \dot{X}^2 - 2\ddot{X}^2)$
$(E{:}Z{:}U{:}T) \leftarrow (\dot{E}\ddot{U}^2{:}\ddot{E}\ddot{U}^2{:}\dot{U}\ddot{U}\ddot{E}{:}\dot{U}^2\ddot{E})$

Algorithm 11 Mapping a field element to $\mathcal{C}(-1, 1/2)$ (jq255s)

Require: $f \in \mathbb{F}_q$
Ensure: $P = (E{:}Z{:}U{:}T) \in \mathcal{C}(-1, 1/2)$
if $f = \pm 1$ **then**
  return $\mathbb{O} = (1{:}1{:}0{:}0)$
$z_1 \leftarrow -2f^6 + 14f^4 - 14f^2 + 2$
$z_2 \leftarrow -z_1 f^2$
$\ddot{x} \leftarrow 1 - f^2$
if $z_1 \in QR$ **then**
  $(\dot{x}, \dot{y}) \leftarrow (-2, \sqrt{z_1})$
else
  $(\dot{x}, \dot{y}) \leftarrow (2f^2, -\sqrt{z_2})$
if $\dot{y} = 0$ **then**
  return $\mathbb{O} = (1{:}1{:}0{:}0)$
$(\dot{u}, \ddot{u}) \leftarrow (\dot{x}\ddot{x}, \dot{y})$ ▷ $(\dot{x}/\ddot{x}, \dot{y}/\ddot{x}^2, \dot{u}/\ddot{u})$ is a point on the dual curve $\mathcal{C}(2, -1)$.
$(\dot{X}, \ddot{X}) \leftarrow (2\dot{u}^2, \ddot{u}^2)$
$(\dot{U}, \ddot{U}) \leftarrow (2\ddot{u}, \dot{x}^2 + 2\ddot{x}^2)$
$(n_1, n_2) = (\dot{X}(2\dot{X} - \ddot{X}), \ddot{X}(\dot{X} - \ddot{X}))$
$(\dot{E}, \ddot{E}) \leftarrow (n_1 + n_2, n_1 - n_2)$
$(E{:}Z{:}U{:}T) \leftarrow (\dot{E}\ddot{U}^2{:}\ddot{E}\ddot{U}^2{:}\dot{U}\ddot{U}\ddot{E}{:}\dot{U}^2\ddot{E})$

7 Short and Fast Signatures

Alternate Signature Encodings. In [31], digital signatures over groups do255e and do255s were defined as classic Schnorr signatures, yielding 64-byte signatures, the same size as standard Ed25519 signatures[21], and with similar performance. Since jq255e and jq255s use a different encoding format, and thus break compatibility with do255e and do255s, we have an opportunity to define a slightly different signature scheme that yields better performance, namely *shorter encodings* (48 bytes instead of 64), and also *faster verification*.

A Schnorr signature scheme[33] can be described as follows:

- The private key is a secret scalar d (an integer modulo the group order r). The corresponding public key is $Q = dG$, with G being the conventional base point in the group.
- To sign a message m, the following computations are performed:
  1. Generate a per-signature secret scalar k (uniformly among integers modulo r).
  2. Compute $R = kG$ (this is the *commitment*).
  3. Compute $c = H(R \parallel Q \parallel m)$, where H is a suitable hash function, computed over an unambiguous encoding of the per-signature commitment, the public key, and the message itself. Value c is the *challenge* and is interpreted as an integer modulo r.
  4. Compute $s = k + cd \bmod r$.

  The signature is nominally the (R, c, s) triplet.
- Verification of the signature entails validating that the challenge c is correct by recomputing it from R, Q and m, and also checking that the verification equation $sG = R + cQ$ is fulfilled.

Since the verifier recomputes the challenge c, that value needs not be transmitted, so the signature can be reduced to the (R, s) pair. This is what happens with Ed25519, where R and s are both encoded over 32 bytes each, for a total signature size of 64 bytes. Another possible way to encode the signature is to include c instead of R; after all, *checking* the verification equation is equivalent to *recomputing* the point $R = sG - cQ$ and comparing it with the received value. As long as c is provided, R can be omitted. The total signature size can thus be reduced, if c is defined to be shorter than the encoding of R; this is what we propose here.

Using (c, s) as the signature instead of (R, s), and reducing the size of c, are not new ideas. They were already in the original paper from Schnorr[33]. They were proposed again by Naccache and Stern[22], and yet again by Neven, Smart and Warinschi[23]. This last paper also includes extensive analysis on the security of the construction depending on the particulars of the hash function H and the size of c; we summarize here the facts from their analysis that are relevant to our proposal:

- The security of Schnorr signatures against forgeries does not depend on the collision resistance of the hash function H. Thus, existing collision attacks on some hash functions are (mostly) irrelevant.
- There are known proofs of security that reduce the security of Schnorr signatures to the discrete logarithm, with some conditions on the size of the output of H. These proofs are not tight; they show that attacks, assuming "generic" group and hash function H, have cost at least 2^n with a hash function of output $2n$ bits, and a group of size at least 2^{3n} (i.e. a $3n$-bit curve, in our context).
- On the other hand, the best known attacks have cost at least 2^n if the group has size at least 2^{2n} and a hash function output of at least n bits, as long as the hash function is not an n-bit "narrow-pipe" design (in practice, this means that H should be a hash function that outputs $2n$ bits, then truncated to its first n bits).
- Existing standard signature schemes tend to be "in between" these two bounds. Notably, Ed25519 uses a group of size about 2^{252}, i.e. at the low end of the range ($2n$-bit curve for n bits of security), but uses a hash function with a massive $4n$-bit output, that gets naturally truncated (through modular reduction) to the $2n$-bit size of scalars. In a sense, Ed25519 is a lot more conservative on the hash function security than on the curve.

The original Ed25519 paper[4] quotes the papers from Schnorr[33] and Neven, Smart and Warinschi[23], and discusses the possibility of using a short hash output, but ultimately rejects it because "the use of double-size hashing helps alleviate concerns regarding hash-function security", without much further explanations. This is probably explained by the historical context: that paper was written at a time when serious weaknesses had been found in MD5 and SHA-1, and nobody really knew whether the SHA-2 function would fall next; the SHA-3 competition was still ongoing, and there was a general unease with regard to the security of hash functions. Another possible reason for the choice of (R, s) representation of signatures in Ed25519 is that it is compatible with the batch verification optimized algorithms presented in the paper, while the (c, s) format is not.

Faster Verification. While we want to use shorter signatures primarily because there are many contexts, notably in the world of embedded systems with severe constraints on communication channels, where any gain on the signature size is important, it turns out that using

a shorter challenge c also promotes efficiency, by making signature verification faster. Let's consider the implementation of a Schnorr signature scheme with a group of size $r \approx 2^{2n}$, and a $2n$-bit challenge c. The verification equation:

$$R = sG - cQ$$

entails computing a linear combination of two points, one of them being the conventional generator point G. This step is commonly implemented with Straus's algorithm[34] (also known as "Shamir's trick"), to the effect that for a group of size 2^{2n}, about $2n$ point doublings and $4n/w$ point additions will be used (for a given "window size" w, which is typically between 4 and 6 bits, depending on the target system, scarcity of RAM, and use of wNAF encoding of scalars). An optimization first described by Antipa *et al*[2] consists of splitting the challenge c into two half-size (n bits) integers c_0 and c_1 such that $c = c_0/c_1 \bmod r$, and transforming the verification equation into:

$$(sc_1)G - c_1R - c_0Q = \mathbb{O}$$

Since G is the fixed generator point, the point 2^nG can be precomputed, and the value sc_1 can be written as $sc_1 = t_0 + 2^n t_1 \bmod r$, leading to:

$$t_0G + t_1(2^nG) - c_1R - c_0Q = \mathbb{O}$$

which is a linear combination of *four* points, but with half-width coefficients (t_0, t_1, c_0 and c_1 all have size n bits, not $2n$ bits). Again using Straus's algorithm, this leads to the same number of general point additions ($4n/w$) as previously, but with only half the number of point doublings (n instead of $2n$). The source challenge c must also be split into c_0 and c_1; the use of Lagrange's lattice basis reduction algorithm, as optimized in [29], makes that step fast enough (about 13k cycles on an x86 Coffee Lake core, for a 253-bit source scalar) to make use of Antipa *et al*'s optimization worthwhile.

Now, suppose that we halve the size of the challenge c. The verification equation can now be written as:

$$R = s_0G + s_1(2^nG) - cQ$$

with a split of s into two n-bit halves: $s = s_0 + 2^n s_1$. This is a linear combination of *three* points instead of four, with again half-width coefficients. It can thus be computed with the same number of point doublings (n) but fewer general point additions ($3n/w$ instead of $4n/w$) than with Antipa *et al*'s method; moreover, the split of s is immediate, and there is no need to run Lagrange's algorithm. We also note that the point R can be obtained as an output, making this verification equation compatible with the use of (c, s) as the signature format.

Benchmarks. We implemented groups jq255s and jq255e, as well as the twisted Edwards curve Ed25519, in the Rust language, as part of the open-source `crrl` library available at:

`https://github.com/pornin/crrl`

The relevant library modules are called `jq255s`, `jq255e` and `ed25519`, respectively. The exact specifications of the implemented signature scheme are provided in appendix A.2 (in a nutshell: the BLAKE2s hash function is used for H, with an output truncated to 128 bits).

Performance was measured on an Intel i5-8259U CPU, clocked at 2.3 GHz (TurboBoost was disabled), running Linux (Ubuntu 22.04) in 64-bit mode, and using the Rust compiler version 1.59.0 ("stable" channel, using LLVM 13). To allow use of all local CPU opcodes, compilation flags "`-C target-cpu=native`" were used. Table 1 shows the results.

Operation	**jq255e**	**jq255s**	**Ed25519**
Decode point	9371	9295	9491
Encode point	7847	7874	7864
Point multiplication	72080	107875	107730
Point multiplication (base)	43570	44915	40456
Sign	54738	56160	51581
Verify	82839	86837	113983
Point size (bytes)	32	32	32
Scalar size (bytes)	32	32	32
Signature size (bytes)	48	48	64

Table 1: Performance of the `crrl` implementation of jq255e, jq255s and Ed25519 on 64-bit x86 (Intel Coffee Lake core); timing measurements in clock cycles.

The "Decode" and "Encode" operations correspond to decoding a group element from 32 bytes, and encoding it into 32 bytes, respectively. "Point multiplication" is multiplication of a dynamically obtained (but already decoded) point by a full-width scalar; it is used in the ECDH key exchange (but not in signatures). "Point multiplication (base)" is the same operation but over the fixed and conventional base point, which allows using multiple precomputed tables with affine coordinates; this is the main operation used in key pair generation and in signature generation. "Sign" is signature generation over a short message (of length 32 bytes); apart from the multiplication of the base point by the scalar k, it also includes generation of k (using a derandomized deterministic process), encoding of the point R, computation of the challenge c, and computation of the signature element s. Signature generation assumes that the encoded public key is already available (i.e. the public key is not dynamically recomputed from the private key for each signature). "Verify" applies the signature verification, from the encoded signature as bytes, and a short message (32 bytes); the verification public key (Q) is provided as an already decoded point (for a verification primitive that receives the public key in encoded format, the cost of the "Decode" operation must be added to that of "Verify").

All operations except "Verify" are fully constant-time. This includes the point decoding and encoding operations; timing-based side channels leak neither information about the involved points, nor whether the operation succeeded or not. Signature verification ("Verify") is assumed to work only on public data and employs non-constant-time implementation strategies (in particular wNAF recoding of scalars).

The implementation of operations modulo $2^{255} - m$ (for integers $m = 18651$, 3957 and 19, respectively) uses the same code, a generic `GF255<m>` type that receives the value of m as a type parameter (i.e. a compile-time constant). The code uses only pure Rust, with two intrisic functions for additions and subtractions with carry propagation, on x86 plat-

forms, but no assembly. Thus, all three curves benefit from the same optimizations at the field level. The jq255e and jq255s implementations use the formulas described in this paper; the Ed25519 implementation uses the classic extended coordinates from Hisil, Wong, Carter and Dawson[18]. In a sequence of successive doublings with Ed25519, all doublings have cost 3M+4S, except the last which has cost 4M+4S.

For point multiplication, 5-bit windows (with constant-time lookups) are used. When working over the base point, four precomputed tables are used (for G, $2^{65}G$, $2^{130}G$ and $2^{195}G$) with affine coordinates (affine extended coordinates (e, u, u^2) for jq255e and jq255s, Duif extended coordinates for Ed25519).

It shall be noted that our performance on Ed25519 is competitive with other optimized implementations. The eBACS/SUPERCOP benchmarking framework[5] contains several implementations of Ed25519 signatures, some of which being optimized with 64-bit x86 assembly (`amd64-51-30k` and `amd64-64-24k`). The list of measurement machines includes a system with a Coffee Lake core (called "r24000"), thus comparable to our test system. Signature generation is reported with cost 48744 cycles; our own figure of 51581 cycles is less than 6% higher[2]. For signature verification, the SUPERCOP implementations achieve 164961 cycles, substantially worse (+34% time) than our code (123474 cycles, when adding the costs of decoding the verification public key, and the verification itself); this is in part due to the fact that our code implements the Antipa *et al* optimization[2,29] while SUPERCOP's code does not. However, if we modify our code so as not to leverage that optimization, then we can still achieve a verification cost of about 129000 cycles (about 138500 cycles with the decoding of the public key), which is still noticeably lower than SUPERCOP's implementation cost.

Another well-known Ed25519 implementation is the Rust library `ed25519-dalek`[14]. That library leverages AVX2 opcodes (using intrinsic functions) and some parallelism inside the point addition formulas to compute up to four field multiplications simultaneously. On our test system, they can perform an Ed25519 verification in about 118000 cycles[3] (not counting the public key decoding). That library does not use the Antipa *et al* optimization; if it did, it could presumably achieve a similar speed-up as in our case, on the order of 15000 cycles or so, which would yield a cost slightly above 100k cycles.

These figures vindicate our implementation strategy (i.e., trust LLVM and don't fiddle with assembly) and mean that our measurements are a faithful representation of the curves' "true speed". We see that, as expected, point decoding and encoding have the same cost for all three curves. Point multiplication with jq255s offers performance very similar to that of Ed25519, but jq255e is substantially faster (cost is only 2/3 of that of Ed25519) because jq255e is a GLV curve[16] and our implementation leverages the endomorphism to avoid half of the doublings. *A contrario*, when working over the base point, Ed25519 is about 10% faster: the multiplicity of precomputed tables for the base point means that much fewer point doublings are performed, which favours Ed25519, whose point doubling is more expensive than that of jq255e and jq255s, but its general point addition is somewhat faster. As expected, the use of a half-width challenge c in signatures provides a substantial boost to signature veri-

[2] We could probably slightly improve performance by using more and/or larger precomputed tables for the base point; our current implementation uses only 6144 bytes for these tables, and could easily be doubled in size without hitting any L1 cache limit.

[3] The AVX2 backend of `ed25519-dalek` needs a "nightly" compiler, so for this test we used Rust 1.65.0, from August 3rd, 2022.

fication; even with the decoding of the public key, both jq255e and jq255s provide signature verification in less than 100k cycles.

It should be remarked that in some contexts, where many signatures should be verified at roughly the same time, then Ed25519 signatures can use a batched verification process, that yields substantial gains: the `ed25519-dalek` authors report a per-verification cost down to 60750 cycles on a Skylake-class Intel CPU[4]. This batch verification is not applicable to our short signatures in (c, s) format. To make our signatures amenable to that process, they must use the (R, s) format, which forfeits the size advantage (signature encoding size would be 64 bytes instead of 48), while still keeping the speed gain on non-batched verifications.

From these figures, we can conclude that indeed jq255e and jq255s are fast. They moreover provide prime order groups, a primitive adequate for building cryptographic protocols. A prime order group can be obtained over Curve25519 with Ristretto[3]; we implemented it as well, and obtain the same performance as Ed25519 (as expected) except for point encoding, which is slightly more expensive, and in fact costs as much as point decoding (which is again expected, both operations involving a square root).

Acknowledgements

We thank Kevin Henry, Giacomo Pope and Javed Samuel, who reviewed this paper.

References

1. C. Ambrose, J. Bos, B. Fay, M. Joye, M. Lochter and B. Murray, *Differential Attacks on Deterministic Signatures*, https://eprint.iacr.org/2017/975
2. A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik and S. Vanstone, *Accelerated Verification of ECDSA signatures*, Selected Areas in Cryptography - SAC 2005, Lecture Notes in Computer Science, vol. 3897, pp. 307-318, 2005.
3. T. Arcieri, I. Lovecruft and H. de Valence, *The Ristretto Group*, https://ristretto.group/
4. D. Bernstein, N. Duif, T. Lange, P. Schwabe and B.-Y. Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering, vol. 2, issue 2, pp. 77-89, 2012.
5. D. Bernstein and T. Lange, *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, https://bench.cr.yp.to (accessed 4 August 2022).
6. D. Bernstein and B.-Y. Yang, *Fast constant-time gcd computation and modular inversion*, https://gcd.cr.yp.to/papers.html#safegcd
7. D. Bernstein, M. Hamburg, A. Krasnova and T. Lange, *Elligator: elliptic-curve points indistinguishable from uniform random strings*, Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, 2013, https://doi.org/10.1145/2508859.2516734
8. O. Billet and M. Joye, *The Jacobi model of an elliptic curve and side-channel analysis*, AAECC-15, Lecture Notes in Computer Science, vol. 2643, pp. 34-42, 2003.

[4] Batch verification can only report whether all signatures are correct or not; if the process fails then identifying the invalid signatures in the batch requires more efforts. However, in most practical contexts, signatures are almost always valid.

9. É. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam and M. TIbouchi, *Efficient Indifferentiable Hashing into Ordinary Elliptic Curves*, Advances in Cryptology - CRYPTO 2010, Lecture Notes in Computer Science, vol. 6223, pp. 237-254, 2010.
10. K. Chalkias, *ed25519-unsafe-libs*, https://github.com/MystenLabs/ed25519-unsafe-libs
11. K. Chalkias, F. Garillot and V. Nikolaenko, *Taming the Many EdDSAs*, Security Standardisation Research - SSR 2020, Lecture Notes in Computer Science, vol. 12529, pp. 67-90, 2020.
12. D. Chudnovsky and G. Chudnovsky, *Sequences of numbers generated by addition in formal groups and new primality and factorization tests*, Advances in Applied Mathematics, vol. 7, issue 4, pp. 385–434, 1986.
13. C. Cremers and D. Jackson, *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman*, IEEE 32nd Computer Security Foundations Symposium (CSF), 2019.
14. I. Lovecruft and H. de Valence, *Dalek cryptography*, https://github.com/dalek-cryptography
15. S. Duquesne, *Improving the arithmetic of elliptic curves in the Jacobi model*, Information Processing Letters, vol. 104, issue 3, pp. 101–105, 2007.
16. R. Gallant, J. Lambert and S. Vanstone, *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms*, Advances in Cryptology - CRYPTO 2001, Lecture Notes in Computer Science, vol. 2139, pp. 190-200, 2001.
17. M. Hamburg, *Decaf: Eliminating cofactors through point compression*, Advances in Cryptology - CRYPTO 2015, Lecture Notes in Computer Science, vol. 9215, pp. 705-723, 2015.
18. H. Hisil, K. Wong, G. Carter and E. Dawson, *Twisted Edwards Curves Revisited*, Advances in Cryptology - ASIACRYPT 2008, Lecture Notes in Computer Science, vol. 5350, pp. 326-343, 2008.
19. H. Hisil, K. Wong, G. Carter and E. Dawson, *Jacobi Quartic Curves Revisited*, Information Security and Privacy - ACISP 2009, Lecture Notes in Computer Science, vol. 5594, pp. 452-468, 2009.
20. C. G. J. Jacobi, *Fundamenta nova theoriae functionum ellipticarum*, Sumtibus fratrum, 1829.
21. S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, https://tools.ietf.org/html/rfc8032
22. D. Naccache and J. Stern, *Signing on a Postcard*, Financial Cryptography - FC 2000, Lecture Notes in Computer Science, vol. 1962, pp. 121-135, 2001.
23. G. Neven, N. P. Smart and B.Warinschi, *Hash function requirements for Schnorr signatures*, Journal of Mathematical Cryptology, vol. 3, issue 1, pp. 69-87, 2009.
24. Information Technology Laboratory, *Secure Hash Standard (SHS)*, National Institute of Standard and Technology, FIPS 180-4, 2015.
25. Information Technology Laboratory, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, National Institute of Standard and Technology, FIPS 202, 2015.
26. J. O'Connor, J.-P. Aumasson, S. Neves and Z. Wilcox-O'Hearn, *BLAKE3*, https://github.com/BLAKE3-team/BLAKE3
27. D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter and P. Rösler, *Attacking Deterministic Signature Schemes Using Fault Attacks*, 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 338-352, 2018.
28. T. Pornin, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, https://tools.ietf.org/html/rfc6979
29. T. Pornin, *Optimized Lattice Basis Reduction In Dimension 2, and Fast Schnorr and EdDSA Signature Verification*, https://eprint.iacr.org/2020/454

30. T. Pornin, *Optimized Binary GCD for Modular Inversion*, https://eprint.iacr.org/2020/972
31. T. Pornin, *Double-Odd Elliptic Curves*, https://eprint.iacr.org/2020/1558
32. M.-J. Saarinen and J.-P. Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)* https://datatracker.ietf.org/doc/html/rfc7693
33. C. Schnorr, *Efficient identification and signatures for smart cards*, Advances in Cryptology - CRYPTO '89, Lecture Notes in Computer Science, vol. 435, pp. 239-252, 1990.
34. E. Straus, *Addition chains of vectors (problem 5125)*, American Mathematical Monthly, vol. 70, pp. 806-808, 1964.
35. H. de Valence, J. Grigg, M. Hamburg, I. Lovecruft, G. Tankersly and F. Valsorda, *The ristretto255 and decaf448 Groups*, https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-03
36. J. Vélu, *Isogénies entre courbes elliptiques*, C.R. Acad. Sc. Paris, Série A, vol. 273, pp. 238-241, 1971.
37. E. Whittaker and G. Watson, *A Course of Modern Analysis*, Cambridge University Press, 1927.

A Cryptographic Algorithm Specifications

In this section, we specify strict rules for cryptographic algorithms built on top of the jq255e and jq255s groups. We try to cover all edge cases, as an attempt to avoid the kind of messy situation that impacts existing deployments of Ed25519, where implementations don't always agree with each other about the validity of some signatures, leading to issues in some applications, especially consensus protocols[11].

In all this section, a *byte* is really an *octet*; it has a numerical value in the 0 to 255 range, and contains eight bits, numbered from 0 (least significant bit) to 7 (most significant bit).

A.1 Encoding Rules

Field Elements. Elements of the finite field $\mathbb{F}_q$ (with $q = 2^{255} - 18651$ for jq255e, $2^{255} - 3957$ for jq255s) are encoded into exactly 32 bytes:

- A field element $x \in \mathbb{F}_q$ is considered as an integer in the 0 to $q - 1$ range, which is then encoded using the unsigned little-endian convention (least significant byte comes first).
- Encoding always uses 32 bytes, even if the integer could have fit into fewer bytes.
- Decoders MUST reject the source bytes, and return INVALID, if any of the following conditions hold:
  - The source length is not exactly 32 bytes.
  - The integer resulting from unsigned little-endian interpretation of the source bytes is not in the 0 to $q - 1$ range.

  In particular, there is no ignored bit: even though the value of q implies that the most significant bit of the last byte is always zero, that bit MUST NOT be ignored by decoders. Moreover, checking that the most significant bit of the last byte is zero is not sufficient validation; for instance, an encoding of the integer q itself, over 32 bytes, MUST be rejected as INVALID, since it is not in the 0 to $q - 1$ range. There is no implicit reduction modulo q.

Scalars. Scalars are integers modulo r, with r being the order of the group. For jq255e and jq255s, r is a prime integer close to 2^{254} (the order of the underlying double-odd curve is $2r$). Scalars are encoded with rules similar to those for field elements:

- A scalar is encoded as an integer in the 0 to $r-1$ range over exactly 32 bytes with unsigned little-endian convention.
- Decoders MUST verify that the received value is in the 0 to $r-1$ range. There is no ignored bit in the last byte. There is no implicit reduction modulo r.

Group Elements. A group element (often called "point", though a group element is, strictly speaking, a set of two curve points, which are called the *representants* of the group element) is encoded by using the process described in section 4, and made explicit in sections 6.1 and 6.2:

- The sign convention (sign function) uses the least significant bit of the representation of the input field element as an integer in the 0 to $q-1$ range; equivalently, the sign is the value of the bit number 0 in the first byte of the encoding of the element as specified previously. Thus, 1 is negative, but -1 (represented as $q-1$) is not. 0 is non-negative.
- Upon encoding, the group element representant whose e coordinate is non-negative is chosen; its u coordinate is encoded into bytes using the rules for field elements.
- Upon decoding, the input bytes are decoded into a field element u, from which the corresponding coordinate e is recomputed. If the input is valid, then there are two choices for the value of e, exactly one of which is non-negative; the non-negative value is chosen.

The neutral element of the group is a valid group element; its u coordinate is zero (for both representants), and therefore it is encoded as the field element 0, which leads to exactly 32 bytes of value `0x00`.

Private Keys. A private key is a non-zero (secret) scalar. A private key is encoded with the same rules as scalars, with one additional rule:

- When decoding a private key from bytes, decoders MUST reject the scalar if it is a valid encoding of the value zero.

A scalar of value zero corresponds to 32 bytes of value `0x00`. Private keys are never equal to zero.

Public Keys. A public key is a non-neutral group element. If the private key is the scalar d, then the public key is the group element dG, where G is the conventional base element in the group (G has order exactly r). Public keys are decoded and encoded using the rules for group elements, with one additional rule:

- When decoding a public key from bytes, decoders MUST reject the group element if it is a valid encoding of the neutral element.

The neutral group element encodes as 32 bytes of value `0x00`. Public keys are never equal to the neutral group element.

A.2 Signature Specification

A.2.1 Pre-Hashing and Hash Function Choice

Apart from the used group (which is jq255e or jq255s in this specification), the signature scheme is configured with a hash function. There are three places where the hash function is used:

- to process the message data (in "pre-hashed" mode);
- to compute the challenge c;
- to generate the per-signature secret scalar k.

In this specification, the BLAKE2s[32] hash function is used for the second and third functionalities. If the message data is pre-hashed, then it is recommended to use BLAKE2s for this step as well.

BLAKE2s can optionally be configured (through its "parameter block") with a specific output size, a secret key, a salt, a personalization string, and a tree mode. We use none of these features. Whenever BLAKE2s is used in this specification, it is run with its default parameters for hashing. In particular, the hash output size is 32 bytes (256 bits); this applies even if we later on truncate that output to its first 128 bits.

Pre-hashing the input message means that the signature is computed and verified not on the message m itself, but on $H'(m)$ for some hash function H'. Pre-hashing facilitates streamed processing. If the data is not pre-hashed (called here the "raw mode"), then the signer must already know the signing key pair when starting to process the message data, and verifiers must known the public key and the signature value when starting to process the message data. In a situation involving a long message, signed by the sender and verified by the recipient, then use of the raw mode implies that either the sender or the verifier must be able to buffer the complete message. On the other hand, raw mode is not impacted by collision attacks in H', since in that case there is no H'. This was the reason raw mode was preferred in the original EdDSA[4]; in the RFC that now specifies Ed25519[21], raw mode is called "PureEdDSA".

Given an input message m (a sequence of bytes), the *prepared message* is the byte sequence M defined as follows:

- In raw mode (no prehashing), M is a single byte of value `0x52`, followed by m itself.
- In pre-hashed mode, M is the concatenation of, in that order:
  - a single byte of value `0x48`;
  - the hash function symbolic name, in ASCII;
  - a single byte of value `0x00` (which marks the end of the hash function name);
  - the hashed message $H'(m)$.

  Hash function names use lowercase ASCII letters and digits; other punctuation signs are removed. Table 2 lists the currently defined names.

A.2.2 Signature Generation

Let d be the signer's private key (a non-zero scalar), and $Q = dG$ the corresponding public key (with G being the conventional base point in the relevant group). The signature generation process on an input message m goes as follows:

Hash function	Symbolic name	Reference
SHA-256	`sha256`	[24]
SHA-384	`sha384`	[24]
SHA-512	`sha512`	[24]
SHA-512/256	`sha512256`	[24]
SHA3-256	`sha3256`	[25]
SHA3-384	`sha3384`	[25]
SHA3-512	`sha3512`	[25]
BLAKE2s	`blake2s`	[32]
BLAKE2b	`blake2b`	[32]
BLAKE3	`blake3`	[26]

Table 2: Defined hash function names (for use with signatures in pre-hashed mode).

1. Process the input message m into the prepared message $\mathcal{M}$, as specified in section A.2.1.
2. Generate a per-signature secret scalar k. The signer is free to use any method which ensures that each value k is used with a single message only (for a given signing key), is secret, and is chosen with a distribution indistinguishable from uniform probability among scalars. A specific generation process is described later on (section A.2.4), and is recommended for proper security in all implementation contexts.
3. Compute the point $R = kG$.
4. Apply the BLAKE2s hash function on the concatenation of, in that order:
   - the point R (encoded into 32 bytes);
   - the signer's public key Q (encoded into 32 bytes);
   - the prepared message $\mathcal{M}$.

   The challenge c is then defined to be the first 16 bytes of the 32-byte output of BLAKE2s.
5. Compute the scalar $s = k + cd$ (interpreting c as an integer in the 0 to $2^{128} - 1$ range, using the unsigned little-endian convention).

The signature is then the concatenation of, in that order:

- the challenge c (16 bytes);
- the scalar s (encoded into 32 bytes).

The total signature length is exactly 48 bytes.

It is theoretically possible (though very improbable) that $k = 0$; in such a case, R is the group neutral. This is considered valid. Similarly, it is acceptable that $s = 0$ or that c is a sequence of 16 bytes of value `0x00`. The probability that any of these cases occurs with a proper implementation of the signature generation is so vanishingly small that it will not happen in practice[5].

A.2.3 Signature Verification

Signature verification uses as inputs a message m, a public key Q, and a signature S, and outputs a Boolean result (TRUE if the signature is valid for the message m relatively to the public key Q, FALSE otherwise). The process is the following:

[5] Or, more accurately, when it happens, it is almost always due to some other hardware failure.

1. Process the input message m into the prepared message M, as specified in section A.2.1.
2. Verify that the signature S has length exactly 48 bytes; otherwise, return FALSE.
3. Split S into the challenge c (first 16 bytes) and the encoded scalar s_b (last 16 bytes), then decode s_b into the scalar s. If the decoding process of s_b returns INVALID, then return FALSE.
4. Compute the point $R = sG - cQ$ (interpreting c as an integer in the 0 to $2^{128} - 1$ range using the unsigned little-endian convention).
5. Using M, Q and the just computed point R, recompute the challenge value c' with the same process as in step 4 of the signature generation process.
6. If $c = c'$, then return TRUE; otherwise, return FALSE.

As previously pointed out, it is acceptable that s or c be zero, or that R be the neutral point. The comparison between c and c' is strict bytewise equality.

A.2.4 Generation of the Per-Signature Scalar *k*

How the signer generates the secret scalar k is invisible to verifiers, and any method may be used as long as it fulfills the proper security expectations of unpredictable uniform randomness. However, it is RECOMMENDED to use the process described here, because it ensures that an adequately safe value is obtained, even if the signer's hardware system does not have a readily available cryptographically strong source of randomness. Moreover, using *exactly* that process makes the implementation more easily testable against known test vectors.

The scalar k is generated over the following inputs:

- the signer's private key d;
- the signer's public key Q;
- the prepared message M;
- an additional seed z, which is a sequence of bytes of (almost) arbitrary length and contents.

The scalar k is then computed as follows:

1. The BLAKE2s hash function is evaluated over the concatenation of, in that order:
   - the signer's private key d (encoded over 32 bytes);
   - the signer's public key Q (encoded over 32 bytes);
   - the length (in bytes) of z, expressed over 8 bytes using the unsigned little-endian convention;
   - the value z itself;
   - the prepared message M.
2. The BLAKE2s output (32 bytes) is then interpreted as an integer in the 0 to $2^{256} - 1$ range, using the unsigned little-endian convention. This integer is then *reduced* modulo r to yield the scalar k.

Note that the BLAKE2s output is reduced modulo r; this is *not* the usual decoding of a scalar from bytes, since it does not reject out-of-range values.

The use of a hash function over the private key and source message is known as *derandomization*. Ed25519[4,21] uses it. It has also been specified in general for DSA and ECDSA

with arbitrary curves[28]. When z has a fixed value (e.g. the empty string), the process is deterministic. It is safe, though it has been noted that fully deterministic signature generation makes the implementation somewhat fragile with regard to some local side-channel attacks, in particular fault attacks[1,27]. To strengthen the process even in that case, a randomly generated seed z can be used; there are no specific requirements on that randomness, and even a simple monotonic counter could be used. Even if the source of randomness for z is so flawed that it is stuck and always returns the same value, then the derandomization process described here still makes the signatures safe.

The process uses both the signer's private key d and the public key Q, even though the latter can be deterministically generated from the former (with $Q = dG$), because that protects against a possible misuse of the API of some possible implementations of the libraries. This was recently reported for Ed25519[10]. The gist of the issue is that since signing generation requires knowledge of the encoded public key (to compute the challenge), and regenerating the public key from the private key is about as expensive as producing the signature itself, some libraries accept that the caller provides the private and public keys as two separate parameters, under the assumption that they will be stored together. Then, some applications would allow attackers to reference a private key and provide a distinct public key separately, and that leads to key reconstruction attacks. Any signature library can be designed to enforce regeneration of the public key from the private key, but that implies that loading the private key incurs the computational cost of recomputing the public key. Using the public key as part of the derandomization process that generates k is a much cheaper way to mitigate this specific attack.

Warning: In general, producing a modular integer by reducing a pseudorandom sequence of bytes of the same length as the modulus yields a biased distribution, which, in the context of Schnorr signatures, would leak information about the private key and eventually allow adverserial reconstruction of that key. This issue is avoided in jq255e and jq255s by the fact that for these groups, the modulus r of the ring of scalars is close enough to a power of two that such biases are negligible (for both groups, $|r - 2^{254}| \leq 2^{127}$). If this specification is ever adapted to other groups, then this part of the generation of the per-signature scalar MUST be reviewed carefully and possibly adjusted.

A.3 ECDH Key Exchange Specification

The ECDH (elliptic curve Diffie-Hellman) protocol is a key exchange mechanism. Since it uses only one message per participant, it can also be used for asymmetric encryption, if combined with a symmetric encryption scheme.

Warning: In ECDH, both parties have private/public key pairs. These key pairs are syntactically identical to the key pairs used in Schnorr signatures, making it possible to use the same key pair for signatures and for key exchange and asymmetric encryption tasks. In general, this is NOT RECOMMENDED. The main flaw with such designs is that private keys for encryption and for signatures normally have incompatible lifecycle requirements: private keys for encryption should have backups, since loss of the private key implies loss of the data that was encrypted against the corresponding public key; but private keys for signature should not have backups, since their value lies in their exclusive control by the signer, and loss of the private signing key does not invalidate previously issued signatures. On a different level, possible interactions between a signing system and a key exchange system working over the same pri-

vate key are not well studied, and some unwanted interaction might allow for faster attacks, e.g. using the ECDH engine as a helper for making signature forgeries, or vice versa.

In ECDH, each party runs the following process:

1. Generate a new key pair (d_1, Q_1). The private key d_1 should be selected as a random non-zero scalar; zero is not an acceptable private key. The public key is computed as $Q_1 = d_1 G$.
2. Encode Q_1 into the 32-byte sequence p_1, and send p_1 to the peer.
3. Receive the 32-byte sequence p_2 from the peer, and decoded it as the public key Q_2. This decoding process may fail (i.e. return INVALID) if the length of p_2 is not exactly 32 bytes, or is not a valid encoding of a group element, or is the valid encoding of the neutral element (the neutral is a valid point but not a valid public key); in such a case, set Q_2 to any other point (e.g. the base point G) but remember that the decoding failed. The Boolean flag OK is set to TRUE if the decoding succeeded, FALSE otherwise.
4. Compute the point $k_1 Q_2$ and encode it into the 32-byte sequence h.
5. Derive h into a 32-byte symmetric key appropriate for further cryptographic operations by computing the BLAKE2s hash function over the concatenation of, in that order:
   - If p_1 is lexicographically lower than p_2, then p_1 followed by p_2; otherwise, p_2 followed by p_1.
   - If OK is TRUE, then a single byte of value `0x53` followed by h; otherwise, a single byte of value `0x46` followed by d_1 (encoded into 32 bytes).
6. Return the BLAKE2s output as the established symmetric key, along with the value of the OK flag.

The use of the OK flag is meant to allow implementations to hide from outsiders observing side channels (such as an error status or computation time) whether the process worked; this can be an interesting feature in some protocols where the involved points are hidden from the attacker, but potentially alterable. If the decoding failed, then the resulting key (BLAKE2s output) is unpredictable by attackers, but still deterministic from the inputs (sending the same p_2 to a recipient reusing the same p_1 will result in the same BLAKE2s output, regardless of whether p_2 is a valid point encoding or not). Hiding the success status of the operation is rarely needed and most implementations are expected to treat the p_1 and p_2 values as public data (though of course d_1, d_2 and h are secret).

If the p_1 and p_2 messages were generated correctly and not altered in transit, then both parties compute the same h (since $k_1 Q_2 = k_2 Q_1$) and thus the same BLAKE2s output. The values p_1 and p_2 are used as extra inputs to BLAKE2s so that the resulting key is bound to the exchanged messages, which can be convenient for proving the security of a protocol that leverages ECDH. The lexicographic ordering of p_1 and p_2 is equivalent to the numerical ordering of integers obtained by interpreting p_1 and p_2 with the unsigned *big-endian* convention (and not the little-endian convention we used everywhere else in this document); however, it is expected that implementations will compare the byte values directly, e.g. with the C function `memcmp()`, if the exchanged messages p_1 and p_2 are public data.

A.4 Hash-to-Group Specification

In section 6.9, maps from field elements to points were defined, for both jq255e and jq255s. We use such maps to specify hash-to-group operations that take as input an arbitrary sequence of bytes, and produce points in the group. The hashing process is one-way and produces an

output distribution indistinguishable from uniformly random selection. The general technique (due to Brier *et al*[9]) is to use a hash function to derive two field elements from the input, then map each field element to a point, and finally add the two points together.

Given an input message m (a sequence of bytes), we first compute the *prepared message* M in exactly the same way as in section A.2.1: M is either a single byte of value `0x52` followed by the raw data, or a single byte of value `0x48` followed by the pre-hash function name (terminated by a zero) and then $h(m)$, for some hash function h. Whether data should be pre-hashed is an application choice and depends on the usage context; as will be seen below, M will be processed twice with BLAKE2s, so that pre-hashing improves performance if m is large. If pre-hashing, then the used hash function h MUST be an appropriately secure hash function; by default, BLAKE2s is recommended.

Once M is obtained, the following steps are applied:

1. Compute BLAKE2s over the concatenation of a single byte of value `0x01` and the prepared message M (in that order); the 32-byte output of BLAKE2s is then interpreted as an integer (using the unsigned little-endian convention) which is then *reduced* modulo q to yield the field element f_1.
2. Map f_1 to the point P_1 using the appropriate map (algorithm 10 for jq255e, algorithm 11 for jq255s).
3. Compute BLAKE2s over the concatenation of a single byte of value `0x02` and the prepared message M (in that order); the 32-byte output of BLAKE2s is then interpreted as an integer (using the unsigned little-endian convention) which is then *reduced* modulo q to yield the field element f_2.
4. Map f_2 to the point P_2 using the appropriate map (algorithm 10 for jq255e, algorithm 11 for jq255s).
5. Compute $P = P_1 + P_2$ as the output of the hash-to-group process.

It shall be noted that the BLAKE2s output is turned into a field element through a modular reduction; this is distinct from *decoding* a field element, in that the latter would reject out-of-range inputs. While, in general, reducing a binary input modulo a prime induces some selection biases, this is not a problem for the fields used in jq255e and jq255s because their respective orders (denoted q) are very close to a power of 2, so that the resulting biases are negligible.