

How to (pre-)compute a ladder

Improving the performance of X25519 and X448

Thomaz Oliveira¹, Julio López², Hüseyin Hisil³, Armando Faz-Hernández²,
and Francisco Rodríguez-Henríquez¹

¹ Computer Science Department, Cinvestav-IPN
thomaz.figueiredo@gmail.com, francisco@cs.cinvestav.mx

² Institute of Computing, University of Campinas
jlopez@ic.unicamp.br, armfazh@ic.unicamp.br

³ Yasar University
huseyin.hisil@yasar.edu.tr

Abstract. In the RFC 7748 memorandum, the Internet Research Task Force specified a Montgomery-ladder scalar multiplication function based on two recently adopted elliptic curves, “curve25519” and “curve448”. The purpose of this function is to support the Diffie-Hellman key exchange algorithm that will be included in the forthcoming version of the Transport Layer Security cryptographic protocol. In this paper, we describe a ladder variant that permits to accelerate the fixed-point multiplication function inherent to the Diffie-Hellman key pair generation phase. Our proposal combines a right-to-left version of the Montgomery ladder along with the pre-computation of constant values directly derived from the base-point and its multiples. To our knowledge, this is the first proposal of a Montgomery ladder procedure for prime elliptic curves that admits the extensive use of pre-computation. In exchange of very modest memory resources and a small extra programming effort, the proposed ladder obtains significant speedups for software implementations. Moreover, our proposal fully complies with the RFC 7748 specification. A software implementation of the X25519 and X448 functions using our pre-computable ladder yields an acceleration factor of roughly 1.20, and 1.25 when implemented on the Haswell and the Skylake micro-architectures, respectively.

Keywords: Montgomery ladder, Elliptic curve scalar multiplication, Diffie-Hellman protocol, RFC 7748.

1 Introduction

Since the last decades, Elliptic Curve Cryptography (ECC) has been used for achieving highly secure and highly efficient cryptographic communication implementations. In particular, ECC has become the prime choice for realizing key exchange and digital signature-verification protocols. However, several reports released in 2013 suggested that the National Security Agency (NSA) secretly introduced backdoors to internationally-used encryption standards [38].

Immediately thereafter, new revelations [37] indicated that the same agency had tampered the elliptic curve-based pseudorandom number generator standard Dual_EC_DRBG, which was consequently removed from the SP 800-90A specification by NIST [30,31].

In 2014, the Transport Layer Security (TLS) working group of the Internet Engineering Task Force reacted to these events requesting from the Crypto Forum Research Group (CFRG), recommendations of new elliptic curves to be integrated into the next version of the TLS protocol [39]. Some of the requirements for the selection of such curves were based on [5,36], which advocate for a number of design practices and elliptic curve properties, including rigidity in the curve-generation process and simplicity in the implementation of cryptographic algorithms. After a long and lengthy discussion, two prime elliptic curves, known as Curve25519 and Curve448, were chosen for the 128-bit and 224-bit security levels, respectively (see §3 for more details). The RFC 7748 [26] memorandum describes the implementation details related to this choice, including the curve parameters and the Montgomery ladder-based scalar multiplication algorithms, also referred to as X25519 and X448 functions.

The Montgomery ladder and Montgomery curves were introduced in [28]. Since then, the Montgomery ladder has been carefully studied by many authors, as discussed for example, in the survey by Costello and Smith in [12] (see also [7]). We know now how to use the Montgomery ladder for computing the point multiplication kP , where P is usually selected as a point that belongs to a prime order r subgroup of an elliptic curve, and k is an integer in the $[1, r - 1]$ interval. Nevertheless, arguably the most important application of the Montgomery ladder lies in the Diffie-Hellman shared-secret computation as described in [26].

The classical Montgomery ladder as it was presented in [28], is a left-to-right scalar multiplication procedure that does not admit in a natural way efficient pre-computation mechanisms. In an effort to obtain this feature, and in the context of binary elliptic curves, the authors of [32] presented a right-to-left Montgomery ladder that can take advantage of pre-computing multiples of the fixed base point P . Notice that this procedure was previously reported by Joye in [21]. However, the procedure presented in [32] crucially depended on the computation of the point halving operation. Although this primitive can be performed at a low computational cost in binary elliptic curves, in general there are no known procedures to compute it efficiently for elliptic curves defined over odd prime fields. Hence, it appeared that the finding of the right-to-left ladder procedure of [32] was circumscribed to binary elliptic curves, as there was no obvious way to extend it to elliptic curves defined over large prime fields.

Our contributions In this paper, we present an alternative way to compute the key exchange protocol presented in [26]. In short, we propose different X25519 and X448 functions which can take advantage of the fixed-point scenario provided by the Diffie-Hellman key generation phase. This algorithm achieves an estimated performance increase of roughly 20% at the price of a small amount of extra memory resources. In addition, it does not intervene with the original RFC

specification and it is straightforward to implement, preserving the simplicity feature of the original design.

The remainder of this paper is organized as follows. In §2 we briefly describe the Diffie-Hellman protocol. In §3 we give more details on the CFRG selected elliptic curves. The Montgomery ladder-based scalar multiplication functions X25519 and X448 are analyzed in §4. Our proposal is discussed in §5 and our concluding remarks and future work are presented in §8.

2 The Diffie-Hellman protocol

The Diffie-Hellman key exchange protocol, introduced by Diffie and Hellman in [13], is a method that allows to establish a shared secret between two parties over an insecure channel. Originally proposed for multiplicative groups of integers modulo p , with p a prime number, the scheme was later adapted to additively-written groups of points on elliptic curves by Koblitz and Miller in [22,27]. Commonly known as elliptic curve Diffie-Hellman protocol (ECDH), this variant is concisely described in Algorithm 1.

Algorithm 1 The elliptic curve Diffie-Hellman protocol

Public parameters: Prime p , curve E/\mathbb{F}_p , point $P = (x, y) \in E(\mathbb{F}_p)$ of order r

Phase 1: Key pair generation

Alice	Bob
1: Select the private key $d_A \xleftarrow{\$} [1, r - 1]$	1: Select the private key $d_B \xleftarrow{\$} [1, r - 1]$
2: Compute the public key $Q_A \leftarrow d_A P$	2: Compute the public key $Q_B \leftarrow d_B P$

Phase 2: Shared secret computation

Alice	Bob
3: Send Q_A to Bob	3: Send Q_B to Alice
4: Compute $R \leftarrow d_A Q_B$	4: Compute $R \leftarrow d_B Q_A$

Final phase: The shared secret is the point R x-coordinate

As shown in Algorithm 1, the ECDH protocol is divided into two phases; in the first phase, both parties generate their private and public key pair. The private key d_A (d_B) is an integer chosen uniformly at random from the interval $[1, r - 1]$ while the public key Q_A (Q_B) is the resulting point of the scalar multiplication of d_A (d_B) by the base-point P . In the majority of the proposed elliptic curve-based standards and specifications (e.g. [14,9,29], including [26]), the point P is fixed and its coordinates are explicitly given in the documentation. At the implementation level, this setting is usually called fixed- or known-point scenario.

After computing their respective public/private key pair, each party sends her public key to the other. Next, they perform the point multiplication of the

received public key by their own private key. The group properties of $E(\mathbb{F}_p)$ guarantee that $R = d_A Q_B = d_A(d_B P) = d_B(d_A P) = d_B Q_A = R$. As a result, the parties have access to a common piece of information, represented by the x -coordinate of R , which is only disclosed to themselves.⁴ Since the public key Q_B (Q_A) is not known a priori by Alice (Bob), the scalar multiplication in the second phase is said to be performed in a variable- or unknown-point scenario.

3 The curves

The [26] memorandum specifies two Montgomery elliptic curves of the form,

$$E_A/\mathbb{F}_p : v^2 = u^3 + Au^2 + u. \quad (1)$$

The standard specification for the 128 bits of security level uses the prime $p = 2^{255} - 19$, and the curve parameter is given by $A = 486662$. This curve is commonly known as Curve25519 and was proposed in 2005 by Bernstein [1]. The point group order is given as $\#E_{486662}(\mathbb{F}_{2^{255}-19}) = h \cdot r \approx 2^{255}$, with $h = 8$ and $r = 2^{252} + 27742317777372353535851937790883648493$. The order- r base-point $P = (u, v)$ is specified as,

$$\begin{aligned} u_P &= 0x9 \\ v_P &= 0x20AE19A1B8A086B4E01EDD2C7748D14C \\ &\quad 923D4D7E6D7C61B229E9C5A27ECED3D9. \end{aligned}$$

The recommendation for the 224-bit security level is to use $p = 2^{448} - 2^{224} - 1$ and $A = 156326$. This curve was originally proposed by Hamburg in the Edwards form as Ed448-Goldilocks [18], but it is referred in [26] as Curve448. The group order $\#E_{156326}(\mathbb{F}_{2^{448}-2^{224}-1}) = h \cdot r \approx 2^{448}$, with $h = 4$ and, $r = 2^{446} - 13818066809895115352007386748515426880336692474882178609894547503885$.

For this curve, the base-point P is given by

$$\begin{aligned} u_P &= 0x5 \\ v_P &= 0x7D235D1295F5B1F66C98AB6E58326FCECBAE5D34F55545D060F75DC2 \\ &\quad 8DF3F6EDB8027E2346430D211312C4B150677AF76FD7223D457B5B1A. \end{aligned}$$

4 The scalar multiplication operation

Let E_A/\mathbb{F}_p be an elliptic curve and P an order- r point in $E_A(\mathbb{F}_p)$. Then, for any n -bit scalar $k = (k_{n-1}, \dots, k_2, k_1, k_0)_2 \in [1, r - 1]$, the scalar multiplication operation is given by $Q = kP = k_{n-1}2^{n-1}P + \dots + k_22^2P + k_12P + k_0P$. As presented in §2, the scalar multiplication function is used in the two first ECDH phases; first, to generate the public keys Q_A and Q_B and later, in the second phase, to compute the common point R .

⁴ Here, we are considering an ideal but unrealistic scenario. In practice, an inappropriate choice of the elliptic curve parameters, the prime p , the order r , the implementation of the scalar multiplication algorithm, among many other aspects, could disqualify this statement.

4.1 Left-to-right Montgomery ladder

Initially proposed to improve the performance of integer factorization algorithms, the Montgomery ladder [28] is now largely used in the design of constant-time scalar multiplication implementations. This is because its ladder step structure assures that the same arithmetic operations are executed independently of the scalar bits k_i values. A high-level description of this procedure is presented in Algorithm 2.

Algorithm 2 Left-to-right Montgomery ladder

Input: $P = (u_P, v_P) \in E_A(\mathbb{F}_p)$, $k = (k_{n-1} = 1, k_{n-2}, \dots, k_1, k_0)_2$

Output: $u_{Q=kP}$

```

1:  $R_0 \leftarrow \mathcal{O}$ ;  $R_1 \leftarrow u_P$ ;
2: for  $i = n - 1$  downto 0 do
3:   if  $k_i = 1$  then
4:      $R_0 \leftarrow R_0 + R_1$ ;  $R_1 \leftarrow 2R_1$ 
5:   else
6:      $R_1 \leftarrow R_0 + R_1$ ;  $R_0 \leftarrow 2R_0$ 
7:   end if
8: end for
9: return  $u_Q \leftarrow R_0$ 

```

If the difference between the points R_1 and R_0 is known, it is possible to derive efficient formulas for computing $R_0 + R_1$ that refer only to the u -coordinates of the operands, a formula that is sometimes named as differential addition [12].⁵ That is the main rationale for Algorithm 2; throughout its execution, the Montgomery ladder maintains the invariant $R_1 - R_0 = P$ by computing at each iteration

$$(R_0, R_1) \leftarrow \begin{cases} (2R_0, 2R_0 + P), & \text{if } k_i = 0 \\ (2R_0 + P, 2R_0 + 2P), & \text{if } k_i = 1. \end{cases}$$

In order to avoid expensive field inversions, one can accelerate the scalar multiplication procedure by using projective coordinates, by means of the transformation $u = U/Z$. In the context of Algorithm 2, the differential addition formula required in Step 6 can be computed as [12,26],

$$\begin{aligned} U_{R_1} &\leftarrow Z_P((U_{R_1} + Z_{R_1}) \cdot (U_{R_0} - Z_{R_0}) + (U_{R_1} - Z_{R_1}) \cdot (U_{R_0} + Z_{R_0}))^2 \\ Z_{R_1} &\leftarrow u_P((U_{R_1} + Z_{R_1}) \cdot (U_{R_0} - Z_{R_0}) - (U_{R_1} - Z_{R_1}) \cdot (U_{R_0} + Z_{R_0}))^2. \end{aligned} \quad (2)$$

where the standard trick of use $Z_P = 1$, saves one field multiplication. Thus, it can be seen that the computational cost of performing the differential addition formula of Eq. (2) is of $3\mathbf{m} + 2\mathbf{s} + 6\mathbf{a}$.

⁵ It is also possible to express the u -coordinate of the resulting point $R_i = 2R_i$, for $i \in \{0, 1\}$, using only the u -coordinate of the operand P , an operation known as differential doubling.

Similarly, the differential point doubling required in Step 6 of Algorithm 2 can be computed as [12,26],

$$\begin{aligned} U_{R_0} &\leftarrow (U_{R_0} + Z_{R_0})^2 \cdot (U_{R_0} - Z_{R_0})^2 \\ T &\leftarrow (U_{R_0} + Z_{R_0})^2 - (U_{R_0} - Z_{R_0})^2 \\ Z_{R_0} &\leftarrow [a_{24} \cdot T + (U_{R_0} - Z_{R_0})^2] \cdot T, \end{aligned} \quad (3)$$

where $a_{24} = \frac{A+2}{4}$. It can be readily seen that the computational cost of performing the differential doubling formula of Eq. (3) is of $2\mathbf{m} + 1\mathbf{m}_{\mathbf{a}24} + 2\mathbf{s} + 4\mathbf{a}$.⁶

A low-level description of the left-to-right ladder on prime elliptic curves in Montgomery form is given in Algorithm 3.⁷ When computed with the parameters listed in §3, this algorithm is called X25519 (with $n = 255$) or X448 (with $n = 448$) [26]. The \oplus notation stands for the exclusive-or logical operator, while the symbols $+$, $-$, \times ,² and $^{-1}$ represent the field \mathbb{F}_p arithmetic operations of addition, subtraction, multiplication, squaring and inversion, respectively.

At each iteration i of Algorithm 3, the conditional swap function (cswap) exchanges the values of the R_0 and R_1 coordinates when the bits k_{i-1} and k_i are different. This function is a countermeasure for potential cache-based attacks [23,24], which could reveal the scalar digits (the private key in Alg. 1) by determining the access order of the points R_0 and R_1 . The cswap function consists only of simple logic operations, so its cost will be disregarded in our estimations. For more details on the implementation of this function see [26,32].

Cost estimations Let \mathbf{m} , $\mathbf{m}_{\mathbf{a}24}$, $\mathbf{m}_{\mathbf{uP}}$, \mathbf{s} , \mathbf{i} and \mathbf{a} represent the cost of a general multiplication, multiplication by the constant $(A+2)/4$, multiplication by the u -coordinate of the base-point P , squaring, inversion and addition/subtraction over the field \mathbb{F}_p , respectively. Then, the computing cost of the left-to-right Montgomery ladder is $n \cdot (4\mathbf{m} + 1\mathbf{m}_{\mathbf{a}24} + 1\mathbf{m}_{\mathbf{uP}} + 4\mathbf{s} + 8\mathbf{a}) + 1\mathbf{m} + 1\mathbf{i}$. More specifically, at the 128 bits of security level, the X25519 function costs

$$1021\mathbf{m} + 255\mathbf{m}_{\mathbf{a}24} + 255\mathbf{m}_{\mathbf{uP}} + 1020\mathbf{s} + 2040\mathbf{a} + 1\mathbf{i},$$

where each operation is performed in the prime field $\mathbb{F}_{2^{255}-19}$. At the 224-bit security level case, the cost for computing the function X448 is

$$1793\mathbf{m} + 448\mathbf{m}_{\mathbf{a}24} + 448\mathbf{m}_{\mathbf{uP}} + 1792\mathbf{s} + 3584\mathbf{a} + 1\mathbf{i},$$

with the arithmetic operations being carried out in the prime field $\mathbb{F}_{2^{448}-2^{224}-1}$.

5 How to (pre-)compute a ladder

Our proposal for improving the performance of the X25519 and X448 functions focuses in the first phase of the Diffie-Hellman protocol (see Alg. 1). There, the

⁶ where $\mathbf{m}_{\mathbf{a}24}$ stands for one multiplication by the constant a_{24} .

⁷ The description is closely related to [26, §5].

Algorithm 3 Low-level left-to-right Montgomery ladder

Input: $P = (u_P, v_P) \in E_A/\mathbb{F}_p$, $k = (k_{n-1} = 1, k_{n-2}, \dots, k_1, k_0)_2$, $a_{24} = (A + 2)/4$ **Output:** $u_{Q=kP}$

```
1: Initialization:  $U_{R_0} \leftarrow 1, Z_{R_0} \leftarrow 0, U_{R_1} \leftarrow u_P, Z_{R_1} \leftarrow 1, s \leftarrow 0$ 
2: for  $i \leftarrow n - 1$  downto 0 do
3:   # timing-attack countermeasure
4:    $s \leftarrow s \oplus k_i$ 
5:    $U_{R_0}, U_{R_1} \leftarrow \text{cswap}(s, U_{R_0}, U_{R_1})$ 
6:    $Z_{R_0}, Z_{R_1} \leftarrow \text{cswap}(s, Z_{R_0}, Z_{R_1})$ 
7:    $s \leftarrow k_i$ 
8:   # common operations
9:    $A \leftarrow U_{R_0} + Z_{R_0}; B \leftarrow U_{R_0} - Z_{R_0}$ 
10:  # addition
11:   $C \leftarrow U_{R_1} + Z_{R_1}; D \leftarrow U_{R_1} - Z_{R_1}$ 
12:   $C \leftarrow C \times B; D \leftarrow D \times A$ 
13:   $U_{R_1} \leftarrow D + C; U_{R_1} \leftarrow U_{R_1}^2$ 
14:   $Z_{R_1} \leftarrow D - C; Z_{R_1} \leftarrow Z_{R_1}^2; Z_{R_1} \leftarrow u_P \times Z_{R_1}$ 
15:  # doubling
16:   $A \leftarrow A^2; B \leftarrow B^2$ 
17:   $U_{R_0} \leftarrow A \times B$ 
18:   $A \leftarrow A - B$ 
19:   $Z_{R_0} \leftarrow a_{24} \times A; Z_{R_0} \leftarrow Z_{R_0} + B; Z_{R_0} \leftarrow Z_{R_0} \times A$ 
20: end for
21:  $U_{R_0}, U_{R_1} \leftarrow \text{cswap}(s, U_{R_0}, U_{R_1})$ 
22:  $Z_{R_0}, Z_{R_1} \leftarrow \text{cswap}(s, Z_{R_0}, Z_{R_1})$ 
23:  $Z_{R_0} \leftarrow Z_{R_0}^{-1}$ 
24:  $u_{R_0} \leftarrow U_{R_0} \times Z_{R_0}$ 
25: return  $u_Q \leftarrow u_{R_0}$ 
```

scalar multiplication is performed in the fixed-point setting. More specifically, the point operand is always the base-point described in the [26] document (see §3 for more details).

One possible solution for taking advantage of this scenario was published in [2], in the context of message signing. In short, the authors pre-compute the points $P_{ij} = i16^jP$, for $1 \leq i \leq 8$ and $0 \leq j \leq 63$ and represent the Curve25519 in Edwards form to process the scalar multiplication through a windowed variant of the traditional double-and-add method. In addition to the significant amount of required memory space, the main drawback of this approach is that complex cache-attack countermeasures need to be applied during the retrieval of the pre-computed points P_{ij} , which go against the principle of implementation simplicity promoted in [5,36].

Thus, instead of designing a timing-protected double-and-add algorithm, we suggest using a slightly modified version of the right-to-left Montgomery ladder presented in [32] as explained in the following subsection.

5.1 Right-to-left Montgomery ladder with pre-computation

Algorithm 4 Right-to-left Montgomery ladder

Input: $P = (u_P, v_P) \in E_A(\mathbb{F}_p)$, $k = (k_{n-1} = 1, k_{n-2}, \dots, k_1, k_0)_2$

Output: $u_{Q=hkP}$

```

1: Pre-computation: Calculate and store  $u_{P_i}$ , where  $P_i = 2^i P$ , for  $0 \leq i \leq n$ 
2: Initialization: Select an order- $h$  point  $S \in E_A(\mathbb{F}_p)$ 
3:  $R_0 \leftarrow u_P$ ,  $R_1 \leftarrow u_S$ ,  $R_2 \leftarrow u_{P-S}$ 
4: for  $i \leftarrow 0$  to  $n - 1$  do
5:   if  $k_i = 1$  then
6:      $R_1 \leftarrow R_0 + R_1$  (with  $R_2 = R_0 - R_1$ )
7:   else
8:      $R_2 \leftarrow R_0 + R_2$  (with  $R_1 = R_0 - R_2$ )
9:   end if
10:   $R_0 \leftarrow u_{P_{i+1}}$ 
11: end for
12: return  $u_Q = hR_1$ 

```

The operating principle of Algorithm 4, is to compute $Q = kP$ using the Montgomery differential arithmetic formulas for the point doubling and point addition operations. This is achieved by recording and storing the difference $R_0 - R_1$ in the point R_2 through the whole execution of the procedure. Indeed, in the case that the bit $k_i = 1$, then R_0 is added to the accumulator R_1 (Step 6) and the difference R_2 does not change, since the operation $2R_0 = R_0 + R_0$ is performed in Step 10. On the other hand, if $k_i = 0$, nothing is added to the accumulator R_1 , so it is necessary to increase the difference R_2 by R_0 (Step 8) in order to account for the unconditional doubling performed in Step 10. Notice that at each iteration, the accumulator R_1 is updated in the same fashion as it would be done in a traditional right-to-left double-and-add algorithm. It follows that at the end of the main loop, $R_1 = kP + S$.

The reason why the accumulator R_1 must be initialized with a point $S \notin \langle P \rangle$ is because the differential formulas are not complete on Montgomery curves. Hence, one must prevent the cases where $R_0 = R_1$ or $R_0 = R_2$. One can eliminate S by performing a scalar multiplication by the cofactor h , thus obtaining

$$hR_1 = h \cdot (kP + S) = hkP + hS = hkP.$$

Notice that for Montgomery curves, the cofactor h is as little as four. So this last correction does not represent a computational burden. Furthermore, in § 5.4 we show a trick specially tailored for the X25519 and X448 functions, which eliminates the point S at almost no cost, and that allows us to return the correct $R_1 = kP$ result. Nevertheless, we stress that the points S and $P - S$ can be clearly specified beforehand and therefore, this matter should not bring any complications for the programmer.

Given that the difference between R_0 and R_1 is volatile, at first glance the differential point addition formula computed in Steps 6 and 8 of Algorithm 4, requires an extra field multiplication as compared with Eq. (2) of the classical ladder shown in Algorithm 2. This is basically because R_2 is now represented in full projective coordinates, which means that its Z -coordinate value will be in general different than one.

We discuss in the following how to compute the differential addition formula of Algorithm 4, without incurring in any additional cost as compared with the cost of Eq. (2) of Algorithm 2.

5.2 Montgomery differential addition with precomputation

Let $R_0 = (u_0, v_0)$ and $R_1 = (u_1, v_1)$, be two points of the elliptic curve of Eq.(1).⁸ Then, the point $R_3 = (u_3, v_3)$, such that, $R_3 = R_0 + R_1$, is determined as,

$$\begin{aligned} (u_3, v_3) &= (u_0, v_0) + (u_1, v_1) \\ &= \left(\frac{u_0 v_1 - v_0 u_1}{u_0 v_1 + v_0 u_1} \cdot \frac{1 - u_0 u_1}{u_0 - u_1}, \frac{u_0 v_1 - v_0 u_1}{u_0 v_1 + v_0 u_1} \cdot \frac{v_0(u_1^2 - 1) - v_1(u_0^2 - 1)}{(u_0 - u_1)^2} \right). \end{aligned} \quad (4)$$

Let us assume that the point $R_2 = (u_2, v_2)$, such that $R_2 = R_0 - R_1$, is known. Then, the addition formulas (4) can be rewritten as the following differential addition formulas,

$$(u_3, v_3) = \left(\frac{1}{u_2} \cdot \frac{(1 - u_0 u_1)^2}{(u_0 - u_1)^2}, \frac{1}{v_2} \cdot \frac{v_0^2(1 - u_1^2)^2 - v_1^2(1 - u_0^2)^2}{(u_0 - u_1)^4} \right) \quad (5)$$

One can perform u -only arithmetic by transforming the above equation to customary projective coordinates as,

$$\begin{aligned} (U_3 : Z_3) &= (Z_2(U_0 U_1 - Z_0 Z_1)^2 : U_2(U_0 Z_1 - Z_0 U_1)^2) \\ &= (Z_2((U_1 + Z_1) + \mu(U_1 - Z_1))^2 : U_2((U_1 + Z_1) - \mu(U_1 - Z_1))^2) \end{aligned} \quad (6)$$

where $\mu = \frac{(U_0 + Z_0)}{(U_0 - Z_0)}$.

The per-point- R_0 constant value μ can be precomputed and stored since it only depends on $(U_0 : Z_0)$. Computing $(U_3 : Z_3)$ in (6) takes only $3\mathbf{m} + 2\mathbf{s} + 4\mathbf{a}$, by reusing $(U_1 + Z_1)$ and $\mu(U_1 - Z_1)$ on both sides. Notice that this exactly matches the computational cost of Eq. (2), which computes the differential addition of the classical Montgomery ladder. In https://github.com/thomazoliveira/rfc7748_verification, a Magma [8] script verifying Eq.(6) is available.

⁸ Notice that in general an Montgomery elliptic curve has the form, $Bv^2 = u^3 + Au^2 + u$.

The corresponding differential addition formulas for Edwards and Huff forms are described in Appendices A and B, respectively. The formulas give exactly the same performance as the formulas presented for the Montgomery form. Therefore, they are not included in the main body of the text.

5.3 Differential addition formulas in Algorithm 4

In the context of Algorithm 4, the differential addition formula required in Steps 6 and 8 can be computed as,

$$\begin{aligned} U_{R_3} &\leftarrow Z_{R_2}((U_{R_1} + Z_{R_1}) + \mu(U_{R_1} - Z_{R_1}))^2 \\ Z_{R_3} &\leftarrow U_{R_2}((U_{R_1} + Z_{R_1}) - \mu(U_{R_1} - Z_{R_1}))^2, \end{aligned} \quad (7)$$

where $\mu = \frac{u_{R_0} + 1}{u_{R_0} - 1}$.

Once again, notice that the μ -values can be pre-computed and stored since they only depend on the u -coordinates of the points $2^i P$.

Timing attacks Notice that no side-channel countermeasures are required to retrieve the values $\mu_i = \frac{u_{2^i P} + 1}{u_{2^i P} - 1}$ from memory, since they are public and do not have any direct correlation to the sensitive information contained in the scalar k . Also, the addition performed in Step 8 is not a dummy operation. The correct value of the R_2 coordinates must be maintained in order to perform further additions in Step 6. Moreover, since $k_{n-1} = 1$, a computational fault induced at any iteration of Algorithm 4 would produce a wrong resulting point Q .

5.4 Implementing the pre-computable ladder

Before presenting a low-level description of the known-point scalar multiplication using Algorithm 4, we must examine the point S selection and how to optimize the processing of the scalar k .

Strategies When selecting the private key k (Alg. 1, Step 1), presumably to facilitate the programming effort, the X25519 specification [26] recommends to generate 32 bytes at random as $k = K_0 + K_1 2^8 + \dots + K_{31} 2^{248}$ with byte-words $K_i \xleftarrow{\$} [0, 255]$, and to perform the following operations:

$$K_0 \leftarrow K_0 \wedge 248, \quad K_{31} \leftarrow K_{31} \wedge 127, \quad K_{31} \leftarrow K_{31} \vee 64,$$

where the symbols \wedge and \vee represent the logical conjunction and disjunction operators. For the X448 function, 56 randomly-chosen bytes are required, which are further processed as

$$K_0 \leftarrow K_0 \wedge 252, \quad K_{55} \leftarrow K_{55} \vee 128.$$

Algorithm Next, in Algorithm 5, we present the low-level details of our approach. Again, the term n represents the bit length of $\#E_A(\mathbb{F}_p) = h \cdot r$ and $q = \log_2 h$.⁹ The pre-computation phase (Step 1) consists of computing and storing the values $\mu_i = \frac{u_{P_i} + 1}{u_{P_i} - 1}$ for the multiples $P_i = 2^i P$. These $n - q$ field elements are computed a priori from the base-point P . Assuming that the architecture is byte-addressable, the memory space required for Curve25519 is approximately $(255 - 3) \cdot 32B \approx 8KB$, while in the Curve448 setting, we need $(448 - 2) \cdot 56B \approx 25KB$.

Algorithm 5 Low-level right-to-left Montgomery ladder

Input: $P = (u_P, v_P), S = (u_S, v_S), P - S = (u_{P-S}, v_{P-S}) \in E_A/\mathbb{F}_p, a_{24} = (A + 2)/4$
 $k = (k_{n-1} = 1, k_{n-2}, \dots, k_1, k_0)_2$

Output: $u_{Q=kP}$

```

1: Pre-computation Let  $P_i = 2^i P$ . Compute and store the values  $\mu_i = \frac{u_{P_i} + 1}{u_{P_i} - 1}$ , for
    $0 \leq i \leq n - q - 1$ 
2: Initialization:  $U_{R_1} \leftarrow u_S, Z_{R_1} \leftarrow 1, U_{R_2} \leftarrow u_{P-S}, Z_{R_2} \leftarrow 1, s \leftarrow 1$ 
3: for  $i \leftarrow 0$  to  $n - q - 1$  do
4:   # timing-attack countermeasure
5:    $s \leftarrow s \oplus k_{i+q}$ 
6:    $U_{R_1}, U_{R_2} \leftarrow \text{cswap}(s, U_{R_1}, U_{R_2})$ 
7:    $Z_{R_1}, Z_{R_2} \leftarrow \text{cswap}(s, Z_{R_1}, Z_{R_2})$ 
8:    $s \leftarrow k_{i+q}$ 
9:   # addition
10:   $A \leftarrow U_{R_1} + Z_{R_1}; B \leftarrow U_{R_1} - Z_{R_1}$ 
11:   $C \leftarrow \mu_i \times B$ 
12:   $D \leftarrow A + C; D \leftarrow D^2$ 
13:   $E \leftarrow A - C; E \leftarrow E^2$ 
14:   $U_{R_1} \leftarrow Z_{R_2} \times D; Z_{R_1} \leftarrow U_{R_2} \times E$ 
15: end for
16: for  $i \leftarrow 0$  to  $q - 1$  do
17:   # doubling
18:    $A \leftarrow U_{R_1} + Z_{R_1}; A \leftarrow A^2$ 
19:    $B \leftarrow U_{R_1} - Z_{R_1}; B \leftarrow B^2$ 
20:    $U_{R_1} \leftarrow A \times B$ 
21:    $A \leftarrow A - B$ 
22:    $Z_{R_1} \leftarrow a_{24} \times A; Z_{R_1} \leftarrow Z_{R_1} + B; Z_{R_1} \leftarrow Z_{R_1} \times A$ 
23: end for
24:  $Z_{R_1} \leftarrow Z_{R_1}^{-1}$ 
25:  $u_{R_1} \leftarrow U_{R_1} \times Z_{R_1}$ 
26: return  $u_Q \leftarrow u_{R_1}$ 

```

⁹ For the sake of simplicity, in the remaining of this paper it will be assumed that h is a small power of two.

The conditional swap function is identical to the one used in Algorithm 3. However, in this case the inputs are the coordinates of the accumulator R_1 and the difference point R_2 . Moreover, the s variable that controls the swap is set to one, since the Montgomery point additions, in terms of memory location, are always performed as $R_1 \leftarrow R_1 + 2^i P$ throughout the algorithm. Also, given that the most significant bit k_{n-1} is always equal to one, it is unnecessary to include another couple of cswap functions after the main loop. At the end of the algorithm (Steps 16-23), we must perform q consecutive point doublings to process the least significant bits of k and to eliminate the point S from the accumulator R_1 .

Cost estimations The cost of the Algorithm 5 can be estimated as $(n - q) \cdot (3\mathbf{m} + 2\mathbf{s} + 4\mathbf{a}) + q \cdot (2\mathbf{m} + 1\mathbf{m}_{\mathbf{a}24} + 2\mathbf{s} + 4\mathbf{a}) + 1\mathbf{m} + 1\mathbf{i}$. If the Curve25519 is used, then $n = 255$ and $q = 3$. As a result, the fixed-point scalar multiplication would cost

$$763\mathbf{m} + 3\mathbf{m}_{\mathbf{a}24} + 510\mathbf{s} + 1020\mathbf{a} + 1\mathbf{i},$$

where the arithmetic operations are over $\mathbb{F}_{2^{255}-19}$. In the Curve448 context, $n = 448$ and $q = 2$. As a consequence, we have the following cost in terms of $\mathbb{F}_{2^{448}-2^{224}-1}$ -operations:

$$1343\mathbf{m} + 2\mathbf{m}_{\mathbf{a}24} + 896\mathbf{s} + 1792\mathbf{a} + 1\mathbf{i}.$$

These results show that, our approach saves more than 25% of general field multiplications. In addition, it completely eliminates the multiplication by u_P ¹⁰ and drastically reduces the number of multiplications by the constant $(A+2)/4$. In addition, it saves half of the field squarings and half of additions/subtractions.

For the programmer, the only extra effort is to organize the pre-computed values in the memory and load them during the main loop execution, since the remaining field and logic operations are very similar to ones presented in Algorithm 3. In the next subsection, we present a comparative based on the arithmetic of state-of-the-art software implementations.

5.5 Comparison

In this part, we present a more concrete analysis of the performance efficiency of Algorithm 5. For this purpose, we measured the field arithmetic cost of different state-of-the-art constant-time software implementations of the Diffie-Hellman protocol on Curve25519 and Curve448. After that, we computed the ratios of $\mathbf{m}_{\mathbf{a}24}$, $\mathbf{m}_{\mathbf{u}P}$, \mathbf{s} and \mathbf{i} to \mathbf{m} , which are considered the most representative field arithmetic operations for scalar multiplication implementations. As a result, we were able to show the practical savings of our proposal in terms of general field multiplications \mathbf{m} .

¹⁰ In fact, given that the difference of the point operands $P_i - R_1$ is variable, the $\mathbf{m}_{\mathbf{u}P}$ operations were changed into two general multiplications and were included in the \mathbf{m} operation count.

Regarding the X25519 implementations, we selected the code from Bernstein et al. [2], which represents the $\mathbb{F}_{2^{255-19}}$ elements in radix-2⁵¹, the AVX2 approach from Faz-Hernández and López [15] and the curve25519-donna library from Langley [25].¹¹ For the X448 function, we considered the original implementation of Hamburg in [18]. The source code of [2,18] were downloaded from the eBACS [3] web page, the [15] implementation was shared by its authors via personal communication and the curve25519-donna library was retrieved from its GitHub repository [25].

Every field arithmetic code was compiled with the clang/LLVM compiler version 3.9 with optimization flags `-O3 -march=haswell -fomit-frame-pointer` and further benchmarked in an Intel Core i7-4700MQ 2.40GHz machine (Haswell architecture) with the Hyper Threading and Turbo Boost technologies disabled. The ratios are presented in Table 1.

Table 1. Ratios of selected arithmetic operations to the general field multiplication in state-of-the-art software implementations

Implementation	Ratios to m				
	m_{a24}	m_{uP}	s	i	a
Bernstein et al. [2]	0.23 [†]	0.23 [†]	0.76	203.29	< 0.1
Faz-Hernández and López [15]	0.28	0.41	0.96	84.33	< 0.1
Langley [25]	0.60	1.00 [‡]	0.82	192.55	< 0.1
Hamburg [18]	0.24	1.00 [‡]	0.75	405.00	< 0.1

[†] Estimated

[‡] The general field multiplication (**m**) is used to implement this operation

The cost of the **m_{a24}** operation in the Bernstein et al. implementation was estimated as follows. After analyzing the assembly code, we concluded that **m_{a24}** takes 10 `movq`, 5 `mov`, 5 `shr`, 5 `add`, 4 `addq`, 5 `mulq` and 1 `imulq` machine instructions. Next, we added its latencies [16] and, to calculate a “lower bound” of our speed improvements, we applied an aggressive throughput of 0.25. Finally, given that the **m_{uP}** is similar to the **m_{a24}** operation, we also assumed a similar cost. In Table 2, we present the performance improvements of our proposal in terms of the general field multiplication.

The above comparison suggests that about 36.01 to 44.04% of speed-up can be reached in the first phase of the ECDH protocol by using Algorithm 5. When considering the complete Diffie-Hellman scheme, the improvement ranges from 18.01 to 22.02%. In practice, these estimated savings can be further improved if we take into consideration compiler optimizations and the machine throughput.

¹¹ The benchmarking reports in [3] shows that the library of Chou [10] currently holds the speed record on computing the scalar multiplication over Curve25519. However, the author decided to embed the field arithmetic functions into the ladder step, in a single assembly code. Isolating the field operations would be impractical and could alter the author’s original intentions.

Table 2. A comparative between Montgomery-ladder approaches in the fixed-point scenario

Implementation	Estimated costs [†]		Diff.
	Mont. ladder left-to-right (Alg. 3)	Mont. ladder right-to-left (Alg. 5)	
Bernstein et al. [2]	2116.89m	1354.58m	-36.01%
Faz-Hernández and López [15]	2260.48m	1337.77m	-40.82%
Langley [25]	2457.95m	1375.55m	-44.04%
Hamburg [18]	4097.52m	2420.48m	-40.93%

[†] Because of its negligible cost, the field addition/subtraction operation was not included

Moreover, while the field addition/subtraction cost is imperceptible if measured separately, it constitutes a significant part in the whole protocol execution timings.

6 Software Implementation on a 64-bit Architecture

In this section, an optimized software implementation of X25519 and X448 targeting 64-bit Intel architectures is presented. Our implementation was developed to take advantage of new instructions, available in Haswell and Skylake micro-architectures, intended to accelerate the calculation of multi-precision integer arithmetic [35]. In this sense, the calculation of multiplications is the most critical operation, and for this reason, we devote a detailed explanation.

Aiming a large usability of the library across different 64-bit platforms, we restrict arithmetic operations to be computed using the 64-bit instruction set; for this reason, we use a radix-2⁶⁴ for representing prime field elements. Thus, for $w = 64$, an element in $\mathbb{F}_{2^{255}-19}$ is stored in $n = 4$ words of 64 bits, whereas an element in $\mathbb{F}_{2^{448}-2^{224}-1}$ requires $n = 7$ words of 64 bits. This representation of elements is compact and does not incur on a large memory footprint for storing the look-up table.

The calculation of prime field multiplications is performed into two steps: integer multiplication followed by modular reduction. Concerning the integer multiplication various methods can be applied targeting different optimization metrics [20,19]. For both fields, we developed the operand scanning technique since its execution pattern benefits from the properties of the MULX instruction, which is part of the BMI2 instruction set.

Like the legacy MUL/IMUL instructions, the MULX instruction also computes a 64-bit integer multiplication (the RDX register times a specified source register) producing a 128-bit product. However, MULX has a three-operand codification to specify the destination registers of the product; this differs from the MUL/IMUL instructions since the product is always deposited in the RAX and RDX registers, which in turn overwrites the RDX register. The fact that RDX is not modified by MULX is crucial for the efficient execution of consecutive multiplications by one common operand, like in the case of the operand scanning technique.

Algorithm 6 Operand scanning method to calculate prime field multiplications.

Output: (x_0, \dots, x_{n-1}) and (y_0, \dots, y_{n-1}) be the radix- 2^w representation of $x, y \in \mathbb{F}_p$.

Input: (z_0, \dots, z_{n-1}) be the radix- 2^w representation of $z = xy \in \mathbb{F}_p$.

```

1:  $c \leftarrow 0$ 
2:  $(H_0 \parallel z_0) \leftarrow x_0 y_0$ 
3: for  $j \leftarrow 1$  to  $n - 1$  do
4:    $(H_j \parallel L) \leftarrow x_0 y_j$ 
5:    $(c \parallel z_j) \leftarrow L + H_{j-1} + c \{x_0 y = (z_0, \dots, z_n)\}$ 
6: end for
7:  $z_n \leftarrow H_{n-1} + c$ 
8: for  $i \leftarrow 1$  to  $n - 1$  do
9:    $c \leftarrow 0$ 
10:   $(H_0 \parallel L_0) \leftarrow x_i y_0$ 
11:   $(d \parallel z_i) \leftarrow z_i + L_0$ 
12:  for  $j \leftarrow 1$  to  $n - 1$  do
13:     $(H_j \parallel L) \leftarrow x_i y_j$ 
14:     $(c \parallel H_{j-1}) \leftarrow L + H_{j-1} + c \{x_i y = (L_0, H_0, \dots, H_{n-1})\}$ 
15:     $(d \parallel z_{i+j}) \leftarrow z_{i+j} + H_{j-1} + d \{z \leftarrow z + 2^{iw} x_i y\}$ 
16:  end for
17:   $H_{n-1} \leftarrow H_{n-1} + c$ 
18:   $z_{i+n} \leftarrow H_{n-1} + d$ 
19: end for return  $(z_0, \dots, z_{n-1}) \leftarrow (z_0 \dots, z_{2n-1}) \bmod p$ 

```

The operand scanning technique calculates the multi-precision integer multiplication $z = xy$ by first calculating $z \leftarrow x_0 y$; followed by the accumulation $z \leftarrow z + 2^{iw} x_i y$ for $0 < i < n$. The schedule of operations is listed in Algorithm 6. Notice that in the steps that compute the $x_i y_j$ product (lines 4 and 13), the y_j operand changes more frequently than the x_i operand; thus once x_i is loaded into the RDX register, it remains there for all the iterations of the j -loop saving $n - 1$ memory accesses. Additionally, this pattern allows scheduling various MULX multiplications to the processor, which can execute them faster by means of the processor's pipeline. These subtle details make that the operand scanning technique be suitable for its implementation using MULX instructions.

The word multiplications are independent to each other, and consequently, no data dependencies occur at all. However, the accumulation of these products is an inherently sequential process. In Algorithm 6, two accumulation steps are identified: first, once the product $(H_j \parallel L) \leftarrow x_i y_j$ was calculated, L must be accumulated into H_{j-1} (lines 5 and 14 of Algorithm 6); and then, the H_{j-1} word is ready to be accumulated into the output z_{i+j} word (line 15 of Algorithm 6). The most relevant fact of these accumulations is that each one produces its own carry bit (the c and d bit-variables); which must be handled using addition with carry instructions.

The ADC/ADD instructions calculate additions with/without carry modifying the FLAGS register according to the result of the addition. Since there is only one carry bit flag, we must compute one accumulation entirely, and after that, we can perform the second one. This is the strategy followed when our software

library is compiled targeting the Haswell micro-architecture; nonetheless, this is not the case of the Skylake micro-architecture.

Unlike the `ADD/ADC` instructions, the new `ADCX` and `ADOX` instructions calculate additions with carry modifying only the `CF` and the `OF` bit, respectively, of the `FLAGS` register. This allows that two sequences of addition instructions depending on carry bits can be computed in parallel by the executing units. Thus, the core of the integer multiplication performs the lines 13, 14, and 15 using respectively one `MULX`, one `ADCX`, and one `ADOX` instruction; it is worth to mention none of these instructions competes to each other for accessing to the same part of the `FLAGS` register. Relying in these features, we developed optimized code for integer multiplication targeting processors supporting the `ADX` instruction set.

The integer multiplication produces a sequence $(z_0 \dots, z_{2n-1})$, which is reduced modulo p . For $p = 2^{255} - 19$, the words $z_i \leftarrow z_i + 38z_{i+4}$ for $0 \leq i < 4$ are updated using four multiplications, then we reduce again $z_0 \leftarrow z_0 + 38z_4$ letting the result in four words of 64 bits. For $p = 2^{448} - 2^{224} - 1$, we perform the modular reduction into three steps:

$$\begin{aligned} z &\leftarrow (z \bmod 2^{672}) + (2^{448} + 2^{224})\lfloor z/2^{672} \rfloor \\ z &\leftarrow (z \bmod 2^{448}) + (2^{224} + 1)\lfloor z/2^{448} \rfloor \\ z &\leftarrow (z \bmod 2^{448}) + (2^{224} + 1)\lfloor z/2^{448} \rfloor \end{aligned}$$

The first two lines take a 224-bit value and add it to z in two different positions, one of them requires a 32-bit shift that we compute using `SHLD` instructions. The last line reduces only the z_7 word. This modular reduction does not require multiplications.

7 Performance Benchmark

The performance timings were measured in two platforms: a Core i7-4770 processor (Haswell micro-architecture) and a Core i7-6700K processor (Skylake micro-architecture). Source code was compiled using the GNU C Compiler (GCC v.6.3.1) and is available at https://github.com/armfazh/rfc7748_precomputed.

Table 3. Prime field arithmetic timings measured in clock cycles

Prime field (\mathbb{F}_p)	Architecture	Arithmetic Operation				
		Add	Mul	Sqr	m_{a24}	Inv
$p = 2^{255} - 19$	Haswell	8	63	54	14	15,032
	Skylake	6	49	41	11	11,441
$p = 2^{448} - 2^{224} - 1$	Haswell	14	161	117	24	54,709
	Skylake	13	122	95	20	45,008

Table 4. Elliptic curve Diffie-Hellman timings measured in clock cycles.

Function	Architecture	DH Operation	
		Key Generation	Shared Secret
X25519	Haswell	90,668	138,963
	Skylake	72,471	107,831
X448	Haswell	401,228	670,754
	Skylake	320,695	527,899

8 Conclusion

In this work, we presented an alternative way to compute the elliptic curve Diffie-Hellman protocol with Montgomery ladders. Particularly, we focused on the key-generation phase, which can be characterized as a fixed-point scenario. For this phase, we assumed that the relevant multiples of the base-point can be pre-computed off-line, which helps to boost the computation of the scalar multiplication via a right-to-left variant of the Montgomery ladder. As a result we achieved, in the Curve25519 setting, performance improvements that range from 36 to 44% of speed-up for the key generation operation, at the price of just $8KB$ of memory space. Our proposal carefully minimizes coding modifications with respect to the specifications given in the RFC 7748 memorandum. Our software implementation of the X25519 and X448 functions using our pre-computable ladder yields an acceleration factor of roughly 1.20, and 1.25 when implemented on the Haswell and the Skylake micro-architectures, respectively.

We also would like to explore the potential savings that our ladder approach can bring for digital signature protocols and other elliptic-curve based protocols. Finally, building on the work of [33], we would like to explore a Montgomery ladder variant, which can be applied to prime elliptic curves equipped with efficient endomorphisms such as the FourQ elliptic curve [11]. For that kind of elliptic curves, the ladder variant presented in [33], allows for an important saving in the number of required point doubling operations when working in the fixed-point scenario.

References

1. D. J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Proceedings of PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.
2. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.
3. D. J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. <https://bench.cr.yp.to>. Accessed in Mar 2017.
4. D. J. Bernstein and T. Lange. Explicit-formulas database. <https://hyperelliptic.org/EFD/g1p/auto-edwards-yzsquared.html>. Accessed in May 2017.
5. D. J. Bernstein and T. Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. <http://safecurves.cr.yp.to>. Accessed in Mar 2017.

6. D. J. Bernstein and T. Lange. A complete set of addition laws for incomplete Edwards curves. *Journal of Number Theory*, 131(5):858–872, 2011.
7. D. J. Bernstein and T. Lange. Montgomery curves and the montgomery ladder. Cryptology ePrint Archive, Report 2017/293, 2017. <http://eprint.iacr.org/2017/293>.
8. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).
9. Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters, 2010. Version 2.0. Standards for Efficient Cryptography. <http://www.secg.org/sec2-v2.pdf>.
10. T. Chou. Sandy2x: New Curve25519 Speed Records. In *Proceedings of SAC 2015*, pages 145–160. Springer International Publishing, 2016.
11. C. Costello and P. Longa. Four \mathbb{Q} : Four-Dimensional Decompositions on a \mathbb{Q} -curve over the Mersenne Prime. In *Proceedings of ASIACRYPT 2015*, volume 9452 of *LNCS*, pages 214–235. Springer, 2015.
12. C. Costello and B. Smith. Montgomery curves and their arithmetic: The case of large characteristic fields. Cryptology ePrint Archive, Report 2017/212, 2017.
13. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
14. ECC Brainpool. Standard Curves and Curve Generation, 2005. Version 1.0. <http://www.ecc-brainpool.org/download/Domain-parameters.pdf>.
15. A. Faz-Hernández and J. López. Fast Implementation of Curve25519 Using AVX2. In *Proceedings of LATINCRYPT 2015*, pages 329–345. Springer International Publishing, 2015.
16. A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. <http://www.agner.org/optimize/>, 2016.
17. P. Gaudry and D. Lubicz. The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009.
18. M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. <http://eprint.iacr.org/2015/625>.
19. M. Hutter and P. Schwabe. Multiprecision multiplication on avr revisited. *Journal of Cryptographic Engineering*, 5(3):201–214, 2015.
20. M. Hutter and E. Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Proceedings of CHES 2011*, pages 459–474. Springer, 2011.
21. M. Joye. Highly regular right-to-left algorithms for scalar multiplication. In P. Pailier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2007.
22. N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of computation*, 48:203–209, 1987.
23. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of CRYPTO 99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
24. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. In *Proceedings of CRYPTO 96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
25. A. Langley. curve25519-donna. <https://github.com/ag1/curve25519-donna>. Accessed in Mar 2017.

26. A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security, 2016. Request for Comments. <https://tools.ietf.org/html/rfc7748>.
27. V. S. Miller. Use of elliptic curves in cryptography. In *Proceedings of CRYPTO 85*, volume 218 of *LNCS*, pages 417–426. Springer, 1986.
28. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48:243–264, 1987.
29. National Institute of Standards and Technology. FIPS PUB 186-4: Digital Signature Standard (DSS). Federal Information Processing Standards, 2013. <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.
30. National Institute of Standards and Technology. NIST Removes Cryptography Algorithm from Random Number Generator Recommendations, 2014. <https://www.nist.gov/news-events/news/2014/04/nist-removes-cryptography-algorithm-random-number-generator-recommendations>.
31. National Institute of Standards and Technology. Special Publication 800-90A Rev.1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators, 2015. <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf>.
32. T. Oliveira, D. F. Aranha, J. López, and F. Rodríguez-Henríquez. Fast Point Multiplication Algorithms for Binary Elliptic Curves with and without Precomputation. In *Proceedings of SAC 2014*, volume 8781 of *LNCS*, pages 324–344. Springer, 2014.
33. T. Oliveira, J. López, and F. Rodríguez-Henríquez. The Montgomery ladder on binary elliptic curves. Submitted to *Journal of Cryptographic Engineering*, 2017.
34. N. G. Orhon and H. Hisil. Speeding up huff form of elliptic curves. *Cryptology ePrint Archive*, Report 2017/320, 2017. <http://eprint.iacr.org/2017/320>.
35. E. Ozturk, J. Guilford, V. Gopal, and W. Feghali. New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors. Intel Corporation, White Paper 327831-001, Aug. 2012. <http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf>.
36. K. Patterson. Formal request from TLS WG to CFRG for new elliptic curves. *Crypto Forum Research Group archives*, 2015. https://mailarchive.ietf.org/arch/msg/cfrg/Hvihr_yQhVB.Qdl-mtwTdVbHGiu.
37. N. Perlroth. Government Announces Steps to Restore Confidence on Encryption Standards. *New York Times*, 2013. <https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/>.
38. N. Perlroth, J. Larson, and S. Shane. N.S.A. Able to Foil Basic Safeguards of Privacy on Web. *New York Times*, 2013. <http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html>.
39. E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, 2017. Internet-Draft. <https://tools.ietf.org/html/draft-ietf-tls-tls13-19>.

A Twisted Edwards differential addition with pre-computation

Bernstein and Lange introduced efficient differential addition formulas for Edwards curves in EFD [4] with links to the arithmetic of Kummer lines [17], and with the following note: “The following formulas are the outcome of a discussion

in 2009 among Daniel J. Bernstein, David Kohel, and Tanja Lange. The core ideas were published by Pierrick Gaudry in 2006.”

This appendix adapts the same formulas to the present setup, with the same assumptions that $r^2 = d$ and $ry^2 = Y/T$. The letter ‘ T ’ is used for $(Y_i : T_i)$ in order to produce a simple verification script that uses the embedding of E in $\mathbb{P}^1 \times \mathbb{P}^1$, see [6]. The numbering in the subscripts are also modified to match the formatting of this paper. The formulas then read:

$$(Y_3 : T_3) = \left(T_2((T_1 - Y_1) - \epsilon(T_1 + Y_1))^2 : Y_2((T_1 - Y_1) + \epsilon(T_1 + Y_1))^2 \right) \quad (8)$$

$$\text{where } \epsilon = \frac{(1-r)(T_0 + Y_0)}{(1+r)(T_0 - Y_0)}.$$

Computing $(Y_3 : T_3)$ in (8) takes only $3\mathbf{M}+2\mathbf{S}+4\mathbf{a}$, assuming that ϵ is pre-computed. In https://github.com/thomazoliveira/rfc7748_verification we presented a Magma script that verifies the above formulas.

B Extended Huff differential addition with pre-computation

The affine addition formulas for the extended Huff curve $H : y(1 + ax^2) = cx(1 + dy^2)$ are given in [34] as

$$\begin{aligned} (x_3, y_3) &= (x_0, y_0) + (x_1, y_1) \\ &= \left(\frac{(x_0 + x_1)(1 - dy_0y_1)}{(1 - ax_0x_1)(1 + dy_0y_1)}, \frac{(1 - ax_0x_1)(y_0 + y_1)}{(1 + ax_0x_1)(1 - dy_0y_1)} \right) \end{aligned} \quad (9)$$

Assuming that we have $(x_2, y_2) = (x_0, y_0) - (x_1, y_1)$, the affine addition formulas (9) can be rewritten as the following differential addition formulas

$$(x_3, y_3) = \left(\frac{1}{x_2} \cdot \frac{x_0^2 - x_1^2}{1 - a^2x_0^2x_1^2}, \frac{1}{y_2} \cdot \frac{y_0^2 - y_1^2}{1 - d^2y_0^2y_1^2} \right) \quad (10)$$

As in [34], we use

$$\mathcal{H} = \{((X : Z), (Y : T)) \in \mathbb{P}^1 \times \mathbb{P}^1 : YT(Z^2 + aX^2) = cXZ(T^2 + dY^2)\},$$

the embedding of H into $\mathbb{P}^1 \times \mathbb{P}^1$, for efficiency purposes. The affine differential addition formulas in (10) translates to the projective differential addition formulas (11), as follows:

$$\begin{aligned}
((X_3 : Z_3), (Y_3 : T_3)) &= ((X_0 : Z_0), (Y_0 : T_0)) + ((X_1 : Z_1), (Y_1 : T_1)) \\
&= \left((Z_2(X_0^2 Z_1^2 - Z_0^2 X_1^2) : X_2(Z_0^2 Z_1^2 - a^2 X_0^2 X_1^2)), \right. \\
&\quad \left. (T_2(Y_0^2 T_1^2 - T_0^2 Y_1^2) : Y_2(T_0^2 T_1^2 - d^2 Y_0^2 Y_1^2)) \right) \\
&= \left((Z_2(Z_1^2 - \kappa X_1^2) : X_2(\kappa Z_1^2 - a^2 X_1^2)), \right. \\
&\quad \left. (T_2(T_1^2 - \lambda Y_1^2) : Y_2(\lambda T_1^2 - d^2 Y_1^2)) \right) \tag{11}
\end{aligned}$$

where $\kappa = Z_0^2/X_0^2$, $\lambda = T_0^2/Y_0^2$, $((X_2 : Z_2), (Y_2 : T_2)) = ((X_0 : Z_0), (Y_0 : T_0)) - ((X_1 : Z_1), (Y_1 : T_1))$, $X_0 \neq 0$, and $Y_0 \neq 0$. Computing $(X_3 : Z_3)$ in (11) takes only $4\mathbf{M}+2\mathbf{S}+1\mathbf{D}+2\mathbf{a}$, assuming that κ is precomputed. Analogous comments apply independently to $(Y_3 : T_3)$, assuming that λ is precomputed.

A better operation count can be achieved if $a = \pm 1$:

$$\begin{aligned}
(X_3 : Z_3) &= \left((Z_2(X_0^2 Z_1^2 - Z_0^2 X_1^2) : X_2(Z_0^2 Z_1^2 - X_0^2 X_1^2)) \right) \\
&= \left(Z_2((X_1^2 - Z_1^2) - \kappa'(X_1^2 + Z_1^2)) : X_2((X_1^2 - Z_1^2) + \kappa'(X_1^2 + Z_1^2)) \right) \tag{12}
\end{aligned}$$

where $\kappa' = \frac{(X_0^2 - Z_0^2)}{(X_0^2 + Z_0^2)}$.

Computing $(X_3 : Z_3)$ now takes only $3\mathbf{M}+2\mathbf{S}+4\mathbf{a}$, assuming that κ' is precomputed. Analogous comments apply independently to $(Y_3 : T_3)$, assuming that $d = \pm 1$ and $\lambda' = (Y_0^2 - T_0^2)/(Y_0^2 + T_0^2)$ is precomputed.

Again, Magma scripts verifying the proposed differential addition formulas are available in https://github.com/thomazoliveira/rfc7748_verification.