Kummer versus Montgomery Face-off over Prime Order Fields

Kaushik Nath and Palash Sarkar

Applied Statistics Unit
Indian Statistical Institute
203, B. T. Road
Kolkata - 700108
India
{kaushikn_r,palash}@isical.ac.in

Abstract

This paper makes a comprehensive comparison of the efficiencies of vectorized implementations of Kummer lines and Montgomery curves at various security levels. For the comparison, nine Kummer lines are considered, out of which eight are new, and new assembly implementations of all nine Kummer lines have been made. Seven previously proposed Montgomery curves are considered and new vectorized assembly implementations have been made for three of them. Our comparisons show that for all security levels, Kummer lines are consistently faster than Montgomery curves, though the speed-up gap is not much.

Keywords: Kummer line, Montgomery curve, Diffie-Hellman, 4-way vectorization, SIMD.

1 Introduction

Diffie-Hellman (DH) [11] key agreement scheme is a cornerstone of modern cryptography. Presently, elliptic curves [20, 22] provide the most efficient way to instantiate DH key agreement. The DH key agreement is a two-phase protocol, consisting of a key generation phase and a shared secret generation phase. Montgomery form elliptic curves [23] are well suited for implementing the shared secret generation phase of the DH key agreement. The famous Curve25519 [4] is a Montgomery form curve which has been proposed for the 128-bit security level. TLS version 1.3 [32] has standardized Curve25519 and another Montgomery form curve named Curve448 which has been proposed for the 224-bit security level.

Kummer lines have been considered [14, 15] as alternatives to elliptic curve for instantiating cryptographic protocols. For the 128-bit security level, three concrete proposals of Kummer lines have been put forward in [19]. This work showed that the field arithmetic operations required for Kummer line operations are naturally amenable to 4-way SIMD operations. Concrete implementations using vector operations available in modern processors have been reported in [19]. Timing results showed significant efficiency improvement over Curve25519.

Our Contributions

At the time the work [19] was done, it was not known how to implement the field arithmetic operations required for Montgomery curve operations using 4-way vectorization. More recently, such 4-way vectorization of Montgomery curve operations have been reported in [17, 24]. In [17], a comparison of the 4-way vectorization of Montgomery curve to the 4-way vectorized implementation of Kummer lines in [19] was made and speed improvements of Curve25519 over the Kummer lines in [19] was reported. Further, [12] also reported speed improvements of a sequential implementation of Curve25519 over the Kummer lines in [19].

In view of the above, the goal of the present work is to make a systematic comparative study of the efficiency of vectorized implementation of Kummer line with that of Montgomery curve. The security levels of 128-bit, 224-bit and 256-bit were considered. The Kummer lines proposed in [19] are at the 128-bit security level. Since, 224-bit and 256-bit security levels are within the ambit of our study, we implemented the search algorithms for Kummer lines at these security levels. This yielded new Kummer lines. Also, at the 128-bit security level, we obtained two new Kummer lines with improved parameters

in comparison to the Kummer lines reported in [19]. In all, a total of nine Kummer lines (three each at the 128-bit, 224-bit and 256-bit) security levels have been considered in the present work, out of which only one is from [19], while the other eight are new. Further, we prove a result which shows that the co-factors of eight of the curves are optimal.

Efficient assembly implementations for all the nine Kummer lines have been made. These implementations make use of the 4-way vector instructions provided for modern Intel processors. The previous vectorized implementations of Kummer lines reported in [19] used Intel intrinsic. More importantly, we identify several new optimization strategies which have not been considered in [19]. The net effect is that we obtain a substantial speed improvement over the implementations reported in [19].

Since our goal is to compare to Montgomery curves, we considered such curves also at the 128-bit, 224-bit and the 256-bit security levels. As mentioned earlier, Curve25519 and Curve448 are targeted at the 128-bit and the 224-bit security levels respectively. Other proposals of Montgomery curves at the 128-bit and the 224-bit security levels have been put forward in [29]. Also, proposals for Montgomery curves at the 256-bit security level have been made in [27]. Vectorized implementations of Curve25519 have been reported in [17]. The work [24] provides faster vectorized implementations of Curve25519 and also provides vectorized implementation of Curve448. Vectorized implementations of the curves proposed in [27] have not appeared in the literature. To obtain a meaningful comparison, we made new assembly implementations of three Montgomery curves from [27]. The assembly codes of all our implementations (i.e., including Kummer lines and Montgomery curves) are available at the following links.

https://github.com/kn-cs/kummer-genus-one
https://github.com/kn-cs/mont256-vec

We have made a detailed performance comparison of Kummer lines and Montgomery curves on the Haswell and Skylake processors of Intel. The timing results show that Kummer lines are faster than Montgomery curves. This is to be expected, since the 4-way vectorization of the Kummer line is simpler than that of the Montgomery curve. We believe the timing results settles the issue of which of Kummer line and Montgomery curve is faster for vectorized implementations. Improvements to the latency and throughput of vector instructions will lead to further improvement of the efficiency gain of Kummer line over Montgomery curve.

While, Kummer lines are indeed faster for vectorized implementations, we note that for sequential implementations Kummer lines will be inherently slower than Montgomery curves. So, the actual choice of Kummer line or Montgomery curve for a particular application will depend on the anticipated balance of sequential versus vectorized implementations. If mostly vectorized implementations are conceived, then it would be better to deploy Kummer lines, otherwise Montgomery curves will be a better choice.

2 Background

We consider elliptic curves and Kummer lines over prime order fields. Let $p \neq 2, 3$ be a prime and $\mathbb{F}_p$ be the finite field of cardinality p. The algebraic closure of $\mathbb{F}_p$ is denoted as $\overline{\mathbb{F}}_p$.

2.1 Elliptic Curves

We provide a brief description of the relevant ideas of elliptic curves to the extent required in the present work. A good introduction to the subject can be found in [33].

An elliptic curve E is the set of all points $(x, y) \in \overline{\mathbb{F}}_p \times \overline{\mathbb{F}}_p$ satisfying an appropriate equation along with a point at infinity denoted as ∞. Under a suitably defined addition operation, an elliptic curve forms a group with ∞ as the identity element. The subgroup $E(\mathbb{F}_p)$ is the set of all $\mathbb{F}_p$-rational points, i.e., along with ∞, it contains the set of all points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ which satisfies the given equation. Points given in the form (x, y) are called affine points. Projective coordinates are of the form $(X : Y : Z)$. If $Z \neq 0$, then $(X : Y : Z)$ corresponds to the affine point $(X/Z, Y/Z)$. The only projective point on E with $Z = 0$ is $(0 : 1 : 0)$ and this is the identity element of the group.

For a point $P = (X : Y : Z)$ on an elliptic curve E, the x-coordinate map $\mathbf{x}$ is the following [10]: $\mathbf{x}(P) = (X : Z)$ if $Z \neq 0$ and $\mathbf{x}(P) = (1 : 0)$ if $P = (0 : 1 : 0)$. Bernstein [4, 3] introduced the map $\mathbf{x}_0$ as follows: $\mathbf{x}_0(X : Z) = XZ^{p-2}$ which is defined for all values of X and Z in $\mathbb{F}_p$.

We will specifically be interested in the following two forms of the equation.

Montgomery form: For $A, B \in \mathbb{F}_p$ such that $B(A^2 - 4) \neq 0$, the Montgomery form $E_{M,A,B}$ is given by the equation $By^2 = x(x^2 + Ax + 1)$ in affine coordinates and by the equation $BY^2Z = X(X^2 + AXZ + Z^2)$ in projective coordinates. It is known that the order of $E_{M,A,B}(\mathbb{F}_p)$ is a multiple of 4.

Legendre form: For $\mu \in \mathbb{F}_p$ such that $\mu \neq 0, 1$, the Legendre form $E_{L,\mu}$ is given by the equation $y^2 = x(x-1)(x-\mu)$ in affine coordinates and by the equation $Y^2Z = X(X-Z)(X-Z\mu)$ in projective coordinates. There are three distinct points of order 2, which in projective coordinates are $(0:0:1)$, $(1:0:1)$ and $(\mu:0:1)$. These three points, along with the identity element form a subgroup of order 4 of $E(\mathbb{F}_p)$ and consequently, 4 divides $\#E(\mathbb{F}_p)$.

For computational efficiency, it is preferable to work with projective coordinates.

Algorithm 1 Differential addition operation on Montgomery curve $E_{M,A,1}$.

function DIFF-ADD-MC($[X_1 : Z_1], [X_2 : Z_2], [X : Z]$)
input: $[X : Z], [X_1 : Z_1], [X_2 : Z_2] \in E_{M,A,B}$.
output: $[X_3 : Z_3] \in E_{M,a,b}$.
    $R \leftarrow (X_1 - Z_1) + (X_2 + Z_2)$
    $S \leftarrow (X_1 - Z_1) - (X_2 + Z_2)$
    $X_3 \leftarrow Z(R+S)^2$
    $Z_3 \leftarrow X(R-S)^2$
    return $[X_3 : Z_3]$
end function.

Algorithm 2 Double operation on Montgomery curve $E_{M,A,1}$.

function DOUBLE-MC($[X_1 : Z_1]$)
input: $[X_1 : Z_1] \in E_{M,a,b}$.
output: $[X_3 : Z_3] \in E_{M,a,b}$.
    $R \leftarrow (X_1 + Z_1)$
    $S \leftarrow (X_1 - Z_1)$
    $X_3 \leftarrow R^2 \cdot S^2$
    $Z_3 \leftarrow (R^2 - S^2)(S^2 + a_{24}(R^2 - S^2))$
    return $[X_3 : Z_3]$
end function.

Algorithm 3 Montgomery ladder.

function LADDER(P, n)
input: P is a projective point $[X : Z] \in E_{M,A,B}$, n is an ℓ-bit scalar such that $n = (n_{\ell-1}, n_{\ell-2} \ldots n_0)$.
output: x-coordinate of nP, the n-times scalar multiple of P.
    $R \leftarrow \mathbf{x}(\infty)$; $S \leftarrow \mathbf{x}(P)$
    for $i \leftarrow \ell - 1$ down to 0 **do**
        $\langle R, S \rangle \leftarrow$ LADDER-STEP(R, S, n_i)
    end for
    let $R = [U : V]$
    return UV^{p-2}
end function.

Scalar multiplication on $E(\mathbb{F}_p)$ is the operation of computing the r-fold addition rP of a point $P \in E(\mathbb{F}_p)$, where r is a non-negative integer. Since $E(\mathbb{F}_p)$ is a group, this can be done using the usual double-and-add algorithm. Montgomery [23] introduced a variant of the usual double-and-add algorithm to compute the value of rP over Montgomery curves that uses only the x-coordinate of the input. This is called the Montgomery ladder and we describe this algorithm using projective coordinates. Let $P = (X : Y : Z)$, and $R = [X_1 : Y_1 : Z_1]$ and $S = [X_2 : Y_2 : Z_2]$ be such that $S - R = P$. The operations differential addition and doubling respectively denoted as DIFF-ADD-MC(R, S, P) and DOUBLE-MC(R) are described in Algorithms 1 and 2. In Algorithm 2, the constant a_{24} is equal to $(A+2)/4$. The operations DIFF-ADD-MC and DOUBLE-MC do not involve the Y-coordinate and the parameter B is not required in the computation. The Montgomery ladder built using the differential addition and doubling operations is shown in Algorithm 3 which uses Algorithm 4 as a sub-routine. In the Montgomery

ladder, Algorithm 1 is used to implement the operation DIFF-ADD and Algorithm 2 is used to implement the operation DOUBLE.

Algorithm 4 A single ladder-step based on the differential add and double operations.

function LADDER-STEP($R, S, \mathfrak{b}$)
  if $\mathfrak{b} = 0$ **then**
    $S \leftarrow$ DIFF-ADD(R, S, P)
    $R \leftarrow$ DOUBLE(R)
  else
    $R \leftarrow$ DIFF-ADD(R, S, P)
    $S \leftarrow$ DOUBLE(S)
  end if
  return $\langle R, S \rangle$
end function.

Algorithm 4 has a conditional branching. In practice, this may result in non-constant time behavior leading to an insecure implementation. There are well known ways of implementing the ladder-step in constant time. We refer to [25] for an overview of several of the methods that have been proposed in the literature for constant time implementation. Constant time vectorized (both 2-way and 4-way) implementations of Algorithm 4 have been reported in the literature [8, 9, 12, 17, 24]. For further details on Montgomery curves we refer to [23, 10, 7].

2.2 Kummer Lines

Kummer lines over genus one are defined using Jacobi theta functions over the complex field [14, 15, 19]. The derivations for the identities involving the theta functions have a good reduction [14, 15] and so the Lefschetz principle [1, 13] can be used to carry over the identities which hold over the complex field to those over a large characteristic field. In view of this, Kummer lines over $\mathbb{F}_p$, where p is a large prime, have been considered in [14, 15, 19, 18] and we also do the same.

Algorithm 5 Differential addition on the Kummer line $\mathcal{K}_{a^2,b^2}$.

function DIFF-ADD-KL($[x_1^2 : z_1^2], [x_2^2 : z_2^2], [x^2 : z^2]$)
input: $[x^2 : z^2], [x_1^2 : z_1^2], [x_2^2 : z_2^2] \in \mathcal{K}_{a^2,b^2}$.
output: $[x_3^2 : z_3^2] \in \mathcal{K}_{a^2,b^2}$.
  $r \leftarrow (a^2 - b^2)(x_1^2 + z_1^2)(x_2^2 + z_2^2)$
  $s \leftarrow (a^2 + b^2)(x_1^2 - z_1^2)(x_2^2 - z_2^2)$
  $x_3^2 \leftarrow z^2(r + s)^2$
  $z_3^2 \leftarrow x^2(r - s)^2$
  return $[x_3^2 : z_3^2]$
end function.

Algorithm 6 Doubling on the Kummer line $\mathcal{K}_{a^2,b^2}$.

function DOUBLE-KL($[x_1^2 : z_1^2]$)
input: $[x_1^2 : z_1^2] \in \mathcal{K}_{a^2,b^2}$.
output: $[x_3^2 : z_3^2] \in \mathcal{K}_{a^2,b^2}$.
  $r \leftarrow (a^2 - b^2)(x_1^2 + z_1^2)^2$
  $s \leftarrow (a^2 + b^2)(x_1^2 - z_1^2)^2$
  $x_3^2 \leftarrow b^2(r + s)^2$
  $z_3^2 \leftarrow a^2(r - s)^2$
  return $[x_3^2 : z_3^2]$
end function.

A Kummer line $\mathcal{K}_{a^2,b^2}$ is a subset of the projective line $\mathbb{P}^1(\mathbb{F}_p)$ and is determined by two parameters a^2 and b^2 both of which are non-zero elements of $\mathbb{F}_p$. It is not required that a and b are in $\mathbb{F}_p$. Points

on $\mathcal{K}_{a^2,b^2}$ do not form a group. Suppose $a^2, b^2 \in \mathbb{F}_p$ with $a^4 \neq b^4$. Then the Kummer line $\mathcal{K}_{a^2,b^2}$ is associated to the Legendre form elliptic curve $E_{L,\mu} : y^2 = x(x-1)(x-\mu)$ where $\mu = a^4/(a^4 - b^4)$. The conditions $\mu \neq 0, 1$ requires $ab \neq 0 \bmod p$. Scalar multiplication on $E_{L,\mu}$ can be performed by moving to its associated Kummer line, performing the scalar multiplication there and then moving back. We refer to [18] for details.

Even though a Kummer line does not form a group, the operations of doubling and differential addition can be carried out. Given $P = [x^2 : z^2]$ on $\mathcal{K}_{a^2,b^2}$, Algorithm 6 shows how to obtain $2P = [x_3^2 : z_3^2]$; given $R = [x_1^2 : z_1^2]$, $S = [x_2^2 : z_2^2]$ such that $S - R = P$, Algorithm 5 shows how to obtain $R + S = [x_3^2 : z_3^2]$.

Given the differential addition and doubling algorithms for the Kummer line, the ladder algorithm given in Algorithm 3 can be used to compute nP, where $P = [x^2 : z^2]$ and n is a scalar. Algorithm 3 calls Algorithm 4 which in turn calls Algorithm 5 and Algorithm 6 for differential addition and doubling respectively. In effect, simply substituting the differential addition and the doubling algorithms of the Montgomery curve with those of the Kummer line provides the scalar multiplication algorithm over the Kummer line. For further details on Kummer lines we refer to [14, 15, 19, 18].

2.3 Diffie-Hellman Key Agreement

The map $\mathbf{x}_0$ defined in Section 2.1 can also be applied to a point $(x^2 : z^2) \in \mathcal{K}_{a^2,b^2}$ to obtain $\mathbf{x}_0(x^2 : z^2) = x^2(z^2)^{p-2}$. Following Miller [22] and Bernstein [4], the Diffie-Hellman key agreement can be carried out on a Montgomery curve as follows. Let Q be a generator of a prime order subgroup of $E_{M,A,B}(\mathbb{F}_p)$. Alice chooses a secret key s_1 and has public key $\mathbf{x}_0(s_1Q)$; Bob chooses a secret key s_2 and has public key $\mathbf{x}_0(s_2Q)$. The shared secret key of Alice and Bob is $\mathbf{x}_0(s_1s_2Q)$ which is computed by Alice from s_1 and $\mathbf{x}_0(s_2Q)$ and by Bob from s_2 and $\mathbf{x}_0(s_1Q)$. The computation has two phases, namely the key generation phase, in which Alice and Bob compute their public keys; and the shared secret computation phase, in which Alice and Bob compute the shared secret. The shared secret computation of both Alice and Bob consists of the following task. Given $(X_1 : Z_1)$ corresponding to a point $P = (X_1 : Y_1 : Z_1)$ and a non-negative integer r, obtain $\mathbf{x}_0(rP)$.

Diffie-Hellman key agreement can be implemented using Legendre form curves and associated Kummer lines. In this case, Q is a generator of a prime order subgroup of $E_{L,\mu}(\mathbb{F}_p)$ and let Q' be the corresponding point on the associated Kummer line $\mathcal{K}_{a^2,b^2}$. Suppose as before that s and t are the secret keys of Alice and Bob respectively. Then the public keys of Alice and Bob are $\mathbf{x}_0(s_1Q')$ and $\mathbf{x}_0(s_2Q')$ respectively and the shared secret is $\mathbf{x}_0(s_1s_2Q')$.

In the above description, the points Q and Q' are fixed and can usually be chosen such that the coordinates are small. Consequently, the scalar multiplication nP, where P is either Q or Q' can be significantly sped up. Such a scalar multiplication is usually called fixed based scalar multiplication. On the other hand, if P is an arbitrary point, the scalar multiplication nP is called a variable base scalar multiplication. Note that the key generation phase requires a fixed base scalar multiplication, whereas the shared secret computation phase requires a variable base scalar multiplication. The computation of the shared secret over either Montgomery form curves, or over Kummer lines can be implemented using the ladder algorithm. The appropriate differential add and doubling algorithms need to be used.

For the key generation phase, on the other hand, it is faster to perform the computation on a (twisted) Edwards form curve. For Montgomery curves proposed in the literature, birationally equivalent Montgomery curves are known [6, 21, 29, 27]. In the case of Kummer lines, for $p \equiv 3 \bmod 4$, it is possible to use Theorems 3.3 and 3.4 of [5] to obtain an Edwards form curve which is birationally equivalent to $E_{L,\mu}$, while for $p \equiv 1 \bmod 4$, it is possible to use the methods of [18] to obtain a desired twisted Edwards form curve. For both Montgomery curves and Kummer lines, the associated (twisted) Edwards form curve can be used to build an efficient signature scheme based on the suggestion in [6]. Since our focus in this paper is the shared secret computation phase of the DH protocol, we skip further details of the key generation phase of the protocol and also of the signature schemes.

2.4 Security

Our consideration of security is based on the recommendations provided in [2].

Let E be an elliptic curve over $\mathbb{F}_p$ and $n = \#E(\mathbb{F}_p)$. From Hasse's theorem, we have $n = p + 1 - t$, where $|t| \leq 2\sqrt{p}$. Suppose, it is possible to write $n = h\ell$, where ℓ is a prime. Then h is called the co-factor of the curve. Cryptography is done over the order ℓ subgroup of $E(\mathbb{F}_p)$. On classical computers, the best known algorithm for computing discrete logarithm in the cryptographic subgroup of order ℓ requires about $O(\ell^{1/2})$ time. If h is small, then $O(\ell^{1/2})$ is about $O(2^{m/2})$, where $m = \lceil \lg p \rceil$. In this case, the security level is said to be $m/2$ bits.

A quadratic twist of E has order $n_T = p + 1 + t$ and so $n + n_T = 2(p + 1)$. If it is possible to write $n_T = h_T \ell_T$, where ℓ_T is a prime, then h_T is the co-factor of a quadratic twist of E. The embedding degrees k and k_T of the curve and its twist are important security parameters. Here k (resp. k_T) is the smallest positive integer such that $\ell | p^k - 1$ (resp. $\ell_T | p^{k_T} - 1$). The value of k (resp. k_T) is equal to the order of p in $\mathbb{F}_\ell$ (resp. $\mathbb{F}_{\ell_T}$) and is found by checking the factors of $\ell - 1$ (resp. $\ell_T - 1$). The complex multiplication field discriminant D is defined in the following manner [2]: Hasse's theorem states that $|t| \leq 2\sqrt{p}$ and in the cases that we considered $|t| < 2\sqrt{p}$ so that $t^2 - 4p$ is a negative integer; let s^2 be the largest square dividing $t^2 - 4p$; define $D = (t^2 - 4p)/s^2$ if $(t^2 - 4p)/s^2 \bmod 4 = 1$ and $D = 4(t^2 - 4p)/s^2$ otherwise. (Note that D is different from the discriminant of E_μ.)

Recommendations in [2] suggest choosing curves such that both h and h_T are small, k and k_T are large and also $|D|$ is large. Note that if h and h_T are small, this implies that ℓ and ℓ_T are large.

2.5 Efficient Implementation of Kummer versus Montgomery

High-speed implementation of the shared secret computation phase of the DH protocol is important from a practical point of view. This requires a high-speed implementation of variable base scalar multiplication. The issue of efficient implementation of variable base scalar multiplication on Montgomery curves has been considered in a number of papers in the literature [4, 30, 27]. Such works have mostly focused on sequential implementations. Modern processors provide opportunities for vectorized implementations. A few works [9, 12] have investigated 2-way vectorization of scalar multiplication over Montgomery curves. More recently, 4-way vectorization of Montgomery ladder has been reported in [17, 24].

The combined differential addition and doubling operation on the Kummer line has a natural 4-way vectorization as has been pointed out in [19]. This makes Kummer lines specially attractive from the point of view of vectorized implementation. Such implementation for three Kummer lines at the 128-bit security level has been reported in [19].

For a comparison of Kummer versus Montgomery, the first point to consider is the number of operations in the computation of Algorithm 4. For Montgomery curves, Algorithm 4 will call Algorithms 1 and 2 for differential addition and doubling respectively. In Algorithm 1, Z may be assumed to be 1. So, the operations required by Algorithm 4 for Montgomery curves consist of 5 multiplications, 4 squarings and 1 multiplication by a (small) field constant. On the other hand, for Kummer lines, Algorithm 4 will call Algorithms 5 and 6 for differential addition and doubling respectively. In Algorithm 5, z^2 may be assumed to be 1. So, the operations required by Algorithm 4 for Kummer lines consist of 3 multiplications, 6 squarings and 6 multiplications by (small) field constants. The trade-off between Montgomery curves and Kummer lines is that 2 less multiplications are required for Kummer lines at the cost of 2 squarings and 5 multiplications by field constants. In practice, a squaring will be noticeably faster than a multiplication. Even then the time for 2 squarings and 5 multiplications by small field constants will be more than the time for 2 multiplications. So, for a sequential implementation, Kummer lines will be slower than Montgomery curves, but not by much.

Next we consider 4-way vectorization of Montgomery curves and Kummer lines. For Montgomery curves, 4-way vectorization have been proposed in [17, 24]. Detailed analysis and implementation results show that the 4-way vectorization proposed in [24] is faster than the one proposed in [17]. A top-level schematic diagram of the 4-way vectorization of Montgomery curves proposed in [24] is shown in Figure 1. A top-level schematic diagram of the 4-way vectorization of Kummer lines is shown in Figure 2. Both the diagrams suggest that the 4-way vectorization can be completed using 2 multiplication rounds, one squaring round and one multiplication-by-constant round. The Kummer line vectorization is simpler in comparison to the Montgomery curve vectorization. This should help in improved efficiency for the vectorized implementation of the Kummer line operations. The detailed implementation that we describe later on confirms this intuition.

3 Searching for a Secure Kummer Line

Consider a Kummer line $\mathcal{K}_{a^2,b^2}$ associated with the Legendre form curve $E_{L,\mu}$ over $\mathbb{F}_p$. As in Section 2.4, let $n = \#E_{L,\mu}(\mathbb{F}_p)$ and n_T be the order of its quadratic twist. Using $n + n_T = 2(p + 1)$ and the fact that 4 divides n, it is not difficult to argue that for $p \equiv 3 \bmod 4$, the minimum possible value of (h, h_T) is $(4, 4)$ and for $p \equiv 1 \bmod 4$, the minimum possible value of (h, h_T) is either $(4, 8)$ or $(8, 4)$. Since the goal is to obtain co-factors with the minimum possible values, in the case of $p \equiv 3 \bmod 4$, one may wish to obtain curves such that $(h, h_T) = (4, 4)$. The next result shows that this will not be possible.

Theorem 1. *Let $p \equiv 3 \bmod 4$ be a prime and $a^2, b^2 \in \mathbb{F}_p$ with $a^4 \neq b^4$. Let $E_{L,\mu}$ be a Legendre form curve with $\mu = a^4/(a^4 - b^4)$. Then 8 divides $\#E_{L,\mu}(\mathbb{F}_p)$.*

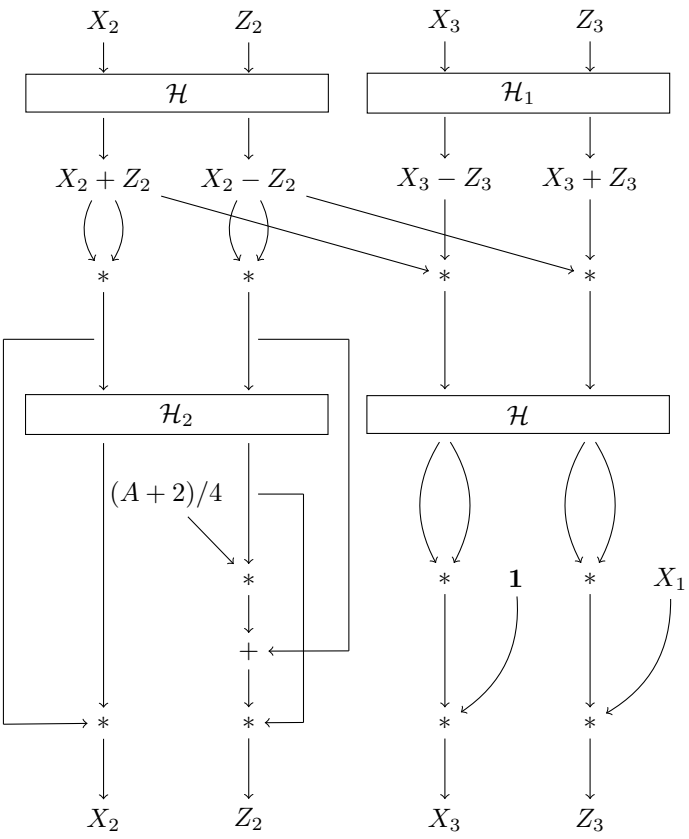

Figure 1: A top-level schematic diagram for combined doubling and differential addition for Montgomery curves which is reproduced here from [24]. In the figure, $\mathcal{H}(a,b) = (a+b, a-b), \mathcal{H}_1(a,b) = (a-b, a+b), \mathcal{H}_2(a,b) = (b, a-b)$.

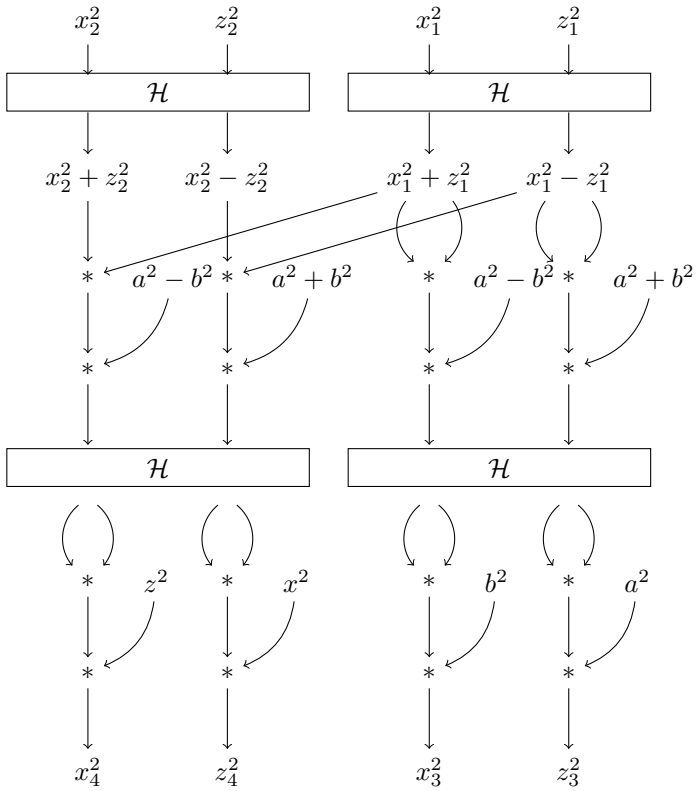

Figure 2: A top-level schematic diagram for combined doubling and differential addition for Kummer lines which is reproduced here from [19]. In the figure, $\mathcal{H}(a,b) = (a+b, a-b)$.

$x_2 = 0$	$x_1 = \sqrt{\mu}$	$y_1 = \pm\sqrt{-\mu^2 + 2\mu^{3/2} - \mu}$
	$x_1 = -\sqrt{\mu}$	$y_1 = \pm\sqrt{-\mu^2 - 2\mu^{3/2} - \mu}$
$x_2 = 1$	$x_1 = 1 + \sqrt{1-\mu}$	$y_1 = \pm(-1 + \mu - \sqrt{1-\mu})$
	$x_1 = 1 - \sqrt{1-\mu}$	$y_1 = \pm(-1 + \mu + \sqrt{1-\mu})$
$x_2 = \mu$	$x_1 = \mu + \sqrt{\mu^2 - \mu}$	$y_1 = \pm\left(2\mu^3 + 2\mu^2\sqrt{\mu^2-\mu} - 3\mu^2 - 2\mu\sqrt{\mu^2-\mu} + \mu\right)^{1/2}$
	$x_1 = \mu - \sqrt{\mu^2 - \mu}$	$y_1 = \pm\left(2\mu^3 - 2\mu^2\sqrt{\mu^2-\mu} - 3\mu^2 + 2\mu\sqrt{\mu^2-\mu} + \mu\right)^{1/2}$

Table 1: Values of x_1, y_1 and x_2 which are solutions to (1).

Proof. The proof uses some results from [18].

First note that $E_{L,\mu}(\mathbb{F}_p)$ has three points of order 2 which along with the identity forms a subgroup of order 4. So, 4 certainly divides $\#E_{L,\mu}(\mathbb{F}_p)$. From Proposition 1 in [18] we have that 8 divides $\#E_{L,\mu}(\mathbb{F}_p)$ if and only if $E_{L,\mu}(\mathbb{F}_p)$ has a point of order 4. So, to prove the result, it is sufficient to show that $E_{L,\mu}(\mathbb{F}_p)$ has a point of order 4.

Proposition 2 in [18] shows that the point (x_1, y_1) is of order 4 if and only if x_1 and y_1 are solutions of the equations

$$\left.\begin{array}{rcl} x_1^3 - 3x_2x_1^2 + (2(\mu+1)x_2 - \mu)x_1 - \mu x_2 & = & 0 \\ 2y_1^2 - (3x_1^2 - 2(\mu+1)x_1 + \mu)(x_1 - x_2) & = & 0 \end{array}\right\} \tag{1}$$

for some $x_2 \in \{0, 1, \mu\}$ and $x_1 \neq x_2$. Further, the explicit formulas for the possible solutions for (x_1, y_1) arising from (1) have been obtained in [18] and are reproduced in Table 1.

The solutions listed in Table 1 are not necessarily in $\mathbb{F}_p$. For $p \equiv 3 \bmod 4$, we argue that one of the solutions in Table 1 is indeed in $\mathbb{F}_p$ giving rise to a point of order 4 in $E_{L,\mu}(\mathbb{F}_p)$ which, as indicated above, will prove the result.

Since $p \equiv 3 \bmod 4$, -1 is a non-square modulo p. The solutions for (x_1, y_1) corresponding to $x_2 = 0$ in Table 1, are either $(\sqrt{\mu}, \pm(\sqrt{mu} - 1)\sqrt{-\mu})$ or $(-\sqrt{\mu}, \pm(\sqrt{\mu}+1)\sqrt{-\mu})$. Since -1 is a non-square, exactly one of μ and $-\mu$ is a square and the other is a non-square and in either case, the solutions (x_1, y_1) are not in $\mathbb{F}_p$. So, we consider the other two cases for x_2, namely $x_2 = 1$ and $x_2 = \mu$.

Note that $\mu = a^4/(a^4 - b^4)$ where a^2 and b^2 are in $\mathbb{F}_p$ and so a^4 and b^4 are squares modulo p. So, $\mu - 1 = b^4/(a^4 - b^4)$ is a square in $\mathbb{F}_p$ if and only if μ is a square in $\mathbb{F}_p$, and consequently, $1 - \mu$ is a square in $\mathbb{F}_p$ if and only if μ is a non-square in $\mathbb{F}_p$. So, if μ is a square in $\mathbb{F}_p$, then $\mu^2 - \mu = \mu(\mu - 1)$ is a square in $\mathbb{F}_p$ and from Table 1, the solutions for (x_1, y_1) corresponding to $x_2 = \mu$ are in $\mathbb{F}_p$. On the other hand, if μ is not a square in $\mathbb{F}_p$, both $1 - \mu$ and $\mu^2 - \mu = \mu(\mu - 1)$ are squares in $\mathbb{F}_p$ so that the solutions corresponding to both $x_2 = 1$ and $x_2 = \mu$ are in $\mathbb{F}_p$. □

In view of Theorem 1, it follows that if $p \equiv 3 \bmod 4$, then 8 divides n and using $n + n_T = 2(p + 1)$, it further follows that 8 also divides n_T. So, in the case $p \equiv 3 \bmod 4$, the minimum possible value of (h, h_T) is $(8, 8)$.

The strategy to search for an appropriate Kummer line $\mathcal{K}_{a^2,b^2}$ is to consider a range of values for (a^2, b^2) and for each pair of values, define $\mu = a^4/(a^4 - b^4)$ and consider the Legendre form curve $E_{L,\mu}$. The goal is to obtain (a^2, b^2) such that (h, h_T) for $E_{L,\mu}$ is the minimum possible (correspondingly, ℓ and ℓ_T is the maximum possible) and k, k_T and $|D|$ are large. In this work, the range for the values for both a^2 and b^2 is considered to be 2-1023 while in [19] this range was considered to be 2-512 for both a^2 and b^2. For two of the three primes considered in [19], the values of (h, h_T) were sub-optimal. For these primes, the expanded search strategy used in the present work has led to curves with optimal values of (h, h_T).

4 Concrete Curves

We consider a total of nine primes in this work at the three security levels 128-bit, 224-bit and 256-bit. At each level of security we consider certain Montgomery curves which have been defined earlier in literature. The different primes are provided in Table 2.

4.1 Montgomery Curves

The Montgomery curves at the various security levels considered in this work are provided in Table 3. The parameters of the above curves are shown in Table 4. The X and Z coordinates of the corresponding

Security level = 128 bits	Security level = 224 bits	Security level = 256 bits
$p251\text{-}9 = 2^{251} - 9$	$p444\text{-}17 = 2^{444} - 17$	$p506\text{-}45 = 2^{506} - 45$
$p255\text{-}19 = 2^{255} - 19$	$p448\text{-}224\text{-}1 = 2^{448} - 2^{224} - 1$	$p510\text{-}75 = 2^{510} - 75$
$p266\text{-}3 = 2^{266} - 3$	$p452\text{-}3 = 2^{452} - 3$	$p521\text{-}1 = 2^{521} - 1$

Table 2: Primes considered in this work at various security levels.

base points are provided using projective coordinates.

Security level	Montgomery curve	Prime	Reference
128 bit	M⟦4698⟧	$p251\text{-}9$	[29]
	Curve25519	$p255\text{-}19$	[4]
224 bit	M⟦4058⟧	$p444\text{-}17$	[29]
	Curve448	$p448\text{-}224\text{-}1$	[16]
256 bit	M⟦996558⟧	$p506\text{-}45$	[27]
	M⟦952902⟧	$p510\text{-}75$	[27]
	M⟦1504058⟧	$p521\text{-}1$	[27]

Table 3: Curves considered in this work at various security levels.

Montgomery Curve	$(\lg \ell, \lg \ell_T)$	(h, h_T)	(κ, κ_T)	$\lg(-D)$	Base point
M⟦4698⟧	(249,249)	(4,4)	$\left(\ell - 1, \frac{\ell_T - 1}{2}\right)$	253	[3:1]
Curve25519	(252,253)	(8,4)	$\left(\frac{\ell-1}{6}, \frac{\ell_T-1}{2}\right)$	254.7	[9:1]
M⟦4058⟧	(442,442)	(4,4)	$\left(\frac{\ell-1}{3}, \ell_T - 1\right)$	446	[3:1]
Curve448	(446,446)	(4,4)	$\left(\frac{\ell-1}{2}, \frac{\ell_T-1}{4}\right)$	449.5	[5:1]
M⟦996558⟧	(504,504)	(4,4)	$\left(\frac{\ell-1}{17}, \ell_T - 1\right)$	508	[3:1]
M⟦996558⟧	(507,508)	(8,4)	$(\ell - 1, \ell_T - 1)$	510	[4:1]
M⟦996558⟧	(519,519)	(4,4)	$(\ell - 1, \ell_T - 1)$	523	[8:1]

Table 4: Montgomery curves and their parameters.

4.2 Kummer Lines

The previous work [19] had considered Kummer lines only at the 128-bit security level and not for higher security levels. Using the search strategy outlined in Section 3, we have implemented a Magma code to search for Kummer lines at the 224-bit and the 256-bit security levels. Also, at the 128-bit level, we have expanded the range of search as mentioned in Section 3. We use the notation $\mathcal{KL}_{p,a^2,b^2}$ to denote a Kummer line associated with a prime p having the parameters a^2 and b^2. The Kummer lines at the various security levels considered in this work are provided in Table 5.

The parameters of the Kummer lines and the corresponding Legendre form curves are shown in Table 6. From the discussion in Section 3, we note that optimal values of (h, h_T) are achieved for $\mathcal{KL}_{p251\text{-}9,81,20}$, $\mathcal{KL}_{p255\text{-}19,838,831}$, $\mathcal{KL}_{p266\text{-}3,683,18}$, $\mathcal{KL}_{p444\text{-}17,659,370}$, $\mathcal{KL}_{p448\text{-}224\text{-}1,410,103}$, $\mathcal{KL}_{p452\text{-}3,913,294}$, $\mathcal{KL}_{p510\text{-}75,1063,198}$, and $\mathcal{KL}_{p521\text{-}1,1559,796}$.

Due to the smaller co-factors, the Kummer lines $\mathcal{KL}_{p255\text{-}19,838,831}$ and $\mathcal{KL}_{p266\text{-}3,683,18}$ are improvements over the Kummer lines $\mathcal{KL}_{p255\text{-}19,82,77}$ and $\mathcal{KL}_{p266\text{-}3,260,139}$ proposed in [19]. The value of (h, h_T) corresponding to $\mathcal{KL}_{p506\text{-}45,856,181}$ is $(8, 16)$ which is sub-optimal (since $p506\text{-}45 \equiv 3 \bmod 4$, the minimum possible value is $(8, 8)$ and $(8, 16)$ is the next best.) For $p506\text{-}45$, in the search range that we considered, no curve has (h, h_T) to be equal to $(8, 8)$.

Remark. For the primes $p255\text{-}19$, $p266\text{-}3$, $p506\text{-}45$ and $p510\text{-}75$ we have also found alternative Kummer lines having values of (a^2, b^2) as (911,784), (762,295), (795,461) and (1609,1496) respectively. For these

Security level	Kummer line	Prime	Reference
128 bit	$\mathcal{KL}_{p251\text{-}9,81,20}$	$p251$-9	[19]
	$\mathcal{KL}_{p255\text{-}19,82,77}$	$p255$-19	[19]
	$\mathcal{KL}_{p255\text{-}19,838,831}$	$p255$-19	this work
	$\mathcal{KL}_{p266\text{-}3,260,139}$	$p266$-3	[19]
	$\mathcal{KL}_{p266\text{-}3,683,18}$	$p266$-3	this work
224 bit	$\mathcal{KL}_{p444\text{-}17,659,370}$	$p444$-17	this work
	$\mathcal{KL}_{p448\text{-}224\text{-}1,410,103}$	$p448$-224-1	this work
	$\mathcal{KL}_{p452\text{-}3,913,294}$	$p452$-3	this work
256 bit	$\mathcal{KL}_{p506\text{-}45,856,181}$	$p506$-45	this work
	$\mathcal{KL}_{p510\text{-}75,1063,198}$	$p510$-75	this work
	$\mathcal{KL}_{p521\text{-}1,1559,796}$	$p521$-1	this work

Table 5: Kummer lines considered in this work at various security levels.

alternative Kummer lines, the values of (h, h_T) are equal to the values of (h, h_T) for the corresponding Kummer lines reported in Table 6. As a criteria we have chosen the Kummer line corresponding to the pair (a^2, b^2) for which the value of the constant $(a^2 + b^2)$ is minimum. This has been done because in the Kummer line scalar multiplication algorithm the constants $a^2, b^2, (a^2 - b^2)$ and $(a^2 + b^2)$ are involved out of which $(a^2 + b^2)$ is the largest.

Kummer line	$(\lg \ell, \lg \ell_T)$	(h, h_T)	(κ, κ_T)	$\lg(-D)$	Base point
$\mathcal{KL}_{p251\text{-}9,81,20}$	(248,248)	(8,8)	$\left(\ell - 1, \frac{\ell_T - 1}{7}\right)$	248.3	[64:1]
$\mathcal{KL}_{p255\text{-}19,82,77}$	(251.4,252)	(12,8)	$(\ell - 1, \ell_T - 1)$	252.9	[31:1]
$\mathcal{KL}_{p255\text{-}19,838,831}$	(253,252)	(4,8)	$(\ell - 1, \ell_T - 1)$	252.9	[10:1]
$\mathcal{KL}_{p266\text{-}3,260,139}$	(262.4,263)	(12,8)	$\left(\frac{\ell - 1}{2}, \ell_T - 1\right)$	263.9	[2:1]
$\mathcal{KL}_{p266\text{-}3,683,18}$	(264,263)	(4,8)	$\left(\frac{\ell - 1}{2}, \ell_T - 1\right)$	263.9	[2:1]
$\mathcal{KL}_{p444\text{-}17,659,370}$	(441,441)	(8,8)	$\left(\ell - 1, \frac{\ell_T - 1}{6}\right)$	445.8	[47:1]
$\mathcal{KL}_{p448\text{-}224\text{-}1,410,103}$	(445,445)	(8,8)	$\left(\frac{\ell - 1}{5}, \ell_T - 1\right)$	449.9	[10:1]
$\mathcal{KL}_{p452\text{-}3,913,294}$	(450,449)	(4,8)	$\left(\frac{\ell - 1}{2}, \frac{\ell_T - 1}{9}\right)$	449.8	[2:1]
$\mathcal{KL}_{p506\text{-}45,856,181}$	(503,502)	(8,16)	$\left(\frac{\ell - 1}{2}, \frac{\ell_T - 1}{2}\right)$	507.9	[17:1]
$\mathcal{KL}_{p510\text{-}75,1063,198}$	(508,507)	(4,8)	$\left(\frac{\ell - 1}{37}, \frac{\ell_T - 1}{24}\right)$	507.3	[18:1]
$\mathcal{KL}_{p521\text{-}1,1559,796}$	(518,518)	(8,8)	$(\frac{\ell - 1}{3}, \ell_T - 1)$	522.5	[29:1]

Table 6: Parameters of the corresponding Legendre form curves of the Kummer lines. The values of κ_T for $\mathcal{KL}_{p510\text{-}75,1063,198}$ and κ, κ_T for $\mathcal{KL}_{p521\text{-}1,1559,796}$ were computed using CADO-NFS [31]. Details of the factorizations involved in these three computations are given in Appendix B.

Set of scalars. For Curve25519, the permitted set of scalars has been defined to be $8(2^{251} + \{0, 1, \ldots, 2^{251} - 1\})$. Defining scalars in this manner precludes small subgroup attacks and also ensures that the loop in Algorithm 3 requires the same number of iterations for all permitted scalars. The procedure of formatting a 256-bit random string into a scalar of the allowed form has been called clamping[1]. Considering the 256-bit string to be a 32-byte quantity, clamping for Curve25519 consists of clearing bits 0, 1 and 2 of the first byte, clearing bit 7 of the last byte, and setting bit 6 of the last byte (see Section 3 of [4]). Clamping ensures that the resulting scalar is in the permitted set of scalars. Following the example of Curve25519, the allowed sets of scalars and clamping strategies have been defined for the Kummer lines and the Montgomery curves proposed in [19, 27, 29].

The permitted sets of scalars for the new Kummer lines proposed in Table 5 are defined in Table 7. In the table, $\mathfrak{s}$ is the number of available bits in the corresponding scalar. Based on the scalars in Table 7,

[1] https://cr.yp.to/ecdh.html, accessed on October 10, 2020

an appropriate clamping strategy has been used. Since the details are quite routine and available from the accompanying code, we do not mention them here.

Kummer line	Set of scalars	$\mathfrak{s}$
$\mathcal{KL}_{p255\text{-}19,838,831}$	$4(2^{252} + \{0, 1, \ldots, 2^{252} - 1\})$	252
$\mathcal{KL}_{p266\text{-}3,683,18}$	$4(2^{263} + \{0, 1, \ldots, 2^{263} - 1\})$	263
$\mathcal{KL}_{p444\text{-}17,659,370}$	$8(2^{440} + \{0, 1, \ldots, 2^{440} - 1\})$	440
$\mathcal{KL}_{p448\text{-}224\text{-}1,410,103}$	$8(2^{444} + \{0, 1, \ldots, 2^{444} - 1\})$	444
$\mathcal{KL}_{p452\text{-}3,913,294}$	$4(2^{449} + \{0, 1, \ldots, 2^{449} - 1\})$	449
$\mathcal{KL}_{p506\text{-}45,856,181}$	$8(2^{502} + \{0, 1, \ldots, 2^{502} - 1\})$	502
$\mathcal{KL}_{p510\text{-}75,1063,198}$	$4(2^{507} + \{0, 1, \ldots, 2^{507} - 1\})$	507
$\mathcal{KL}_{p521\text{-}1,1559,796}$	$8(2^{517} + \{0, 1, \ldots, 2^{517} - 1\})$	517

Table 7: Allowed set of scalars for the new Kummer lines proposed in this paper.

5 Field Arithmetic

Differential addition and doubling over both Kummer line and Montgomery curves require arithmetic over the underlying field $\mathbb{F}_p$. Let $m = \lceil \log_2 p \rceil$. Elements of $\mathbb{F}_p$ can be represented as m-bit strings formatted into κ η-bit words called limbs, where $m = \eta(\kappa - 1) + \nu$ such that $1 \leq \nu \leq \eta < 32$. So, the first $(\kappa - 1)$ limbs are η bits long, while the size of the last limb is ν which lies between 1 and η.

Note that there are 2^m m-bit strings whereas the number of elements in $\mathbb{F}_p$ is $p < 2^m$. So, working with m-bit strings as if they are field elements has the consequence that a few field elements have two representations. This does not affect the correctness of the computation. At the very end of the computation, the unique representation of the result is obtained using a constant time algorithm. For the details of the constant time algorithm to obtain unique representation, we refer to [26].

Representations of elements in the various fields are summarized in Table 8. For the elements of $\mathbb{F}_{2^{255}-19}$ we also consider the 10-limb representation introduced by Bernstein [4], which is not given in Table 8. In this representation, a field element $A \in \mathbb{F}_{2^{255}-19}$ is written as $A = \sum_{i=0}^{9} a_i 2^{\lceil 25.5i \rceil}$ where $0 \leq a_0 \leq 2^{26} - 19$, $0 \leq a_2, a_4, a_6, a_8 < 2^{26}$ and $0 \leq a_1, a_3, a_5, a_7, a_9 < 2^{25}$.

Field	m	κ	η	ν	$\eta - \nu$
$\mathbb{F}_{2^{251}-9}$	251	9	28	27	1
$\mathbb{F}_{2^{255}-19}$	255	9	29	23	6
$\mathbb{F}_{2^{266}-3}$	266	10	27	23	4
$\mathbb{F}_{2^{444}-17}$	444	16	28	24	4
$\mathbb{F}_{2^{448}-2^{224}-1}$	448	16	28	28	0
$\mathbb{F}_{2^{452}-3}$	452	16	29	17	12
$\mathbb{F}_{2^{506}-45}$	506	18	29	13	16
$\mathbb{F}_{2^{510}-75}$	510	18	29	17	12
$\mathbb{F}_{2^{521}-1}$	521	18	29	28	1

Table 8: Representations of field elements.

The goal of determining the representations is to be able to use 4-way SIMD instructions. Each limb will be packed into the lower 32 bits of a 64-bit portion of a 256-bit word. The entire 256-bit word then contains four limbs of four different field elements. Two such 256-bit words are multiplied using a single SIMD instruction and the result is also a 256-bit word, where each 64-bit portion of the output 256-bit word contains the result of the multiplication of the two corresponding limbs of the input. The requirement of fitting each limb into a 32-bit word dictates the choice of η to be less than 32.

Field arithmetic for p251-9, p255-19, p448-224-1, and p444-17 using the representations in Table 8 have been reported in [24]. Field arithmetic for the other primes have been freshly implemented as part of this work. In Appendix A, we provide the details of the multiplication and squaring algorithms for

the prime $p521$-1. The details for the other primes for which the implementations have been freshly done are similar.

Multiplication and squaring in $\mathbb{F}_p$. The major operations required in differential addition and doubling are field multiplications and squarings. At a conceptual level, a field multiplication consists of an integer multiplication followed by a reduction. The integer multiplication takes as input two κ-limb quantities and produce as output a $(2\kappa - 1)$-limb quantity, to which the reduction procedure is applied to obtain a κ-limb quantity. For the purpose of implementation, however, the separation of integer multiplication and reduction is not always explicit. We identify two strategies for performing the multiplication and squaring operations.

Strategy 1: Two κ-limb quantities are multiplied using schoolbook multiplication and simultaneously the results are partially reduced to obtain a κ-limb quantity where the size of all the limbs are at most 64 bits.

Strategy 2: Two κ-limb quantities are multiplied using a variant of the Karatsuba algorithm to obtain a $(2\kappa - 1)$-limb result. The result is expanded to a 2κ-limb quantity which is then folded into a κ-limb quantity where the size of all the limbs are at most 64 bits. The reason for the expansion before folding is that if the $(2\kappa - 1)$-limb result is directly attempted to be folded into a κ-limb quantity, then some of the limbs require more than 64 bits to be represented.

The second strategy is essentially the mulle algorithm in [19]. Strategy 1 has been used for $p251$-9, $p255$-19 with 10-limb representation, and for $p266$-3. Strategy 2 has been used for the 9-limb representation of $p255$-19 using (5+4)-Karatsuba; $p448$-224-1, $p444$-17 and $p452$-3 using (8+8)-Karatsuba; and for $p506$-45, $p510$-75 and $p521$-1 using (9+9)-Karatsuba. Here, by $(k_1 + k_2)$-Karatsuba, we mean that at the top level the κ-limb quantity is divided into k_1-limb and k_2-limb quantities to which the Karatsuba algorithm is applied. At the lower levels, the schoolbook algorithm is applied to complete the entire integer multiplication algorithm.

Both Strategy 1 and Strategy 2 provide a κ-limb result where each limb has at most 64 bits. A reduction algorithm is applied to this result to ensure that the sizes of the limbs reduce to the appropriate representations. This reduction algorithm essentially consists of computing a carry and forwarding it to the next limb and after the last limb, the produced carry is reduced and added back to the first limb. For the primes $p251$-9, $p255$-19, $p266$-3, $p444$-17, $p448$-224-1 and $p452$-3 an interleaved carry chain suggested by [9] in the context of $p255$-19 has been used. On the other hand, for the primes $p506$-45, $p510$-75 and $p521$-1 we use the simple carry chain instead of the interleaved carry chain. The details of the various carry chains are as follows.

- For $p251$-9 and $p255$-19 with $\kappa = 9$, we interleave the chains $c_0 \to c_1 \to \cdots \to c_4 \to c_5$ and $c_4 \to c_5 \to \cdots \to c_8 \to c_0 \to c_1$.
- For $p255$-19 and $p266$-3 with $\kappa = 10$, we interleave the chains $c_0 \to c_1 \to \cdots \to c_4 \to c_5 \to c_6$ and $c_5 \to c_6 \to \cdots \to c_9 \to c_0 \to c_1$.
- For $p444$-17, $p448$-224-1 and $p452$-3 with $\kappa = 16$, we interleave the chains $c_0 \to c_1 \to \cdots \to c_7 \to c_8 \to c_9$ and $c_8 \to c_9 \to \cdots \to c_{15} \to (c_0, c_8) \to (c_1, c_9)$.
- For $p506$-45, $p510$-75 and $p521$-1 with $\kappa = 18$, we use the simple carry chain $c_0 \to c_1 \to \cdots \to c_{17} \to c_0 \to c_1$.

We term this reduction algorithm as reduce_1. The multiplication/squaring algorithms without applying reduction will be termed as mul/sqr.

It should be noted that the number of steps in an interleaved carry chain is actually more than that in the simple carry chain. The reason that interleaved carry chains are faster for the smaller primes is due to the fact that internal processor pipelining can take advantage of the independence of the chains. For the larger primes, however, the number of limbs is 18 while the number of ymm registers (i.e., the registers where the quantities are to be stored) is 16. So, using the interleaved carry chain for these primes increases the number of load/stores making its performance inferior to that of the simple carry chain.

Multiplication by a small constant in $\mathbb{F}_p$. Let $A \in \mathbb{F}_p$ have a κ-limb representation $(a_0, a_1, \ldots, a_{\kappa-1})$. Let $\mathfrak{c}$ be an element in $\mathbb{F}_p$, which can be represented using a single limb. Then multiplication of A by $\mathfrak{c}$ provides κ limbs of the form $(c_0, c_1, \ldots, c_{\kappa-1}) = (a_0 \cdot \mathfrak{c}, a_1 \cdot \mathfrak{c}, \ldots, a_{\kappa-1} \cdot \mathfrak{c})$ where each limb is at most 64 bits. This needs to be reduced so that the limb values respect the appropriate representations.

- For $p251$-9 and $p255$-19 with $\kappa = 9$, we interleave the chains $c_0 \to c_1 \to \cdots \to c_3 \to c_4$ and $c_4 \to c_5 \to \cdots \to c_8 \to c_0$.
- For $p255$-19 and $p266$-3 with $\kappa = 10$, we interleave the chains $c_0 \to c_1 \to \cdots \to c_4 \to c_5$ and $c_5 \to c_6 \to \cdots \to c_9 \to c_0$.
- For $p444$-17, $p448$-224-1 and $p452$-3 with $\kappa = 16$, we interleave the chains $c_0 \to c_1 \to \cdots \to c_7 \to c_8$ and $c_8 \to c_9 \to \cdots \to c_{15} \to (c_0, c_8)$.
- For $p506$-45, $p510$-75 and $p521$-1 with $\kappa = 18$, we use the simple carry chain $c_0 \to c_1 \to \cdots \to c_{17} \to c_0$.

We term this reduction as reduce_2. Note that reduce_2 is slightly more efficient that reduce_1 because the lengths of the chains are one less. The algorithm to multiply with a small constant without applying reduction will be termed as mulc.

Negation. Given a field element A, the goal is to compute $-A \bmod p$. To avoid handling negative numbers, the negation of A is computed by subtracting A from an appropriate multiple of p to ensure that all limbs of the result are non-negative. This requires the following representation of p as a κ-limb quantity $\mathfrak{P} = \sum_{i=0}^{\kappa-1} \mathfrak{p}_i \theta^i$, where the values of $\mathfrak{p}_i$ are obtained from the appropriate representation as shown in Table 8.

Let n be the least integer such that all the limbs of $(2^n\mathfrak{P} - A)$ are non-negative. The negation of the element A is then defined by $\mathsf{negate}(A) = 2^n\mathfrak{P} - A = C$ in unreduced form, while reducing C modulo p gives us the desired value in $\mathbb{F}_p$. Let $C = \sum_{i=0}^{\kappa-1} c_i \theta^i$ so that $c_i = 2^n\mathfrak{p}_\mathfrak{i} - a_i \geq 0 \; \forall i$. Considering all values to be α-bit quantities, the computation of c_i is done as

$$c_i = ((2^\alpha - 1) - a_i) + (1 + 2^n\mathfrak{p}_\mathfrak{i}) \bmod 2^\alpha. \tag{2}$$

The operation $(2^\alpha - 1) - a_i$ is equivalent to taking the bitwise complement of a_i, which is equivalent to $1^\alpha \oplus a_i$. The values of α will be considered as 32 and 64 in our implementations.

Subtraction. Subtraction is done by first negating the subtrahend B and then adding the obtained result to the minuend A.

Reduction. The bit-sizes of the limbs of the output of the subtraction procedure are at most two more than the bit-sizes of the input limbs. This can be reduced using the following parallel carry chain.

$$(c_0, c_{\lceil(\kappa-1)/2\rceil}) \to (c_1, c_{\lceil(\kappa-1)/2\rceil+1}) \to \cdots \to (c_{\lceil(\kappa-1)/2\rceil-1}, c_{2\lceil(\kappa-1)/2\rceil-1})$$

Here, the notation $(c_i, c_j) \to (c_k, c_\ell)$ means performing the reductions $c_i \to c_k$ and $c_j \to c_\ell$ simultaneously. We denote this reduction operation as reduce_3.

Inversion in $\mathbb{F}_p$. The output of the ladder algorithm is $x^2(z^2)^{p-2}$. The computation of $(z^2)^{p-2}$ requires squaring and multiplication in $\mathbb{F}_p$. There is no scope to utilize SIMD instructions for computing $(z^2)^{p-2}$. So, for the computation of $x^2(z^2)^{p-2}$, the quantities x^2 and z^2 are re-packed into multi-limb quantities, where each limb is at most 64 bits. This reduces the number of limbs and hence makes the individual field multiplication and squaring faster. Extensive details along with explicit algorithms for field multiplication, squaring and inversion have been provided in [26]. We have used these to implement the inversion algorithms required in the present context.

6 Vector Operations

Our goal is to obtain 4-way vector implementations of the ladder for Kummer arithmetic. To this end, we introduce the required vector operations. This requires packing four field elements into κ 256-bit words. Each of the field elements has κ limbs, where each limb is at most 32 bits. We consider two kinds of packings. These packings have been considered in [24] and in [8, 17] a similar packing strategy has been called squeeze/unsqueeze.

Notation. In the following sections, for uniformity of description, we use expressions of the form $\sum_{i=\ell}^{h} f_i \theta^i$. For $p255$-19, while dealing with 10-limb representations θ^i should be considered as $2^{\lceil 25.5i \rceil}$.

Dense packing of field elements. Let $A = \sum_{i=0}^{\kappa-1} a_i\theta^i$. Consider that every limb a_i is less than 2^{32} and is stored in a 64-bit word. Then it is possible to pack $a_{\lfloor\kappa/2\rfloor}$with a_0, $a_{\lfloor\kappa/2\rfloor+1}$with $a_1, \ldots, a_{2\lfloor\kappa/2\rfloor-1}$with $a_{\lfloor\kappa/2\rfloor-1}$, so that every pair can be represented using a 64-bit word without losing any information. If κ is odd then $a_{\kappa-1}$ can be left alone. We denote this operation as dense packing of limbs and is denoted by $\underline{A}$.

Vector representation of field elements. Define $\mathbf{A} = \langle A_0, A_1, A_2, A_3\rangle$ where $A_k = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i \in \mathbb{F}_p$. Hence, $\mathbf{A}$ is a 4-element vector. Each $a_{k,i}$ is stored in a 64-bit word, and conceptually one may think of $\mathbf{A}$ to be given by a $\kappa \times 4$ matrix of 64-bit words. If we consider $\underline{A_k}$, i.e., densely packed form of A_k, then we have $\underline{\mathbf{A}} = \langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle$ where $\underline{A_k} = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{a_{k,i}}\theta^i$. Then we can conceptually think of $\underline{\mathbf{A}}$ as a $\lceil\kappa/2\rceil \times 8$ matrix of 32-bit words.

We can also visualize $\mathbf{A}$ and $\underline{\mathbf{A}}$ by the following alternative representation. Let $\mathbf{a}_i = \langle a_{0,i}, a_{1,i}, a_{2,i}, a_{3,i}\rangle$. Define $\mathbf{a}_i\theta^i = \langle a_{0,i}\theta^i, a_{1,i}\theta^i, a_{2,i}\theta^i, a_{3,i}\theta^i\rangle$. Then, we can write $\mathbf{A} = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$. Each $\mathbf{a}_i$ is stored as a 256-bit value. Similarly, let $\underline{\mathbf{a}_i} = \langle \underline{a_{0,i}}, \underline{a_{1,i}}, \underline{a_{2,i}}, \underline{a_{3,i}}\rangle$. Define $\underline{\mathbf{a}}\theta^i = \langle \underline{a_{0,i}}\theta^i, \underline{a_{1,i}}\theta^i, \underline{a_{2,i}}\theta^i, \underline{a_{3,i}}\theta^i\rangle$. Then, we can write $\underline{\mathbf{A}} = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{\mathbf{a}_i}\theta^i$. Like $\mathbf{a}_i$, each $\underline{\mathbf{a}_i}$ is stored as a 256-bit value.

Dense packing of vector elements. Let $\langle A_0, A_1, A_2, A_3\rangle = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$, where $A_k = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i$. The vectorized normal to dense packing operation Pack-N2D($\langle A_0, A_1, A_2, A_3\rangle$) returns the 4-tuple $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{\mathbf{a}_i}\theta^i$, where $\underline{A_k} = \text{N2D}(A_k)$, such that $\underline{A_k} = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{a_{k,i}}\theta^i$. Let $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{\mathbf{a}}\theta^i$, where $\underline{A_k} = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{a_{k,i}}\theta^i$. The vectorized dense to normal operation Pack-D2N($\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle$) returns the 4-tuple $\langle A_0, A_1, A_2, A_3\rangle = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$, where $A_k = \text{D2N}(\underline{A_k})$, such that $A_k = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i$. For details of implementations of D2N and N2D we refer to [24].

Vector reduction. Three types of vector reduction operations will be used, namely Reduce$_1$, Reduce$_2$ and Reduce$_3$ out of which Reduce$_3$ will be used on densely packed limbs after the Hadamard transformations.

- Reduce$_1$($\langle A_0, A_1, A_2, A_3\rangle$): This is used in the vectorized field multiplication and squaring algorithms which returns $\langle \mathsf{reduce}_1(A_0), \mathsf{reduce}_1(A_1), \mathsf{reduce}_1(A_2), \mathsf{reduce}_1(A_3)\rangle$.
- Reduce$_2$($\langle A_0, A_1, A_2, A_3\rangle$): This is used in the vectorized algorithm for multiplication by a field constant which returns $\langle \mathsf{reduce}_2(A_0), \mathsf{reduce}_2(A_1), \mathsf{reduce}_2(A_2), \mathsf{reduce}_2(A_3)\rangle$. The same reduction is also used after addition and subtraction of two normally packed vector elements.
- Reduce$_3$($\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle$): This is used in the vectorized algorithms for Hadamard transformations which returns $\langle \mathsf{reduce}_3(\underline{A_0}), \mathsf{reduce}_3(\underline{A_1}), \mathsf{reduce}_3(\underline{A_2}), \mathsf{reduce}_3(\underline{A_3})\rangle$.

Vector multiplication and squaring. Vector multiplication and squaring are done over normally packed field elements which are defined as below.

- Mul($\langle A_0, A_1, A_2, A_3\rangle, \langle B_0, B_1, B_2, B_3\rangle$): returns $\mathbf{C} = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ such that $\mathbf{C}$ = Reduce$_1$($\langle C_0, C_1, C_2, C_3\rangle$), where $C_k = \mathsf{mul}(A_k, B_k)$.
- Sqr($\langle A_0, A_1, A_2, A_3\rangle$): returns C $= \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$, such that $\mathbf{C}$ = Reduce$_1$($\langle C_0, C_1, C_2, C_3\rangle$), where $C_k = \mathsf{sqr}(A_k)$.

Vector multiplication by a field constant. Vector multiplication by a field constant is done with a normally packed field element. The function is defined as Mulc($\langle A_0, A_1, A_2, A_3\rangle, \langle d_0, d_1, d_2, d_3\rangle$), which returns $\mathbf{C} = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$, such that $\mathbf{C}$ = Reduce$_2$($\langle C_0, C_1, C_2, C_3\rangle$). Here $d_0, d_1, d_2, d_3 \in \mathbb{F}_p$ and $C_k = \mathsf{mulc}(A_k, d_k)$. The Mulc operation without reduction will be termed as Unreduced-Mulc.

Addition, Subtraction and Hadamard transforms. We will require vector versions of addition, subtraction and Hadamard transforms. Let $A = \sum_{i=0}^{\kappa-1} a_i\theta^i$, $B = \sum_{i=0}^{\kappa-1} b_i\theta^i$ be two elements in $\mathbb{F}_p$. Using the operation N2D on A and B we obtain the densely packed elements $\underline{A} \leftarrow \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{a_i}\theta^i$ and $\underline{B} \leftarrow \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{b_i}\theta^i$ respectively.

Addition. Two normally packed vector elements $\langle A_0, A_1, A_2, A_3\rangle$ and $\langle B_0, B_1, B_2, B_3\rangle$ when added returns $\textsc{Reduce}_2(\langle C_0, C_1, C_2, C_3\rangle)$ where

$$C_k = A_k + B_k = \sum_{i=0}^{\kappa-1}(a_i + b_i)\theta^i = \sum_{i=0}^{\kappa-1} c_i\theta^i.$$

When we apply a similar addition operation over densely packed vector elements we can exploit 2-way parallelism to compute a field addition. The addition $\underline{c_i} \leftarrow \underline{a_i}+\underline{b_i}$ computes the additions $c_i \leftarrow a_i+b_i$ and $c_{\lceil\kappa/2\rceil+i} \leftarrow a_{\lceil\kappa/2\rceil+i}+b_{\lceil\kappa/2\rceil+i}$ simultaneously for $i = 0, 1, \ldots, \lceil\kappa/2\rceil-1$. The quantity $c_{\kappa-1} \leftarrow a_{\kappa-1}+b_{\kappa-1}$ can be computed as a single addition if κ is odd.

Negation. The negation operation for a single field element has already been defined. It can be applied to $\underline{A} = \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{a_i}\theta^i$ in parallel similar to addition.

Subtraction. This operation can be done over $\underline{A}$ and $\underline{B}$ simultaneously similar to addition.

Hadamard transformations. Let $A,\ B$ be two elements in $\mathbb{F}_p$ and $\underline{A},\ \underline{B}$ be their dense representations. The Hadamard transform $\mathfrak{H}_1(A, B)$ outputs the pair $\langle C, D\rangle$ where

$$C = \mathsf{reduce}_2(A + B) \text{ and } D = \mathsf{reduce}_2(A + \mathsf{negate}(B)).$$

The Hadamard transform $\mathfrak{H}_2(\underline{A}, \underline{B})$ outputs the pair $\langle \underline{C}, \underline{D}\rangle$ where

$$\underline{C} = \mathsf{reduce}_3(\underline{A} + \underline{B}) \text{ and } \underline{D} = \mathsf{reduce}_3(\underline{A} + \mathsf{negate}(\underline{B})).$$

We define the operation $\mathsf{unreduced}\text{-}\mathfrak{H}_1(A, B)$ which is the same as $\mathfrak{H}_1(A, B)$ except that the reduce_2 operation is dropped. Similarly, the operation $\mathsf{unreduced}\text{-}\mathfrak{H}_2(\underline{A}, \underline{B})$ is same as $\mathfrak{H}_2(\underline{A}, \underline{B})$ except that the reduce_3 operation is dropped.

Remark. When the linear operations are applied in parallel over densely packed elements α is considered as 32, else α is taken as 64.

Algorithms for vectorized Hadamard operations. For a normally packed 256-bit vector quantity $\mathbf{a} = \langle a_0, a_1, a_2, a_3\rangle$ we define $\mathsf{copy}_1(\mathbf{a}) = \langle a_0, a_0, a_2, a_2\rangle$ and $\mathsf{copy}_2(\mathbf{a}) = \langle a_1, a_1, a_3, a_3\rangle$. Similarly, for a densely packed 256-bit quantity $\underline{\mathbf{a}} = \langle \underline{a_0}, \underline{a_1}, \underline{a_2}, \underline{a_3}\rangle$ we define $\mathsf{copy}_3(\underline{\mathbf{a}}) = \langle \underline{a_0}, \underline{a_0}, \underline{a_2}, \underline{a_2}\rangle$ and $\mathsf{copy}_4(\underline{\mathbf{a}}) = \langle \underline{a_1}, \underline{a_1}, \underline{a_3}, \underline{a_3}\rangle$.

The operations copy_1, copy_2, copy_3 and copy_4 can be implemented using the assembly instruction `vpshufd`. The instruction `vpshufd` uses an additional parameter known as the shuffle mask, whose values for $\mathsf{copy}_1(\cdot)$ and $\mathsf{copy}_3(\cdot)$ is 68 and for $\mathsf{copy}_2(\cdot)$ and $\mathsf{copy}_4(\cdot)$ is 238. The vector Hadamard operation HH and Dense-HH are described in Algorithm 7 and Algorithm 8 respectively. HH implements the transformation $\mathfrak{H}_1\mathfrak{H}_1$ over normally packed vector elements and Dense-HH implements $\mathfrak{H}_2\mathfrak{H}_2$ over densely packed vector elements.

Algorithm 7 Vectorized Hadamard transformation over normally packed elements.

function HH($\langle A_0, A_1, A_2, A_3\rangle$)
input: $\langle A_0, A_1, A_2, A_3\rangle = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$.
output: $\mathbf{C} = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $\langle A_0 + A_1, A_0 - A_1, A_2 + A_3, A_2 - A_3\rangle$ where each component is reduced modulo p.
    for $i \leftarrow 0$ to $\kappa - 1$ **do**
        $\mathbf{s} \leftarrow \mathsf{copy}_1(\mathbf{a}_i)$
        $\mathbf{t} \leftarrow \mathsf{copy}_2(\mathbf{a}_i)$
        $\mathbf{t} \leftarrow \mathbf{t} \oplus \langle 0^{64}, 1^{64}, 0^{64}, 1^{64}\rangle$
        $\mathbf{t} \leftarrow \mathbf{t} + \langle 0^{64}, 2\mathfrak{p}_i + 1, 0^{64}, 2\mathfrak{p}_i + 1\rangle$
        $\mathbf{c}_i \leftarrow \mathbf{s} + \mathbf{t}$
    end for
    return $\textsc{Reduce}_2(\mathbf{C})$
end function.

Algorithm 8 Vectorized Hadamard transformation over densely packed elements.

function DENSE-HH($\langle \underline{A}_0, \underline{A}_1, \underline{A}_2, \underline{A}_3 \rangle$)
input: $\langle \underline{A}_0, \underline{A}_1, \underline{A}_2, \underline{A}_3 \rangle = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}}_i \theta^i$.
output: $\underline{\mathbf{C}} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{c}}_i \theta^i$ representing $\langle \underline{A}_0 + \underline{A}_1, \underline{A}_0 - \underline{A}_1, \underline{A}_2 + \underline{A}_3, \underline{A}_2 - \underline{A}_3 \rangle$, where each component is reduced modulo p.
    for $i \leftarrow 0$ to $\lceil \kappa/2 \rceil - 1$ **do**
        $\underline{\mathbf{s}} \leftarrow \mathsf{copy}_3(\underline{\mathbf{a}}_i)$
        $\underline{\mathbf{t}} \leftarrow \mathsf{copy}_4(\underline{\mathbf{a}}_i)$
        $\underline{\mathbf{t}} \leftarrow \underline{\mathbf{t}} \oplus \langle 0^{32}, 0^{32}, 1^{32}, 1^{32}, 0^{32}, 0^{32}, 1^{32}, 1^{32} \rangle$
        $\underline{\mathbf{t}} \leftarrow \underline{\mathbf{t}} + \langle 0^{32}, 0^{32}, 2\mathfrak{p}_i + 1, 2\mathfrak{p}_{i+\lceil \kappa/2 \rceil} + 1, 0^{32}, 0^{32}, 2\mathfrak{p}_i + 1, 2\mathfrak{p}_{i+\lceil \kappa/2 \rceil} + 1 \rangle$
        $\underline{\mathbf{c}}_i \leftarrow \underline{\mathbf{s}} + \underline{\mathbf{t}}$
    end for
    return REDUCE$_3$($\underline{\mathbf{C}}$)
end function.

Previous works [19, 17, 24] on Kummer ladder and vectorized Montgomery have used Hadamard transforms. We mention the similarities and the differences. Neither HH nor DENSE-HH were required in [17, 24]. The idea of HH was used in [19], but the implementation was done using costly permutation instructions instead of shuffle which has been used here; the DENSE-HH was not used in [19].

Vector duplication. For the 256-bit quantity $\underline{\mathbf{a}} = \langle \underline{a}_0, \underline{a}_1, \underline{a}_2, \underline{a}_3 \rangle$ let us define the operation $\mathsf{copy}_3(\underline{\mathbf{a}}) = \langle \underline{a}_0, \underline{a}_1, \underline{a}_0, \underline{a}_1 \rangle$, which can be implemented using the assembly instruction `vpermq`. The instruction `vpermq` uses an additional parameter known as the shuffle mask, whose value for $\mathsf{copy}_3(\cdot)$ is 68. Let $\underline{\mathbf{A}} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}}_i \theta^i$. Define the operation DENSE-CDUP($\underline{\mathbf{A}}$) to return $\sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \mathsf{copy}_3(\underline{\mathbf{a}}_i) \theta^i$. If $\underline{\mathbf{A}}$ represents $\langle \underline{A}_0, \underline{A}_1, \underline{A}_2, \underline{A}_3 \rangle$, then DENSE-CDUP($\underline{\mathbf{A}}$) $= \langle \underline{A}_0, \underline{A}_1, \underline{A}_0, \underline{A}_1 \rangle$.

The DENSE-CDUP operation was not used in the earlier works [19, 17, 24].

Vector swapping. Let $\underline{\mathbf{a}} = \langle \underline{a}_0, \underline{a}_1, \underline{a}_2, \underline{a}_3 \rangle$ and $\mathfrak{b}$ be a bit. We define an operation $\mathsf{swap}(\underline{\mathbf{a}}, \mathfrak{b})$ as

$$\mathsf{swap}(\underline{\mathbf{a}}, \mathfrak{b}) \leftarrow \begin{cases} \langle \underline{a}_0, \underline{a}_1, \underline{a}_2, \underline{a}_3 \rangle & \text{if } \mathfrak{b} = 0, \\ \langle \underline{a}_2, \underline{a}_3, \underline{a}_0, \underline{a}_1 \rangle & \text{if } \mathfrak{b} = 1. \end{cases}$$

The operation $\mathsf{swap}(\underline{\mathbf{a}}, \mathfrak{b})$ is implemented using the assembly instruction `vpermd`. Let $\underline{\mathbf{A}} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}}_i \theta^i$. We define the operation DENSE-SWAP($\underline{\mathbf{A}}, \mathfrak{b}$) to return $\sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \mathsf{swap}(\underline{\mathbf{a}}_i, \mathfrak{b}) \theta^i$. If $\underline{\mathbf{A}}$ represents the vector $\langle \underline{A}_0, \underline{A}_1, \underline{A}_2, \underline{A}_3 \rangle$, then

$$\text{DENSE-CSWAP}(\underline{\mathbf{A}}, \mathfrak{b}) \leftarrow \begin{cases} \langle \underline{A}_0, \underline{A}_1, \underline{A}_2, \underline{A}_3 \rangle & \text{if } \mathfrak{b} = 0, \\ \langle \underline{A}_2, \underline{A}_3, \underline{A}_0, \underline{A}_1 \rangle & \text{if } \mathfrak{b} = 1. \end{cases}$$

The DENSE-CSWAP operation (under a slightly different name) has been used in [24].

Remark. The vector operations MUL, SQR, MULC, PACK-N2D are applied to normally packed field elements. and DENSE-CSWAP, DENSE-HH, DENSE-CDUP, PACK-D2N are applied to densely packed field elements.

7 Vectorized Ladder

For the Montgomery curves over the primes $p266$-3, $p452$-3, $p506$-45, $p510$-75 and $p521$-1, we have implemented the vectorized algorithm of the ladder for Montgomery curves given in [24].

A vectorized version of the ladder for Kummer lines has been previously proposed in [19]. This did not incorporate the advantages of using dense packing of field elements. We provide a new vectorized algorithm of the ladder for Kummer lines in Algorithm 9. This algorithm computes variable base scalar multiplication.

The correctness of Algorithm 9 is easy to derive from Algorithms 3, 4, 5 and 6. At a top-level, the main loop of Algorithm 9 is the same as that of Algorithms 3 and 4 except that the conditional statement of Algorithm 4 has been implemented using a constant time procedure. The body of the main loop in Algorithm 3 is essentially the vectorized version of the combined field operations of Algorithm 5 and 6.

Since the first two columns of the output vector in the SQR operations of INITIALIZE-P are unutilized we can apply the optimization technique used in the SQR operation of Algorithm 8 of [24] for the Kummer lines at 224-bit and 256-bit security levels.

Algorithm 9 can be slightly modified to obtain a vectorized algorithm for fixed base scalar multiplication. In this case, the vector $\langle T_5, T_6, T_7, T_8\rangle$ in Step 16 is small and a single limb quantity. Consequently, the vector multiplication MUL of Step 18 is actually MULC which is a vector multiplication by a vector constant. The operation DENSE-CSWAP in Step 16 in this case is actually CSWAP because we have the single limb vector constant $\langle b^2, a^2, z^2, x^2\rangle$ involved in the operation instead of $\langle \underline{b^2}, \underline{a^2}, \underline{z^2}, \underline{x^2}\rangle$.

10-limb implementations of $\mathcal{KL}_{p255\text{-}19,838,831}$. For the 10-limb implementations of the Kummer line over the prime $p255$-19, the UNREDUCED-MULC and the HH operations in Steps 8, 9 of INITIALIZE-P and Steps 13, 14 of KUMMER-LINE-SCALARMULT are replaced by the MULC, PACK-N2D, UNREDUCED-DENSE-HH and PACK-D2N operations. This replacement makes the implementations for the Kummer line more efficient. Also, the other DENSE-HH operations can be kept unreduced in the ladder for these implementations.

Algorithm 9 Ladder to compute scalar multiplication over Kummer lines using 4-way vectorization.

1: **function** KUMMER-LINE-SCALARMULT(P, n)
2: **input:** An x-coordinate only projective point $P = [x^2 : z^2]$ on $\mathcal{KL}_{p,a^2,b^2}$ and an ℓ-bit clamped scalar n given as $n = (1, n_{\ell-2}, n_{\ell-3}, \ldots, n_0)$.
3: **output:** The x-coordinate of the scalar multiple nP.
4: $\langle \underline{b^2}, \underline{a^2}, \underline{z^2}, \underline{x^2}\rangle \leftarrow$ PACK-N2D($\langle b^2, a^2, z^2, x^2\rangle$)
5: $\langle T_1, T_2, T_3, T_4\rangle \leftarrow$ INITIALIZE(P)
6: **for** $i \leftarrow \ell - 2$ **down to** 0 **do**
7: $\langle \underline{T_1}, \underline{T_2}, \underline{T_3}, \underline{T_4}\rangle \leftarrow$ PACK-N2D($\langle T_1, T_2, T_3, T_4\rangle$)
8: $\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8}\rangle \leftarrow$ DENSE-HH($\langle \underline{T_1}, \underline{T_2}, \underline{T_3}, \underline{T_4}\rangle$)
9: $\langle T_5, T_6, T_7, T_8\rangle \leftarrow$ PACK-D2N($\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8}\rangle$)
10: $\langle \underline{T_1}, \underline{T_2}, \underline{T_3}, \underline{T_4}\rangle \leftarrow$ DENSE-CDUP($\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8}\rangle, n_i$)
11: $\langle T_1, T_2, T_3, T_4\rangle \leftarrow$ PACK-D2N($\langle \underline{T_1}, \underline{T_2}, \underline{T_3}, \underline{T_4}\rangle$)
12: $\langle T_1, T_2, T_3, T_4\rangle \leftarrow$ MUL($\langle T_1, T_2, T_3, T_4\rangle, \langle T_5, T_6, T_7, T_8\rangle$)
13: $\langle T_1, T_2, T_3, T_4\rangle \leftarrow$ UNREDUCED-MULC($\langle T_1, T_2, T_3, T_4\rangle, \langle a^2 - b^2, a^2 + b^2, a^2 - b^2, a^2 + b^2\rangle$)
14: $\langle T_5, T_6, T_7, T_8\rangle \leftarrow$ HH($\langle T_1, T_2, T_3, T_4\rangle$)
15: $\langle T_1, T_2, T_3, T_4\rangle \leftarrow$ SQR($\langle T_5, T_6, T_7, T_8\rangle$)
16: $\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8}\rangle \leftarrow$ DENSE-CSWAP($\langle \underline{b^2}, \underline{a^2}, \underline{z^2}, \underline{x^2}\rangle, n_i$)
17: $\langle T_5, T_6, T_7, T_8\rangle \leftarrow$ PACK-D2N($\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8}\rangle$)
18: $\langle T_1, T_2, T_3, T_4\rangle \leftarrow$ MUL($\langle T_1, T_2, T_3, T_4\rangle, \langle T_5, T_6, T_7, T_8\rangle$)
19: **end for**
20: $\langle x^2, z^2, *, *\rangle \leftarrow$ REDUCE$_2$($\langle T_1, T_2, T_3, T_4\rangle$)
21: **return** $x^2(z^2)^{p-2}$
22: **end function**.

8 Implementations and Timings

We have developed constant-time assembly implementations targeting modern Intel architectures. This was done for nine Kummer lines and three Montgomery curves. Timing results were obtained for the following two platforms.

Haswell: Intel®Core™ i7-4790 4-core CPU 3.60 Ghz. The OS was 64-bit Ubuntu 14.04 LTS and the source code was compiled using GCC version 7.3.0.

Skylake: Intel®Core™ i7-6500U 2-core CPU @ 2.50GHz. The OS was 64-bit Ubuntu 14.04 LTS and the source code was compiled using GCC version 7.3.0.

The timing experiments were carried out on a single core of Haswell and Skylake processors. During measurement of the CPU-cycles, TurboBoost© and Hyper-Threading© features were turned off.

Along with the timings of our implementations, we also report timings of previous implementations. For a fair comparison, we have downloaded the relevant codes and have measured the timings of these

Algorithm 10 Initializes the variables for ladder computation.

function INITIALIZE(P)
input: An x-coordinate only projective point $P = [x^2 : z^2]$ on $\mathcal{KL}_{p,a^2,b^2}$.
output: $\langle P, 2P \rangle$.
  $\langle \underline{\mathbf{0}}, \underline{\mathbf{0}}, \underline{x}^2, \underline{z}^2 \rangle \leftarrow$ PACK-N2D($\langle \mathbf{0}, \mathbf{0}, x^2, z^2 \rangle$)
  $\langle *, *, \underline{T_1}, \underline{T_2} \rangle \leftarrow$ DENSE-HH($\langle \underline{\mathbf{0}}, \underline{\mathbf{0}}, \underline{x}^2, \underline{z}^2 \rangle$)
  $\langle *, *, T_1, T_2 \rangle \leftarrow$ PACK-D2N($\langle *, *, \underline{T_1}, \underline{T_2} \rangle$)
  $\langle *, *, T_3, T_4 \rangle \leftarrow$ SQR($\langle *, *, T_1, T_2 \rangle$)
  $\langle *, *, T_1, T_2 \rangle \leftarrow$ UNREDUCED-MULC($\langle *, *, T_3, T_4 \rangle, \langle *, *, a^2 - b^2, a^2 + b^2 \rangle$)
  $\langle *, *, T_3, T_4 \rangle \leftarrow$ HH($\langle *, *, T_1, T_2 \rangle$)
  $\langle *, *, T_1, T_2 \rangle \leftarrow$ SQR($\langle *, *, T_3, T_4 \rangle$)
  $\langle *, *, T_3, T_4 \rangle \leftarrow$ MULC($\langle *, *, T_1, T_2 \rangle, \langle *, *, b^2, a^2 \rangle$)
  $\langle T_1, T_2, *, * \rangle \leftarrow \langle x^2, z^2, *, * \rangle$
  return $\langle T_1, T_2, T_3, T_4 \rangle$
end function.

Curve	Haswell	Skylake	κ	Strategy	Implementation	Implementation type
M⟦4698⟧	-	87807	4	64-bit seq	[28]	`maax`, assembly
	114937	91203	9	4-way SIMD [24]	[28]	`AVX2`, assembly
Curve25519	-	98694	4	64-bit seq	[28]	`maax`, assembly
	120108	99194	9	4-way SIMD [24]	[24]	`AVX2`, assembly
	123899	95437	10	4-way SIMD [24]	[24]	`AVX2`, assembly
$\mathcal{KL}_{p251\text{-}9,81,20}$	128322	112275	9	4-way SIMD [19]	[19]	`AVX2`, intrinsics
$\mathcal{KL}_{p255\text{-}19,82,77}$	169696	140908	10	4-way SIMD [19]	[19]	`AVX2`, intrinsics
$\mathcal{KL}_{p266\text{-}3,260,139}$	164078	139318	10	4-way SIMD [19]	[19]	`AVX2`, intrinsics
$\mathcal{KL}_{p251\text{-}9,81,20}$	106640	83424	9	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p255\text{-}19,838,831}$	113545	91837	9	4-way SIMD	this work	`AVX2`, assembly
	118959	91151	10	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p266\text{-}3,683,18}$	135243	105328	10	4-way SIMD	this work	`AVX2`, assembly

Table 9: CPU-cycle counts for variable base scalar multiplication at the 128-bit security level.

Curve	Haswell	Skylake	κ	Strategy	Implementation	Implementation type
M⟦4058⟧	-	384905	7	64-bit seq	[29]	`maax`, assembly
	476866	401809	16	4-way SIMD [24]	[29]	`AVX2`, assembly
Curve448	-	434831	7	64-bit seq	[28]	`maax`, assembly
	441715	357095	16	4-way SIMD [24]	[24]	`AVX2`, assembly
$\mathcal{KL}_{p444\text{-}17,659,370}$	471372	385300	16	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p448\text{-}224\text{-}1,410,103}$	439902	350065	16	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p452\text{-}3,913,294}$	487605	397713	16	4-way SIMD	this work	`AVX2`, assembly

Table 10: CPU-cycle counts for variable base scalar multiplication at the 224-bit security level.

codes on the same platforms where we measured the timings of our implementations. The timings thus obtained have been reported in the tables.

Timings in form of CPU-cycles are shown in Tables 9, 10 and 11 for Haswell and Skylake processors. For comparison, we report the timings of the previously most efficient (to the best of our knowledge) and publicly available sequential and vectorized implementations. In the tables, `maax`-type implementations refer to 64-bit sequential implementations which have been developed using the modern `mulx/adcx/adox` instructions. In the discussion below, speed-up percentages are measured as $100(t_1 - t_2)/t_1$, where t_1 and t_2 are the old and new timings respectively.

Curve	Haswell	Skylake	κ	Strategy	Implementation	Implementation type
M⟦996558⟧	-	558757	8	64-bit seq	[27]	`maax`, assembly
	671196	568938	18	4-way SIMD [24]	this work	`AVX2`, assembly
M⟦952902⟧	-	566088	8	64-bit seq	[27]	`maax`, assembly
	677102	573672	18	4-way SIMD [24]	this work	`AVX2`, assembly
M⟦1504058⟧	-	689588	9	64-bit seq	[27]	`maax`, assembly
	651211	542726	18	4-way SIMD [24]	this work	`AVX2`, assembly
$\mathcal{KL}_{p506\text{-}45,856,181}$	666856	543248	18	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p510\text{-}75,1063,198}$	672097	554248	18	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p521\text{-}1,1559,796}$	650107	532177	18	4-way SIMD	this work	`AVX2`, assembly

Table 11: CPU-cycle counts for variable base scalar multiplication at the 256-bit security level.

New implementations of Kummer lines at 128-bit security level. The timing results for the new implementations of variable base scalar multiplication of the Kummer lines at 128-bit security level are shown in Table 9. We achieve the following speed-ups.

1. Our implementations of $\mathcal{KL}_{p251\text{-}9,81,20}$ are 17% and 26% faster in Haswell and Skylake respectively over the $\mathcal{KL}_{p251\text{-}9,81,20}$ implementations of [19].
2. Our 9-limb implementations of $\mathcal{KL}_{p255\text{-}19,838,831}$ are 33% and 35% faster in Haswell and Skylake respectively over the $\mathcal{KL}_{p255\text{-}19,82,77}$ implementations of [19].
3. Our 10-limb implementations of $\mathcal{KL}_{p255\text{-}19,838,831}$ are 39% and 35.3% faster in Haswell and Skylake respectively over the $\mathcal{KL}_{p255\text{-}19,82,77}$ implementations of [19].
4. Our implementations of $\mathcal{KL}_{p266\text{-}3,683,18}$ are 18% and 24.4% faster in Haswell and Skylake respectively over the $\mathcal{KL}_{p266\text{-}3,260,139}$ implementations of [19].

Vectorized implementations of Montgomery curves at 256-bit security level. The timing results for the new implementations of variable base scalar multiplication of the Montgomery curves at 256-bit security level are shown in Table 9. We achieve the following speed-ups.

1. Our vectorized implementation of M⟦1504058⟧ is 21% faster in Skylake over the 64-bit `maax`-type sequential implementation of M⟦1504058⟧ from [27].
2. Our vectorized implementation of M⟦996558⟧ and M⟦952902⟧ are about 3% slower in Skylake over the 64-bit `maax`-type sequential implementation of the curves from [27].
3. It has been mentioned in [29] that the vectorized implementations of the Montgomery curves at the 128-bit and 224-bit security levels outperform the sequential implementations in Haswell. On the basis of this we have left out exploring the sequential implementations of the Montgomery curves at 256-bit security level targeting the Haswell architecture.

Comparison between Kummer lines and Montgomery curves. From the timings provided in the Tables 9, 10 and 11 it can be observed that the performances of variable base scalar multiplication over the Kummer lines are uniformly better than the variable base scalar multiplication over the corresponding Montgomery curves at all the security levels.

Key generation. We have also developed vectorized assembly code for fixed base scalar multiplication for the Kummer lines at all the security levels. This corresponds to the key generation phase of the Diffie-Hellman protocol. The timings are reported in Table 12. For comparison, we report the corresponding timings of fixed-base scalar multiplication over the Kummer lines at 128-bit security level from [19]. It can be observed that the timings of our implementations are substantially better than those of [19].

Additionally, we have developed the vectorized implementations of fixed-base scalar multiplications over the Montgomery curves at 256-bit security level. These have been done following Algorithm 7 of [24]. The timings of the implementations have been reported at the end of Table 12. Here too it can be seen that the implementations over the Kummer lines outperform the implementations over the Montgomery curves.

Operation	Haswell	Skylake	κ	Strategy	Implementation	Implementation type
$\mathcal{KL}_{p251\text{-}9,81,20}$	10202	91261	9	4-way SIMD [19]	[19]	`AVX2`, intrinsics
	84892	67491	9	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p255\text{-}19,82,77}$	120353	107558	10	4-way SIMD [19]	[19]	`AVX2`, intrinsics
$\mathcal{KL}_{p255\text{-}19,838,831}$	94203	72945	9	4-way SIMD	this work	`AVX2`, assembly
	91399	70872	10	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p266\text{-}3,683,18}$	123071	110691	10	4-way SIMD [19]	[19]	`AVX2`, intrinsics
	104768	82598	10	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p444\text{-}17,659,370}$	347215	273842	16	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p448\text{-}224\text{-}1,410,103}$	325031	244571	16	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p452\text{-}3,913,294}$	360158	283582	16	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p506\text{-}45,856,181}$	489930	391305	18	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p510\text{-}75,1063,198}$	497224	399064	18	4-way SIMD	this work	`AVX2`, assembly
$\mathcal{KL}_{p521\text{-}1,1559,796}$	476917	386050	18	4-way SIMD	this work	`AVX2`, assembly
M⟦996558⟧	567250	480584	18	4-way SIMD [24]	this work	`AVX2`, assembly
M⟦952902⟧	573289	486202	18	4-way SIMD [24]	this work	`AVX2`, assembly
M⟦1504058⟧	555958	475849	18	4-way SIMD [24]	this work	`AVX2`, assembly

Table 12: CPU-cycle counts required by the curves for fixed base scalar multiplication.

9 Conclusion

This work considered the comparative efficiencies of vectorized implementations of Kummer lines and Montgomery curves at the 128-bit, 224-bit and 256-bit security levels. A total of nine Kummer lines were included for comparison, out of which eight were obtained in this work. New implementations of all the nine Kummer lines have been made in assembly for Intel processors. A total of seven Montgomery curves were considered and new vectorized assembly implementations of five of these have been made. Timing results of all the implementations show that Kummer lines are consistently faster than Montgomery curves at all security levels.

Acknowledgements. We thank Sabyasachi Karati for some preliminary discussion regarding arithmetic modulo $2^{521} - 1$.

References

[1] Jon Barwise and Paul C. Eklof. Lefschetzs principle. *J. Algebra*, 13(4):554–570, 1969.

[2] D. J. Bernstein and T. Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to/index.html, accessed on November 04, 2020.

[3] Daniel J. Bernstein. Can we avoid tests for zero in fast elliptic-curve arithmetic? https://cr.yp.to/ecdh/curvezero-20060726.pdf, 2006. Accessed on 16 September, 2019.

[4] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[5] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008.

[6] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[7] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the Montgomery ladder. In Joppe W. Bos and Arjen K. Lenstra, editors, *Topics in Computational Number Theory inspired by Peter L. Montgomery*, pages 82–115. Cambridge University Press, 2017.

[8] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.

[9] Tung Chou. Sandy2x: New Curve25519 speed records. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2015.

[10] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic - the case of large characteristic fields. *J. Cryptographic Engineering*, 8(3):227–240, 2018.

[11] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions of Information Theory*, 22(6):644–654, 1976.

[12] Armando Faz-Hernández, Julio López Hernandez, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, 45(3):25:1–25:35, 2019.

[13] Gerhard Frey and Hans -G. Rück. The strong lefschetz principle in algebraic geometry. *Manuscripta Mathematica*, 55(3):385–401, 1986.

[14] Pierrick Gaudry. Fast genus 2 arithmetic based on theta functions. *J. Mathematical Cryptology*, 1(3):243–265, 2007.

[15] Pierrick Gaudry and David Lubicz. The arithmetic of characteristic 2 kummer surfaces and of elliptic kummer lines. *Finite Fields Their Applications*, 15(2):246–260, 2009.

[16] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive*, 2015:625, 2015.

[17] Hüseyin Hisil, Berkan Egrice, and Mert Yassi. Fast 4 way vectorized ladder for the complete set of montgomery curves. *IACR Cryptology ePrint Archive*, 2020:388, 2020.

[18] Sabyasachi Karati and Palash Sarkar. Connecting Legendre with Kummer and Edwards. *Adv. Math. Commun.*, 13(1):41–66, 2019.

[19] Sabyasachi Karati and Palash Sarkar. Kummer for Genus One Over Prime-Order Fields. *J. Cryptology*, 33(1):92–129, 2020.

[20] Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, 1987.

[21] Adam Langley and Mike Hamburg. Elliptic curves for security. Internet Research Task Force (IRTF), Request for Comments: 7748, https://tools.ietf.org/html/rfc7748, 2016. Accessed on 16 September, 2019.

[22] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO'85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 417–426. Springer Berlin Heidelberg, 1985.

[23] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[24] K. Nath and P. Sarkar. Efficient 4-way vectorizations of the montgomery ladder. *IEEE Transactions on Computers*, Feb 2021.

[25] Kaushik Nath and Palash Sarkar. Constant Time Montgomery Ladder. *IACR Cryptol. ePrint Arch.*, 2020:956, 2020.

[26] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)Mersenne prime order fields. *Advances in Mathematics of Communications*, 2020.

[27] Kaushik Nath and Palash Sarkar. Efficient elliptic curve Diffie-Hellman computation at the 256-bit security level. *IET Inf. Secur.*, 14(6):633–640, 2020.

[28] Kaushik Nath and Palash Sarkar. Reduction modulo $2^{448} - 2^{224} - 1$. *Mathematical Cryptology*, 1(1):8–21, Jan. 2021.

[29] Kaushik Nath and Palash Sarkar. Security and Efficiency Trade-offs for Elliptic Curve Diffie-Hellman at the 128-bit and 224-bit Security Levels. *Journal of Cryptographic Engineering*, Accepted, 2021.

[30] Thomaz Oliveira, Julio López Hernandez, Hüseyin Hisil, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder - improving the performance of X25519 and X448. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, volume 10719 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2017.

[31] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm. http://cado-nfs.gforge.inria.fr, 2017. Release 2.3.0.

[32] Version 1.3 TLS Protocol. RFC 8446. https://datatracker.ietf.org/doc/rfc8446/?include_text=1, 2018. Accessed on 16 September, 2019.

[33] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman and Hall/CRC, 2003.

A Field Multiplication/Squaring over $\mathbb{F}_{2^{521}-1}$

Here we discussion the field multiplication/squaring over the field $\mathbb{F}_{2^{521}-1}$. It has two phases, the integer multiplication which is done through a Karatsuba and the reduction after that.

A.1 Multiplication in $\mathbb{F}_{2^{521}-1}$ when $\kappa = 18$

Define $p = 2^{521} - 1$ and $\theta = 2^{29}$. Let $A = \sum_{i=0}^{17} a_i\theta^i$ and $B = \sum_{i=0}^{17} b_i\theta^i$ be two elements in $\mathbb{F}_{2^{521}-1}$. We can perform a schoolbook multiplication of A and B and reduce the product as was done for $p = 2^{255}-19$. But this might be costly. Instead, we divide the problem and use the Karatsuba technique to achieve a better field multiplication algorithm, which we discuss below. Define $\phi = \theta^9 = 2^{261}$ and write the field element A as

$$\begin{aligned} A &= a_0 + a_1\theta + \cdots + a_{15}\theta^{17} \\ &= (a_0 + a_1\theta + \cdots + a_8\theta^8) + (a_9 + a_{10}\theta + \cdots + a_{17}\theta^8)\theta^9 \\ &= U + V\phi \end{aligned} \tag{3}$$

where $U = a_0 + a_1\theta + \cdots + a_8\theta^8$ and $V = a_9 + a_{10}\theta + \cdots + a_{17}\theta^8$. Similarly, consider the field element

$$\begin{aligned} B &= b_0 + b_1\theta + \cdots + b_{15}\theta^{17} \\ &= (b_0 + b_1\theta + \cdots + b_8\theta^8) + (b_9 + b_{10}\theta + \cdots + b_{17}\theta^8)\theta^9 \\ &= W + Z\phi \end{aligned} \tag{4}$$

where $W = b_0 + b_1\theta + \cdots + b_8\theta^8$ and $Z = b_9 + b_{10}\theta + \cdots + b_{17}\theta^8$. The bounds on the limbs of A and B are

$$0 \le a_0, a_2, \ldots, a_{16} < 2^{29}, 0 \le a_1 < 2^{30} \text{ and } 0 \le a_{17} < 2^{28} \tag{5}$$

$$0 \le b_0, b_2, \ldots, b_{16} < 2^{29}, 0 \le b_1 < 2^{30} \text{ and } 0 \le b_{17} < 2^{28} \tag{6}$$

The product of the input polynomials A and B is given by

$$\begin{aligned} C &= AB \\ &= (U + V\phi)(W + Z\phi)) \\ &= UW + (UZ + VW)\phi + VZ\phi^2 \\ &\equiv (UW + 2VZ) + (UZ + VW)\phi \mod p \\ &= (UW + 2VZ) + ((U + V)(W + Z) - (UW + VZ))\phi. \end{aligned} \tag{7}$$

We now compute the three products UW, VZ and $(U + V)(W + Z)$ with the schoolbook method using $3 \times 9 \times 9 = 243$ limb-multiplications and combine the results to find the product C. This gives us a saving of 81 limb-multiplications as compared to the schoolbook method when applied to the entire 18-limb polynomials A and B. We can find similar equation for squaring as

$$\begin{aligned} C &= A^2 \\ &= (U^2 + 2V^2) + ((U + V)^2 - (U^2 + V^2))\phi. \end{aligned} \tag{8}$$

The product UV is computed as the polynomial $R = UW = \sum_{j=0}^{16} r_j\theta^j$, where

$$r_j = \sum_{i=0}^{j} a_i b_{j-i}, \text{ for } j = 0, 1, \ldots, 8; \tag{9}$$

$$r_{j+8} = \sum_{i=j}^{8} a_i b_{8-i+j}, \text{ for } j = 1, 2, \ldots, 8. \tag{10}$$

Similarly, let the products VZ and $(U + V)(W + Z)$ be denoted by $S = \sum_{j=0}^{16} s_j\theta^j$ and $T = \sum_{j=0}^{16} t_j\theta^j$ respectively. Then we can write

$$\begin{aligned} C &= (R + 2S) + (T - R - S)\phi \\ &= E + F\phi. \end{aligned} \tag{11}$$

A.2 Reduction in $\mathbb{F}_{2^{521}-1}$ when $\kappa = 18$

First phase of the reduction. To perform the first phase of reduction on the product $C = E + F\phi$, we perform some carry-less additions with specific coefficients of the polynomial C to arrive to a certain polynomial on which the second phase of the reduction can be applied. We describe the method below.

$$\begin{aligned}
C &= E + F\phi \\
&= \sum_{j=0}^{16} e_j\theta^j + \sum_{j=0}^{16} f_j\theta^{j+9} \\
&= \sum_{j=0}^{8} e_j\theta^j + \sum_{j=9}^{16} (e_j + f_{j-9})\theta^j + \sum_{j=17}^{25} f_{j-9}\theta^j \\
&= \sum_{j=0}^{8} (r_j + 2s_j)\theta^j + \sum_{j=9}^{16} (r_j + 2s_j + t_{j-9} - r_{j-9} - s_{j-9})\theta^j + \sum_{j=17}^{25} (t_{j-9} - r_{j-9} - s_{j-9})\theta^j \\
&= \sum_{j=0}^{25} c_j\theta^j, \text{ (say)} \\
&\equiv \sum_{j=0}^{7} (c_j + 2c_{j+18})\theta^j + \sum_{j=8}^{17} c_j\theta^j \mod p \qquad [\text{ since } 2^{522} \equiv 2 \bmod p\ ] \\
&= \sum_{j=0}^{7} (r_j + 2s_j + 2t_{j+9} - 2r_{j+9} - 2s_{j+9})\theta^j + (r_8 + 2s_8)\theta^8 + \\
&\quad \sum_{j=9}^{16} (r_j + 2s_j + t_{j-9} - r_{j-9} - s_{j-9})\theta^j + (t_8 - r_8 - s_8)\theta^{17} \\
&= \sum_{j=0}^{17} h_j\theta^j = H \text{ (say)}. \qquad (12)
\end{aligned}$$

Second phase of the reduction. We now apply the second phase of reduction on the polynomial H. This is done through a simple carry chain on the coefficients of H as

$$h_0 \to h_1 \to \cdots \to h_{17} \to h_0 \to h_1$$

which performs a partial reduction on the coefficients of H by keeping an extra bit in the second limb of the reduced polynomial. A single carry step $h_{j \bmod 18} \to h_{(j+1) \bmod 18}$ perform the following operations.

- Logically right shift the 64-bit word in $h_{j \bmod 18}$ by 29 bits. For $j = 17$ the amount of shift is 28 bits. Let this amount be $\mathfrak{c}$.
- Add $\mathfrak{c}$ to $h_{(j+1) \bmod 18}$.
- Mask out the most significant 35 bits of $h_{j \bmod 18}$. For $j = 17$ the masking amount is 26 bits.

It has to be noted that an interleaved carry chain similar to $p255$-19 [9] can also be applied here as well. We have implemented this strategy, but it did not lead to any efficiency gain in this case.

B Difficult Factorizations using CADO-NFS

- For the Kummer line $\mathcal{KL}_{p510\text{-}75,1063,198}$, the value of ℓ_T is

  418993997810706159361688281193932691483730181893512293053861295116305125939807271481018349009896963361626533879377337777689648106792207292991624363411297

  and the different factors of $(\ell_T - 1)$ found by CADO-NFS are

  2, 3, 11549, 392569, 2767795850519536795089261258030828799684574536724641257218031159366846281208186788815745408812572669324117361067657462133881331049785856264069

  in which 2 has multiplicity of 5 and 3 has multiplicity of 2.

- For the Kummer line $\mathcal{KL}_{p521\text{-}1,1559,796}$, the value of ℓ is

  858099707516326214372737599885174152158679412517913176174307932398192897924706658162526282103673152533760615791573986996809484874259004814711790456624840489

  and the different factors of $(\ell - 1)$ found by CADO-NFS are

  3, 1157964121682084988272973724079800734311154368548591893923, 2, 19 and 1625091459197308998032556831933275475624427132870879953381761328096934319354460433632689599511951

  in which 2 has multiplicity of 3.

- For the Kummer line $\mathcal{KL}_{p521\text{-}1,1559,796}$, the value of ℓ_T is

  858099707516326214372737599885174152158679412517913176174307932398192897924707354868113628061690486210563462056296227512471012125670156138431716616153923799

  and the different factors of $(\ell_T - 1)$ found by CADO-NFS are

  744530470339986806241554824273069466780194283690308870415228581592384185643 09, 2, 3 and 213432974974415376531928914629269546394997549996790267692841033840218928274093

  in which 3 has multiplicity of 3.