Binary Kummer Line

Sabyasachi Karati

School of Computer Sciences
National Institute of Science Education and Research
Bhubaneswar, India.
skarati@niser.ac.in

Abstract. In this work, we explore the problem of secure and efficient scalar multiplication on binary field using Kummer lines. Gaudry and Lubicz first introduced the idea of Kummer line in [12]. We investigate the possibilities of speedups using Kummer lines compared to binary Edwards curve and Weierstrass curves. Firstly, we propose a binary Kummer line BKL251 on binary field $\mathbb{F}_{2^{251}}$ where the associated elliptic curve satisfies the required security conditions and offers 124.5-bit security which is same as the BBE251 and CURVE2251. BKL251 also has small parameter and small base point. We implement the software of BKL251 using the instruction `PCLMULQDQ` of modern Intel processors. For fair comparison, we also implement the software BEd251 for binary Edwards curve introduced in [4] using the same field arithmetic library of the BKL251 and thus this work also complements the works of [7, 4]. In both the implementations, scalar multiplications take constant time which use Montgomery ladder. Binary Kummer line requires $4[\mathsf{M}]+5[\mathsf{S}]+1[\mathsf{C}]+1[\mathsf{B}]$ field operations for each ladder step where ladder step of binary Edwards curve requires $4[\mathsf{M}]+4[\mathsf{S}]+2[\mathsf{C}]+1[\mathsf{B}]$. Our experimental results show that fixed-base scalar multiplication of BKL251 is $8.36\%-9.33\%$ faster than that of BEd251. On the other hand, variable-base scalar multiplications take almost same time for both the curves (variable-base scalar multiplication of BKL251 is $0.25\%-1.55\%$ faster than that of BEd251).

Keywords: Binary Finite Field Arithmetic, Elliptic Curve Cryptography, Kummer Line, Edwards Curve, Montgomery Ladder, Scalar Multiplication.

1 Introduction

Elliptic curve cryptography provides a secure and efficient platform for public-key cryptography. Since the introduction of elliptic curve cryptography by Miller [21] and Koblitz [17] and hyperelliptic curve cryptography by Koblitz [18], this area of research remains an extremely important one. The security of elliptic curve cryptography is derived from the hardness of discrete logarithm problem on the underlying group of the elliptic curve. To ensure security, it remains an important question that which curve should be chosen for use. Choice of the curve normally starts with determination of the underlying field of the elliptic curve based on the applications: large characteristics field (characteristics greater than 3) or binary field.

It is a common practice to choose large characteristics field for software implementation and binary field for the hardware implementation. For a long time, the speed records of the fastest variable-base elliptic curve scalar multiplications were held by curves defined over large characteristics field such as:

- 59,000 clock cycles (`clks`) and 1,09,000 `clks` for finite field $\mathbb{F}_{(2^{127}-1)^2}$ by Four($\mathbb{Q}$) curve with and without endomorphism, respectively, on the Haswell architecture [9, 16],
- 1,56,060 `clks` for variable-base scalar multiplications by Curve25519 on finite field $\mathbb{F}_{2^{255}-19}$ on Haswell architecture [3, 16],

– 95,424 clks, 1,28,178 clks and 1,23,818 clks for large prime fields $\mathbb{F}_{2^{251}-9}$, $\mathbb{F}_{2^{255}-19}$ and $\mathbb{F}_{2^{266}-3}$ respectively by Kummer lines KL2519, KL25519 and KL2663 [16].

Curve25519 and Kummer lines (KL2519, KL25519 and KL2663) have small base points and as a consequence they achieve $8\% - 24\%$ faster fixed-base scalar multiplications compared to variable-base scalar multiplication. Curve25519, KL2519, KL25519 and KL2663 require only 1,44,224 clks, 93,020 clks, 1,16,832 clks and 1,19,564 clks for fixed-base scalar multiplication, respectively, without any precomputation on Haswell architecture [16].

On the other hand, for the binary fields,

– the reported fastest scalar multiplication is 3,14,323 clks by batch binary Edwards curve software BBE251 [4] on Core 2 Quad Q6600 processor.
– [8] reports 5,37,000 clks for binary Weierstrass curve CURVE2251 on Core-i7 processor.

Both of these curves are defined over binary field $\mathbb{F}_{2^{251}}$. These softwares use "bitslicing" technique to achieve the best possible result by avoiding the shifting operations. Also they compute 128 scalar multiplications in batches. In other words, without a requirement of a large number scalar multiplications, we can not get the speedups mentioned for these software. For a busy server, these softwares are effective ones. But for resource-constraint clients, these software are not suitable. Previous to these works, [13] reports 8,55,036 clks for one scalar multiplication on the same field $\mathbb{F}_{2^{251}}$.

From the above mentioned results, two things can be observed. Firstly, the works of [4, 8] focus on batch implementation using variable-base point. But if we consider the most important two protocols of public key cryptography: Diffie-Hellman key exchange [10] and digital signatures [25], we see that fixed-based scalar multiplications also take an important role to determine the performance of these protocols. In Diffie-Hellman key exchange, each party has to compute one fixed-base and one variable-base scalar multiplication, and the performance is measured as the sum of those two scalar multiplication. If we consider the x-coordinate based digital signature algorithm qDSA [16, 26], then key generation and signing algorithm use only fixed-base scalar multiplications. On the other hand, verification uses one fixed base and one variable-base scalar multiplication. Also it is shown in [16, 3] that small base point can improve the performance of fixed-base scalar multiplication compared to variable-base scalar multiplication.

Secondly, It clear that software implementations of binary elliptic curves are slower than that of elliptic curves over large characteristics fields. But multiplications on binary fields are similar to the multiplications on prime fields without any forwarded carry and squaring is just rearrangement of the coefficients. The reason behind the slower results is the absence of $\mathbb{F}_2[t]$ multiplication circuit in the modern processors while the existence of highly optimized mid-level $\mathbb{Z}$ multiplication circuit [4]. Recently, Intel has introduced $\mathbb{F}_2[t]$ multiplication circuit (as the instructions PCLMULQDQ and VPCLMULQDQ) in the processors which can multiply 2 64-bit long $\mathbb{F}_2[t]$ elements or can perform multiple $\mathbb{F}_2[t]$ multiplications in parallel [15]. In this work, we analysis the effect of PCLMULQDQ instruction on binary Edwards curve BEd251 [4] and binary Kummer line BKL251 (introduced in this work) and achieve fastest Diffie-Hellman key exchange [10] on binary field. The details of our contributions are as given below.

Our Contributions:

1. **Binary Kummer Line.** The idea of Kummer line over binary field is introduced by Gaudry and Lubicz in [12]. But it was not clear whether it is possible to achieve competitive speed

using binary Kummer line compared to the binary Edwards curve or Kummer lines over prime fields. Our main contribution is that we show that it is possible by developing software using instruction `PCLMULQDQ`. To achieve this, we further develop the theory of binary Kummer line from [12]. We show the consistency between scalar multiplication on binary Kummer line and that of binary Weierstrass which in turn proves the equivalence of discrete logarithm problem on binary Kummer line and the associated elliptic curve. We propose a binary Kummer line BKL251 whose details are as given below:

- **Choice of Finite Field.** Our target is 128-bit security. For fair comparison with BEd251 and CURVE2251, we also choose the finite field $\mathbb{F}_{2^{251}}$.
- **Choice of Kummer line.** The binary Kummer line BKL251 is one with the smallest curve parameter $b = t^{13} + t^9 + t^8 + t^7 + t^2 + t + 1$ of the underlying field $\mathbb{F}_{2^{251}}$ such that it satisfies certain security conditions. For the line, we also identify a small base point $(t^3 + t^2, 1)$ with small coordinates. Later we provide the details of the above mentioned security details and show that BKL251 offers 124-bit security similar to BEd251 and CURVE2251.

2. **Implementation of Scalar Multiplication over binary Kummer line.** One of the major concern of implementation of scalar multiplication is the resistance against side-channel attack like timing attacks [7]. The solution is the constant time scalar multiplication and one of the popular choices is the use of Montgomery ladder [24] like x-coordinate-based ladder with differential additions and doubling operations. In binary Kummer line, one ladder step requires 4 field multiplications, 5 field squaring, 1 field multiplication by Kummer line parameter and 1 field multiplication by base point. By carefully choosing small Kummer line parameter and small base point, we achieve one ladder step at the cost of 4 field multiplications, 5 field squaring and 2 field multiplications by small constants for fixed-base scalar multiplications and 5 field multiplications, 5 field squaring and 1 field multiplications by small constants for variable-base scalar multiplications.
   In binary Edwards Curve [7], one ladder step needs 4 field multiplications, 4 field squaring, 2 field multiplication by curve parameter and 1 field multiplication by base point in WZ-coordinate system. Because of WZ-coordinate system, base point becomes a full length element of the field. As a consequence, the operation count for each ladder step becomes 5 field multiplications, 4 field squaring and 2 field multiplications by small constant.
   By trading off 1 field multiplication at the cost of 1 field squaring, Kummer line gains significant amount of speed ups for fixed-base scalar multiplication. On the other hand, the required times for variable-base scalar multiplications are very close for Kummer line and Edwards curve.
3. **Implementation of Scalar Multiplication over binary Edwards Curve.** To make a fair comparison, the possible candidates are mpfq [13], CURVE2251 and BEd251. But there exist certain problems associated with the available software of each of the curves.
   (a) mpfq uses similar curve arithmetic as the binary Kummer line but the software is approximately 12 years old and it does not take advantage of the present processors. The software is developed for the curve
   $$E_m : Y^2 + XY = X^3 + (t^{13} + t^9 + t^8 + t^7 + t^2 + t + 1),$$
   and uses the base point $(t^3 + 1, 1)$ for fixed-based scalar multiplications. But $(t^3 + 1, 1)$ is not a point of E_m rather it is a point on the twist of E_m [11].
   (b) Latest implementation of CURVE2251 is available as a part of the RELIC [1] but each ladder step takes 6 field multiplications, 5 field squaring [27] which is higher than the operation count of ladder step of the binary Edwards curve.

(c) The only available software of BEd251 is BBE251 which is available at [5] and uses bitslicing technique to compute 128 variable-base scalar multiplications in batches.

As BEd251 has lower operation counts than CURVE2251, we choose to implement the binary Edwards curve BEd251 and make a comparison with the proposed binary Kummer line BKL251. Our experiments show that BKL251 is 9.3% faster than BEd251 in case of fixed-base scalar multiplication and for variable-base scalar multiplication both of them are takes approximately same time (BKL251 is 1.5% faster than BEd251).

Both the softwares are publicly available at the following links:

BKL251 `https://github.com/sk5891/BKL251`
BEd251 `https://github.com/sk5891/BEd251`

2 Binary Kummer Line

Binary Kummer Line is introduced by Gaudry and Lubicz in [12]. Here we only give the brief description of the arithmetic of binary Kummer line and we refer [12] to interested reader for further details. Let k be a finite field of characteristics two. Let $\mathsf{BKL}_{(1,b)}$ be a Kummer Line defined by the parameter $b \in k$ where $b \neq 0$.

The arithmetic of Kummer line is given in projective coordinate system. Let $\mathbf{P} = (x_1, z_1)$ and $\mathbf{Q} = (x_2, z_2)$ be two points on the Kummer line such that $\mathbf{P} \neq (0, 0)$ and $\mathbf{Q} \neq (0, 0)$. We say that $\mathbf{P}$ and $\mathbf{Q}$ are equivalent if there exists a non-zero $\xi \in k$ such that

$$x_1 = \xi x_2 \text{ and } z_1 = \xi z_2.$$

Suppose that $\mathbf{P} = (x_1, z_1)$ be a projective point on the Kummer line $\mathsf{BKL}_{(1,b)}$. Given the point $\mathbf{P}$, doubling Algorithm `dbl` of Table 1 shows that how to compute $2\mathbf{P} = (x_3, z_3)$. Let $\mathbf{Q} = (x_2, z_2)$ be another point on the $\mathsf{BKL}_{(1,b)}$ and we want to compute $\mathbf{P} + \mathbf{Q} = (x_4, z_4)$. Given the point $\mathbf{P} - \mathbf{Q} = (x, z)$, computation of (x_4, z_4) is shown by the differential addition Algorithm `diffAdd` of Table 1.

$(x_3, z_3) = \mathtt{dbl}(x_1, z_1)$:	$(x_4, z_5) = \mathtt{diffAdd}(x_1, z_1, x_2, z_2, x, z)$:
$x_3 = b(x_1^2 + z_1^2)^2$;	$x_4 = z(x_1x_2 + z_1z_2)^2$;
$z_3 = (x_1z_1)^2$;	$x_4 = x(x_1z_2 + x_2z_1)^2$;

Table 1. Doubling and Differential Addition on Binary Kummer Line

In Kummer Line $\mathsf{BKL}_{(1,b)}$, the point $(1, 0)$ acts as an identity. This can be proved by showing

$$\left.\begin{aligned} \mathtt{diffAdd}(x, z, 1, 0, x, z) &= (x^2z, xz^2) \sim (x, z) \\ \mathtt{diffAdd}(1, 0, x, z, x, z) &= (x^2z, xz^2) \sim (x, z) \\ \mathtt{dbl}(1, 0) &= (b, 0) \sim (1, 0) \end{aligned}\right\} \tag{1}$$

It also can be shown that the point $(0, 1)$ is a point of order 2 as given in Equation 2

$$\mathtt{dbl}(0, 1) = (b, 0) \sim (1, 0) \tag{2}$$

In the rest of the paper, we will consider Kummer line $\mathsf{BKL}_{(1,b)}$ for some non-zero $b \in k$.

2.1 Scalar Multiplication

Let $\mathbf{P} = (x, z)$ be a point on the Kummer line $\mathsf{BKL}_{(1,b)}$ and n be a l-bit scalar as $n = \{1, n_{l-2}, \ldots, n_0\}$. Our objective is to compute $n\mathbf{P} = (x_n, z_n)$. We use Montgomery ladder like ladder to perform this operation [22]. This methods loops for $l-1$ times and each ladder step performs a `dbl` and a `diffAdd` operation.

Assume that at a ladder step, the inputs are the points (x_1, z_1) and (x_2, z_2). At the end of the ladder step, the outputs are two point (x_3, z_3) and (x_4, z_4). Suppose, that we need to compute double of the point (x_1, z_1), and addition of the points (x_1, z_1) and (x_2, z_2), then during ladder step we compute $(x_3, z_3) = \texttt{dbl}(x_1, z_1)$ and $(x_4, z_4) = \texttt{diffAdd}(x_1, z_1, x_2, z_2, x, z)$. The details of the Algorithms `scalarMult` and `ladderStep` are given in the Table 2. Notice that, Algorithm `ladderStep` uses "If" condition, but in our implementation and code of the ladder step we do not use any branching instruction.

We start Algorithm `scalarMult` with two points $\mathbf{P}$ and $2\mathbf{P} = \texttt{dbl}(\mathbf{P})$. Let at i-th iteration, the inputs be the points $m\mathbf{P}$ and $(m+1)\mathbf{P}$. Then if $n_i = 0$, the `ladderStep` outputs the points $2m\mathbf{P}$ and $(2m+1)\mathbf{P}$. On the other hand, if $n_i = 1$, the `ladderStep` computes the points $(2m+1)\mathbf{P}$ and $2(m+1)\mathbf{P}$.

$n\mathbf{P} = \texttt{scalarMult}(\mathbf{P}, n)$:	`ladderStep`$(\mathbf{S}, \mathbf{R}, n_i)$:
1. Let $n = \{1, n_{l-2}, \ldots, n_0\}$;	1. If $n_i = 0$ then
2. Set $\mathbf{S} = \mathbf{P}$ and $\mathbf{R} = \texttt{dbl}(\mathbf{P})$;	2. $\mathbf{R} = \texttt{diffAdd}(\mathbf{S}, \mathbf{R}, \mathbf{P}); \mathbf{S} = \texttt{dbl}(\mathbf{S})$;
3. For $i = l-2$ to 0 do	3. Else If $n_i = 1$ then
4. `ladderStep`$(\mathbf{S}, \mathbf{R}, n_i)$;	4. $\mathbf{S} = \texttt{diffAdd}(\mathbf{S}, \mathbf{R}, \mathbf{P}); \mathbf{R} = \texttt{dbl}(\mathbf{R})$;
5. End For;	5. End If;
6. Return $\mathbf{S}$;	

Table 2. Scalar Multiplication and Ladder Step

2.2 Binary Kummer Line to Elliptic Curve

Let $b \in k$ and $b \neq 0$. Let E_b be an elliptic curve defined over field k by Equation 3

$$E_b : Y^2 + XY = X^3 + b^4. \tag{3}$$

We can map a point of Kummer Line to elliptic curve by the mapping $\pi : \mathsf{BKL}_{(1,b)} \to E_b/\{\pm 1\}$ [12] which is defined as

$$\pi(\mathbf{P} = (x, z)) = \begin{cases} (bz, \cdot, x), \text{ if } x \neq 0 \\ \infty, \text{ if } x = 0 \end{cases} \tag{4}$$

Putting $X = \frac{bz}{x}$ in Equation 3, we can compute the Y-coordinate upto to the sign. We can also convert a point of $E_b/\{\pm 1\}$ to a point of Kummer line using the inverse mapping π^{-1} as defined by Equation 5. Let $P = (X, \cdot, Z)$ be a point of E_b then

$$\pi^{-1}(P) = \begin{cases} (bZ, \cdot, X), \text{ if } X \neq 0 \\ (0, 1), \text{ if } X = 0 \end{cases} \tag{5}$$

But the mapping π along does not conserve the consistency of the scalar multiplications between Kummer line $\mathsf{BKL}_{(1,b)}$ and the elliptic curve E_b. We also need help of a point of order two of the elliptic curve E_b as given in the next section.

2.3 Equivalence between $\mathsf{BKL}_{(1,b)}$ and E_b

Let $\mathsf{BKL}_{(1,b)}$ be a Kummer line on the field k and E_b be the associated elliptic curve as defined by Equation 3. Let $\mathbf{P}$ be a point on Kummer line $\mathsf{BKL}_{(1,b)}$. Also consider the point $T_2 = (0, b^2)$ which is a point of order two on E_b. As $\pi(\mathbf{P})$ is a point on E_b, $\pi(\mathbf{P}) + T_2$ is also a point on E_b. Now we extend the mapping π to $\hat{\pi}$ as given Equation 6

$$\hat{\pi}(\mathbf{P}) = \pi(\mathbf{P}) + T_2. \tag{6}$$

The inverse of the mapping of $\hat{\pi}$ is defined as

$$\hat{\pi}^{-1}(P) = \pi(P + T_2), \tag{7}$$

where P is a point on E_b. Let $\mathbf{P} = (x, z)$ be a point on the Kummer line such that it is not a point of order 2 or identity, then Equation 8 holds.

$$2\hat{\pi}(\mathbf{P}) = \hat{\pi}(\mathtt{dbl}(\mathbf{P})). \tag{8}$$

Let P_1 and P_2 be any two points on E_b such that $P_1 \neq \pm P_2$ and neither of them is point at infinity nor of order 2. Then Equations 9 hold.

$$\left.\begin{aligned} \hat{\pi}\left(\mathtt{dbl}(\hat{\pi}^{-1}(P_i))\right) &= 2P_i, i = 1, 2 \\ \hat{\pi}\left(\mathtt{diffAdd}(\hat{\pi}^{-1}(P_1), \hat{\pi}^{-1}(P_2), \hat{\pi}^{-1}(P_1 - P_2))\right) &= P_1 + P_2 \end{aligned}\right\} \tag{9}$$

Notice that $2\hat{\pi}(\mathbf{P}) = 2\left(\pi(\mathbf{P}) + T_2\right) = 2\pi(\mathbf{P})$. The proofs of Equations 8 and 9 are trivial, but the expressions grow very fast and hard to compute manually. Therefore, we used GP/PARI script to verify them symbolically and made available along with the software. Equations 8 and 9 are important as they forms the consistency between the scalar multiplication between binary Kummer Line $\mathsf{BKL}_{(1,b)}$ and elliptic curve E_b. Let $\mathbf{P} \in \mathsf{BKL}_{(1,b)}$. Then we have $\hat{\pi}(n\mathbf{P}) = n\hat{\pi}(\mathbf{P})$. Again, we have that $\hat{\pi}(n\mathbf{P}) = \pi(n\mathbf{P}) + T_2$ and $n\hat{\pi}(\mathbf{P}) = n\left(\pi(\mathbf{P}) + T_2\right) = n\pi(\mathbf{P}) + n \pmod 2\ T_2$. Therefore, $\hat{\pi}(n\mathbf{P}) = n\hat{\pi}(\mathbf{P})$ can be rewritten as

$$\pi(n\mathbf{P}) = n\pi(\mathbf{P}) + (1 + n \pmod 2)\ T_2$$

which is pictorially shown in Figure 1. The equivalence of scalar multiplication on Kummer line

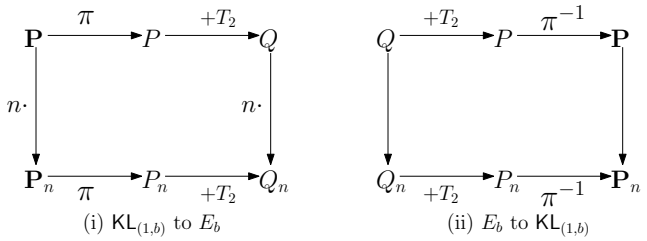

Fig. 1. Consistency of scalar multiplications between $\mathsf{BKL}_{(1,b)}$ and E_b

and the associated elliptic curve is exactly the same as the Kummer line on prime field [16]. From Figure 1, it can be concluded that discrete logarithm problem is equally hard on the Kummer line $\mathsf{BKL}_{(1,b)}$ and on the elliptic curve E_b.

3 Binary Edwards Curve

In this section, we give a brief description of binary Edward curve to make the paper self-contained. Let k be a field of characteristics 2. Let $d \in k \setminus \{0\}$. We define binary Edwards curve [7, 4] by Equation 10.

$$\mathsf{BEd} : d(x + x^2 + y + y^2) = (x + x^2)(y + y^2) \tag{10}$$

The neutral element is the point $(0, 0)$. The point $(1, 1)$ has order 2. The curve Edwards curve BEd is birationally equivalent to the ordinary curve E_d of Equation 11.

$$E_d : X^2 + XY = X^3 + (d^2 + d)X + d^8. \tag{11}$$

The mapping from BEd to E_d is given by Equation 12.

$$\begin{aligned} \phi &: (x, y) \mapsto (X, Y) \\ X &= \frac{d^3(x + y)}{xy + d(x + y)} \\ Y &= d^3 \left(\frac{x}{xy + d(x + y)} + d + 1 \right) \end{aligned} \tag{12}$$

[7, 4] suggests the use of WZ-coordinate system, where $W = X+Y$, which provides the minimum operation count for the ladder step of Montgomery like scalar multiplication on binary Edwards curve. Let $P, Q \in \mathsf{BEd}$ such that $P = (w_2, z_2)$, $Q = (w_3, z_3)$ and $P - Q = (w_1, 1)$ are given. We compute $2P = (w_3, z_3)$ and $P + Q = (w_4, z_4)$ using mixed differential and doubling operation as given in Table 3. We refer [7, 4] for further details of binary Edwards curve.

$c = w_2(w_2 + z_2);$	$w_4 = c^4;$	$z_4 = d(z_2^2)^2 + w_4;$
$v = cw_3(w_3 + z_3)$	$z_5 = v + d(z_2 z_3)^2$	$w_5 = v + z_5 w_1;$

Table 3. Mixed differential and doubling of binary Edwards Curve

4 Kummer Line over Field $\mathbb{F}_{2^{251}}$

Let q be an integer and $k = \mathbb{F}_{2^q}$ be a finite field of characteristics 2 with 2^q elements. We choose Kummer line $\mathsf{BKL}_{(1,b)}$ such that $b \neq 0$ and $b \in \mathbb{F}_{2^q}$ and then we check whether the associated elliptic curve E_b is one with all the required security criteria like curve and the twist of it have near prime orders, has large prime subgroups, resistance against pairing attacks and others which we discuss in details for the proposed Kummer line later. In this work, we target 128-bit security and we choose field $\mathbb{F}_{2^{251}} = \mathbb{F}_2[t]/(t^{251} + t^7 + t^4 + t^2 + 1)$ as the binary Edwards curve $\mathsf{BEd251}$ [7, 4] and the binary Weierstrass curve $\mathsf{CURVE2251}$ [27]. For $\mathsf{BEd251}$, $d = t^{57} + t^{54} + t^{44} + 1$.

To find the appropriate Kummer line, the value of b is increased from 1 onwards, and then we compute the associated elliptic curve and checked the security details. In our experiment, we found that $b = t^{13} + t^9 + t^8 + t^7 + t^2 + t + 1$ is the smallest value for which the associated elliptic curve E_b of the Kummer line $\mathsf{BKL}_{(1,b)}$ satisfies the following security details.

1. Order of the curve is $4p_1$ where $p_1 = 2^{249} - \delta_1$ and $\delta_1 = 160978630352464458983623\backslash 660609333279$. Therefore, the curve order is near prime with cofactor $h = 4$ [4].
2. Order of the twist curve is $2p_2$ where $p_2 = 2^{250} + \delta_2$ and $\delta_2 = 3219572607049289179672\backslash 4613321218666559$. Similarly, the twist curve order is near prime with cofactor $h_T = 2$ [4].
3. The largest prime subgroup has order p_1 and of size 249-bit. Therefore the curve provides approximately 124.5-bit security against discrete logarithms problem.
4. Avoiding subfields: The j-invariant $1/b^4$ is a primitive element of the field $\mathbb{F}_{2^{251}}$.
5. The discriminant of the curve is $\Delta = \left((2^{251} + 1 - 4p_1) - 4 \cdot 2^{251}\right)$ which is 1 (mod 4) and a square-free term. The discriminant is also divisible by the large prime $\Delta/(-7 \cdot 31 \cdot 599 \cdot 2207)$.
6. The multiplicative order of $2^{251} \pmod{p_1}$ and $2^{251} \pmod{p_2}$ are very large and they are respectively $\lambda = (p_1 - 1)/2$ and $\lambda_T = (p_2 - 1)/6$. Therefore, the curve is resistant against pairing attacks.
7. Similar to the BEd251, it is also resistant against GHS attack as the degree of the extension field is 251 which is a prime [4, 20].

From hereon, BKL251 denotes the $\mathsf{BKL}_{(1,b)}$ with $b = t^{13} + t^9 + t^8 + t^7 + t^2 + t + 1$ over the finite field $\mathbb{F}_{2^{251}}$. The Kummer line BKL251 also has a small base point $(t^3 + t^2, 1)$, where other two curves have long base points. Table 4 lists the comparative study of (estimates of) the sizes of the various parameters of the associated elliptic curve of the proposed Kummer lines $\mathsf{BKL}_{(1,b)}$ with respect to the BEd251 and the CURVE2251.

	$(\lg p_1, \lg p_2)$	(h, h_T)	(λ, λ_T)	$\lg(-\Delta)$	Base point
BEd251 [7, 4]	(249,250)	(4,2)	$(\frac{p_1-1}{2}, \frac{p_2-1}{2})$	252	-
CURVE2251 [27]	(249,250)	(4,2)	$(\frac{p_1-1}{2}, \frac{p_2-1}{6})$	253	(α_1, γ_1)
BKL251 (**this work**)	(249,250)	(4,2)	$(\frac{p_1-1}{2}, \frac{p_2-1}{6})$	253	$(\alpha_2, 1)$

$\alpha_1 = 0x6AD0278D8686F4BA4250B2DE565F0A373AA54D9A154ABEFACB90\backslash DC03501D57C$,
$\gamma_1 = 0x50B1D29DAD5616363249F477B05A1592BA16045BE1A9F218180C5150\backslash ABE8573$,
$\alpha_2 = 0$xC.

Table 4. Comparison of BKL251 against BEd251 and CURVE2251

4.1 Set of Scalars

In this work, the allowed range of scalars are of length 251 bits and have the from

$$2^{250} + 4 \cdot \{0, 1, 2, \ldots, 2^{248} - 1\}.$$

We call this scalars as **clamped scalar** following the terminology of [16]. Use of clamped scalars, ensures two things:

1. **Resistance to Small Subgroup Attacks.** Small subgroup attacks are effective when curves have small cofactors [19]. The attack becomes infeasible if the scalars are the multiples of the cofactor. The clamped scalars here are all multiples of 4 for BKL251 where 4 is the cofactor of the curve.

2. **Constant time scalar multiplication.** The most traditional way to achieve constant time scalar multiplication is the use of Montgomery ladder. In Montgomery ladder, the ladder step iterates $(l-1)$ times where l is the bit-length of the scalar. This implies that the constant time is relative to the length of scalar. By clamping, we ensure the use of constant number of iterations in the ladder step irrespective of the choice of the scalar. For BKL251, we always need 250 differential additions and 250 doubling.

Generation of Clamped Scalars One can create a clamped scalar from a 32-byte random binary string. First, clear the least significant two bits (that is, set zero to bit number 0 and 1 of 0-th byte). Second, clear the most significant 5 bits (that is, set 0 to bit number 7, 6, 4, 5, 4, and 3 of 31-st byte). Lastly set the 3-rd least significant bit of 31-st byte to 1 (that is, set 1 to bit number 2 of 31-st byte).

5 Field Arithmetic

Efficient field arithmetic are extremely necessary to have efficient scalar multiplication. There are many efficient algorithms are available for binary field arithmetic, but we focus only on the finite field $\mathbb{F}_{2^{251}} = \mathbb{F}_2[t]/(t^{251}+t^7+t^4+t^2+1)$ where $f(t) = t^{251}+r(t)$ is a irreducible polynomial with $r(t) = (t^7+t^4+t^2+1)$. Each element of $u \in \mathbb{F}_{2^{251}}$ can be represented as a polynomial of the form

$$u = u_{250}t^{250} + u_{249}t^{249} + \cdots + u_1 t + u_0, \text{ where each } u_i \in \mathbb{F}_2, \forall 0 \leqslant i \leqslant 250.$$

Element u can also be represented as binary vector of the form $(u_{250}, u_{249}, \ldots, u_1, u_0)$. We can divide this vector into ν small vectors and we call this small vectors as limbs. Assume that the least significant $\nu-1$ limbs have length κ and then length the most significant limb is $\eta = 251-\kappa\cdot(\nu-1)$ as given in Figure 2.

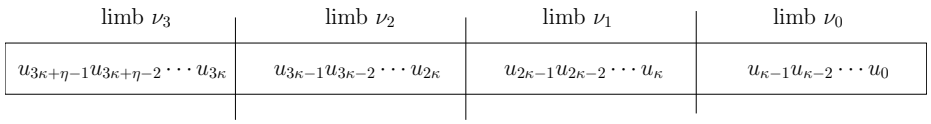

Fig. 2. Field element representation

In our implementation, our main objective is to explore the operation `PCLMULQDQ` of Intel Intrinsic [15]. Let x and y be two 128-bit registers as `__m128i`. We represent x as a vector (x_0, x_1) where x_0 is the least significant 64 bits and x_1 is the most significant 64 bits. Similarly we also represent the y as (y_0, y_1). The instruction `PCLMULQDQ` takes two `__m128i` variables and an 8-bit integer `0x`ij (`0x` stands for hexadecimal representation), where $i, j \in \{0, 1\}$, as inputs. Let z be another `__m128i` register. The `PCLMULQDQ` outputs

$$z = (z_0, z_1) = \texttt{PCLMULQDQ}(x, y, \texttt{0x}ij) = x_i \odot_2 y_j,$$

where $(z_1 \| z_0)$ is the binary multiplication of x_i and y_j, $\|$ denotes string concatenation and $\odot_2$ denotes multiplication on $\mathbb{F}_2$. Notice that `PCLMULQDQ` can only multiply two binary elements of length at most 64. Because of this, we choose $\kappa = 64$ and consequently we have $\nu = 4$. The length of the ν_3 is $\eta = 59$ bits.

5.1 Notation

The following notions are used to explain the algorithms of field arithmetic used to implement the software.

$\|$ concatenation of binary strings,
$\ll_\ell$ left shift by ℓ bits,
$\gg_\ell$ right shift by ℓ bits,
$\&$ bitwise AND,
$\oplus$ bitwise XOR,
$\odot_2$ multiplication of two elements in $\mathbb{F}_2[t]$ by `PCLMULQDQ`,
$\text{len}(u)$ length of a binary string u,
$\text{lsb}_\ell(u)$ least significant ℓ bits of a binary string u, and
$\deg(f)$ degree of the polynomial f.

5.2 Field Element Representation

Let $\theta = t^{64} \in \mathbb{F}_2[t]$. Each element $u \in \mathbb{F}_{2^{251}}$ is represented as $u(\theta)$ as given below:

$$u(\theta) = u_0 + u_1\theta + u_2\theta^2 + u_3\theta^3,$$

where each u_i is the i-th limb as shown in Figure 2. We call $u(\theta)$ is of proper representation if each $u_i < \theta$ for $i = 0, 1, 2$ and $u_3 < t^{59}$. In other words, $\text{len}(u_i) \leqslant 64$ for $i = 0, 1, 2$ and $\text{len}(u_3) \leqslant 59$ as $\text{len}(u_i) = \deg(u_i) + 1$ where u_i is a binary polynomial.

5.3 Reduction

Reduction is one of the most important algorithms of field arithmetic. Let $u(t) \in \mathbb{F}_2[t]$ such that $\deg(u) = 251 + i$. Then we can write $u = h(t)t^{251} + g(t)$ where $h(t), g(t) \in \mathbb{F}_2[t]/f(t)$ such that $\deg(h(t)) = i$ and $\deg(g(t)) \leqslant 250$. Now we have

$$u(t) = h(t)t^{251} + g(t) = r(t)h(t) + g(t) \pmod{f(t)}.$$

Let $u(\theta) = \sum_{i=0}^{3} u_0\theta^i$ where $\deg(u_i) < 127$ for $i = 0, 1, 2$ and $\deg(u_3) < 118$. Now if for any $i = 0, 1, 2$, $\deg(u_i) > 63$ and/or $\deg(u_3) > 58$ then $u(\theta)$ is not a proper one. Following the ideas of [3, 14, 2, 16], the reduction algorithm `reduce` is given in Table 5. Notice that the returned $v(\theta)$ is a proper one. After the For loop at line 4, $\text{len}(v_i) \leqslant 64$ for $i = 0, 1, 2$ and $\text{len}(v_3) \leqslant \max\{\text{len}(u_3), \text{len}(w_2)\} \leqslant \max\{118, 127 - 64\} = 118$. After line 5, $\text{len}(v_3) \leqslant 59$ and $\text{len}(w_3) \leqslant 59$. This implies that w_3 is a binary polynomial of maximum degree 58. As r is a polynomial of degree 7, $\deg(w_3)$ can be at most 65 after line 6 that is $\text{len}(w_3) \leqslant 66$. Then v_0 can be of length 66 bits after line 7 which is just 2-bit greater than the allowed 64-bit. Line 8 takes care of this overflow from v_0. As XOR do not increase the length of the input binary strings and $\text{len}(v_1) \leqslant 64$ at the beginning of line 8, then $\text{len}(v_1)$ is still at most 64 bits after line 8. This concludes that the output $v(\theta)$ is the proper representation of $u(\theta)$.

$v(\theta) = \texttt{reduce}(u(\theta))$ where $u(\theta) = \sum_{i=0}^{3} u_0\theta^i$:
1. $v_0 = u_0$;
2. For $i = 0$ to 2 do
3. $w_i = v_i \gg_{64}$; $v_i = \text{lsb}_{64}(v_i)$; $v_{i+1} = u_{i+1} \oplus w_i$;
4. End For;
5. $w_3 = v_3 \gg_{59}$; $v_3 = \text{lsb}_{59}(v_3)$;
6. $w_3 = w_3 \odot_2 r$, where r is the binary vector representation of $r(t)$;
7. $v_0 = v_0 \oplus w_3$;
8. $w_0 = v_0 \gg_{64}$; $v_0 = \text{lsb}_{64}(v_0)$; $v_1 = v_1 \oplus w_0$;
9. Return $v(\theta) = \sum_{i=0}^{3} v_0\theta^i$;

Table 5. Algorithm `reduce`

5.4 Addition and Subtraction

Let $u(\theta) = \sum_{i=0}^{3} u_0\theta^i$ and $v(\theta) = \sum_{i=0}^{3} v_0\theta^i$ be two elements of $\mathbb{F}_{2^{251}}$ with proper representations. Let $w(\theta) = u(\theta) + v(\theta) \in \mathbb{F}_{2^{251}}$. Addition over binary field is very easy and only needs XOR operations. We compute addition algorithm `add` as $w_i = u_i \oplus v_i$ for all $i = 0, 1, 2, 3$ where $w(\theta) = \sum_{i=0}^{3} w_i\theta^i$ is also in proper representation. As binary addition operation is component-wise XOR, it is not followed by the `reduce` algorithm.

On binary field, subtraction is the same as the addition as $-1 = 2 - 1 = 1$. Therefore, we do not define subtraction separately.

5.5 Multiplication by Small Constant

Let $u(\theta) = \sum_{i=0}^{3} u_i\theta^i$ be an element of the field $\mathbb{F}_{2^{251}}$. We would like to multiply u by a small constant c (let). By small constant, we mean that $c \in \mathbb{F}_2[t]$ and $\deg(c) \leqslant 63$. This also implies that c can be stored using only one limb. We compute the multiplication as $u'(\theta) = \texttt{constMult}(u(\theta), c) = \sum_{i=0}^{3} (c \odot_2 u_i)\theta^i$ and then apply `reduce` on $u'(\theta)$ to achieve proper representation. The Algorithm is summarized in Table 6.

$v(\theta) = \texttt{constMult}(u(\theta), c)$:
1. For $i = 0$ to 3 do
2. $u'_i = u_i \odot_2 c$;
3. End For;
4. $v = \texttt{reduce}(u'(\theta))$
5. Return $v(\theta) = \sum_{i=0}^{3} v_0\theta^i$;

Table 6. Algorithm `constMult`

5.6 Field Multiplication

Let $u(\theta) = \sum_{i=0}^{3} u_0\theta^i$ and $v(\theta) = \sum_{i=0}^{3} v_0\theta^i$ be two elements to be multiplied. Let u and v are in proper representation. Then the multiplication algorithm is given in the Table 7. The function `polyMult` of $u(\theta)$ and $v(\theta)$ computes a polynomial of degree 6 in θ. Let the polynomial be $w =$

$\sum_{i=0}^{6} w_i\theta^i$. We apply `expandM` function on $w(\theta)$ to achieve a polynomial of 8 limbs where each limb is of at most 64-bit. The steps of the Algorithm `expandM` are given in the Table 7. Let the expanded polynomial be $w = \sum_{i=0}^{7} w_i\theta^i$ with $\text{len}(w_i) \leqslant 64$. Then we can derive Equation 14 from the output of the function `expandM` (that is Equation 13) using the function `fold`$(w(\theta))$ as given below.

$$
\begin{aligned}
w &= w_7\theta^7 + w_6\theta^6 + w_5\theta^5 + w_4\theta^4 + w_3\theta^3 + w_2\theta^2 + w_1\theta + w_0 \qquad (13)\\
&= (w_7\theta^3 + w_6\theta^2 + w_5\theta + w_4)\theta^4 + (w_3\theta^3 + w_2\theta^2 + w_1\theta + w_0)\\
&= (w_7\theta^3 + w_6\theta^2 + w_5\theta + w_4)t^5t^{251} + (w_3\theta^3 + w_2\theta^2 + w_1\theta + w_0)\\
&= (w_7\theta^3 + w_6\theta^2 + w_5\theta + w_4)rt^5 + (w_3\theta^3 + w_2\theta^2 + w_1\theta + w_0) \text{ [As } f(t) = t^{251} + r(t)]\\
&= (w_3 + w_7rt^5)\theta^3 + (w_2 + w_6rt^5)\theta^2 + (w_1 + w_5rt^5)\theta + (w_0 + w_4rt^5) \qquad (14)
\end{aligned}
$$

$w(\theta) =$ `mult`$(u(\theta), v(\theta))$:	$w =$ `expandM`$(w(\theta))$:
1. $w(\theta) =$ `polyMult`$(u(\theta), v(\theta))$	1. $w_7 = 0$;
2. $w(\theta) =$ `expandM`(w);	2. For $i = 0$ to 6 do
3. $w(\theta) =$ `fold`(w);	3. $w_{i+1} = w_{i+1} \oplus (w_i \gg_{64})$; $w_i = \text{lsb}_{64}(w_i)$;
4. $w(\theta) =$ `reduce`$(w(\theta))$	4. End For;
5. Return $w(\theta) = \sum_{i=0}^{3} w_i\theta^i$;	5. Return $w(\theta) = \sum_{i=0}^{7} w_i\theta^i$;

Table 7. Algorithms `mult` and `expandM`

Notice that the `expandM` is absolutely necessary. In the absence of `expandM` function, consider the term w_4rt^5. After `polyMult`$(u(\theta), v(\theta))$, w_4 is a polynomial of degree at most 127. As $\deg(r) = 7$, then w_4rt^5 can be a polynomial of degree $127 + 7 + 5 = 139$ which requires 140 bits to store. In our implementation, we use `__m128i` whose capacity is 128-bit. Therefore, without `expand`, there will be overflow. This issue of overflow is also true for w_6rt^5 and w_5rt^5. As `expand` introduces the term w_7, w_7rt^5 will not be there in the absence of `expand`.

Computation `polyMult`$(u(\theta), v(\theta))$ Let $u(\theta)$ and $v(\theta)$ be in proper representation with 4 limbs and let $w(\theta) =$ `polyMult`$(u(\theta), v(\theta))$. $w(\theta)$ is a polynomial of the form $w(\theta) = \sum_{i=0}^{6} w_i\theta^i$. The main objective of the function `polyMult` is the computation of the coefficients of $w(\theta)$ that is w_i for $i = 0, 1, \ldots, 6$. We use `PCLMULQDQ` and XOR to compute the coefficients using a hybrid method of 2-2 Karatsuba [23] method and school-book method. The 2-2 Karatsuba method used in implementation is as given below:

$$
\begin{aligned}
w(\theta) &= \texttt{polyMult}(u(\theta), v(\theta))\\
&= \texttt{polyMult2}(u_1\theta + u_0, v_1\theta + v_0) + \texttt{polyMult2}(u_3\theta + u_2, v_3\theta + v_2)\theta^4 +\\
&\quad (\texttt{polyMult2}((u_1 \oplus u_3)\theta + (u_0 \oplus u_2), (v_1 \oplus v_3)\theta + (v_0 \oplus v_2)) +\\
&\quad (\texttt{polyMult2}(u_1\theta + u_0, v_1\theta + v_0) + \texttt{polyMult2}(u_3\theta + u_2, v_3\theta + v_2)))\,\theta^2
\end{aligned}
$$

On the other hand, `polyMult2` is computed using school-book method as

$$
\begin{aligned}
\texttt{polyMult2}(u_1\theta + u_0, v_1\theta + v_0) = (u_1 \odot_2 v_1)\theta^2 + + ((v_1 \odot_2 u_0) \oplus (v_0 \odot_2 u_1))\,\theta\\
(u_0 \odot_2 v_0)
\end{aligned}
$$

Each `polyMult2` requires 4 `PCLMULQDQ` operations and 1 XORs. Consequently, `polyMult` requires 12 `PCLMULQDQ` operations and 13 XORs.

Unreduced Field Multiplication (`multUnreduced`) Let $u(\theta)$ and $v(\theta)$ be two elements of $\mathbb{F}_{2^{251}}$ with proper representation that is $u(\theta) = \sum_{i=0}^{3} u_i\theta^i$ and $v(\theta) = \sum_{i=0}^{3} v_i\theta^i$. Let $w(\theta)$ is a polynomial of the form $w(\theta) = \sum_{i=0}^{6} w_i\theta^i$. We define `multUnreduced` as

$$w(\theta) = \texttt{multUnreduced}\left(u(\theta), v(\theta)\right),$$

where $\texttt{multUnreduced}\left(u(\theta), v(\theta)\right) = \texttt{polyMult}(u(\theta), v(\theta))$. $\texttt{multUnreduced}\left(u(\theta), v(\theta)\right)$ is exactly the same as the `mullt` without `expandM`, `fold` and `reduce`.

Field Addition of unreduced field elements (`addReduce`) Let $u(\theta) = \sum_{i=0}^{6} u_i\theta^i$ and $v(\theta) = \sum_{i=0}^{6} v_i\theta^i$ be two elements of $\mathbb{F}_{2^{251}}$ where you can assume that $u(\theta)$ and $v(\theta)$ are outputs of `multUnreduced`. As addition over binary field is simply the bit-wise XOR of the inputs, it does not increase the length and thus there is no issue of overflow. The details of the algorithm of `addReduce` is given in Table 8. On the XORed value, we apply `expandM`, `fold` and `reduce` to attend a proper representation.

$w(\theta) = \texttt{addReduce}(u(\theta), v(\theta))$:
1. For $i = 0$ to 6 do
2. $\quad w_i = u_i \oplus v_i$;
3. End For;
4. $w(\theta) = \texttt{expandM}(w)$;
5. $w(\theta) = \texttt{fold}(w)$;
6. $w(\theta) = \texttt{reduce}(w(\theta))$;
7. Return $w(\theta) = \sum_{i=0}^{3} w_i\theta^i$;

Table 8. Algorithms `addReduce`

5.7 Field Squaring

Field squaring is much less expensive in binary fields compared to prime fields as here squaring means relabeling the exponent of the input binary element. Let $u(\theta) = \sum_{i=0}^{3} u_0\theta^i$ be the element to be squared. The squaring algorithm is given in Table 9. The `polySq` function creates a polynomial $w(\theta) = \sum_{i=0}^{6} w_i\theta^i$ from u as given in Equation 15.

$$w_i = \begin{cases} u_{i/2}^2, i = 0 \pmod 2 \\ 0,, \quad i = 1 \pmod 2 \end{cases} \tag{15}$$

As a consequence, the `expandS` is also slightly different that `expandM`. In function `expandS`, if $i = 0 \pmod 2$, then we divide the w_i in to two parts and assign the least significant 64 bits to w_i and the remaining most significant bits to w_{i+1}. If $i = 1 \pmod 2$, we do nothing. The details of the function `expandS` in Table 9. On the output of `expandS`, we apply `fold` and `reduce` to compute the proper representation of the squared value.

Notice that, `polySq` only needs 4 `PCLMULQDQ` operations and no XORs which is less than half of the operation counts of `polyMult`. As a result, `sq` is significantly faster than `mult`.

$v(\theta) = \texttt{sq}(u(\theta))$:	$w = \texttt{expandS}(w(\theta))$:
1. $w(\theta) = \texttt{polySq}(u(\theta), v(\theta))$	1. For $i = 0, 2, 4, 6$ do
2. $w(\theta) = \texttt{expandS}(w)$;	2. $w_{i+1} = (w_i \gg_{64})$; $w_i = \text{lsb}_{64}(w_i)$;
3. $w(\theta) = \texttt{fold}(w)$;	3. End For;
4. $w(\theta) = \texttt{reduce}(w(\theta))$	4. Return $w(\theta) = \sum_{i=0}^{7} w_i \theta^i$;
5. Return $w(\theta) = \sum_{i=0}^{3} w_i \theta^i$;	

Table 9. Algorithms `sq` and `expandS`

5.8 Field Inverse

We compute Kummer Line scalar multiplication in projective coordinate system and receive an projective point (x_n, z_n) at the end of the iterations of ladder steps. Therefore, we compute the affine output as x_n/z_n which requires one field inversion and one field multiplication. We compute field inversion as $z_n^{-1} = z_n^{2^{251}-2}$ using 250 field squaring and 10 field multiplications following the sequence given in [4]. The multiplications produce the terms z_n^3, z_n^7, $z_n^{2^6-1}$, $z_n^{2^{12}-1}$, $z_n^{2^{24}-1}$, $z_n^{2^{25}-1}$, $z_n^{2^{50}-1}$, $z_n^{2^{100}-1}$, $z_n^{2^{125}-1}$ and $z_n^{2^{250}-1}$.

5.9 Conditional Swap

The `ladderstep` of Table 2 uses conditional swap based on the input bit from the scalar. But to achieve constant time scalar multiplication, no use of branching instructions is prerequisite. Therefore, we perform the conditional swap without any branching instruction as given in Table 10. In Table 10, the algorithm `condSwap` uses branching instruction where `condSwapConst` performs the same job as the `condSwap` without any branching instruction. In computer, 0 is represented as a binary sting of all zeros and -1 is represented as a binary string of all ones in 2's complement representation. Therefore, if $\mathfrak{b}$ is 0 then $w = 0$ at the end of the line 2 of Algorithm `condSwapConst` else it is $w = u_i \oplus v_i$. As a consequence, if $\mathfrak{b} = 0$, there is no change of values in u_i and v_i as $u_i = u_i \oplus 0 = u_i$ and $v_i = v_i \oplus 0 = v_i$. On the other hand, if $\mathfrak{b} = 1$, then u_i and v_i get swapped as $u_i = u_i \oplus w = u_i \oplus u_i \oplus v_i = v_i$ and $v_i = v_i \oplus w = v_i \oplus u_i \oplus v_i = u_i$.

`condSwap`$(u(\theta), v(\theta), \mathfrak{b})$	`condSwapConst`$(u(\theta), v(\theta), \mathfrak{b})$
1. If $(\mathfrak{b} = 1)$ then	1. For $i = 0$ to 4 do
2. For $i = 0$ to 4 do	2. $w = u_i \oplus v_i$; $w = w\&(-\mathfrak{b})$;
3. $w = u_i$; $u_i = v_i$; $v_i = w$;	3. $u_i = u_i \oplus w$; $v_i = v_i \oplus w$;
4. End For;	4. End For;
5. End If;	

Table 10. Algorithm Conditional Swap

6 Scalar Multiplication

In this section, we explain the details of the algorithms implemented to compute scalar multiplication for the Kummer line BKL251 and binary Edwards curve BEd251.

6.1 Scalar Multiplication of BKL251

The algorithms for scalar multiplication BKLscalarMult and BKLscalarMultFB are given in the Table 11. Algorithms BKLscalarMult is scalar multiplication algorithm when the base point is unknown from before hand and we call it variable-base scalar multiplication. On the other hand, FB of BKLscalarMultFB stands for fixed-base and, we pre-compute the point $\mathtt{dbl}(\mathbf{P})$ and keep it in memory. We alway consider that the z-coordinate of the input base point is 1 and the implementation is designed to take advantage of that. The total operation counts of ladder step of BKLscalarMult is $5[\mathsf{M}]+5[\mathsf{S}]+1[\mathsf{C}]$ where $[\mathsf{M}]$ denotes field multiplication mult, $[\mathsf{S}]$ stands for field squaring sq and we denote multiplication by small constant by $[\mathsf{C}]$. $1[\mathsf{C}]$ refers the multiplication by Kummer Line parameter b (line 25 of BKLscalarMult and line 22 of BKLscalarMultFB). In out implementation, the base point is a small one $(t^3+t^2,1)$ which can be stored in one limb and consequently the field multiplication in line 20 of BKLscalarMult becomes a multiplication by constant in BKLscalarMultFB (line 17). The total operation counts of ladder step of BKLscalarMultFB becomes $4[\mathsf{M}]+5[\mathsf{S}]+2[\mathsf{C}]$.

BKLscalarMult$(\mathbf{P}, n)$:	BKLscalarMultFB$(\mathbf{P}, \mathtt{dbl}(\mathbf{P}), n)$:
Input: Base Point $= (x(\theta), 1)$	Input: Base Points $= (x(\theta), 1, x_2(\theta), z_2(\theta))$
$n = \{1, n_{l-2}, n_{l-3}, \ldots, n_0\}$	$n = \{1, n_{l-2}, n_{l-3}, \ldots, n_0\}$
Output: $x_n(\theta)$	Output: $x_n(\theta)$
1. $\mathsf{sx}(\theta) = x(\theta)$; $\mathsf{sz}(\theta) = 1$;	1. $\mathsf{sx}(\theta) = x(\theta)$; $\mathsf{sz}(\theta) = 1$;
2. $t_1(\theta) = \mathtt{add}(\mathsf{sx}(\theta), \mathsf{sz}(\theta))$;	2. $\mathsf{rx}(\theta) = x_2(\theta)$; $\mathsf{rz}(\theta) = z_2(\theta)$;
3. $\mathsf{rx}(\theta) = \mathtt{sq}(t_1(\theta))$;	3. $\mathfrak{pb} = 0$;
4. $\mathsf{rx}(\theta) = \mathtt{multConst}(\mathsf{rx}(\theta), b)$;	4. For $i = (l-2)$ to 0 do
5. $\mathsf{rz}(\theta) = \mathtt{sq}(\mathsf{sx}(\theta))$;	5. $\mathfrak{b} = (\mathfrak{pb} \oplus n_i)$;
6. $\mathfrak{pb} = 0$;	6. $\mathtt{condSwapConst}(\mathsf{sx}(\theta), \mathsf{rx}(\theta), \mathfrak{b})$;
7. For $i = (l-2)$ to 0 do	7. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), \mathfrak{b})$;
8. $\mathfrak{b} = (\mathfrak{pb} \oplus n_i)$;	8. $t_1(\theta) = \mathtt{add}(\mathsf{sx}(\theta), \mathsf{sz}(\theta))$;
9. $\mathtt{condSwapConst}(\mathsf{sx}(\theta), \mathsf{rx}(\theta), \mathfrak{b})$;	9. $t_2(\theta) = \mathtt{add}(\mathsf{rx}(\theta), \mathsf{rz}(\theta))$;
10. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), \mathfrak{b})$;	10. $t_3(\theta) = \mathtt{mult}(t_1(\theta), t_2(\theta))$;
11. $t_1(\theta) = \mathtt{add}(\mathsf{sx}(\theta), \mathsf{sz}(\theta))$;	11. $t_3(\theta) = \mathtt{sq}(t_3(\theta))$;
12. $t_2(\theta) = \mathtt{add}(\mathsf{rx}(\theta), \mathsf{rz}(\theta))$;	12. $t_4(\theta) = \mathtt{multUnreduced}(\mathsf{sx}(\theta), \mathsf{rz}(\theta))$;
13. $t_3(\theta) = \mathtt{mult}(t_1(\theta), t_2(\theta))$;	13. $t_5(\theta) = \mathtt{multUnreduced}(\mathsf{sz}(\theta), \mathsf{rx}(\theta))$;
14. $t_3(\theta) = \mathtt{sq}(t_3(\theta))$;	14. $\mathsf{rz}(\theta) = \mathtt{addReduce}(t_4(\theta), t_5(\theta))$;
15. $t_4(\theta) = \mathtt{multUnreduced}(\mathsf{sx}(\theta), \mathsf{rz}(\theta))$;	15. $\mathsf{rz}(\theta) = \mathtt{sq}(\mathsf{rz}(\theta))$;
16. $t_5(\theta) = \mathtt{multUnreduced}(\mathsf{sz}(\theta), \mathsf{rx}(\theta))$;	16. $\mathsf{rx}(\theta) = \mathtt{add}(t_3(\theta), \mathsf{rz}(\theta))$;
17. $\mathsf{rz}(\theta) = \mathtt{addReduce}(t_4(\theta), t_5(\theta))$;	17. $\mathsf{rz}(\theta) = \mathtt{multConst}(\mathsf{rz}(\theta), x(\theta))$;
18. $\mathsf{rz}(\theta) = \mathtt{sq}(\mathsf{rz}(\theta))$;	18. $\mathsf{sz}(\theta) = \mathtt{mult}(\mathsf{sx}(\theta), \mathsf{sz}(\theta))$;
19. $\mathsf{rx}(\theta) = \mathtt{add}(t_3(\theta), \mathsf{rz}(\theta))$;	19. $\mathsf{sz}(\theta) = \mathtt{sq}(\mathsf{sz}(\theta))$;
20. $\mathsf{rz}(\theta) = \mathtt{mult}(\mathsf{rz}(\theta), x(\theta))$;	20. $\mathsf{sx}(\theta) = \mathtt{sq}(t_1(\theta))$;
21. $\mathsf{sz}(\theta) = \mathtt{mult}(\mathsf{sx}(\theta), \mathsf{sz}(\theta))$;	21. $\mathsf{sx}(\theta) = \mathtt{sq}(\mathsf{sx}(\theta))$;
22. $\mathsf{sz}(\theta) = \mathtt{sq}(\mathsf{sz}(\theta))$;	22. $\mathsf{sx}(\theta) = \mathtt{multConst}(\mathsf{sx}(\theta), b(\theta))$;
23. $\mathsf{sx}(\theta) = \mathtt{sq}(t_1(\theta))$;	23. $\mathfrak{pb} = n_i$;
24. $\mathsf{sx}(\theta) = \mathtt{sq}(\mathsf{sx}(\theta))$;	24. End For;
25. $\mathsf{sx}(\theta) = \mathtt{multConst}(\mathsf{sx}(\theta), b(\theta))$;	25. $\mathtt{condSwapConst}(\mathsf{sx}(\theta), \mathsf{rx}(\theta), n_0)$;
26. $\mathfrak{pb} = n_i$;	26. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), n_0)$;
27. End For;	27. Return $(\mathsf{sx}(\theta)/\mathsf{sz}(\theta))$;
28. $\mathtt{condSwapConst}(\mathsf{sx}(\theta), \mathsf{rx}(\theta), n_0)$;	
29. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), n_0)$;	
30. Return $(\mathsf{sx}(\theta)/\mathsf{sz}(\theta))$;	

Table 11. Algorithm BKLscalarMul and BKLscalarMultFB

6.2 Scalar Multiplication of BEd251

For scalar multiplication on binary Edwards curve BEd251, we use WZ-coordinate system as suggested in [7, 4, 6] which provides the fastest result. The algorithms for scalar multiplication `BEdscalarMult` and `BEdscalarMultFB` are given in the Table 12. Similar to binary Kummer line, `BEdscalarMult` takes care variable-base, where `BEdscalarMultFB` is the algorithm for fixed-base scalar multiplication on binary Edwards curve. The operation counts of the ladder steps of `BEdscalarMult` is $5[\mathsf{M}] + 4[\mathsf{S}] + 2[\mathsf{C}]$ where $[\mathsf{C}]$ is the multiplication by Edwards curve parameter d (lines 18 and 23 of `BEdscalarMult` and, lines 15 and 20 of `BEdscalarMultFB`). In WZ-coordinate system, $W = X + Y$. It is very hard to find a base point (x, y) on Edwards curve such that (x, y) is the generator of the largest prime subgroup and $w = x + y$ becomes small enough to be stored in a limb. Even if we try to make x small, y becomes a random element of the field which satisfies the Equation 10, that is y becomes the roots of the quadratic Equation 16

$$\left(1 + \frac{d}{x + x^2}\right) y^2 + \left(1 + \frac{d}{x + x^2}\right) y + d = 0. \tag{16}$$

Similar thing happens if we try to control the size of y. In our experiment we could not find such a point and it seems only way is to check all the points of BEd251 by brute-forced method[1]. As a result, the operation counts in ladder step of `BEdscalarMultFB` remains the same as that of `BEdscalarMult`, that is $5[\mathsf{M}] + 4[\mathsf{S}] + 2[\mathsf{C}]$.

7 Implementations and Timings

We have implemented the softwares using the Intel intrinsic instructions of __m128i. All the modules of the field arithmetics and the ladder step are written in assembly language to achieve the most optimized implementation. The 64×64 bit binary field multiplications are done using `pclmulqdq` instruction. We compute 128-bit bit-wise XOR and AND operations are implemented using instructions `pxor` and `pand` respectively. For byte-wise and bit-wise right-shift, we use `psrldq` and `psrlq`. The rest of the details of the implementation can be found from the publicly available softwares. The code implements the ladder algorithm with constant-time conditional swap algorithm which takes the same amount of time for all scalars. Consequently, our code also runs in constant time.

We use `reduce` algorithm with `mult`, `sq`, `multConst` and `addReduce`. In case of `mult` and `sq`, the size of the limbs are at most 76 bits after `fold` operations. Therefore, the w_3 of line 5 of Algorithm `reduce` (Table 5) will be 17 bits at most and in turn w_3 becomes 24-bit after line 6. Therefore, there will be no overflow from v_0 of line 7 of Table 5. Similar situation will happen for the `addReduce`.

In case of `multConst` in scalar multiplications `BKLscalarMult` and `BKLscalarMultFB`, the maximum length of the constant is the length of the Kummer line parameter b which is 14-bit (where the base point is of 4 bits). Therefore, the maximum possible length of u'_3 after line 3 of Algorithm `multConst` is 72-bit. During `reduce` of `multConst`, w_3 of line 7 of Table 5 becomes 20-bit long and in turn there will be no overflow from v_0.

In case of `multConst` in scalar multiplications `BEdscalarMult` and `BEdscalarMultFB`, the maximum length of the constant is curve parameter d which is of degree 57. Therefore, the maximum possible length of u'_3 after line 3 of Algorithm `multConst` is 116-bit. During `reduce` of `multConst`,

[1] [7, 4] also do not mention about small base-point

BEdscalarMultKL(P, n) :	BEdscalarMultFB$(\mathbf{P}, \mathtt{dbl}(P), n)$:
Input: Base Point $= (w(\theta), 1)$	Input: Base Points $= (w(\theta), 1, w_2(\theta), z_2(\theta))$
$n = \{1, n_{l-2}, n_{l-3}, \ldots, n_0\}$	$n = \{1, n_{l-2}, n_{l-3}, \ldots, n_0\}$
Output: $x_n(\theta)$	Output: $x_n(\theta)$
1. $\mathsf{sw}(\theta) = w(\theta)$; $\mathsf{sz}(\theta) = 1$;	1. $\mathsf{sw}(\theta) = w(\theta)$; $\mathsf{sz}(\theta) = 1$;
2. $t_1(\theta) = \mathtt{add}(\mathsf{sw}(\theta)), \mathsf{sz}(\theta))$;	2. $\mathsf{rw}(\theta) = w_2(\theta)$; $\mathsf{sz}(\theta) = z_2(\theta)$;
3. $t_2(\theta) = \mathtt{mult}(\mathsf{sw}(\theta), t_1(\theta))$;	3. $\mathfrak{pb} = 0$;
4. $\mathsf{rw}(\theta) = \mathtt{sq}(t_2(\theta))$;	4. For $i = (l-2)$ to 0 do
5. $\mathsf{rz}(\theta) = \mathtt{add}(\mathsf{rw}(\theta), d(\theta))$;	5. $\mathfrak{b} = (\mathfrak{pb} \oplus n_i)$;
6. $\mathfrak{pb} = 0$;	6. $\mathtt{condSwapConst}(\mathsf{sw}(\theta), \mathsf{rw}(\theta), \mathfrak{b})$;
7. For $i = (l-2)$ to 0 do	7. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), \mathfrak{b})$;
8. $\mathfrak{b} = (\mathfrak{pb} \oplus n_i)$;	8. $t_2(\theta) = \mathtt{add}(\mathsf{sw}(\theta), \mathsf{sz}(\theta))$;
9. $\mathtt{condSwapConst}(\mathsf{sw}(\theta), \mathsf{rw}(\theta), \mathfrak{b})$;	9. $t_2(\theta) = \mathtt{mult}(t_2(\theta), \mathsf{sw}(\theta))$;
10. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), \mathfrak{b})$;	10. $t_3(\theta) = \mathtt{add}(\mathsf{rw}(\theta), \mathsf{rz}(\theta))$;
11. $t_2(\theta) = \mathtt{add}(\mathsf{sw}(\theta), \mathsf{sz}(\theta))$;	11. $t_3(\theta) = \mathtt{mult}(t_3(\theta), \mathsf{rw}(\theta))$;
12. $t_2(\theta) = \mathtt{mult}(t_2(\theta), \mathsf{sw}(\theta))$;	12. $t_3(\theta) = \mathtt{mult}(t_2(\theta), t_3(\theta))$;
13. $t_3(\theta) = \mathtt{add}(\mathsf{rw}(\theta), \mathsf{rz}(\theta))$;	13. $t_4(\theta) = \mathtt{mult}(\mathsf{sz}(\theta), \mathsf{rz}(\theta))$;
14. $t_3(\theta) = \mathtt{mult}(t_3(\theta), \mathsf{rw}(\theta))$;	14. $t_4(\theta) = \mathtt{sq}(t_4(\theta))$;
15. $t_3(\theta) = \mathtt{mult}(t_2(\theta), t_3(\theta))$;	15. $t_4(\theta) = \mathtt{multConst}(t_4(\theta), d)$;
16. $t_4(\theta) = \mathtt{mult}(\mathsf{sz}(\theta), \mathsf{rz}(\theta))$;	16. $t_4(\theta) = \mathtt{add}(t_3(\theta), t_4(\theta))$;
17. $t_4(\theta) = \mathtt{sq}(t_4(\theta))$;	17. $\mathsf{sw}(\theta) = \mathtt{sq}(t_2(\theta))$;
18. $t_4(\theta) = \mathtt{multConst}(t_4(\theta), d)$;	18. $\mathsf{sz}(\theta) = \mathtt{sq}(\mathsf{sz}(\theta))$;
19. $t_4(\theta) = \mathtt{add}(t_3(\theta), t_4(\theta))$;	19. $\mathsf{sz}(\theta) = \mathtt{sq}(\mathsf{sz}(\theta))$;
20. $\mathsf{sw}(\theta) = \mathtt{sq}(t_2(\theta))$;	20. $\mathsf{sz}(\theta) = \mathtt{multConst}(\mathsf{sz}(\theta), d)$;
21. $\mathsf{sz}(\theta) = \mathtt{sq}(\mathsf{sz}(\theta))$;	21. $\mathsf{sz}(\theta) = \mathtt{add}(\mathsf{sz}(\theta), \mathsf{sw}(\theta))$;
22. $\mathsf{sz}(\theta) = \mathtt{sq}(\mathsf{sz}(\theta))$;	22. $\mathsf{rw}(\theta) = \mathtt{mult}(t_4(\theta), w(\theta))$;
23. $\mathsf{sz}(\theta) = \mathtt{multConst}(\mathsf{sz}(\theta), d)$;	23. $\mathsf{rw}(\theta) = \mathtt{add}(\mathsf{rw}(\theta), t_3(\theta))$;
24. $\mathsf{sz}(\theta) = \mathtt{add}(\mathsf{sz}(\theta), \mathsf{sw}(\theta))$;	24. $\mathsf{rz}(\theta) = t_4(\theta)$;
25. $\mathsf{rw}(\theta) = \mathtt{mult}(t_4(\theta), w(\theta))$;	25. $\mathfrak{pb} = n_i$;
26. $\mathsf{rw}(\theta) = \mathtt{add}(\mathsf{rw}(\theta), t_3(\theta))$;	26. End For;
27. $\mathsf{rz}(\theta) = t_4(\theta)$;	27. $\mathtt{condSwapConst}(\mathsf{sw}(\theta), \mathsf{rw}(\theta), n_0)$;
28. $\mathfrak{pb} = n_i$;	28. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), n_0)$;
29. End For;	29. Return $(\mathsf{sw}(\theta)/\mathsf{sz}(\theta))$;
30. $\mathtt{condSwapConst}(\mathsf{sw}(\theta), \mathsf{rw}(\theta), n_0)$;	
31. $\mathtt{condSwapConst}(\mathsf{sz}(\theta), \mathsf{rz}(\theta), n_0)$;	
32. Return $(\mathsf{sw}(\theta)/\mathsf{sz}(\theta))$;	

Table 12. Algorithm `BEdscalarMult` and `BEdscalarMultFB`

w_3 of line 7 of Table 5 can be at most 64-bit long and in turn there will also be no overflow from v_0.

As there will be no overflow from v_0 after line 7 of `reduce` in all possible cases, we further optimize the field arithmetic by removing the line 8 of Table 5 during implementation.

In the modules of field multiplications and squaring, a significant amount of time is taken by the attempt of achieving the proper representation. In other words, the operations `expandM`/`expandS`, `fold` and `reduce` are most time consuming it total compared to `polyMult`/`polySq`. Therefore using `multUnreduced` and `addReduce` in `BKLscalarMult` (lines 15, 16 and 17) and in `BKLscalarMultFB` (lines 12, 13 and 14), we reduce two sets of `expandM`, `fold` and `reduce` in to one set and it produced a significant speed up also.

Timing experiments were carried out on a single core of three platforms and their setups are listed in Table 13. OS of the computer with Sandy Bridge processor is 64-bit Ubuntu 16.04 and

the code was compiled using GCC version 9.2.1. On the other hand, the OS of the computers with Haswell and Skylake processors is 64-bit Ubuntu 14.04 with GCC version 7.3.0.

Processor Architecture	Processor Make	Ubuntu 64-bit	GCC Version
Sandy Bridge	Intel®Xeon® CPU E5-1620 v4 @ 3.50GHz	16.04	9.2.1
Haswell	Intel Core™i7-4790 4-core CPU @ 3.60 Ghz	14.04	7.3.0
Skylake	Intel Core™i7-6500U 2-core CPU @ 2.50GHz	14.04	7.3.0

Table 13. Experimental Setup

During timing measurements, turbo boost and hyperthreading were turned off. An initial cache warming was done with 25,000 iterations and then the median of 1,00,000 iterations was recorded. The Time Stamp Counter (TSC) was read from the CPU to RAX and RDX registers by RDTSC instruction. The timings are listed in the Table 14.

In Sandy Bridge, BKL251 is faster than BEd251 by 9.33% and 1.55% for fixed-base and variable-base scalar multiplications respectively. In Haswell architecture, we found that BKL251 is faster than BEd251 by 8.36% and 0.80% for fixed-base and variable-base scalar multiplications respectively. We get the similar results for Skylake architecture where the speedups fixed-base and variable-base scalar multiplication are 8.81% and 0.25% respectively.

	Sandy Bridge		Haswell		Skylake	
	Fixed-base	Var-base	Fixed-base	Var-base	Fixed-base	Var-base
BKL251	82,914	90,437	1,17,592	1,27,812	86,748	95,104
BEd251	91,454	91,865	1,28,356	1,28,852	95,132	95,348

Table 14. Timings of Scalar Multiplications of BKL251 and BEd251 in clock cycles (clk)

Diffie-Hellman Key Exchange. In two-party Diffie-Hellman key exchange [10] protocol, each party has to compute two scalar multiplication: one fixed-base and one variable-base. Ignoring the communication time, the total computation time required by each party is the sum of the computation time of both the scalar multiplication. The results are given in Table 15. In Sandy Bridge, Haswell and Skylake platforms, Diffie-Hellman key exchange which uses BKL251 is 5.43%, 4.58% and 4.52% faster than Diffie-Hellman key exchange with BEd251 respectively.

	Sandy Bridge	Haswell	Skylake
BKL251	1,73,351	2,45,404	1,81,852
BEd251	1,83,319	2,57,208	1,90,480

Table 15. Timings of Diffie-Hellman Key Exchange for BKL251 and BEd251 in clock cycles

8 Conclusion

This work exhibits that Binary Kummer line based scalar multiplication offers competitive performance compared to existing proposal like BEd251 and CURVE2251 over finite field of characteristics 2 using `PCLMULQDQ`. Previous implementations of BEd251 and CURVE2251 focuses on batch implementation using bitslicing technique. This work presents the first ever implementation of the proposed BKL251 and BEd251 using the instruction `PCLMULQDQ` (best to our knowledge). From the experimental results, we conclude that BKL251 is approximately 9% and 1% faster than BEd251 for fixed-base and variable-base scalar multiplication respectively.

References

1. D. F. Aranha. Relic-toolkit, 2019. `https://github.com/relic-toolkit`.
2. R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & HallCRC, 1st edition, 2006.
3. D. J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography – PKC*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.
4. D. J. Bernstein. Batch binary edwards. In *Advances in Cryptology - CRYPTO*, volume 5677 of *LNCS*, pages 317–336. Springer, 2009.
5. D. J. Bernstein. Batch binary edwards, 2017. `https://binary.cr.yp.to/edwards.html`.
6. D. J. Bernstein and T. Lange. Explicit-formulas database, 2019. `https://www.hyperelliptic.org/EFD/`.
7. D. J. Bernstein, T. Lange, and R. R. Farashahi. Binary edwards curves. In *Cryptographic Hardware and Embedded Systems - CHES*, volume 5154 of *LNCS*, pages 244–265, 2008.
8. B. B. Brumley, S. ul Hassan, A. Shaindlin, N. Tuveri, and Kide Vuojärvi. Batch binary weierstrass. In *Advances in Cryptology - LATINCRYPT*, volume 11774 of *LNCS*, pages 364–384. Springer, 2019.
9. C. Costello and P. Longa. Four($\mathbb{Q}$): Four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In *Advances in Cryptology - ASIACRYPT Part I*, volume 9452 of *LNCS*, pages 214–235. Springer, 2015.
10. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions of Information Theory*, 22(6):644–654, 1976.
11. P. Gaudry, 2019. Personal Communication.
12. P. Gaudry and D. Lubicz. The arithmetic of characteristic 2 kummer surfaces and of elliptic kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009.
13. P. Gaudry and E. Thomé. The mpfq library and implementing curve-based key exchanges. *SPEED: Software Performance Enhancement for Encryption and Decryption*, pages 49–64, 06 2007.
14. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Publishing Company, Incorporated, 1st edition, 2010.
15. Intel. Intel intrinsics guide, 2019. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/\#`.
16. S. Karati and P. Sarkar. Kummer for genus one over prime-order fields. *Journal of Cryptology*, pages 1–38, 2019. `https://doi.org/10.1007/s00145-019-09320-4`.
17. N. Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, 1987.
18. Neal Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3):139–150, 1989.
19. C. H. Lim and P. J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Advances in Cryptology - CRYPTO*, volume 1294 of *LNCS*, pages 249–263. Springer, 1997.
20. A. Menezes and M. Qu. Analysis of the weil descent attack of gaudry, hess and smart. In *Topics in Cryptology – CT-RSA*, volume 2020 of *LNCS*, pages 308–318. Springer, 2001.
21. V. S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO*, LNCS, pages 417–426. Springer, 1985.
22. P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243â243, 1987.
23. P. L. Montgomery. Five, six, and seven-term karatsuba-like formulae. *IEEE Trans. Computers*, 54(3):362–369, 2005.
24. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

25. NIST. Fips pub 186-4: Digital signature standard (dss), 2013. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.
26. J. Renes and B. Smith. qDSA: Small and secure digital signatures with curve-based Diffie-Hellman key pairs. In *Advances in Cryptology - ASIACRYPT*, volume 10625 of *LNCS*, pages 273–302. Springer, 2017.
27. J. Taverne, A. Faz-Hernández, D. F. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López Hernandez. Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. *J. Cryptographic Engineering*, 1(3):187–199, 2011.