Faster Repeated Doublings on Binary Elliptic Curves

Christophe Doche and Daniel Sutantyo*

Department of Computing, Macquarie University, Australia
<christophe.doche, daniel.sutantyo>@mq.edu.au

Abstract. The use of precomputed data to speed up a cryptographic protocol is commonplace. For instance, the owner of a public point P on an elliptic curve can precompute various points of the form $[2^k]P$ and transmit them together with P. One inconvenience of this approach though may be the amount of information that needs to be exchanged. In the situation where the bandwidth of the transmissions is limited, this idea can become impractical. Instead, we introduce a new scheme that needs only one extra bit of information in order to efficiently and fully determine a point of the form $[2^k]P$ on a binary elliptic curve. It relies on the *x-doubling* operation, which allows to compute the point $[2^k]P$ at a lower cost than with k regular doublings. As we trade off regular doublings for x-doublings, we use multi-scalar multiplication techniques, such as the Joint Sparse Form or interleaving with NAFs. This idea gives rise to several methods, which are faster than Montgomery's method in characteristic 2. A software implementation shows that our method called x-JSF$_2$ induces a speed-up between 4 and 18% for finite fields $\mathbb{F}_{2^d}$ with d between 233 and 571. We also generalize to characteristic 2 the scheme of Dahmen et al. in order to precompute all odd points $[3]P$, $[5]P, \ldots, [2t-1]P$ in affine coordinates at the cost of a single inversion and some extra field multiplications. We use this scheme with x-doublings as well as with the window NAF method in López–Dahab coordinates.

Keywords. Public key cryptography, elliptic curves, scalar multiplication.

1 Introduction

We refer readers to [28] for a general introduction to elliptic curves. An ordinary elliptic curve E defined over the finite field $\mathbb{F}_{2^d}$ can always be written with an equation of the form

$$E : y^2 + xy = x^3 + a_2x^2 + a_6, \text{ with } a_2 \in \{0, 1\}, a_6 \in \mathbb{F}_{2^d}^*. \tag{1}$$

This is an example of a curve in Weierstraß form. A point $P = (x_1, y_1)$ satisfying (1) is an *affine point* and the set of rational points of E, denoted by $E(\mathbb{F}_{2^d})$, corresponds to

$$E(\mathbb{F}_{2^d}) = \{(x_1, y_1) \in \mathbb{F}_{2^d} \times \mathbb{F}_{2^d} \mid y_1^2 + x_1y_1 = x_1^3 + a_2x_1^2 + a_6\} \cup P_\infty,$$

* This work was partially supported by ARC Discovery grant DP110100628.

where P_∞ is a special point called the *point at infinity*. The set $E(\mathbb{F}_{2^d})$ can be endowed with an abelian group structure under a point addition operation, denoted by $+$, with P_∞ as the identity element. Naturally, this addition leads to the *scalar multiplication*

$$[n]P = \underbrace{P + P + \cdots + P}_{n \text{ times}},$$

for an integer $n \geqslant 1$ and a point $P \in E(\mathbb{F}_{2^d})$. Given n and P, there are very efficient techniques to compute $[n]P$, see Section 2.3. But the converse problem, known as the *Elliptic Curve Discrete Logarithm Problem* (ECDLP), appears to be intractable for a well chosen elliptic curve defined over $\mathbb{F}_{2^d}$ with d prime. Therefore the security of many protocols in elliptic curve cryptography relies on the hardness of the ECDLP. This makes scalar multiplication the most ubiquitous operation in any elliptic curve based protocols. See [3, 14, 5, 6] for further discussions specific to the use of elliptic curves in cryptography.

2 State of the Art

2.1 Affine and López–Dahab Coordinates

Consider the elliptic curve E as in (1) and the point $P = (x_1, y_1) \in E(\mathbb{F}_{2^d})$. The double $P_2 = [2]P$ with coordinates (x_2, y_2) can be obtained with one inversion, two multiplications and one squaring. We abbreviate this as I+2M+S. There are similar formulas to compute the addition of two points at the cost of $\mathrm{I} + 2\mathrm{M} + \mathrm{S}$ as well. In $\mathbb{F}_{2^d}$, represented by a polynomial basis, a squaring involves only a reduction. Thus it costs much less than a generic multiplication. Computing the inverse of an element $\alpha \in \mathbb{F}_{2^d}$ is more complex. We rely usually on the Extended Euclidean gcd algorithm or on the relation $\alpha^{-1} = \alpha^{2^d-2}$. In software, an inversion can be cheaper than 10M, whereas the same operation can be extremely time consuming on platforms such as embedded devices. This explains why an alternate representation of E in projective coordinates is often considered. The equation

$$Y^2 + XYZ = X^3Z + a_2X^2Z^2 + a_6Z^4 \tag{2}$$

is an homogenized version of the equation E where the point P is represented in projective-like *López–Dahab* (LD) coordinates [17] by the triple $(X_1 : Y_1 : Z_1)$. When $Z_1 = 0$, the point P is the point at infinity whereas for $Z_1 \neq 0$, P corresponds to the affine point $(X_1/Z_1, Y_1/Z_1^2)$. The complexity of a doubling in LD coordinates is $3\mathrm{M} + 1 \times a_6 + 5\mathrm{S}$, including one multiplication by the fixed element a_6.

The addition of two points requires $13\mathrm{M} + 4\mathrm{S}$ in general, but only $8\mathrm{M} + 5\mathrm{S}$ in case at least one of the points is in affine coordinates, or in other words its Z-coordinate is equal to 1. This operation, that is very useful in practice, is referred to as a *mixed addition* [8]. See [4] for all the corresponding formulas.

2.2 Decompression techniques in LD Coordinates

In the following, we show how the x-coordinate of a point together with one extra bit is enough to fully recover the point in affine coordinates. Indeed, let us consider a point P that is not the point at infinity nor a point of order 2 and which is represented by $(X_1 : Y_1 : Z_1)$ in LD coordinates. Assuming that we ignore Y_1 but know X_1 and Z_1 as well as the last bit of $Y_1/(X_1Z_1)$, let us see how to determine the affine coordinates x_1 and y_1 of P. Before we start, note that $Y_1/(X_1Z_1)$ is equal to y_1/x_1, so it is independent of the choice of the Z-coordinate of P in LD format.

Since P is on (2), it follows that Y_1 is a root of the quadratic equation

$$T^2 + TX_1Z_1 = X_1^3Z_1 + a_2X_1^2Z_1^2 + a_6Z_1^4.$$

The other root is $Y_1 + X_1Z_1$. The two solutions correspond to the points P and $-P$. Because of the assumption on P, we have $X_1Z_1 \neq 0$ so the change of variable $U = T\alpha$ where $\alpha = 1/(X_1Z_1)$ is valid and leads to the new equation

$$U^2 + U = \beta \tag{3}$$

with $\beta = \alpha X_1^2 + a_2 + \alpha^2 a_6 Z_1^4$. To solve this equation let us introduce the half-trace function H defined by

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

When d is odd, it is well known that the solutions of equation (3) are $H(\beta)$ and $H(\beta) + 1$.

Clearly, we have $x_1 = \alpha X_1^2$. Now, let us explain how to find the correct value of y_1. It is easy to see that the solutions of (3) correspond to y_1/x_1 and $y_1/x_1 + 1$. So we can use the least significant bit of a root to identify the correct y_1. Let b be the last bit of y_1/x_1, then if the last bit of $H(\beta)$ is equal to b, set $y_1 = H(\beta)x_1$, otherwise $y_1 = (H(\beta)+1)x_1$. The public point P can therefore be represented as (x_1, b) and we can fully determine P in affine coordinates with $\mathrm{I} + \mathrm{H} + 4\mathrm{M} + 1 \times \sqrt{a_6} + 2\mathrm{S}$, where H is the complexity to evaluate the half-trace function.

As the half-trace function is linear, its computation can be sped up significantly provided that there is enough memory to store some precomputed values. With those enhancements, we have in general $\mathrm{H} \sim \mathrm{M}$. See [15, 14, 2] for details.

Next, we review how to compute $[n]P$. Note that throughout the paper the coordinates of $[n]P$ are denoted by (x_n, y_n) or $(X_n : Y_n : Z_n)$.

2.3 Classical Scalar Multiplication Techniques

The simplest, yet efficient, way to perform a scalar multiplication $[n]P$ is the *double and add* method, which is a straightforward adaptation of the square and multiply algorithm used to compute an exponentiation. Given the binary representation of n, denoted by $(n_{\ell-1} \dots n_0)_2$, a doubling is performed at each

step followed by an addition if $n_i = 1$. The double and add method is therefore intrinsically linked to the binary representation of n. This is no surprise as the method used to perform the scalar multiplication $[n]P$ is often related to the representation of the integer n. Other choices are available to represent n, for instance in base 2^k for a fixed parameter k, or signed digits. A signed-digit expansion for an integer n is of the form

$$n = \sum_{i=0}^{\ell-1} c_i 2^i, \text{ with } c_i \in S,$$

where S is a finite set of acceptable coefficients. This is particularly interesting as a negative coefficient $-c$ in the representation of n induces the use of the point $-[c]P$ in the computation of $[n]P$. Note that $-[c]P$ can be obtained virtually for free from the point $[c]P$ in affine coordinates. The *Non-Adjacent Form* (NAF) [21, 20] is especially relevant as the density of the expansion, i.e. the number of nonzero terms divided by the total length is equal to $\frac{1}{3}$ on average. Also for a given n, the NAF has the lowest density among all signed-digit expansions of n with coefficients in $\{-1, 0, 1\}$. A generalization of the NAF, called *window NAF of size* w [18, 23, 27] and denoted by NAF_w achieves an average density equal to $\frac{1}{w+1}$ for the set S of digits containing 0 and all the odd integers strictly less than 2^{w-1} in absolute value. For maximal efficiency, the points $[c]P$ are precomputed in affine coordinates for all the positive c in S. See [11] or [14] for more details on NAF and window NAF expansions.

It is often required, for instance during the verification phase of the ECDSA signature protocol, to perform a double-scalar multiplication of the form $[n_1]Q_1 + [n_0]Q_0$. Instead of computing each scalar multiplication separately, it is more efficient to combine the binary expansions of n_1 and n_0 in

$$\binom{n_1}{n_0} = \binom{u_{k-1} \dots u_0}{v_{k-1} \dots v_0}.$$

Mimicking the double and add method, we process a sequence of joint-bits, instead of a sequence of bits. At each step, a doubling is then followed by an addition of Q_1, Q_0, or $Q_1 + Q_0$ depending on the value of the joint-bit that is considered, i.e. $\binom{1}{0}$, $\binom{0}{1}$, or $\binom{1}{1}$. This idea, often attributed to Shamir is in fact a special case of an idea of Straus [26]. Mixing this idea with signed-digit representations gives birth to the *Joint Sparse Form* (JSF) [24] that can be seen as a generalization of the NAF for double-scalar multiplication. Indeed, the joint-density of the JSF is equal to $\frac{1}{2}$ on average and is optimal across all the joint signed-digit representations of n_1 and n_0. Note that the points Q_1, Q_0, $Q_1 + Q_0$, and $Q_1 - Q_0$ must be precomputed in affine coordinates for maximal efficiency. The JSF method is the standard way for computing a double-scalar multiplication when both points are not known in advance or when the amount of memory available on the device performing the computation is limited and does not allow the use of more precomputed values. Otherwise, *interleaving with NAFs* [14, Algorithm 3.51] gives excellent results. The principle of this approach

is simply to form

$$\binom{n_1}{n_0} = \binom{v_{k-1} \dots v_0}{u_{k-1} \dots u_0}$$

where $(v_{k-1} \dots v_0)$ and $(u_{k-1} \dots u_0)$ are the window NAF expansions of n_1 and n_0, possibly padded with a few zeroes at the beginning. Note that we precompute only the points $[3]Q_i$, $[5]Q_i, \dots, [2^w - 1]Q_i$ for $i = 0$ and 1 and not all the combinations $[2s+1]Q_1 \pm [2t+1]Q_0$ as this option is too costly in most situations. It remains to compute all the doublings together and then perform at most two mixed additions at each step. Obviously, it is easy to generalize this idea to efficiently compute the $k + 1$ scalar multiplications $[n_k]Q_k + \dots + [n_0]Q_0$ simultaneously provided that the points $[3]Q_i$, $[5]Q_i$, $\dots$, $[2^w - 1]Q_i$, for $i = 0, \dots, k$ are all precomputed.

2.4 Fixed Point Scalar Multiplication Using Precomputations

There are faster scalar multiplications techniques when the point P is known in advance and when some precomputations are available. For instance, we could considerably reduce the number of doublings, if not totally avoid them, by considering precomputed points of the form $[2^{ki}]P$ for a fixed k. Three methods, namely the Euclidean, Yao, and fixed-base comb, make use of this space-time trade-off to greatly reduce the complexity of a scalar multiplication. See [11, §9.3] for a presentation of those different methods.

If we consider a specific protocol such as ECDSA, Antipa et al. [1] show how to speed up the signature verification process by introducing the precomputed multiple $[2^k]P$ of the public point P, with $k = \lceil \ell/2 \rceil$. Tests show that the speed-up is significant, more than 35% of the time saved, but this approach shares a drawback with the techniques discussed so far: it requires to transmit an important amount of additional data on top of the public point P.

When exchanging large volume of data is not practical, for instance because the bandwidth of the network is limited, the methods described in this part do not apply, even if the point P is known in advance. Instead, Montgomery's method is perfectly suited, as it allows to perform arithmetic on an elliptic curve using only the x-coordinate of P.

2.5 Montgomery's Method

Montgomery developed an efficient arithmetic for special elliptic curves defined over a prime field [19]. Ultimately, it relies on the possibility to derive the x-coordinate of $P + Q$ from the x-coordinates of P, Q, and $P - Q$. This approach has been generalized to binary curves by López and Dahab [16]. Indeed, starting from P in LD coordinates and assuming that we know at each step the X and Z coordinates of $[u]P$ and of $[u + 1]P$, we can determine the X and Z coordinates of $[2u]P$ and $[2u + 1]P$. See [16, 25] for the actual formulas. The full scalar multiplication $[n]P$ can then be computed with Montgomery's ladder, which requires an addition and a doubling at each step. If the square root of a_6 is

precomputed, the complexity of the scheme is $(5\mathrm{M} + 1 \times \sqrt{a_6} + 4\mathrm{S})$ per bit of the scalar.

3 Applications of x-doublings to Scalar Multiplication

We have just seen that the x-coordinate of $[2]P$ only depends on the x-coordinate of the point P itself. Namely, if $P = (X_1 : Y_1 : Z_1)$, we have $[2]P$ given by $X_2 = X_1^4 + a_6 Z_1^4 = (X_1^2 + \sqrt{a_6} Z_1^2)^2$ and $Z_2 = X_1^2 Z_1^2$. In the following, we refer to this operation as an *x-doubling*. We can compute an x-doubling with $\mathrm{M} + 1 \times \sqrt{a_6} + 3\mathrm{S}$. This means that an x-doubling saves $2\mathrm{M} + 2\mathrm{S}$ compared to a regular doubling in LD coordinates. A doubling in LD coordinates costs $3\mathrm{M} + 1 \times a_6 + 5\mathrm{S}$ in the same conditions. From now on, we assume that $\sqrt{a_6}$ is precomputed and does not enjoy any special property so that $1 \times \sqrt{a_6} = 1\mathrm{M}$.

We propose to speed up the scalar multiplication $[n]P$ by replacing some regular LD doublings with x-doublings. To take full advantage of this operation, our idea is to determine the x-coordinate of $[2^k]P$ using k successive x-doublings and then recover the y-coordinate using decompression techniques and one extra known bit of information.

3.1 Double Scalar Multiplication and x-Doublings

Let ℓ be the bit size of the order of a point P and $k = \lfloor \ell/2 \rfloor$. Let (x_1, b) be the public information corresponding to the point P, as explained in Section 2.2. We assume that the owner of the point P has also precomputed b_k, i.e. the last bit of y_{2^k}/x_{2^k}, and made it public.

Let n be an integer of size ℓ bits. It is clear that $[n]P = [n_1]Q_1 + [n_0]Q_0$ where $n = 2^k n_1 + n_0$, $Q_1 = [2^k]P$, and $Q_0 = P$. The X and Z coordinates of Q_1 can be obtained with k straight x-doublings at a cost of $(2\mathrm{M}+3\mathrm{S})k$. Then we can recover the y-coordinates of Q_0 and Q_1 using b and b_k with decompression techniques. For that, we compute $\beta = \alpha x_1^2 + a_2 + a_6 \alpha^2$ and $\beta_k = \alpha_k X_{2^k}^2 + a_2 + a_6 \alpha_k^2 Z_k^4$ where $\alpha = 1/x_1$ and $\alpha_k = 1/(X_{2^k} Z_{2^k})$. Montgomery's trick [19] allows to obtain α and α_k with only one inversion and three extra multiplications. Then we compute the half-trace of β and β_k and identify the correct root with the bits b and b_k. Assuming that a_6 is a random element of $\mathbb{F}_{2^d}$ and that $\sqrt{a_6}$ has been precomputed, the complexity to determine Q_0 and Q_1 in affine coordinates is I+2H+12M+4S. This approach can easily be generalized to retrieve an arbitrary number of points. It then takes $\mathrm{I} + t\mathrm{H} + (7t - 2)\mathrm{M} + 2t\mathrm{S}$ to fully determine Q_0, $Q_1, \ldots, Q_{t-1}$ in affine coordinates once the X and Z coordinates of those points have been obtained using x-doublings.

Now that we have Q_0 and Q_1 in affine coordinates, we can compute $[n]P = [n_1]Q_1 + [n_0]Q_0$ with any standard double scalar multiplication technique, for instance the JSF, see Section 2.3. In this case, the complexity is approximately $\ell/2$ x-doublings, $\ell/2$ regular doublings, and $\ell/4$ mixed additions using the four points Q_0, Q_1, $Q_0 + Q_1$, and $Q_0 - Q_1$. Note that $Q_0 + Q_1$ and $Q_0 - Q_1$ should be computed simultaneously with $\mathrm{I} + 4\mathrm{M} + 2\mathrm{S}$.

This new approach, called x-JSF, compares favorably against the window NAF_w method in LD coordinates and Montgomery's approach. Both are very popular methods in practice for a wide range of extension degrees when only the x-coordinate of the point P is transmitted. The x-JSF requires essentially $5\mathrm{M} + 5.25\mathrm{S}$ per bit compared to $6\mathrm{M} + 4\mathrm{S}$ for Montgomery's method. Adding the complexities of all the steps involved, including the cost of precomputations, and assuming that $\mathrm{H} = \mathrm{M}$, we obtain the following result.

Proposition 1 *Let n be an integer of binary length ℓ. The average complexity of the x-JSF method to compute the scalar multiplication $[n]P$ is*

$$\big(5\mathrm{M} + \tfrac{21}{4}\mathrm{S}\big)\ell + 3\mathrm{I} + 20\mathrm{M} + 7\mathrm{S}.$$

Proof. Starting from the x-coordinate of P, we do $\frac{\ell}{2}$ consecutive x-doublings with $\big(\mathrm{M} + \frac{3}{2}\mathrm{S}\big)\ell$. We need $\mathrm{I} + 12\mathrm{M} + 4\mathrm{S}$ to form the two quadratic equations and $2\mathrm{M}$ to solve them in order to obtain P and $[2^{\lfloor \ell/2 \rfloor}]P$ in affine coordinates. We then need $\mathrm{I} + 4\mathrm{M} + 2\mathrm{S}$ to compute the sum and difference of those two points in affine coordinates again. Then, we perform $\frac{\ell}{2}$ regular doublings, with $\big(2\mathrm{M} + \frac{5}{2}\mathrm{S}\big)\ell$, and $\frac{\ell}{4}$ mixed additions on average, with $\big(2\mathrm{M} + \frac{5}{4}\mathrm{S}\big)\ell$. Finally, we express the point in affine coordinates with $\mathrm{I} + 2\mathrm{M} + \mathrm{S}$.

3.2 Trading Off More Doublings for x-Doublings

Previously, we explained how to replace approximately 50% of normal doublings by x-doublings. Since an x-doubling is significantly cheaper than a doubling, it is natural to try to increase this ratio, i.e. replace more doublings with x-doublings. A simple idea is for instance to work with three shares instead of two, i.e. fix $k = \lfloor \ell/3 \rfloor$ and write $n = 2^{2k}n_2 + 2^k n_1 + n_0$. If we fix $Q_0 = P$, $Q_1 = [2^k]P$ and $Q_2 = [2^{2k}]P$, we see that $[n]P = [n_2]Q_2 + [n_1]Q_1 + [n_0]Q_0$. We denote by w-w-w the generalization of the interleaving with NAFs method, where the NAF_w expansions of n_2, n_1, and n_0 are stacked together as follows

$$\begin{pmatrix} n_2 \\ n_1 \\ n_0 \end{pmatrix} = \begin{pmatrix} w_k & \dots & w_0 \\ v_k & \dots & v_0 \\ u_k & \dots & u_0 \end{pmatrix}.$$

and processed from left to right using Straus' idea, see Section 2.3. The only points that we precompute are $[3]Q_i$, $[5]Q_i, \dots, [2^{w-1} - 1]Q_i$, for $i \in [0, 2]$.

Proposition 2 *Let n be an integer of binary length ℓ. The average complexity of the w-w-w interleaving with NAFs method to compute the scalar multiplication $[n]P$ with $w > 3$ is*

$$\big((8w + 32)\mathrm{M} + (11w + 26)\mathrm{S}\big)\frac{\ell}{3(w+1)} + 3 \times 2^{w-2}(\mathrm{I} + 2\mathrm{M} + \mathrm{S}) + 2\mathrm{I} + 24\mathrm{M} + 7\mathrm{S}.$$

For $w = 3$, the complexity is simply

$$\big(\tfrac{14}{3}\mathrm{M} + \tfrac{59}{12}\mathrm{S}\big)\ell + 5\mathrm{I} + 42\mathrm{M} + 19\mathrm{S}.$$

Proof. The proof is similar to Proposition 1. It is clear that we need $2\frac{\ell}{3}$ successive x-doublings and $\frac{\ell}{3}$ normal doublings, plus a certain number of additions, which depends on the size of the window w. The exact number can be derived via some probabilistic analysis. Given that the density of the NAF_w is $\frac{1}{w+1}$, we may assume that the probability for a coefficient in a long expansion to be nonzero is $\frac{1}{w+1}$. It follows that the w-w-w interleaving method requires $\frac{\ell}{w+1}$ mixed additions on average. To determine the precomputations, we could compute $[2]Q_0$, $[2]Q_1$, and $[2]Q_2$ in LD coordinates, make those three points affine simultaneously, before performing $3(2^{w-2}-1)$ LD mixed additions, and converting all the resulting points to affine simultaneously again. This approach only needs two inversions. Instead, in our implementation as well as in this analysis, we perform all the computations directly in affine coordinates for simplicity. The formula follows by adding all the different contributions.

The complexity is very low, but in practice the main drawback of this approach is the number of precomputations, which grows as $3 \times 2^{w-2}$. Another method along those lines, called x-JSF_2, requires less storage and involves splitting n in four shares as $2^{3k}n_3+2^{2k}n_2+2^k n_1+n_0$ with $k = \lfloor \ell/4 \rfloor$. The points $Q_3 = [2^{3k}]P$, $Q_2 = [2^{2k}]P$, $Q_1 = [2^k]P$, $Q_0 = P$ are determined after $3\frac{\ell}{4}$ straight x-doublings. Four extra bits of information are necessary to fully recover the points. Then we compute the JSF expansions of n_3 and n_2 and of n_1 and n_0, together with the precomputed affine points $Q_3 \pm Q_2$, and $Q_1 \pm Q_0$. We then need $\frac{\ell}{4}$ regular LD doublings as well as $\frac{\ell}{4}$ mixed additions. The following result follows immediately.

Proposition 3 *Let n be an integer of binary length ℓ. The average complexity of the x-JSF_2 method to compute the scalar multiplication $[n]P$ is*

$$\left(\tfrac{9}{2}\mathrm{M} + \tfrac{19}{4}\mathrm{S}\right)\ell + 4\mathrm{I} + 40\mathrm{M} + 13\mathrm{S}.$$

Proof. We perform $3\frac{\ell}{4}$ x-doublings, then need $\mathrm{I} + 30\mathrm{M} + 8\mathrm{S}$ to recover the four points Q_0, Q_1, Q_2, and Q_3 in affine coordinates. We need $2(\mathrm{I} + 4\mathrm{M} + 2\mathrm{S})$ to compute $Q_3 \pm Q_2$ and $Q_1 \pm Q_0$. Then, we perform $\frac{\ell}{4}$ regular doublings and $\frac{\ell}{4}$ mixed additions on average. Finally, we express the point in affine coordinates with $\mathrm{I} + 2\mathrm{M} + \mathrm{S}$.

Tests show that the x-JSF_2 achieves a speed-up close to 18% over the fastest known method in $\mathbb{F}_{2^{571}}$, i.e. Montgomery's method. See Section 5 for details.

3.3 Trading Off Even More Doublings for x-Doublings

In some sense, the x-JSF_2 relies on interleaving with JSFs. Generalizing this idea, the x-JSF_t uses 2^t shares, each of size $\ell/2^t$ bits, and needs $\frac{2^t-1}{2^t}$ x-doublings and $\frac{1}{2^t}$ regular doublings. Arranging the scalars two by two, and computing their corresponding JSF expansions, we see that $\frac{\ell}{4}$ mixed additions are necessary on average, provided that we precompute 2^{t-1} pairs of points of the form $Q_{2i+1} \pm Q_{2i}$.

To further reduce the number of regular doublings without using precomputations, we turn our attention to a method first described by de Rooij, but credited to Bos and Coster [10]. As previously, write n in base 2^k, for a well chosen k. It follows that $[n]P = [n_{t-1}]Q_{t-1} + \cdots + [n_0]Q_0$, where $Q_i = [2^{ki}]P$.

The main idea of the Bos–Coster method is to sort the shares in decreasing order according to their coefficients and to recursively apply the relation

$$[n_1]Q_1 + [n_2]Q_2 = [n_1](Q_1 + [q]Q_2) + [n_2 - qn_1]Q_2$$

where $n_2 > n_1$ and $q = \lfloor n_2/n_1 \rfloor$. The process stops when there is only one nonzero scalar remaining. Because the coefficients are roughly of the same size throughout the process, we have $q = 1$ at each step almost all the time. This implies that $[n]P$ can be computed almost exclusively with additions once the shares Q_i's are obtained via x-doublings.

Clearly, this approach requires $k(t-1)$ successive x-doublings and t point reconstructions. The precise number of additions involved is much harder to analyze but can be approximated by $\frac{\ell}{\log k}$ for reasonable values of k. So, with the Bos–Coster method, we have replaced almost all the doublings by x-doublings, but one detail plays against us. Most of the additions that we need are full additions and not mixed additions as it was the case for the x-JSF and interleaving with NAFs. Indeed, even if the different points $Q_0, \ldots, Q_{t-1}$ are initially expressed in affine coordinates, then after a few steps of the algorithm, it is no longer the case and subsequent additions need to be performed in full. Those full additions are too expensive to make this scheme competitive with Montgomery's method.

Next, we investigate Yao's method [29]. As for the previous approach, we express n in base 2^k as $(n_{t-1} \ldots n_0)_{2^k}$ and we consider the points $Q_i = [2^{ki}]P$, for $i = 0, \ldots, t-1$, obtained with x-doublings only. Note that we can rewrite the sum $[n]P = [n_{t-1}]Q_{t-1} + \cdots + [n_0]Q_0$ as

$$[n]P = \sum_{j=1}^{2^k-1} [j]\Big(\sum_{n_i=j} Q_i\Big).$$

We deduce the following algorithm. Starting from $T = P_\infty$, $R = P_\infty$, and $j = 2^k - 1$, we repeat $R = R + Q_i$ for each i such that $n_i = j$, followed by $T = T + R$ and $j = j - 1$ until $j = 0$.

To update R, we use mixed additions, but the statement $T = T + R$ requires a full LD addition. Therefore, the complexity of this approach is essentially $k(t-1)$ successive x-doublings, t point reconstructions, $\big(1 - \frac{1}{2^k}\big)2^k$ mixed additions on average, and 2^k full additions. In order to minimize the number of full additions, we need to keep k low, which means increasing t. Unfortunately, it is quite expensive to retrieve a point in affine coordinates from its X, Z-coordinates and the last bit of y/x. As explained in Section 3.1, we need H + 7M + 2S per additional point Q_i to fully recover it in affine coordinates. This proves to be too much and all the parameters k that we tried failed to introduce any improvement over Montgomery's method.

3.4 Generic Protocol Setup Compatible with x-doublings

The purpose of this article is to evaluate the relevance of x-doublings to perform a scalar multiplication, not to precisely describe how to use this operation in a specific protocol. However, it seems that the most realistic setup to use one of the schemes presented in Sections 3.1 and 3.2 is for the owner of a point P to precompute and store the last bit b_k of y_{2^k}/x_{2^k}, for all $k \in [0, d+1]$. This of course is done only once at the very beginning and does not affect the security of the scheme. Since P is public, anybody can perform the computations and retrieve those bits. The other party can then access x_1, the x-coordinate of P, as well as a few bits b_k. At most four bits are sufficient to deliver a significant speed-up with the x-JSF$_2$ approach, see Section 5. The choice of those bits does not reveal anything on the scalar n except maybe its size, which we do not see as a problem.

4 Affine Precomputations with Sole Inversion in Char 2

The other contribution of this paper is a generalization of the work of Dahmen et al. [9] to precompute all the affine points required by the NAF$_w$ method with just one inversion. Indeed, starting from the affine point P, Dahmen et al. show how to obtain $[3]P$, $[5]P$,..., $[2t-1]P$ also in affine coordinates with $\mathrm{I} + (10t - 11)\mathrm{M} + 4t\mathrm{S}$. But their work only addresses the case of large odd characteristic.

With a generalized scheme, we can precompute all the points necessary for the w-w-w interleaving with NAFs method with just three inversions instead of $3 \times 2^{w-2}$, using affine arithmetic. We mimic their approach and follow three easy steps. First, we compute all the denominators involved. Then we apply Montgomery's inversion trick [19] that combines j inversions in $\mathbb{F}_{2^d}$ at the expense of one inversion and $3j - 3$ field multiplications. Finally, we reconstruct all the points. The total complexity to compute $[2]P$, $[3]P$, $[5]P$, ..., $[2t-1]P$ for $t > 2$ is $\mathrm{I} + (11t - 13)\mathrm{M} + 2t\mathrm{S}$. See the Appendix for the actual algorithm. When we only need $[3]P$, for instance for the NAF$_3$, we compute $[3]P$ directly following the approach explained in [7]. Note however that $\mathrm{I} + 6\mathrm{M} + 4\mathrm{S}$ are enough to determine $[3]P$, saving one multiplication.

5 Tests and Results

To validate the use of the x-doubling operation and the methods described in Section 3, we have implemented all of them in `C++` using `NTL 6.0.0` [22] built on top of `GMP 5.1.2` [13]. The program is compiled and executed on a `quad core i7-2620` at `2.70Ghz`.

In the following, we test some of the techniques described in Sections 2 and 3 to perform a scalar multiplication on a random curve defined over $\mathbb{F}_{2^d}$ for $d =$ 233, 409, and 571. Namely, we compare the following methods: Montgomery (Mont.), window NAF in LD coordinates with $w \in [2, 6]$ (NAF$_w$), x-JSF, x-JSF$_2$,

and interleaving with NAFs (w-w-w). Note that $\underline{\text{NAF}}_w$ and $\underline{w\text{-}w\text{-}w}$ are slightly different variants where the precomputations are obtained with the sole inversion technique, explained in Section 4. We generate a total of 100 curves of the form

$$E : y^2 + xy = x^3 + x^2 + a_6,$$

where a_6 is a random element of $\mathbb{F}_{2^d}^*$. For each curve, a random point P is created as well as 100 random scalars selected in the interval $[0, 2^d + 2^{d/2} - 1]$. We assume that the point P needs to be decompressed for all the methods. The different methods are then tested against the same curves, points, and scalars. The computations are timed over 10 repetitions.

Degree 233: $\mathrm{I}_{233}/\mathrm{M}_{233} = 8.651$ and $\mathrm{S}_{233}/\mathrm{M}_{233} = 0.226$							
	$\#\mathcal{P}$	I_{233}	H_{233}	M_{233}	S_{233}	Time (ms)	Speed-up
Mont.	1	2	0	1402	928	1.102	0%
NAF_5	8	10	0	1253	1360	1.221	−10.81%
$\underline{\text{NAF}}_5$	8	3	0	1312	1368	1.241	−12.66%
x-JSF	4	3	2	1181	1229	1.118	−1.51%
x-JSF_2	8	4	4	1094	1126	1.053	4.43%
4-4-4	12	14	3	1043	1108	1.079	2.05%
$\underline{3\text{-}3\text{-}3}$	6	5	3	1143	1165	1.083	1.70%
Degree 409: $\mathrm{I}_{409}/\mathrm{M}_{409} = 9.289$ and $\mathrm{S}_{409}/\mathrm{M}_{409} = 0.140$							
	$\#\mathcal{P}$	I_{409}	H_{409}	M_{409}	S_{409}	Time (ms)	Speed-up
Mont.	1	2	0	2457	1631	4.289	−0.68%
NAF_5	8	10	0	2192	2386	4.267	−0.16%
$\underline{\text{NAF}}_5$	8	3	0	2251	2394	4.260	0%
x-JSF	4	3	2	2061	2153	3.928	7.79%
x-JSF_2	8	4	4	1885	1962	3.650	14.33%
4-4-4	12	14	3	1793	1929	3.667	13.94%
$\underline{4\text{-}4\text{-}4}$	12	5	3	1862	1941	3.752	11.94%
Degree 571: $\mathrm{I}_{571}/\mathrm{M}_{571} = 9.212$ and $\mathrm{S}_{571}/\mathrm{M}_{571} = 0.153$							
	$\#\mathcal{P}$	I_{571}	H_{571}	M_{571}	S_{571}	Time (ms)	Speed-up
Mont.	1	2	0	3430	2280	10.986	0%
NAF_6	16	18	0	2961	3269	12.154	−10.64%
$\underline{\text{NAF}}_6$	16	3	0	3092	3285	12.464	−13.46%
x-JSF	4	3	2	2871	3004	10.153	7.58%
x-JSF_2	8	4	4	2618	2735	9.014	17.94%
5-5-5	24	26	3	2355	2601	8.843	19.51%
$\underline{5\text{-}5\text{-}5}$	24	5	3	2532	2625	8.810	19.81%

Table 1. Comparison of different methods for degrees 233, 409, and 571

Together with the average timings of the best methods in each category, we present the average number of basic operations required to compute $[n]P$.

Those basic operations, i.e. inversion, half-trace computation, multiplication, and squaring in $\mathbb{F}_{2^d}$ are respectively represented by I_d, H_d, M_d, and S_d. In any case we have, $\mathrm{I}_d/\mathrm{M}_d$ between 8 and 10, and $\mathrm{S}_d/\mathrm{M}_d$ between 0.14 and 0.23.

See Table 1 for the actual figures, which features timings and operation counts of the x-JSF, the x-JSF$_2$, as well as the fastest interleaving with NAFs methods among w-w-w, for $w \in [2, 5]$ and among $\underline{w\text{-}w\text{-}w}$ again for $w \in [2, 5]$. Table 1 also includes the number of stored points required by each method ($\#\mathcal{P}$), and the improvement, if any, over Montgomery's method and the fastest window NAF method.

With our implementation, the x-JSF$_2$ breaks even with Montgomery's method around $d = 233$ and enjoys a much bigger speed-up for larger degrees, reflecting Proposition 3. The interleaving with NAFs method $\underline{5\text{-}5\text{-}5}$ is the fastest of all for $d = 571$, with a speed-up that is close to 20%.

Remark 1 *A careful reader would have noticed that for $d = 233$, $\underline{3\text{-}3\text{-}3}$ is faster than $\underline{4\text{-}4\text{-}4}$. This is surprising for two reasons. First, 4-4-4 is faster than 3-3-3 Second, it is more efficient, given the value of the ratio $\mathrm{I}_{233}/\mathrm{M}_{233}$, to determine the precomputations using the single inversion approach. So $\underline{4\text{-}4\text{-}4}$ should be faster. This is confirmed by an analysis of the average numbers of multiplications and squarings required. Indeed, we need $1112\mathrm{M} + 1120\mathrm{S}$ for $\underline{4\text{-}4\text{-}4}$, against $1143\mathrm{M} + 1165\mathrm{S}$ for $\underline{3\text{-}3\text{-}3}$. We observe the same phenomenon for degrees $d = 163$ and $d = 283$, but not for $d = 409$ or $d = 571$. We explain this by the large number of variables needed to determine the precomputations when $w > 3$. See the Appendix for details. For $w = 3$, the formulas are simpler, requiring much less intermediate storage. This overhead of declaring and manipulating extra variables tends to have less impact for larger degrees because multiplications and squarings take relatively longer.*

6 Conclusion and Future Work

We have shown how to make use of x-doublings to compute a scalar multiplication on a binary elliptic curve. Our main approach is to trade off regular doublings for cheaper x-doublings using classical multi-scalar multiplication techniques.

Unfortunately, it seems impossible to generalize the use of x-doublings in large characteristic, since solving quadratic equations is much slower than in characteristic 2.

A possible generalization of this work would be to investigate which endomorphisms different from doublings enjoy similar properties, i.e. have an x-coordinate that can be computed efficiently and independently from the y-coordinate. Certain endomorphisms $[k]P$ that can be split as the product of two isogenies on special families of curves are known to have this property [12]. It would be interesting to see what kind of improvements those endomorphisms could bring when it comes to computing a scalar multiplication.

7 Acknowledgments

We would like to thank Tanja Lange and Daniel J. Bernstein as well as the reviewers of this article for their numerous comments and suggestions, which greatly contributed to improve its contents.

References

Numbers in curly brackets at the end specify the pages where the citations occur.

1. A. Antipa, D. R. L. Brown, R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Accelerated Verification of ECDSA Signatures. In *Selected Areas in Cryptography, 12th Annual International Workshop, SAC 2005*, volume 3897 of *Lecture Notes in Comput. Sci.*, pages 307–318. Springer, 2005. {5}
2. R. M. Avanzi. Another look at square roots (and other less common operations) in fields of even characteristic. In *Selected Areas in Cryptography, 14th Annual International Workshop, SAC 2007*, volume 4876 of *Lecture Notes in Comput. Sci.*, pages 138–154. Springer, 2007. {3}
3. R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2005. {2, 13}
4. D. J. Bernstein and T. Lange. Explicit-formulas database. See `http://www.hyperelliptic.org/EFD/` {2}
5. I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1999. {2}
6. I. F. Blake, G. Seroussi, and N. P. Smart. *Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 2005. {2}
7. M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. Trading Inversions for Multiplications in Elliptic Curve Cryptography. *Des. Codes Cryptogr.*, 39(2):189–206, 2006. {10}
8. H. Cohen, A. Miyaji, and T. Ono. Efficient Elliptic Curve using Mixed Coordinates. In *Advances in Cryptography - Proceedings of ASIACRYPT 1998*, volume 1514 of *Lecture Notes in Comput. Sci.*, pages 51–65. Springer, 1998. {2}
9. E. Dahmen, K. Okeya, and D. Schepers. Affine Precomputation with Sole Inversion in Elliptic Curve Cryptography. In *Information Security and Privacy, 12th Australasian Conference, ACISP 2007*, volume 4586 of *Lecture Notes in Computer Science*, pages 245–258. Springer, 2007. {10}
10. P. de Rooij. Efficient Exponentiation using Precomputation and Vector Addition Chains. In *Advances in Cryptography - EUROCRYPT 1994*, volume 950 of *Lecture Notes in Computer Science*, pages 389–399. Springer, 1995. {9}
11. C. Doche. Exponentiation. In *[3]*, pages 145–168. 2005. {4, 5}
12. C. Doche, T. Icart, and D. R. Kohel. Efficient Scalar Multiplication by Isogeny Decompositions. In *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Comput. Sci.*, pages 191–206. Springer, 2006. {12}

13. Free Software Foundation. GNU Multiple Precision Library.
See `http://gmplib.org/` {10}
14. D. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, Berlin, 2003. {2, 3, 4}
15. E. W. Knudsen. Elliptic Scalar Multiplication Using Point Halving. In *Advances in Cryptography - Proceedings of ASIACRYPT 1999*, volume 1716 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999. {3}
16. J. López and R. Dahab. Fast Multiplication on Elliptic Curve over $GF(2^m)$ without Precomputation. In Ç. K. Koç and C.Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 1999*, volume 1717, pages 316–327. Springer, 1999. {5}
17. J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In *Advances in Cryptology - Proceedings of Selected Areas in Cryptography 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 1999. {2}
18. A. Miyaji, T. Ono, and H. Cohen. Efficient Elliptic Curve Exponentiation. In *Information and Communication - ICICS'97*, volume 1334 of *Lecture Notes in Comput. Sci.*, pages 282–291. Springer, 1997. {4}
19. P. L Montgomery. Speeding the Pollard and Elliptic Curves Methods of Factorisation. *Math. of Comp.*, 48:243–264, 1987. {5, 6, 10}
20. F. Morain and J. Olivos. Speeding up the Computations on an Elliptic Curve using Addition-Subtraction Chains. *Inform. Theor. Appl.*, 24:531–543, 1990. {4}
21. G. Reitwiesner. Binary arithmetic. *Adv. Comput.*, 1:231–308, 1962. {4}
22. V. Shoup. NTL: A Library for doing Number Theory.
See `http://www.shoup.net/ntl` {10}
23. J. A. Solinas. Improved Algorithms for Arithmetic on Anomalous Binary Curves. Technical Report CORR 99-46, CACR, 1999.
See `http://cacr.uwaterloo.ca/techreports/1999/corr99-46.pdf` {4}
24. J. A. Solinas. Low-weight binary representations for pairs of integers. Combinatorics and Optimization Research Report CORR 2001-41, University of Waterloo, 2001. {4}
25. M. Stam. On Montgomery-Like Representations for Elliptic Curves over $GF(2^k)$. In *Public Key Cryptography – PKC 2003*, Lecture Notes in Comput. Sci. Springer, 2003. {5}
26. E. G. Straus. Addition chains of vectors (problem 5125). *Amer. Math. Monthly*, 70:806–808, 1964. {4}
27. T. Takagi, S.-M. Yen, and B.-C. Wu. Radix-r non-adjacent form. In *Information Security Conference – ISC 2004*, volume 3225 of *Lecture Notes in Comput. Sci.*, pages 99–110. Springer, Berlin, 2004. {4}
28. L. C. Washington. *Elliptic Curves*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2003. Number theory and cryptography. {1}
29. A. C. C. Yao. On the evaluation of powers. *SIAM J. Comput.*, 5(1):100–103, 1976. {9}

Appendix: Affine Precomputations with Sole Inversion in Characteristic 2

Let $P = (x_1, y_1)$ be a point on the curve $y^2 + xy = x^3 + a_2x^2 + a_6$. For $t > 2$, the following procedure computes the points $[2]P$, $[3]P$, $[5]P, \ldots, [2t-1]P$ with $\mathrm{I} + (11t - 13)\mathrm{M} + 2t\mathrm{S}$.

`Step 1. Computing all the denominators` d_i`'s to be inverted`
$d_1 \leftarrow x_1, \quad s_1 \leftarrow d_1^2, \quad c_1 \leftarrow s_1 \cdot d_1, \quad n_1 \leftarrow s_1 + y_1$
$A \leftarrow n_1(n_1 + d_1), \quad d_2 \leftarrow A + (d_1 + a_2)s_1$
$B \leftarrow d_1 \cdot d_2, \quad C \leftarrow B + s_1^2, \quad n_2 \leftarrow n_1 \cdot d_2 + C$
$s_2 \leftarrow d_2^2, \quad A \leftarrow A \cdot s_2, \quad c_2 \leftarrow s_2 \cdot d_2, \quad d_3 \leftarrow A + n_2(n_2 + B) + c_2$
for $i = 3$ **to** $t - 1$ **do**
  $B \leftarrow B \cdot d_i, \quad C \leftarrow C \cdot c_{i-1}, \quad n_i \leftarrow n_{i-1} \cdot d_i + B + C$
  $s_i \leftarrow d_i^2, \quad A \leftarrow A \cdot s_i, \quad c_i \leftarrow s_i \cdot d_i, \quad d_{i+1} \leftarrow A + n_i(n_i + B) + c_i$
end for

`Step 2. Montgomery's inversion trick`
$B \leftarrow B \cdot d_t, \quad \mathrm{INV} \leftarrow B^{-1}, \quad e_1 \leftarrow c_1$
for $i = 2$ **to** $t - 1$ **do**
  $e_i \leftarrow e_{i-1} \cdot c_i$
end for

`Step 3. Reconstructing the points`
for $i = t$ **down to** 2 **do**
  $j_i \leftarrow \mathrm{INV} \cdot e_{i-1}, \quad \mathrm{INV} \leftarrow \mathrm{INV} \cdot d_i$
end for
$j_1 \leftarrow \mathrm{INV}$
$\lambda_2 \leftarrow j_1 \cdot n_1$
$x_2 \leftarrow \lambda_2^2 + \lambda_2 + a_2$
$y_2 \leftarrow \lambda_2(x_2 + x_1) + x_2 + y_1$
$\lambda_3 \leftarrow j_2(y_2 + y_1)$
$x_3 \leftarrow \lambda_3^2 + \lambda_3 + x_2 + x_1 + a_2,$
$y_3 \leftarrow \lambda_3(x_2 + x_3) + x_3 + y_2$
for $i = 4$ **to** $t + 1$ **do**
  $\lambda_{2i-3} \leftarrow j_{i-1}(y_2 + y_{2i-5})$
  $x_{2i-3} \leftarrow \lambda_{2i-3}^2 + \lambda_{2i-3} + x_2 + x_{2i-5} + a_2$
  $y_{2i-3} \leftarrow \lambda_{2i-3}(x_2 + x_{2i-3}) + x_{2i-3} + y_2$
end for