Efficient and Complete Formulas for Binary Curves

Thomas Pornin

NCC Group, `thomas.pornin@nccgroup.com`

5 October, 2022

Abstract. *Binary elliptic curves* are elliptic curves defined over finite fields of characteristic 2. On software platforms that offer carryless multiplication opcodes (e.g. `pclmul` on x86), they have very good performance. However, they suffer from some drawbacks, in particular that non-supersingular binary curves have an even order, and that most known formulas for point operations have exceptional cases that are detrimental to safe implementation.

In this paper, we show how to make a prime order group abstraction out of standard binary curves. We describe a new canonical compression scheme that yields a canonical and compact encoding. We also describe complete formulas for operations on the group. The formulas have no exceptional case, and are furthermore faster than previously known complete and incomplete formulas (general point addition in cost 8M + 2S + 2$\mathrm{m_b}$ on all curves, 7M + 2S + 2$\mathrm{m_b}$ on half of the curves). We also show how the same formulas can be applied to computations on the entire original curve, if full backward compatibility with standard curves is needed. Finally, we implemented our method over the standard NIST curves B-233 and K-233. Our strictly constant-time code achieves generic point multiplication by a scalar on curve K-233 in as little as 29600 clock cycles on an Intel x86 CPU (Coffee Lake core).

1 Introduction

Elliptic curves used in cryptography are defined over finite fields. When the base field is of characteristic 2, i.e. the field is $GF(2^m)$ for some degree m, then the curve is called a *binary elliptic curve*. When elliptic curves were selected and chosen as standards by SEC[6], and adopted by NIST[17], a number of binary curves were defined.

In practice, binary curves are not widely used, for a number of reasons, not all of them being fully rational:

- Binary curves, at least when non-supersingular, cannot have a prime order, since they necessarily contain a point of order 2. This implies potential *cofactor issues*; checking that a given point is in a specific prime order subgroup can be done efficiently, but this is not specified by relevant standards.
- Performance of binary curves can be poor on systems that do not offer a “carryless multiplication” opcode (i.e. support for multiplication of polynomials over $GF(2)[z]$). When such an opcode is available, operations on binary curves are very fast, but much less so on other systems. Small embedded software platforms (microcontrollers), in particular, find such curves to be quite expensive to use.
- Most known formulas for adding points together are incomplete, i.e. they have exceptional cases that must be handled specially. Such handling is detrimental to either security

(if done with conditional execution, then that leads to side-channel leaks) or performance (if all cases are computed, and the correct one selected in a side-channel-free way). There are known complete formulas for curves when converted into binary Edwards curves[3], but at a steep cost ($18M + 2S + 7m_b$ in all generality).
– Known algorithms for solving discrete logarithm in the multiplicative subgroup of a binary field are more efficient than their counterparts in fields of large characteristic, especially when the field degree is composite. Thus, pairing-friendly binary curves must use a very large base field and even larger extension field to hope to achieve some security, making them impractical for pairing-based cryptography.
– There are some known results that indicate that discrete logarithm on binary curves is "less secure" asymptotically than what we would expect from generic discrete logarithm attacks[20]. The effect would impact only very large degrees ($m \geq 2000$), making the issue purely theoretical, but it highlights that binary curves do not necessarily follow the same patterns as curves over large characteristic fields, and require some specific analysis. More generally, binary fields are just weird, and that generates some unease among cryptographers, who tend to feel that they have a better understanding of what happens with prime fields (whether that feeling is actually correct or not is another question).
– Many implementation techniques pertaining to implementation of binary curves, especially in custom hardware and using normal bases, have been patented, generating a chilling effect on open-source implementations and work on binary curves in general. Note that many such patents were filed in the 1990s and have since expired.

The potential security issues remain, indeed, potential; no actual reduction in security was ever found for binary curves in general with sizes used in practical deployments, and in particular for NIST curves. Some newer curves have also been proposed, notably binary GLS curves[10], that feature an easily computed endomorphism that can be used to speed up some operations, in particular multiplication of a point by a scalar. One specific instance of GLS curves is GLS254[19]; a recent work[1] offers a very optimized implementation of multiplication of a point by a scalar, completed in 35739 cycles on an Intel x86 "Kaby Lake" core. A substantial part of this last article is devoted to the analysis of exceptional cases, working out where they may happen in the computation, and adding corresponding workarounds; this highlights the complexities that arise from relying on incomplete formulas.

In this article, we try to alleviate some of these issues. Namely, in the following sections, we will show the following:

– We show how to leverage the (already known) tests for membership in the prime-order subgroup of interest in order to obtain a prime order group abstraction. This is applicable to all non-supersingular binary elliptic curves.
– We define a new coordinate system (called (x, s) coordinates) that allows representing all members of the group, including the neutral element. On this system, we show complete formulas. The formulas are also quite efficient, with general point addition in cost $8M + 2S + 2m_b$ for all curves ($7M + 2S + 2m_b$ for half of all curves) and doubling in cost $3M + 4S + 4m_b$ (or $3M + 5S + 2m_b$).
– We describe a new encoding/compression scheme with which a group element can be encoded into a fixed-length sequence of bytes. The encoding is canonical (a given element can be encoded only in a unique way) and verified (the decoding process inher-

ently checks that the source indeed used that exact encoding). The encoding works over all group elements, including the neutral.

- We also show how the new formulas can be used to perform computations over the full curve. Normally, cryptographic protocols should be built over the prime order subgroup only; however, in some scenarios, backward compatibility requirements mandate support of operations on arbitrary curve points. In this way, we can obtain the same efficiency and completeness on the full curve.
- We implemented our formulas and techniques in C, for standard NIST curves B-233 and K-233. On the latter, we achieve a point multiplication by a scalar in as little as 29602 cycles on an Intel x86 "Coffee Lake" system.[1]

A Note on Patents. Binary elliptic curves have the reputation of being a "patent minefield". It is true that many patents were filed about various aspects of their implementation, especially in the 1990s. Most of these patents have expired by now. To our knowledge, none of the techniques described in this paper are currently covered by a patent. This is not legal advice; we are not entitled to provide such advice. Moreover, the state of patent law is such that in many jurisdictions, it is not even possible to ever offer a guarantee of applicability or non-applicability of any patent over a given implementation. We can *informally* say that after some cursory scans, we found only patent WO2001035573A1, which covers point halving and was filed in 2000; however, `patents.google.com` shows it as being expired, discontinued, abandoned or withdrawn, depending on the involved jurisdiction.

We also claim that we have not and do not intend to file any patent on these techniques.

2 Background on Binary Fields and Curves

In this section, we recall some core definitions and properties of binary fields and binary elliptic curves. All these results are well-known and can be found in textbooks[7,11,23,25].

2.1 Binary Fields

Field Construction. A binary field is a finite field of characteristic 2. Its order is equal to 2^m for some integer $m \geq 1$. We denote it $GF(2^m)$. That field can be instantiated with binary polynomials. Let $GF(2)[z]$ be the ring of polynomials in the variable z, with coefficients in $GF(2)$ (in all this text, z will designate the formal variable for binary polynomials). Let $M \in GF(2)[z]$ be an irreducible polynomial of degree m. The quotient ring $GF(2)[z]/M$ is then a finite field with 2^m elements.

It can be shown that two finite fields with the same cardinal are isomorphic to each other, and the isomorphisms can be efficiently computed. Therefore, it does not matter, for security, which modulus polynomial M we choose, as long as it is irreducible; we can thus use a modulus that promotes implementation efficiency. In practice, for slightly more than half of the

[1]Though nominally distinct, the Skylake, Kaby Lake, Cannon Lake and Coffee Lake cores "are based on the same design" and "differ mainly in processing technology, number of cores, and cache sizes."[8]. These parameters should have about no influence on the measured performance for tight cryptographic implementations that entirely fit in L1 cache; thus, benchmarks for these four core types should be directly comparable.

possible degrees m, there exists an irreducible trinomial $M = z^m + z^u + 1$ for some $1 < u < m/2$; in case there are several such trinomials, we normally use the one with the smallest degree u for the intermediate element.[2] When m is such that no irreducible trinomial exists, then there are irreducible pentanomials (with three non-zero "intermediate" coefficients) and we use the smallest one in lexicographic order.

When $m = m_1 m_2$ is composite, then $GF(2^m)$ can be mapped to a tower of field extensions, i.e. as an extention of degree m_2 over the field $GF(m_1)$ (or vice versa). When a composite degree is used, the GHS attack may apply[9]; thus, any use of a non-prime degree m requires extra analysis, and can be deemed as slightly "riskier", in a fuzzy way. One case is the use of $m = 254 = 2 \times 127$, in particular for the GLS254 curve; [10] offers extensive arguments on why that specific choice is still safe. More conservative standards such as NIST FIPS 186-4[17] stick to prime degrees m.

Representation and Operations. The most commonly used representation of elements in $GF(2^m)$ is a polynomial basis: the coefficients of the polynomial in $GF(2)[z]$ are listed in increasing sequence order, as so many bits. Other representations are possible (e.g. normal bases) but they usually don't yield improvements in software implementations. With a polynomial basis, addition in $GF(2^m)$ is a simple bitwise XOR, with mostly negligible cost. Subtraction is the same operation as addition.

Multiplication in $GF(2^m)$ is fast on platforms that offer a carryless multiplication opcode (e.g. `pclmul` on x86 CPUs); the product is evaluated over polynomials, then reduced modulo M. The reduction itself is relatively inexpensive, if M was chosen as a sparse polynomial, with its non-zero coefficients only in the low degrees. On software platforms without a carryless multiplication opcode, multiplication in $GF(2^m)$ is expensive. Karatsuba's method can be applied repeatedly to reduce the operation to a sequence of individual multiplications over smaller polynomials[13]. A bit-by-bit algorithm can be improved with table lookups[4], but that tends to lead to side-channel leaks when operating over secret values. SIMD vector instructions can help[2]. Integer multiplications can also be used, provided that they are used on values with enough zeros inserted between data bits to allow carries to accumulate without spilling over other bits.

Squaring in $GF(2^m)$ is a field automorphism: for any elements x and y, $(xy)^2 = x^2y^2$ and $(x + y)^2 = x^2 + y^2$. This makes squaring a linear operation (when considering $GF(2^m)$ as a vector space of dimension m over $GF(2)$). In a square-and-reduce algorithm, the polynomial squaring simply becomes an "expansion" in which a bit of value zero is inserted between any two consecutive data bits; even without access to a carryless multiplication opcode, this can be done with a few mask-and-shift operations. On platform with a carryless multiplication opcode, squaring cost is typically between 0.5 and 0.75 times that of multiplication cost; on other platforms, in particular small embedded microcontrollers, the ratio is often much lower, with 0.1 being the traditional value retained for analysis.

Every element in $GF(2^m)$ is a quadratic residue, and has a single square root. Extracting a square root is also a field automorphism ($\sqrt{x + y} = \sqrt{x} + \sqrt{y}$). Computation of $\sqrt{x}$ can be

[2]Not always! E.g. when working in $GF(2^{127})$, using $z^{127} + z^{63} + 1$ may yield slightly better performance than $z^{127} + z + 1$, by computing modulo $z^{128} + z^{64} + z$, with only one element which is not 64-bit aligned.

done easily by splitting x into its even-degree and odd-degree coefficients:

$$\begin{aligned} x &= \left(\sum_{i=0}^{(m-1)/2} x_{2i} z^{2i}\right) + z\left(\sum_{j=0}^{(m-3)/2} x_{2j+1} z^{2j}\right) \\ &= x_{\text{even}} + z x_{\text{odd}} \end{aligned}$$

Then $\sqrt{x} = \sqrt{x_{\text{even}}} + \sqrt{z}\sqrt{x_{\text{odd}}}$. The square roots of x_{even} and x_{odd} can be done with "squeezing" (the reverse of the expansion done in squaring), while the constant $\sqrt{z}$ is usually of low Hamming weight, if M was itself chosen as a sparse polynomial. On small software platforms, square root extraction is about as cheap as squaring; on large systems with an efficient carryless multiplication opcode, a square root cost is typically similar or slightly above that of a multiplication.

Inversion is more expensive. The main strategy for implementing inversions in $GF(2^m)$ is to use a variant of Fermat's little theorem described by Itoh and Tsujii[12]: $1/x$ is computed as:

$$\frac{1}{x} = \left(x^{2^m-2}\right) = \left(x^{2^{m-1}-1}\right)^2$$

with the exponent $2^{m-1}-1$ having a regular format (only ones, no zeros) that is amenable to efficient addition chains, so that the exponentiation can be performed with a limited number of field multiplications (roughly proportional to $\log m$), and sequences of successive squarings. Since squaring is a linear operation, each such sequence can be performed with a precomputed matrix multiplication.[3] Another method for inversion is to use a binary GCD variant[5], which is easier to implement and more efficient than the binary GCD on integers due to the absence of carry propagation when working with polynomials in $GF(2)[z]$. On platforms with a carryless multiplication opcode, the Itoh-Tsujii method tends to be faster, with a cost typically between 40 and 100 times that of a multiplication; on smaller systems, the binary GCD may be preferred, especially since it avoids the use of large precomputed matrices.

Trace and Quadratic Equations. The *trace* of a field element x, denoted $\text{Tr}(x)$, is defined as:

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

The trace has some important characteristics:

- The trace of any x is always equal to 0 or 1. Exactly 2^{m-1} elements of $GF(2^m)$ have trace 0, and the 2^{m-1} other elements of $GF(2^m)$ have trace 1.
- The trace is linear: for any x and y, $\text{Tr}(x+y) = \text{Tr}(x) + \text{Tr}(y)$.
- For any x, $\text{Tr}(x^2) = \text{Tr}(x) = (\text{Tr}(x))^2$.

If the degree m is odd, then $\text{Tr}(1) = 1$. If m is even, then $\text{Tr}(1) = 0$; however, there is always at least one degree i such that $\text{Tr}(z^i) = 1$, so that one can always find an element of minimal Hamming weight whose trace is 1.

[3]Itoh and Tsujii initially described their method using a normal basis representation, where squarings and sequences of squarings are basically free. However, even with polynomial bases, the algorithm remains efficient.

As a linear operation that outputs a single bit, the trace of x can be computed as a XOR of some of the bits of x. When the field modulus M is sparse, very few bits of x actually contribute to the trace. For instance, with $m = 233$ and $M = z^{233} + z^{74} + 1$ (the field and modulus used in the standard NIST curves B-233 and K-233), $\mathrm{Tr}(x) = x_0 + x_{159}$, i.e. the sum of only two of the value bits. In general, the cost of computing the trace is negligible.

Quadratic equations in $GF(2^m)$ can be solved by reducing the equation to $x^2 + x = d$ for some field element d and unknown x. Note that $\mathrm{Tr}(x^2 + x) = \mathrm{Tr}(x)^2 + \mathrm{Tr}(x) = 0$; therefore, such an equation can be solved only if $\mathrm{Tr}(d) = 0$. The converse is also true: solutions exist for any d of trace zero. When x is a solution, the other solution is $x + 1$.

When m is odd, a solution is given by the halftrace:

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

Since $x^{2^m} = x$ for all $x \in GF(2^m)$, it is easily seen that $H(d)^2 + H(d) = d + \mathrm{Tr}(d)$. If m is even (e.g. when working in $GF(2^{254})$), the expression of the solution of the quadratic equation is slightly more complicated, but it boils down to the same result: a solution x can be found from d with a linear operation, which can thus be implemented with a multiplication by a constant matrix. Even with a constant-time implementation that scans the whole matrix, this is a relatively inexpensive operation (less than 20 times the cost of a multiplication).

We define as $\mathrm{QSolve}(d)$ the linear operation that returns a solution x to the equation $x^2 + x = d + \mathrm{Tr}(d)$ for any d. The other solution is $x + 1 = \mathrm{QSolve}(d) + 1$.

2.2 Binary Elliptic Curves

Curve Equation. Any ordinary (non-supersingular) binary elliptic curve equation can be transformed, through changes of variables, into a short Weierstraß equation:

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

for two constants A and B in $GF(2^m)$, with $B \neq 0$. The curve is the set of points (x, y) that fulfill the equation, together with a formal "point-at-infinity" (hereafter denoted $\mathbb{O}$) which does not have defined x and y coordinates. The curve is well-defined for any choice of constants A and B, as long as $B \neq 0$.

Curve isomorphisms are mappings $(x, y) \mapsto (x, y + C)$ for any constant $C \in GF(2^m)$; this modifies the constant A into $A + C + C^2$, while leaving B unchanged. This means that through such an isomorphism, it is always possible to change A into any other constant A' as long as $\mathrm{Tr}(A) = \mathrm{Tr}(A')$ (it suffices to choose $C = \mathrm{QSolve}(A + A')$). In particular, when a curve uses A such that $\mathrm{Tr}(A) = 0$, then an isomorphic curve works with $A = 0$, which tends to make computations easier and faster. Similarly, when $\mathrm{Tr}(A) = 1$, we can arrange for A to be a value with minimal Hamming weight; if m is odd, then we can make $A = 1$, which again promotes implementation efficiency. In all of this article, we assume that multiplication by A or A^2 has negligible cost.

Point Addition Law. An addition law, that grants the curve an Abelian group structure, is defined as follows:

- For any point P, $P + \mathbb{O} = \mathbb{O} + P = P$.
- For $P = (x, y)$, its opposite is $-P = (x, y + x)$. For any given $x \in GF(2^m)$, there are at most two points on the curve with that x coordinate, and the two points are opposite of each other.
- The double of a point $P = (x, y)$ is $2P = (x', y')$ with:

$$\begin{aligned} \lambda &= x + y/x \\ x' &= \lambda^2 + \lambda + A \\ y' &= \lambda(x + x') + x' + y \end{aligned}$$

  Note that if $x = 0$, then P is a point of order two, and $2P = \mathbb{O}$.
- If $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, with $x_1 \neq x_2$ (equivalently, $P_1 \neq \pm P_2$), then their sum is $P_1 + P_2 = (x_3, y_3)$ with:

$$\begin{aligned} \lambda &= (y_1 + y_2)/(x_1 + x_2) \\ x_3 &= \lambda^2 + \lambda + A + x_1 + x_2 \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \end{aligned}$$

A binary elliptic curve has a single point of order two, which is $N = (0, \sqrt{B})$. This is the only point with $x = 0$.

Group Structure. Hasse's theorem on elliptic curves over finite fields also applies to binary elliptic curves, so the order of a binary elliptic curve over $GF(2^m)$ is n such that $|2^m + 1 - n| \leq 2^{1+m/2}$. The order can always be split into an even and odd parts: $n = 2^t r$, for some integer $t \geq 1$, and odd integer r. Standard curves are normally chosen so that t is small and r is prime; however, all the treatment in this article also applies to large t values and non-prime r values. When r is prime, 2^t is called the *cofactor*. The main goal of using such a binary elliptic curve $\mathcal{E}$ in cryptographic protocols is to compute operations in the subgroup of points of r-torsion, which is then a cyclic group with a prime order:

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

Any point $P \in \mathcal{E}$ can be split into the sum of a point of r-torsion and a point of 2^t-torsion. Moreover, since there is a single point of order 2, it follows that there must be 2^{t-1} points of order exactly 2^t: the subgroup of points of 2^t-torsion is cyclic. If we call T one such point of order exactly 2^t, then any point P on the curve splits as $P = Q + kT$ for some $Q \in \mathcal{E}[r]$, and integer k taken modulo 2^t. This decomposition is unique; we can generically obtain Q as:

$$Q = 2^t((1/2^t \bmod r)P)$$

We will see in section 4.2 a more efficient method to obtain Q and k. Note that $N = 2^{t-1}T$.

Point Halving. Point halving consists in computing a point P from a point P' such that $P' = 2P$. It works as follows:

1. Find λ such that $x' = \lambda^2 + \lambda + A$. If $\mathrm{Tr}(x' + A) = 1$ then there is no solution (the point P' is not the double of any point over the curve). Otherwise, set $\lambda = \mathrm{QSolve}(x' + A)$.

2. Since $y' = (\lambda+1)x' + \lambda x + y = (\lambda+1)x' + x^2$, we can compute $x = \sqrt{y' + (\lambda+1)x'}$.
3. From x and λ we get $y = x(x+\lambda)$.

When P exists, then there are two solutions, which are P and $P+N$.

The first step highlights the important property that a point P' can be halved (i.e. is the double of another point) if and only if $\mathrm{Tr}(x' + A) = 0$. When applied to the point $N = (0, \sqrt{B})$, we find the following:

– When $\mathrm{Tr}(A) = 1$, N cannot be halved; the curve $\mathcal{E}$ has order $2r$ for some odd integer r (i.e. the curve is *double-odd*). Points that can be halved are then exactly the points of r-torsion. This implies that $P = (x, y)$ is a part of the subgroup of order r if and only if $\mathrm{Tr}(x) = 1$.
– When $\mathrm{Tr}(A) = 0$, N can be halved; the curve $\mathcal{E}$ has order $2^t r$ for some $t \geq 2$ and odd integer r. The cofactor is at least 4. If $A = 0$ (as we saw, this can always be arranged with an isomorphism when $\mathrm{Tr}(A) = 0$), then the two points $R = (\sqrt[4]{B}, \sqrt{B})$ and $R + N = -R = (\sqrt[4]{B}, \sqrt{B} + \sqrt[4]{B})$ are the two points such that $2R = N$.

3 Prime Order Group Definition

The point addition law has exceptional cases, related to the point-at-infinity (which does not have defined coordinates), and when adding a point to itself. To work around these issues, and obtain complete formulas, we apply the folllowing methodology, that leverages the same core ideas as double-odd curves[22]:

– We restrict ourselves to points of r-torsion, which we *represent* by other points: each group element will be a point $P + N$, where P is a point of r-torsion. The sum in the group of $P_1 + N$ and $P_2 + N$ is then defined as $(P_1 + P_2) + N$. This representation makes N the neutral element in the group, hence with defined coordinates.
– When adding $P_1 + N$ to $P_2 + N$ in the group, we have to compute (on the curve) the sum $P_1 + P_2 + N$, which we can compute as $(P_1 + N) + P_2$, with P_2 being a point of r-torsion, while $P_1 + N$ is *not* a point of r-torsion; thus, it cannot happen that $P_1 + N = \pm P_2$. This avoids the exceptional case of the additional law when adding a point to itself; we thus immediately obtain *unified* formulas, where the only remaining exceptional cases are related to the neutral (here, N) as an operand or as a result.
– In order to make the computation of P_2 from $P_2 + N$ easier, we apply a change of variable on the curve so that N becomes the point $(0, 0)$.
– We then use a somewhat different coordinate sytem to remove exceptional cases related to the neutral.

This methodology hinges on the ability to efficiently check or otherwise ensure that a given point P+N is indeed the sum of N and an r-torsion point. As seen in the previous section, this is immediate when $\mathrm{Tr}(A) = 1$: non-r-torsion points (x, y) are exactly the points for whom $\mathrm{Tr}(x) = 0$. For the general case, see section 4.1.

The title of this section is slightly misleading because everything described below only requires r to be odd, not necessarily prime. However, the most useful case for cryptography is when r is prime.

New Curve Equation. We first move the point N to $(0,0)$ by applying the change of variable $y \mapsto y + \sqrt{B}$. We then obtain the following equation:

$$y^2 + xy = x(x^2 + ax + b)$$

for two constants $(a, b) = (A, \sqrt{B})$. We will now use these constants throughout the rest of this paper.

This new equation is *not* a "short Weierstraß" equation. The addition law (in (x, y) coordinates) is mostly unchanged, except for point doublings: λ is now computed as:

$$\lambda = x + \frac{y + b}{x}$$

instead of $x + y/x$. The other formulas are unchanged.

Given a point $P = (x, y)$, we get $P + N$ as:

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

New Coordinate System. We replace the y coordinate with another value s:

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

It is easy to see that given x, s can be computed from y and vice versa. Moreover, when $x \neq 0$ (i.e. all curve points except N and $\mathbb{O}$), we have $s = y^2/x$. The s coordinate of N is equal to b. In (x, s) coordinates, the curve equation can be transformed into the following:

$$s^2 + xs = x^4 + a^2x^2 + b^2$$

Group Definition. We can now formally define our group $\mathcal{G}$:

$$\mathcal{G} = \left\{(x, s) \in GF(2^m) \times GF(2^m) \mid (x, s + x^2 + ax + b) + N \in \mathcal{E}[r]\right\}$$

In other words, $\mathcal{G}$ consists of the points $P + N$, for all r-torsion points P. The addition law on $\mathcal{G}$, denoted "$\boxplus$", is:

$$(P_1 + N) \boxplus (P_2 + N) = P_1 + P_2 + N$$

The neutral is N, with coordinates $(0, b)$. The opposite of $P+N$ is $-P+N$. Since the opposite of (x, y) on $\mathcal{E}$ is $(x, y + x)$, it follows that the opposite of (x, s) in $\mathcal{G}$ is $(x, s + x)$.

The group $\mathcal{G}$ is obviously homomorphic to $\mathcal{E}[r]$ (the group of r-torsion points on the curve). If r is prime, then this is a prime-order group appropriate for building cryptographic protocols.

Addition Formulas. Starting from the affine formulas in (x, y) coordinates, we can derive formulas for addition in $\mathcal{G}$. We are given $P_1 + N = (x_1, s_1)$ and $P_2 + N = (x_2, s_2)$, and want to compute the coordinates of $P_3 + N = (P_1 + N) \boxplus (P_2 + N) = (x_3, s_3)$. We initially assume that neither $P_1 + N$ nor $P_2 + N$ is the point N; hence, $x_1 \neq 0$ and $x_2 \neq 0$. Since $P_1 + N$

and $P_2 + N$ are both in $\mathcal{G}$, it cannot happen that $x_1 = b/x_2$ (value b/x_2 is the x coordinate of P_2, which is a point of r-torsion, while x_1 is the x coordinate of $P_1 + N$, which is not a point of r-torsion); equivalently, $x_1x_2 + b \neq 0$. We can thus apply the general addition law on $P_1 + N$ and P_2 without ever encountering the special case of point doublings. A lengthy but straightforward calculation leads to the following formulas:

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

These formulas were derived under the assumption that neither $P_1 + N$ nor $P_2 + N$ was equal to N, but it is easily verified that they still return the proper result in those cases. Thus, the formulas are *complete*.

4 Group Membership and Extension to Full Curves

In this section, we study the problem of validating whether a given curve point is part of the group $\mathcal{G}$ or not. We then extend the process into a procedure that allows extending the complete formulas on $\mathcal{G}$ (shown in section 3) to perform computations on the full curve. Finally, we introduce a new point compression method that is appropriate for canonically encoding and decoding elements of $\mathcal{G}$.

We recall that the base curve $\mathcal{E}$ has order $2^t r$, for $t \geq 1$ and an odd integer r.

4.1 Group Membership Test

The general tool for testing whether a curve point Q is a member of $\mathcal{G}$ is point halving. Note that we envision here halving on the original curve, not in the group. We are given Q in (x, y) coordinates (or some other coordinate system) and we want to know whether $Q = P + N$ for some point $P \in \mathcal{E}[r]$. The cyclic structure of the 2^t-torsion subgroup of $\mathcal{E}$ implies the following result.

Lemma 1. *$Q \in \mathcal{G}$ if and only if Q can be halved successively exactly $t - 1$ times.*

Proof. If we decompose Q into $Q = P + kT$, as in section 2.2, then $Q \in \mathcal{G}$ if and only if $k = 2^{t-1}$. The P part can be halved indefinitely since it is an r-torsion point and r is odd. In the T part, halving corresponds to dividing k by 2, which is possible modulo 2^t only if k is an even integer (in the 0 to $2^t - 1$ range). Halving yields two possible points, which correspond to $k/2$ and $(k + 2^t)/2$ (which are the two possible halves of an even integer k modulo 2^t).

If the initial value k is not zero, then it is equal to $2^e f$ for some integers e and f such that $0 \leq e \leq t - 1$ and $f = 1 \bmod 2$, and it can be halved if and only if $e \geq 1$; the two possible halves are then $2^{e-1}f$ and $2^{e-1}(f + 2^{t-1-e})$. Note that $f + 2^{t-1-e}$ is odd. Thus, starting with the point Q, we can halve it exactly e times; this does not depend on which solution we choose for each halving.

If the initial value k is zero, then the first halving yields $k/2 = 0$ or $(k + 2^t)/2 = 2^{t-1}$. We can chain an indefinite number of halvings by staying on $k = 0$; otherwise, the choice

$k = 2^{t-1}$ then yields exactly $t-1$ more possible halvings. In total, starting with $k = 0$ (i.e. $Q \in \mathcal{E}[r]$), we can perform at least t successive halvings.

The only case that leads to exactly $t-1$ halvings is when $k = 2^{t-1}$, i.e. when $Q \in \mathcal{G}$, which completes the proof. □

We can thus test whether a point $Q = (x, y)$ is in $\mathcal{G}$ by halving it $t-1$ times, and checking that the final point *cannot* be halved. This can be done with algorithm 1:

Algorithm 1 Test membership in group $\mathcal{G}$

```
Input:  $Q = (x, y)$ in curve $\mathcal{E} : y^2 + xy = x(x^2 + ax + b)$ (curve order is $2^t r$ with $r = 1 \bmod 2$)
Output: TRUE if $Q \in \mathcal{G}$, FALSE if $Q \notin \mathcal{G}$
 1: for $i = 1$ to $t-1$ do
 2:     if $\mathrm{Tr}(x + a) \neq 0$ then
 3:         return FALSE
 4:     $\lambda \leftarrow \mathrm{QSolve}(x + a)$
 5:     $x \leftarrow \sqrt{y + \lambda x + x + b}$        ▷ Halving formulas adapted to curve $y^2 + xy = x(x^2 + ax + b)$
 6:     $y \leftarrow \lambda x + x^2 + b$
 7: if $\mathrm{Tr}(x + a) \neq 0$ then
 8:     return TRUE
 9: else
10:     return FALSE
```

Note that we are not, in this algorithm, particularly interested in the actual point obtained from all these halvings. In particular, in the last loop iteration (when $i = t-1$), we do not have to compute y, since that value will not be used afterwards. In the same iteration, we can also skip the square root computation to get x, because at the end, instead of testing $\mathrm{Tr}(x + a)$, we can use $\mathrm{Tr}((x + a)^2) = \mathrm{Tr}(x^2 + a^2)$. With these two optimizations, and in the case of $t = 1$ (cofactor 2) and $t = 2$ (cofactor 4), algorithm 1 becomes exactly the process already described by Solinas in his classic paper on Koblitz curves ([24], appendix A). In the general case, ignoring the additions and traces (which have negligible cost), algorithm 1 requires $t-1$ calls to QSolve, $t-1$ multiplications, $t-2$ squarings and $t-2$ square roots.

4.2 Full Curve Support

A group membership test is sufficient for obtaining a proper prime-order group abstraction (on a curve with a prime r). It is usually not necessary to ever consider computations on the whole curve. However, there are conceivable scenarios where such computations are made necessary. One example is a distributed system that processes signed data with a consensus protocol, in which all systems must fully agree on whether any given purported signature is acceptable or not. Some of these systems might employ a traditional ECDSA verification mechanism that does not check whether the public key is really part of the r-torsion group of the curve. A normal, legitimate public key should always be in the r-torsion group; it is not "wrong" to reject keys which are not in that group. However, the combination of the consensus requirement and of backward compatibility implies that new implementations (based on the methods described in this paper) should, in that scenario, support operations on all curve points, not just on an appropriate subgroup of order r.

If faced with such a situation, then the following method can be employed. The idea is to decompose the input point Q into $Q = P+kT$, where P is an r-torsion point, T is a fixed point of order exactly 2^t, and k is an integer modulo 2^t. We then represent Q as the pair $(P+N, k)$, with $P+N \in \mathcal{G}$ and $k \in \mathbb{Z}_{2^t}$. Such pairs form a group of order $2^t r$, isomorphic to the full curve, with the law $(P_1+N, k_1) + (P_2+N, k_2) = ((P_1+N) \boxplus (P_2+N), k_1+k_2 \bmod 2^t)$. Computations on the k values are trivial (additions modulo a power of 2, usually small); if using a double-odd curve ($t = 1$, i.e. $\mathrm{Tr}(a) = 1$), then the k values are Boolean flags and the addition is a simple XOR. Computations on the $P+N$ parts simply use the group $\mathcal{G}$. Conversion of $(P+N, k)$ back to Q can be done by computing $Q = (P+N)+(k+2^{t-1} \bmod 2^t)T$.

In this way, complete formulas are obtained for computing over the full original curve, provided that we can reasonably efficiently perform the decomposition. This can be done by recovering k on a bit-by-bit basis, as shown in algorithm 2.

Algorithm 2 Curve point decomposition

```
Input:  $Q$ in curve $\mathcal{E} : y^2 + xy = x(x^2 + ax + b)$ (curve order is $2^t r$ with $r = 1 \bmod 2$)
Output: $(P+N, k)$ with $P+N \in \mathcal{G}$ and $k \in \mathbb{Z}_{2^t}$ such that $Q = P + kT$
 1: if $Q = \mathbb{O}$ then
 2:     return $(N, 0)$
 3: $Q' \leftarrow Q$
 4: $k \leftarrow 0$
 5: for $i = 0$ to $t-1$ do
 6:     $(x, y) \leftarrow Q'$
 7:     if $\mathrm{Tr}(x+a) \neq 0$ then
 8:         $Q' \leftarrow Q' - T$                         ▷ Addition of curve points
 9:         $k \leftarrow k + 2^i \bmod 2^t$
10:         if $Q' = \mathbb{O}$ then
11:             return $(N, k)$
12:         $(x, y) \leftarrow Q'$
13:     $\lambda \leftarrow \mathrm{QSolve}(x+a)$       ▷ This can be skipped for the final iteration (see text below)
14:     $x \leftarrow \sqrt{y + \lambda x + x + b}$
15:     $y \leftarrow \lambda x + x^2 + b$
16:     $Q' \leftarrow (x, y)$
17: $P \leftarrow 2^t Q'$                                  ▷ $t$ successive point doublings
18: return $(P+N, k)$
```

Algorithm 2 works as follows: upon entry of iteration i of the loop, the current point Q' and integer k are such that $2^i Q' = Q - kT$. This is true when entering the first iteration ($Q' = Q$, $k = 0$, $i = 0$). When Q' can be halved, we simply do that; if Q' cannot be halved, then subtracting T transforms Q' into a new point that can be halved (since T itself cannot be halved); to maintain the invariant, we add 2^i to k. After all t iterations, we have $2^t Q' = Q - kT$. We double that Q' point t times, which necessarily yields a point P of r-torsion, since P can then be halved t times. We therefore have found the decomposition $Q = P + kT$. It only remains to add N to P to get its $P + N$ representation in the group $\mathcal{G}$.

The following notes apply:

- Subtraction of T must be performed over the full curve, with the incomplete addition formulas. It may happen that $Q' = T$ or $Q' = -T$, both cases being exceptional. A correct implementation can be challenging, especially if constant-time processing is required (though in general such decomposition would happen on publicly known points, e.g. public keys). Note that the case of the point-at-infinity is explicitly handled in algorithm 2; there again, a constant-time implementation would have to keep going in such cases.
- Subtraction of T must furthermore yield a result in *affine* coordinates, which involves an inversion in the field. Inversion is not too expensive, but can still cost as much as 1/20 of a point multiplication by a scalar. If the curve is such that t is large, then this process can become relatively expensive. Fortunately, standard curves tend to have low cofactors (i.e. $2^t = 2$ or 4).
- In the final iteration, most computations can be optimized away:
  - The final halving does not need to be performed, since it is immediately cancelled by a doubling when computing P. We can thus skip it, and instead compute $P = 2^{t-1}Q'$.
  - Similarly, the subtraction of T can be skipped in the final iteration, because subtracting T from Q' at this point is equivalent to subtracting $2^{t-1}T = N$ from the point $P = 2^{t-1}Q'$. That last-iteration subtraction of T can then be transformed into a conditional subtraction of N from the point P, which neatly combines with the final addition of N to P at the end of the algorithm.

With the optimization above, algorithm 2 becomes trivial for double-odd curves (i.e. $t = 1$), and remains reasonably simple and efficient for curves with cofactor 4 ($t = 2$). We nonetheless stress that in general we should not have to apply algorithm 2 at all; it is required only for rather contrived scenarios involving strict backward compatibility with support of nominally invalid values.

4.3 Point Compression

All of the above could be applied on any points in (x, y) coordinates, and thus is compatible with any encoding scheme, including the various compression schemes that have been defined. A most common one (often called "IEEE p1363 compression" because it was specified in the IEEE p1363-2000 standard, a document which has since been inactivated). In that scheme, a point (x, y) on the short Weierstraß curve $y^2 + xy = x^3 + Ax^2 + B$ is encoded as x and an extra bit v; the y coordinate is then rebuilt by first computing $w = \text{QSolve}(x + A + B/x^2)$, and replacing w with $w + 1$ if necessary so that its least significant bit matches the value v; the value y is obtained as $y = xw$. This encoding scheme can be applied as is on elements of $\mathcal{G}$, provided that such elements are first converted into (x, y) coordinates (with $y = s + x^2 + ax + b$) then mapped back to the short Weierstraß coordinates by adding b to y. The fact that $\mathcal{G}$ does not contain the point-at-infinity $\mathbb{O}$ even simplifies such operations.

We here define another compression scheme, which is applicable to elements of $\mathcal{G}$ (not to the full curve). It relies on the fact that the point addition on curves has a geometric interpretation: the line that goes from point P_1 to point P_2 also intersects the curve on a third point, which is $-(P_1 + P_2)$. It follows that the line that goes from point N to point P also intersects the curve on point $-P + N$, and these are the only three curve points that belong to that specific line. Note that only at most one of points P and $-P + N$ can be part of $\mathcal{G}$. Therefore, if we define $w = y/x = \sqrt{s/x}$, then w is the slope of the line from N to point (x, s) and the map from points in $\mathcal{G}$ to their w coordinate is injective. We can thus encode an element of $\mathcal{G}$ by its w coordinate.

The point N itself does not have a defined w coordinate. We conventionally choose a field element w_0 that will serve as the formal w coordinate of N; that value just needs to be "free", i.e. not corresponding to $\sqrt{s/x}$ for any point of $(x, s) \in \mathcal{G}$. We will detail later on how to choose w_0. In order to ultimately encode N as the value zero, we define that the encoding of a point $(x, s) \in \mathcal{G}$ is the field element $w_0 + \sqrt{s/x}$.

Decompression, i.e. decoding a field element into a point (x, s) in $\mathcal{G}$, is performed using algorithm 3.

Algorithm 3 Curve point decoding

```
Input:  $c \in GF(2^m)$
Output: $P + N \in \mathcal{G}$, or INVALID
 1: if $c = 0$ then
 2:     return $N$
 3: $w \leftarrow c + w_0$                    ▷ $w_0$ is a conventional constant, often zero
 4: $d \leftarrow w^2 + w + a$
 5: if $d = 0$ then                           ▷ This case cannot happen if $\mathrm{Tr}(a) = 1$
 6:     return INVALID
 7: $e \leftarrow b/d^2$
 8: if $\mathrm{Tr}(e) = 1$ then
 9:     return INVALID
10: $f \leftarrow \mathrm{QSolve}(e)$
11: $x \leftarrow df$
12: $(x', y') \leftarrow (x, xw)$
13: for $i = 1$ to $t - 1$ do
14:     if $\mathrm{Tr}(x' + a) \neq 0$ then
15:         return INVALID
16:     $\lambda \leftarrow \mathrm{QSolve}(x' + a)$
17:     $x' \leftarrow \sqrt{y' + \lambda x' + x' + b}$
18:     $y' \leftarrow \lambda x' + x'^2 + b$
19: if $\mathrm{Tr}(x' + a) = 0$ then
20:     $x \leftarrow x + d$
21: return $(x, xw^2)$
```

We recognize in steps 13 to 18 the sequence of point halvings which was already in the group membership test (algorithm 1). As in that previous algorithm, the value y' does not need to be computed in the last iteration, and the square root extraction for x' can be omitted as well since $\mathrm{Tr}(x' + a) = \mathrm{Tr}(x'^2 + a^2)$. When $t = 1$ (for double-odd curves), the entire loop disappears.

Algorithm 3 first finds a point (x, y) that matches the input w value; the curve equation ($y^2 + xy = x(x^2 + ax + b)$) is easily turned into:

$$x(w^2 + w + a) = x^2 + b$$

which is a quadratic equation in x solved by transforming it into a shape amenable to the QSolve function:

$$\left(\frac{x}{w^2 + w + a}\right)^2 + \frac{x}{w^2 + w + a} = \frac{b}{w^2 + w + a}$$

Note that $\mathrm{Tr}(d) = \mathrm{Tr}(w^2+w+a) = \mathrm{Tr}(a)$, so d cannot be zero if $\mathrm{Tr}(a) \neq 0$. On curves where $\mathrm{Tr}(a) = 0$, there can exist a curve point R (distinct from $\mathbb{O}$ and N) such that $w^2 + w + a = 0$; however, that implies that $x = \sqrt{b} = b/x$, which implies that $R = -R + N$, i.e. that point is a point of order 4 (and therefore not a valid element of $\mathcal{G}$).

The solution f obtained from QSolve is turned into the candidate x by simply multiplying by $d = w^2 + w + a$; the other solution to QSolve is $f + 1$, corresponding to $x + d$. At most one of the two resulting points is in $\mathcal{G}$. To know which solution to use, and whether it actually designates a point in the group, we apply some point halvings on the decoded point Q:

– If $Q \in \mathcal{G}$, then it can be halved exactly $t - 1$ times, but no more than that.
– If $Q \in \mathcal{E}[r]$ (i.e. $Q + N \in \mathcal{G}$), then it can be halved at least t times.
– If Q is neither in $\mathcal{G}$ nor $\mathcal{E}[r]$, then it can be halved only at most $t - 2$ times.

The loop computes the $t - 1$ halvings (if one fails, then neither Q nor $Q + N$ is in $\mathcal{G}$), then checks (in step 19) whether another extra halving could be performed or not, to know whether Q was in $\mathcal{G}$ or $\mathcal{E}[r]$.

For the choice of w_0, conventionally used as the w coordinate of the neutral, it suffices to choose a value that would make algorithm 3 return Invalid. Any value w_0 such that either $w_0^2 + w_0 + a = 0$ or $\mathrm{Tr}(b/(w_0^2 + w_0 + a)) = 1$ will work; in about half of the curves, $w_0 = 0$ happens to be an appropriate value. We also note that in the case of curves with $a = 0$, the value $w_0 = 0$ is always appropriate.

5 Formulas

In order to implement the point addition and doubling formulas efficiently, we define a representation of a point $P + N = (x, s) \in \mathcal{G}$, consisting of four field elements $(X{:}S{:}Z{:}T)$, with the following rules:

$$\begin{aligned} Z &\neq 0 \\ x &= \sqrt{b}X/Z \\ s &= \sqrt{b}S/Z^2 \\ T &= XZ \end{aligned}$$

The neutral N uses $X = T = 0$, and $S = \sqrt{b}Z^2$.

All the formula costs are expressed in terms of field element multiplications (M), squarings (S) and multiplications by a constant ($\mathrm{m_b}$). In almost all formulas, the latter constant is

$\sqrt{b}$. We ignore additions, whose cost is negligible, as well as multiplications by a or a^2, because, as was pointed out in section 2.2, it is always possible to apply a curve isomorphism to turn the constant a into a value of minimal Hamming weight. Indeed, all standard NIST curves use $a = 0$ or $a = 1$.

When selecting a new curve, it is normally possible to focus on values for b such that $\sqrt{b}$ has a low Hamming weight, and multiplications by $\sqrt{b}$ are inexpensive.[4] Some standard curves, in particular the NIST non-Koblitz curves (B-233, B-283...) were selected with pseudorandomly generated constants B, leading to expensive multiplications by $\sqrt{b}$ (practically speaking, about as expensive as generic multiplications in the field). For support of such curves, it is thus desirable to reduce the "$\mathrm{m_b}$" cost. The constant $\sqrt{b}$ was used in the definitions of the X and S coordinates precisely for that reason: it reduces the number of multiplications by $\sqrt{b}$ in the point addition and doubling formulas.

Standard Koblitz curves represent an ideal case, because they use $B = 1$, making multiplications by $\sqrt{b}$ trivial (and thus with no cost at all). Moreover, all of them (except K-163) use $a = 0$, which further helps with performance since it removes one multiplication from the point addition formulas.

In the following subsections, the formulas are expressed mathematically, then illustrated with pseudocode in a Python/Sage syntax. The costs correspond to the operations shown in the pseudocode figures.

5.1 Point Addition

Given points $P_1 + N = (X_1{:}S_1{:}Z_1{:}T_1)$ and $P_2 + N = (X_2{:}S_2{:}Z_2{:}T_2)$, their sum in $\mathcal{G}$ is $P_3 + N = (P_1 + N) \boxplus (P_2 + N) = (X_3{:}S_3{:}Z_3{:}T_3)$, which can be computed as follows:

$$\begin{aligned}
X_3 &= T_1T_2 + S_1T_2 + S_2T_1 \\
S_3 &= \sqrt{b}((Z_1Z_2)^2(S_1S_2 + a^2T_1T_2) + (X_1X_2)^2((a^2+1)T_1T_2 + S_1T_2 + S_2T_1 + S_1S_2)) \\
Z_3 &= \sqrt{b}(X_1X_2 + Z_1Z_2)^2 \\
T_3 &= X_3Z_3
\end{aligned}$$

These formulas can be computed in cost 8M + 2S + 2$\mathrm{m_b}$, as illustrated in figure 1. The computation of $a^2T_1T_2$ (variable `E` in the pseudocode) disappears when $a = 0$, which lowers the cost to 7M + 2S + 2$\mathrm{m_b}$. Note that these costs are quite lower than that of the previously best known formulas (11M + 2S with lambda coordinates[18]).

[4]It may be noted that we work here with $\sqrt{b} = \sqrt[4]{B}$, with B being the constant chosen in the original short Weierstraß curve. If B was chosen so that multiplications by B are fast, but multiplications by $\sqrt[4]{B}$ turn out to be not so fast, then a possible optimization strategy is to raise everything to the power 4, i.e. define $(x', y') = (x^4, y^4)$, and work in the curve $y'^2 + x'y' = x'^3 + A^4x'^2 + B^4$, i.e. with curve constants $A' = A^4$ and $B' = B^4$. We then have $\sqrt[4]{B'} = B$.

```
# Addition in the group, XSZT coordinates.
def point_add(P1, P2):
    (X1, S1, Z1, T1) = P1
    (X2, S2, Z2, T2) = P2
    X1X2 = X1*X2
    S1S2 = S1*S2
    Z1Z2 = Z1*Z2           # for mixed addition (Z2 == 1), Z1Z2 = Z1
    T1T2 = T1*T2
    D = (S1 + T1)*(S2 + T2)
    E = aa*T1T2            # aa == a^2; this disappears if a == 0
    F = X1X2**2
    G = Z1Z2**2
    X3 = D + S1S2
    S3 = sqrt_b*(G*(S1S2 + E) + F*(D + E))
    Z3 = sqrt_b*(F + G)
    T3 = X3*Z3
    return (X3, S3, Z3, T3)
```

Fig. 1: Point addition in $\mathcal{G}$.

Mixed Addition. When the second operand $(P_2 + N)$ is a precomputed constant, it will often be already normalized to affine (x, s) coordinates, i.e. with $Z_2 = 1$. In that case, the computation of Z_1Z_2 (into variable Z1Z2 in the code) simplifies to using Z_1 wherever Z_1Z_2 would be used. This reduces the cost to 7M + 2S + 2m$_b$ (6M + 2S + 2m$_b$ for curves with $a = 0$).

We also note that such affine points only need two coordinates for storage, since $Z = 1$ implies that $T = X$.

5.2 Point Doubling

The generic point addition formulas also work for adding a point to itself. However, some extra optimizations are possible when it is statically known that the two addition operands are the same point. Given input point $P + N = (X{:}S{:}Z{:}T)$, its double (in $\mathcal{G}$) $P' + N = 2P + N = (X'{:}S'{:}Z'{:}T')$ can be computed as:

$$
\begin{aligned}
X' &= T^2 \\
S' &= \sqrt{b}((X^2 + Z^2)(S + aT) + X^2T)^2 \\
Z' &= \sqrt{b}(X^2 + Z^2)^2 \\
T' &= X'Z'
\end{aligned}
$$

There are several ways to translate these formulas into code, and optimization trade-offs leveraging the fact that $b(X + Z)^4 = S^2 + a^2T^2 + ST$ (this comes directly from the curve equation in (x, s) coordinates). Figure 2 shows one such possibility, with cost 3M + 4S + 4m$_b$. If working with a curve whose $\sqrt{b}$ constant is not "simple", then performance may be improved by using the code from figure 3, which has cost $3M + 5S + 2m_b$ (in essence, exchanging two multiplications by $\sqrt{b}$ for one squaring).

```
# Doubling in the group, XSZT coordinates (cost 3M+4S+4mb)
def point_double(P):
    (X, S, Z, T) = P
    D = (S + a*T)*(S + a*T + T)
    E = (S + a*T + T + sqrt_b*Z**2)**2
    F = T**2
    Xp = sqrt_b*F
    Zp = D + a*F
    Sp = sqrt_b*(E*(E + Zp + F) + (D + sqrt_b*Xp)**2)
    Tp = Xp*Zp
    return (Xp, Sp, Zp, Tp)
```

Fig. 2: Point doubling in $\mathcal{G}$.

```
# Doubling in the group, XSZT coordinates (alternate, cost 3M+5S+2mb)
def point_double_alt(P):
    (X, S, Z, T) = P
    XX = X**2
    ZZ = Z**2
    Xp = T**2
    Sp = sqrt_b*((XX + ZZ)*(S + a*T) + XX*T)**2
    Zp = sqrt_b*((XX + ZZ)**2)
    Tp = Xp*Zp
    return (Xp, Sp, Zp, Tp)
```

Fig. 3: Point doubling in $\mathcal{G}$ (alternate version).

Multiple successive doublings can be performed by invoking the point doubling routine repeatedly; however, some extra optimizations are possible for curves on which multiplications by $\sqrt{b}$ are expensive. The doubling procedure shown in figure 2 is really the combination of two successive operations:

1. Convert point $P + N$ into point P in (x, λ) coordinates (cost: $1S + 1m_b$).
2. Double P (in (x, λ) coordinates) into $2P+N$ (in (x, s) coordinates) (cost: $3M+3S+3m_b$).

(x, λ) coordinates are described in [18]; translated to our curve (which does not uses a short Weierstraß equation), $\lambda = x + (y + b)/x$. These coordinates are represented by the triplet

$(X{:}L{:}Z)$ with $x = X/Z$ and $\lambda = L/Z$. The first of the two steps above consists in computing the $(X'{:}L'{:}Z')$ coordinates of P, given $P + N = (X{:}S{:}Z{:}T)$:

$$\begin{aligned} X' &= \sqrt{b}Z^2 \\ L' &= S + aT + T \\ Z' &= T \end{aligned}$$

and the second step, which computes $2P + N = (X''{:}S''{:}Z''{:}T'')$ from P is the following:

$$\begin{aligned} X'' &= \sqrt{b}Z'^2 \\ Z'' &= L'(L' + Z') + aZ'^2 \\ S'' &= \sqrt{b}((L' + X')^2((L' + X')^2 + Z'' + Z'^2) + (L'(L' + Z') + \sqrt{b}X'')^2) \\ T'' &= X''Z'' \end{aligned}$$

Between the two steps, it is possible to insert additional doublings in (x, λ) coordinates, using the formulas from [18]. Each such extra doubling requires three field multiplications, four squarings, and only one multiplication by a non-trivial constant (that constant is $a^2 + b^2$ on our curve). In total, this method computes n doublings with cost $(3n)\mathrm{M} + (4n)\mathrm{S} + (n + 3)\mathrm{m_b}$, which is preferable (for curves with a non-simple $\sqrt{b}$ constant) over calling n times the simple doubling function (from figure 3), which would lead to cost $(3n)\mathrm{M}+(5n)\mathrm{S}+(2n)\mathrm{m_b}$.

This multiple-doubling process is detailed in figure 4. It shall be noted that the neutral N is properly handled, i.e. this function is complete. (x, λ) coordinates are not nominally defined for either N or $\mathbb{O}$. However, it can be easily verified that the doubling formulas in (x, λ) coordinates are, in fact, complete, provided that the point N is represented with $(X{:}L{:}Z) = (0{:}L{:}0)$ for some $L \neq 0$, and $\mathbb{O}$ is represented with $(L{:}L{:}0)$ for some $L \neq 0$.

5.3 Point Negation and Subtraction

The opposite of $P + N \in \mathcal{G}$ is $-P + N$, which is obtained by replacing s with $s + x$. In our coordinate system, this is done by adding T to S. The cost is negligible (a single field element addition). Subtraction of a point from another can be done by simply negating the second operand, then performing a point addition.

5.4 Montgomery Ladders

López-Dahab Formulas. Multiplication of a point P_0 by a scalar n can be performed efficiently through a *Montgomery ladder*. A pair of points (P_1, P_2) is maintained, with the rule that $P_2 = P_0 + P_1$. Suppose that at some point, $P_1 = kP_0$ for some integer k (and thus $P_2 = (k + 1)P_0$). Then, the point $P_3 = P_1 + P_2 = (2k + 1)P_0$ is computed, as well as the double of either P_1 or P_2. The pair (P_1, P_2) is then replaced with either $(2P_1, P_3)$, or the pair $(P_3, 2P_2)$. In both cases, the invariant is maintained (the difference of the two points is P_0), and the new P_1 is equal to $(2k)P_0$ (first case) or $(2k + 1)P_0$ (second case). In that way, we can compute nP_0 for arbitrary scalars n by processing the bits of n in high-to-low order.

The main advantage of the technique is that such computations can be performed with the x coordinates only: given the x coordinates of P_0, P_1 and P_2, the x coordinate of P_3 can

```
# Multiple doublings in the group (cost (3n)M+(4n)S+(n+3)mb)
def point_xdouble(P, n):
    (X, S, Z, T) = P

    # P+N (x,s) -> P (x,lambda)
    X = sqrt_b*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
        # aa = a^2, bb = b^2 (curve constants)

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

Fig. 4: Multiple doublings in $\mathcal{G}$ (for curves with a non-simple $\sqrt{b}$ constant).

be computed without knowing the y coordinates of any of the points. When the final point is obtained (in P_1 at the end of the algorithm), its y coordinate can nonetheless be efficiently rebuilt from the x coordinates of P_0, P_1 and P_2, and the y coordinate of P_0.

In 1999, López and Dahab[15] showed how to apply a Montgomery ladder to any non-supersingular binary elliptic curve, with a cost of $5\mathrm{M} + 4\mathrm{S} + 1\mathrm{m_b}$ per scalar bit. The formulas have only one exceptional case, when the base point P_0 has order two (i.e. when $P_0 = N$). We can use that ladder with our group $\mathcal{G}$ in the following way:

1. Inputs are $P_0 + N = (X{:}S{:}Z{:}T) \in \mathcal{G}$, and the scalar n. We assume that $P_0 + N \neq N$; we will handle the case of multiplying the group neutral at the end with a single corrective step. We first compute $P_0 = (x_0, y_0)$ with:

$$\begin{aligned} x_0 &= \sqrt{b}Z/X \\ y_0 &= \sqrt{b}(S + aT + T + \sqrt{b}(X + Z)^2)/X^2 + b \end{aligned}$$

   The addition of b to y_0 means that we get P_0 on the original short Weierstraß curve ($y^2 + xy = x^3 + Ax^2 + B$). The point P_0 is furthermore an r-torsion point (by definition of $\mathcal{G}$); in particular, it cannot be a point of order two, which avoids the exceptional case of the López-Dahab formulas. These formulas imply one inversion (to compute $1/X$). If we do not obtain the affine version of x_0, then the cost per scalar bit is increased by one multiplication; computing a single inversion at the start yields better overall performance.
2. The x coordinate of P_1 is represented as X_1/Z_1; The point-at-infinity uses $Z_1 = 0$; for all other points, $Z_1 \neq 0$. Note that $X_1 \neq 0$ for all points (including $\mathbb{O}$), since P_0 cannot be a point of order two. The formulas for computing $P_3 = P_1 + P_2$ and $P_4 = 2P_1$ are:

$$\begin{aligned} Z_3 &= (X_1Z_2 + X_2Z_1)^2 \\ X_3 &= (X_1Z_2)(X_2Z_1) + x_0Z_3 \\ Z_4 &= (X_1Z_1)^2 \\ X_4 &= (X_1 + \sqrt{b}Z_1)^4 \end{aligned}$$

   (The formula for X_4 is expressed slightly differently from the original in [15] so that the constant is equal to $\sqrt{b}$, like in our other formulas.)
3. Once the x coordinates of $P_1 = nP_0$ and $P_2 = P_1 + P_0$ are obtained, the full $(X'{:}S'{:}Z'{:}T')$ coordinates of the result, back in group $\mathcal{G}$, are computed as follows (note that the original source point $(X{:}S{:}Z{:}T)$ coordinates are used):

$$\begin{aligned} Z_r &= \sqrt{b}ZZ_1^2Z_2 \\ Y_r &= X(X_1 + x_0Z_1)((X_1 + x_0Z_1)(X_2 + x_0Z_2) + (x_0^2 + y_0)Z_1Z_2) + y_0Z_r \\ X' &= \sqrt{b}ZZ_1Z_2 \\ Z' &= X_1ZZ_2 \\ S' &= ZZ_2(Y_r + \sqrt{b}Z'(X_1 + aZ_1 + Z_1)) \\ T' &= X'Z' \end{aligned}$$

4. If $P_2 = \mathbb{O}$ (i.e $Z_2 = 0$), then the above formulas return zero in all four coordinates, which is invalid. This happens only when $n = -1 \bmod r$, in which case the result should be $-P_0 + N$. This can be readily detected by checking whether $Z' = 0$; in that case, the four coordinates should be replaced with the coordinates of $-P_0 + N = (X{:}S + T{:}Z{:}T)$.
5. If the original source point $P_0 + N$ was the neutral (i.e. if $X = 0$), then all of the above is wrong, and the four output coordinates should be set to a representation of the neutral. Note that $-P_0 + N$, in that case, *is* a representation of the neutral. Thus, this corrective action can be merged into the previous one: the output coordinates are to be replaced with those of $-P_0 + N$ if $Z' = 0$ or $X = 0$.

Ladder Formulas in the Group. Efficient ladder formulas also exist in the group $\mathcal{G}$ itself; their cost is 5M + 4S + 2m$_b$ per scalar bit, which is equivalent to the López-Dahab formulas cost when $\sqrt{b}$ is "simple", but slightly more expensive if multiplications by $\sqrt{b}$ have a non-negligible cost. Using these formulas avoids conversions between curves and makes initial and final formulas somewhat simpler.

The source point $P_0 + N$ is first normalized to affine coordinates (x_0, s_0), which requires inverting the source value Z. Since $Z \neq 0$, this has no special case.

Suppose that you have points $P_0 + N$, $P_1 + N$ and $P_2 + N$ in $\mathcal{G}$, such that $(P_1 + N) \boxplus (P_0 + N) = P_2 + N$. The x coordinate of $P_0 + N$ is x_0, while the x coordinates of $P_1 + N$ and $P_2 + N$ are known as the fractions $\sqrt{b}X_1/Z_1$ and $\sqrt{b}X_2/Z_2$. Then the x coordinates of $P_3 + N = (P_1 + N) \boxplus (P_2 + N)$ and $P_4 + N = 2P_1 + N$ can be computed as fractions $\sqrt{b}X_3/Z_3$ and $\sqrt{b}X_4/Z_4$, such that:

$$\begin{aligned}
Z_3 &= \sqrt{b}(X_1X_2 + Z_1Z_2)^2 \\
X_3 &= X_1X_2Z_1Z_2 + x_0(X_1X_2 + Z_1Z_2)^2 \\
Z_4 &= \sqrt{b}(X_1 + Z_1)^4 \\
X_4 &= (X_1Z_1)^2
\end{aligned}$$

When the scalar has been processed and the x coordinate of the result is $\sqrt{b}X_1/Z_1$, we can rebuild the complete $(X'{:}S'{:}Z'{:}T')$ of the result with the following formulas (that use the affine coordinate of $P_0 + N = (x_0, s_0)$):

$$\begin{aligned}
X' &= x_0X_1Z_2 \\
S' &= x_0Z_2(X_1Z_1Z_2(x_0 + s_0) + X_2(x_0X_1 + \sqrt{b}Z_1)^2) \\
Z' &= x_0Z_1Z_2 \\
T' &= X'Z'
\end{aligned}$$

These last conversion formulas, unfortunately, admit one exceptional case, which happens when $x_0 = 0$ (i.e. the source point $P_0 + N$ was N, the neutral element of $\mathcal{G}$). In that case, all output coordinates are zero, which is invalid. That case can be easily detected (we can check whether $x_0 = 0$, or whether $Z' = 0$, these tests are equivalent); the corrective action is then to set S' and Z' to $\sqrt{b}$ and 1, respectively, so that a valid representation of N is obtained.

Figure 5 illustrates the whole process with these formulas. The figure uses two conditional swap operations of all four running coordinates per loop iteration; they could be merged into a single conditional swap (of all four coordinates), as is done in the Montgomery ladder on Curve25519 specified in RFC 7748 ([14], section 5).

```
# Multiplication of a group element by a scalar (ladder algorithm)
def point_mul_ladder(P, n):
    (X, S, Z, T) = P
    n = int(n)
    if n < 0:
        S += T
        n = -n

    # Normalize the point to affine.
    iZ = 1/Z
    x0 = sqrt_b*X*iZ
    s0 = sqrt_b*S*(iZ**2)

    # Ladder.
    (X1, Z1) = (0, 1)
    (X2, Z2) = (x0, 1)
    for i in reversed(range(0, n.bit_length()):
        # swap P1 and P2 if the scalar bit is 1
        bit = (n >> i) & 1
        if bit != 0:
            (X1, Z1, X2, Z2) = (X2, Z2, X1, Z1)
        X1X2 = X1*X2
        Z1Z2 = Z1*Z2
        D = (X1X2 + Z1Z2)**2
        X3 = X1X2*Z1Z2 + x0*D
        Z3 = sqrt_b*D
        X4 = (X1*Z1)**2
        Z4 = sqrt_b*(X1 + Z1)**4
        if bit != 0:
            (X3, Z3, X4, Z4) = (X4, Z4, X3, Z3)
        (X1, Z1, X2, Z2) = (X3, Z3, X4, Z4)

    # Rebuild the full result coordinates.
    Z1Z2 = Z1*Z2
    D = x0*X1
    Xp = D*Z2
    Sp = x0*Z2*(X1*Z1Z2*(x0 + s0) + X2*(D + sqrt_b*Z1)**2)
    Zp = x0*Z1Z2
    Tp = Xp*Zp

    # Corrective action if the source was the neutral.
    if x0 == 0:
        Sp = sqrt_b
        Zp = 1

    return (Xp, Sp, Zp, Tp)
```

Fig. 5: Multiplication of a point by a scalar (with a Montgomery ladder).

5.5 Comparisons

We can compare two points together by noticing that the mapping from group elements to their $w = y/x$ coordinate is injective (see section 4.3). Since $w^2 = s/x$, we can thus check whether points $(X_1{:}S_1{:}Z_1{:}T_1)$ and $(X_2{:}S_2{:}Z_2{:}T_2)$ are in fact the same point by verifying the following:

$$S_1 T_2 \stackrel{?}{=} S_2 T_1$$

This equation directly translates the affine relation $s_1/x_1 = s_2/x_2$; it obviously works if s_1, x_1, s_2 and x_2 are non-zero. To see why it in fact works in all cases, consider the following:

- N is the only point such that $T = 0$.
- There can be only at most two points on the curve such that $s = 0$ (since that implies $x^2 + ax + b = 0$, which can have at most two solutions). Their x coordinates are then x and b/x for some $x \neq 0$, i.e. the points are V and $-V + N$ for some curve point V. Only one of these two points (at most) can be in $\mathcal{G}$.

Therefore, the only potential exceptional cases for this comparison relation would be related to using $N = (0, b)$ or $V = (x, 0)$ as operand. Simply enumerating all possibilities of comparing N, V and $P \neq N, V$ shows that the formula works in all cases.

6 Implementation

We implemented the formulas described in this paper, for the NIST curves B-233 and K-233. The code is available at:

https://github.com/pornin/c-xs233

C was used. The implementation assumes that a 64-bit platform is used, and that the compiler provides a 128-bit `unsigned __int128` integer type; it was tested on x86 and ARMv8 platforms, with both the Clang (14.0.0) and GCC (11.2.0) compilers, on a Linux operating system. When compiled for an x86 target that offers the `pclmul` opcodes and the SSE4.1 vector instructions, then a backend using these instructions is used, which greatly improves performance; otherwise, a generic and portable backend is employed.

The groups $\mathcal{G}$ for curves B-233 and K-233 are called xsb233 and xsk233, respectively. In both cases, a flexible API is offered, that includes functions for the following operations:

- Statically allocated neutral and generator elements. The generator corresponds to the standard generator defined in the NIST curves.
- Point decoding and encoding, as per the compression method described in section 4.3.
- Comparisons (equality of two points, and equality with the neutral element).
- Point addition, subtraction, negation, doubling, and sequences of successive doublings.
- Conditional negation and selection, based on a flag.
- Multiplication of a point by a scalar. A classic double-and-add algorithm with 5-bit windows and Booth recoding (16 points per window) is offered (under the name "`mul`"), as well as an alternate implementation using a Montgomery ladder ("`mul_ladder`"), as described in section 5.4.

– An optimized point multiplication routine, when the base point is the conventional generator ("`mulgen`"). This operation typically corresponds to key pair generation in cryptographic protocols. It uses a double-and-add algorithm with 5-bit windows; four precomputed windows are used (for points $G + N$, $(2^{60})G + N$, $(2^{120})G + N$ and $(2^{180})G + N$) and these windows use affine coordinates.
– On xsk233 only, extra variants of `mul` and `mulgen` are also implemented, under the names `mul_frob` and `mulgen_frob`. These functions leverage the Frobenius endormorphism on Koblitz curves, using the methods described by Solinas[24].

Everything is strictly constant-time. This applies to all implemented operations, including encoding and decoding functions (side channel leaks will not disclose the decoded point, nor even whether the decoding succeeded or failed). For the point multiplication routines on xsk233 with acceleration from the Frobenius endomorphism, this required "un-NAFfing" of the output of scalar reduction, so that a constant-time lookup with 5-bit windows could be performed: the scalar reduction uses a complicated process which inherently outputs signed digits (of value −1, 0 or +1) in a non-adjacent format (no two consecutive digits are non-zero); regrouping them into chunks of 5 digits yields 43 possible combinations, out of which one consists only of zeros, and the 42 others are opposite pairs that can thus be reduced to 21 points per window.

With `pclmul`. We measured the performance of the point multiplication operations on an Intel i5-8259U (Coffee Lake) system, clocked at 2.3 GHz, running under Linux (Ubuntu 22.04) in 64-bit mode. Compiler is Clang-14.0.0, with flags "`-O2 -mpclmul -mavx2`" (the code does not explicitly use AVX or AVX2 registers or opcodes, but the compiler can still take advantage of these features to automatically speed up some operations such as memory copies). The cycle counter register is used (TurboBoost is disabled). A single core is employed. The obtained performance is shown in table 1.

Operation	xsb233	xsk233
`mul`	60621	49378
`mul_ladder`	51537	46365
`mul_frob`	-	29602
`mulgen`	25754	20084
`mulgen_frob`	-	16546

Table 1: Performance of operations on groups xsb233 and xsk233 on 64-bit x86 "Coffee Lake", with use of `pclmul` and SSE4.1 (values in clock cycles).

We may note that the ladder implementations outperform the generic window-based implementations on both curves, but less so in the case of xsk233. For every 5 scalar bits, the window based method is expected to have cost $23\mathrm{M} + 22\mathrm{S} + 10\mathrm{m_b}$, versus $25\mathrm{M} + 20\mathrm{S} + 5\mathrm{m_b}$ for the ladder method (for xsb233). Given that multiplications by $\sqrt{b}$ cost as much as generic multiplications for that curve, it is no wonder that the ladder is faster. On xsk233, one multiplication is saved for each point addition (since $a = 0$ for that curve) and multiplications by $\sqrt{b}$ cost nothing (since $b = 1$), and the end result is more balanced. Both algorithms have some overhead (window building and constant-time window lookups for the `mul` function, initial inversion for `mul_ladder`); the measurements should be interpreted as the overhead for `mul_ladder` being lower.

Compared to the ladder code, though, the generic window-based addition can be extended to other situations, including use of multiple precomputed windows (as in `mulgen` and `mul_frob`) or multiscalar multiplication, i.e. when computing $uP+vQ$ for two points P and Q and two scalars u and v, as is commonly used in ECDSA or Schnorr signature verification (we did not implement this functionality, but the completeness of the formulas should help).

The performance achieved by `mul_frob` on xsk233 is, by elliptic curve standards, very good. In fact it is close to three times faster than what could be achieved with, for instance, Curve25519[16]; it also beats the fastest reported performance for GLS-254 (at 35739 cycles on a Kaby Lake core). Of course, K-233 offers a security of "only" 112 bits instead of 128 bits, though in practice the difference is rather meaningless; with foreseeable improvements in computing technology, neither 112-bit nor 128-bit security will be broken within the next few decades, for rather fundamental energy consumption reasons, unless a working quantum computer finally emerges, and such a machine would destroy security of both curves with similar ease. The main advantage of the 128-bit level is psychological, since 128 is a more round number, thus aesthetically more pleasing, and magically more potent.

Without `pclmul`. If we compile the same code on the same system but *without* the options that enable use of `pclmul` and SSE4.1, i.e. invoking Clang with only `-O2`, then the implementation switches to a portable backend that relies on integer multiplications (over 64 bits, with a 128-bit result). Field elements are split into limbs of 58 bits each (59 bits for the top limb), and the multiplication of two limbs as polynomials in $GF(2)[z]$ uses integer multiplication with "holes": three bits out of four are masked out, so that carries inherent to the additions induced by the multiplication accumulate only inside these holes without spilling over the next data bit. Each multiplication of two limbs then requires 16 integer multiplications, and some masking and combining of values. This is a technique which has already been used in several places, e.g. as part of some of the GHASH implementations in the BearSSL library[21].

Even though the integer multiplications in modern x86 cores are fast and well pipelined (a new multiplication can be issued at every cycle, and results are obtained with a latency of 4 cycles), this implementation strategy yields much worse performance. This can already be seen in the speed of field element multiplications, going from about 30.5 cycles (with `pclmul`) to a whooping 415 cycles (without `pclmul`). Note that such values are only estimates, because they were measured over long sequences of co-dependent multiplications: the output of each multiplication is used as one operand for the next one. This implies that what is measured is the *latency* of the operation, but not the throughput. Indeed, such CPUs can not only issue several instructions in parallel, but will routinely dynamically reorder instructions over a pipeline which can extend over more than a hundred instructions; thus, two implementations that seem to offer the same performance through this latency-based measurement may yield different overall performance when integrated in larger operations.[5] We can still expect things to be considerably slower with this non-`pclmul` code. More interestingly, squarings are also slower, but less so comparatively speaking: their measured latency jumps from 22.2 to 78.5 cycles. This situation is thus a data point that illustrates a larger multiplication-to-squaring cost ratio. Measured performance is shown in table 2.

Operation	**xsb233**	**xsk233**
`mul`	818079	577398
`mul_ladder`	712140	607688
`mul_frob`	-	267945
`mulgen`	332388	231132
`mulgen_frob`	-	153376

Table 2: Performance of operations on groups xsb233 and xsk233 on 64-bit x86 "Coffee Lake", with generic portable code (no use of `pclmul` nor SSE2+ intrinsics) (values in clock cycles).

As expected, cost of each operation is about 12 to 14 times the cost of the same operation in the `pclmul` implementation; however, the ratio is only 9 times for the squaring-heavy "`frob`" versions, since squarings have become much faster relatively to multiplications. We also note that the double-and-add `mul` is now a bit faster than the ladder-based `mul_ladder` on xsk233: this is due partly to the higher use of multiplications in the latter (25 vs 22 for each chunk of 5 scalar bits), and partly to the lower relative cost of window lookups in this version (lookups with 64-bit registers are a bit slower than their SSE2-aware counterparts, but the slowdown is less dramatic than that of multiplications or squarings).

7 Conclusion

In this article, we have detailed a construction for using the odd-order subgroup of a binary curve as a safe abstraction, with canonical and verified encoding, no cofactor issue, and complete formulas for computing operations in a safe way, with no timing-based side channels and no exceptional cases. While safety and convenience were the main point, it turned out that the new formulas are also faster than the best previously known formulas for point addition (which were moreover incomplete). Furthermore, we showed that with a bit of extra effort upon decoding, it is possible to apply the new formulas to the complete curve, not just its subgroup of interest, if backward compatibility considerations make such a functionality desirable.

[5] Indeed, during the development of our code, a previous version of the field squaring with `pclmul` yielded the same latency as the current version, but the current version is still 15% faster for point multiplications. The new version uses fewer instructions, thanks to judicious use of the `palignr` opcode, giving the CPU more free slots in which to schedule elements of other non-dependent operations. This effect is not well captured by latency-based benchmarks.

We cannot fully recommend binary elliptic curve for general usage, because their performance on small embedded systems is likely to be poor. More research is warranted in that area: the traditional lore is that performance on small CPUs will be abysmal, but microcontrollers and smartcards in 2022 are not the same hardware as what they were in 1995. A very common low-power microcontroller is the ARM Cortex M0+, which is a 32-bit core that can perform $32 \times 32 \rightarrow 32$ integer multiplications in one cycle; it is possible that a properly optimized implementation of a fast binary curve such as K-233 might actually be competitive on such a platform. The question is still open for now.

On large CPUs with carryless multiplication opcodes, a quite large category since such opcodes tend to be part of any modern hardware acceleration of AES/GCM, it is known that binary elliptic curves are very fast, more so than elliptic curves of similar security defined over large characteristic fields. Among fast binary curves, Koblitz curves such as K-233 are especially efficient, as demonstrated here; GLS curves such as GLS254 are also quite fast. The formulas described in this paper apply to all binary elliptic curves, and that includes GLS254. It is plausible that even with GLS254, these formulas would not only grant safety through completeness, but also some tangible performance benefit. This is also an open question for now.

Acknowledgements

We thank Gérald Doussot and Giacomo Pope, who reviewed the paper; Giacomo also showed us how to make a nice and tastefully coloured rendering of source code in LaTeX.

References

1. M. Aardal and D. Aranha, *2DT-GLS: Faster and exception-free scalar multiplication in the GLS254 binary curve*, `https://eprint.iacr.org/2022/748`
2. D. Aranha, J. López and D. Hankerson, *Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets*, Progress in Cryptology - LATINCRYPT 2010, Lecture Notes in Computer Science, vol. 6212, pp. 144-161, 2010.
3. D. Bernstein, T. Lange and R. Rezaeian Farashahi, *Binary Edwards Curves*, Cryptographic Hardware and Embedded Systems - CHES 2008, Lecture Notes in Computer Science, vol. 5154, pp. 244-265, 2008.
4. R. Brent, P. Gaudry, E. Thomé and P. Zimmerman, *Faster multiplication in GF(2)[x]*, Algorithmic Number Theory - ANTS-VIII, Lecture Notes in Computer Science, vol. 5011, pp. 153-166, 2008.
5. H. Brunner, A. Curriger and M. Hofstetter, *On computing multiplicative inverses in GF(2^m)*, IEEE Transactions on Computers, vol. 42, issue 8, pp. 1010-1015, 1993.
6. Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*, version 2.0, 2010, `https://www.secg.org/sec2-v2.pdf`
7. H. Cohen and G. Frey (editors), *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Chapman and Hall/CRC, 2006.
8. A. Fog, *The microarchitecture of Intel, AMD, and VIA CPUs*, `https://www.agner.org/optimize/microarchitecture.pdf`
9. P. Gaudry, F. Hess and N. Smart, *Constructive and destructive facets of Weil descent on elliptic curves*, Journal of Cryptology, vol. 15, issue 1, pp. 19-46, 2002.

10. D. Hankerson, K. Karabina and A. Menezes, *Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields*, IEEE Transactions on Computers, vol. 58, issue 10, pp. 1411-1420, 2009.
11. D. Hankerson, A. Menezes and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer/Sci-Tech/Trade, 2004.
12. T. Itoh and S. Tsujii, *A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases*, Information and Computation, vol. 78, pp. 171-177, 1988.
13. А. Карацуба and Ю. Офман, *Умножение Многозначных Чисел На Автоматах*, Доклады Академи и наук СССР, vol. 145, pp. 293-294, 1962.
14. A. Langley, M. Hamburg and S. Turner, *Elliptic Curves for Security*, https://www.rfc-editor.org/rfc/rfc7748
15. J. López and R. Dahab, *Fast Multiplication on Elliptic Curve Over $GF(2^m)$ without precomputation*, Cryptographic Hardware and Embedded Systems - CHES'99, Lecture Notes in Computer Science, vol. 1717, pp. 316-327, 1999.
16. K. Nath and P. Sarkar, *Efficient 4-way Vectorizations of the Montgomery Ladder*, https://eprint.iacr.org/2020/378
17. Information Technology Laboratory, *Digital Signature Standard (DSS)*, National Institute of Standard and Technology, FIPS 186-4, 2013.
18. T. Oliveira, J. López-Hernández, D. Aranha and F. Rodríguez-Henríquez, *Two is the fastest prime: lambda coordinates for binary elliptic curves*, Journal of Cryptographic Engineering, vol. 4, issue 1, pp. 3-17, 2014.
19. T. Oliveira, J. López-Hernández, D. Aranha and F. Rodríguez-Henríquez, *Improving the performance of the GLS254*, Presented at the CHES 2016 Rump Session.
20. C. Petit and J.-J. Quisquater, *On Polynomial Systems Arising from a Weil Descent*, Advances in Cryptology - ASIACRYPT 2012, Lecture Notes in Computer Science, vol. 7658, pp. 451-466, 2012.
21. T. Pornin, *BearSSL: a smaller SSL/TLS library*, https://www.bearssl.org/
22. T. Pornin, *Double-Odd Elliptic Curves*, https://eprint.iacr.org/2020/1558
23. J. Silverman, *The Arithmetic of Elliptic Curves* (2nd edition), Springer New York, NY, 2009.
24. J. Solinas, *Efficient Arithmetic on Koblitz Curves*, Designs, Codes and Cryptography, vol. 19, issue 2-3, pp. 195-249, 2000.
25. L. Washington, *Elliptic Curves: Number Theory and Cryptography* (2nd edition), Chapman and Hall/CRC, 2008.