Efficient Constant Time Conditional Branching in the Montgomery Ladder

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

The Montgomery ladder has a conditional statement. Existing constant time implementations of the Montgomery ladder are based on constant time conditional swaps or conditional selection of field elements. Implementations of the underlying field arithmetic require a multi-limb representation of the field elements. So, a swap or a selection of two field elements require a number of data movement operations which is proportional to the number of limbs. In this work, we introduce a new method for constant time implementation of the conditional statement. Our method does not require any swap or selection of field elements. Further, the number of involved data movement operations in our method is independent of the size of the underlying field. This leads to substantial savings in the number of data movement operations required for Montgomery ladder computation. We have implemented the new idea using 64-bit arithmetic for Curve25519 and Curve448, two elliptic curves which have been proposed in the Transport Layer Security, Version 1.3. Timing measurements on the Skylake and the Kaby Lake processors of Intel show that for Curve25519 about 11% and for Curve448 about 13% speed-ups are achieved.

Keywords: Montgomery ladder, Diffie-Hellman protocol, constant time implementation, elliptic curve cryptography, Curve25519, Curve448.

1 Introduction

Diffie-Hellman (DH) [6] key agreement is one of the fundamental primitives of modern cryptography. The currently most efficient implementation of this primitive is done over groups arising from elliptic curves [7, 8]. Several models of elliptic curves are used in cryptography. The Montgomery form [9] elliptic curve is the most efficient for implementing the shared secret computation phase of DH key agreement. A concrete Montgomery form curve, called Curve25519, has been proposed [2] to provide security at the 128-bit security level. Since its proposal, Curve25519 has gained wide acceptance and is used in many important applications. Details can be found at [1].

The Transport Layer Security (TLS) protocol, Version 1.3 [13] specifies elliptic curve cryptography for DH shared secret computation targeted at the 128-bit and the 224-bit security levels. For the 128-bit security level, Curve25519 is specified. For the 224-bit security level, a Montgomery form curve called Curve448 is specified. In view of the importance of the TLS protocol and also the widespread adoption of Curve25519, efficient implementation of the shared secret computation phase of the DH key agreement scheme has major implications to practical deployment.

Suppose p is a prime and $\mathbb{F}_p$ be the finite field of p elements. A Montgomery form elliptic curve $M_{A,B}$ is specified by two parameters $A \in \mathbb{F}_p \setminus \{2, -2\}$ and $B \in \mathbb{F}_p \setminus \{0\}$, and is given by an equation $M_{A,B} : By^2 = x^3 + Ax^2 + x$. For $i \geq 1$, the $\mathbb{F}_{p^i}$-rational points of $M_{A,B}$ are points $(x, y) \in \mathbb{F}^2_{p^i}$ satisfying the equation of the curve. Following [13], we consider the case where p is a large prime and cryptography is done over a prime order subgroup G of the $\mathbb{F}_p$-rational points of $M_{A,B}$.

The DH shared secret computation on $M_{A,B}$ requires performing the following computation. Let P be a point in G and n be a secret scalar. Suppose the x-coordinate of P is x_P. Given x_P and n, it is required to compute the x-coordinate of the point nP. Montgomery [9] introduced a particularly efficient way of performing this computation which has since then come to be known as the Montgomery ladder. The basic structure of the Montgomery ladder and a single ladder step are shown in Algorithms 1 and 2.

A requirement for secure implementation of any cryptographic primitive is that the run time should not depend on any secret value. Note that the Montgomery ladder shown in Algorithm 1 has a conditional instruction where the condition is based on a secret bit. So, a straightforward implementation of the ladder algorithm will not be constant time and has the potential to leak the secret bit. This problem has been addressed in the literature and several constant time implementations are known.

The basic computations in the Montgomery ladder are on elements of $\mathbb{F}_p$. Typically, multi-precision arithmetic would be used to implement operations in $\mathbb{F}_p$. So, an element x of $\mathbb{F}_p$ will be represented by several words which are also called the limbs of x. For concreteness, consider the case of Curve25519 which is defined over $\mathbb{F}_{p_1}$ with $p_1 = 2^{255} - 19$. Using 64-bit arithmetic, an element of $\mathbb{F}_{p_1}$ can have a 4-limb representation. Similarly, Curve448 is defined over $\mathbb{F}_{p_2}$, with $p_2 = 2^{448} - 2^{224} - 1$; using 64-bit arithmetic, an element of $\mathbb{F}_{p_2}$ can have a 7-limb representation.

Going back to constant time conditional swap, we note that a swap of two field elements will require swapping all the limbs storing the two field elements. So, a swap of two field elements will require a number of 64-bit data movement operations which is proportional to the number of limbs. The exact number of data movement operations will depend on the actual implementation, but, since all the limbs will have to be swapped, this number must necessarily be linear in the number of limbs. Consequently, it follows that the number of data movement operations to implement a swap of field element increases as the number of limbs increases. For example, the number of 64-bit data movement operations to implement a swap over $\mathbb{F}_{p_2}$, will be more than the number of 64-bit data movement operations to implement a swap over $\mathbb{F}_{p_1}$. The conditional statement is part of the main loop of the Montgomery ladder. So, a substantial number of 64-bit data movement operations are executed to implement the swaps of field elements. This consumes a significant portion of the total time required for the entire ladder computation.

Our Contributions

We describe a new way of implementing the conditional statement in the Montgomery ladder in constant time. Our method does not require swapping or selection between field elements. Further, the number of 64-bit move instructions is independent of the size of the underlying field, i.e., it remains the same irrespective of the number of limbs used to represent an element of $\mathbb{F}_p$. This leads to substantial savings in the number of 64-bit move instructions that need to be executed to perform the ladder computation.

Our idea works with addresses of memory locations storing the field elements. At a conceptual level, two arrays U and V store the addresses of the relevant field elements, but in two different orders. The start address of U is loaded to a memory location X. Then the present bit of the scalar is compared to 1. Next, the assembly instruction `cmove` is used to copy the address of V to X. Depending upon the outcome of the prior comparison, after the execution of the `cmove` instruction, X stores the address of either U or V according as whether the present bit of the scalar is 0 or 1. Using X as a pointer it becomes possible to access and update the relevant field elements in the proper order. This strategy does not require swap or movement of any field element. Consequently, substantial speed improvement is obtained.

We note that the `cmove` instruction has earlier been used to implement the conditional statement of the Montgomery ladder in constant time. Such implementations, however, used the `cmove` instruction to implement a conditional swap or a conditional selection of field elements. We propose a new use of the `cmove` instruction to implement the conditional branching of the ladder that does not require swapping or selection of field elements.

To demonstrate the practicability of our idea, we have carried out 64-bit assembly language implementations of the algorithm targeting the Intel Skylake and later generation processors. For the implementations, we chose Curve25519 and Curve448 due to their importance in being part of TLS Version 1.3. The above mentioned savings in 64-bit data movement operations combined with carefully optimised assembly code lead to substantial speed-up over the previously known implementations [12] on Skylake and the Kaby Lake processors.

1. For Curve25519, about 11% speed-up is obtained on Skylake and Kaby Lake.
2. For Curve448, about 13% speed-up is obtained on Skylake and Kaby Lake.

Our source codes are publicly available at the following link.

`https://github.com/kn-cs/shared-secret-curve25519-curve448`.

These can be used to replace the existing codes in deployed softwares to obtain substantial speed-ups.

2 The Montgomery Ladder

Let $M_{A,B} : By^2 = x^3 + Ax^2 + x$ be a Montgomery curve over a field $\mathbb{F}_p$. As mentioned earlier, following [13], we will consider p to be a large prime such that cryptography is done over a suitable subgroup of the $\mathbb{F}_p$-rational points of $M_{A,B}$.

The standard description of the Montgomery ladder is given in Algorithm 1. In the algorithm, $m = \lceil \lg p \rceil$, n is the scalar and it is required to compute the scalar multiplication nP. Following the idea of clamping introduced in [2], we will assume that the $(m-1)$-th bit of the scalar n is set to 1. This ensures that the number of iterations is the same for all scalars. Another option to achieve a constant number of iterations is mentioned in Section 5.3 of [5]. A single step of the ladder is described in Algorithm 2. For details of the background theory and correctness of these algorithms we refer to [9, 4, 5].

Algorithm 1 Montgomery ladder

function MontLadder(x_P, n)
input: A scalar n and the x-coordinate x_P of a point P.
output: (X_{nP}, Z_{nP}), with $x_{nP} = X_{nP}/Z_{nP}$.
    $X_1 \leftarrow x_P; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
    for $i \leftarrow m - 1$ down to 0 **do**
        if the bit at index i of n is 1 **then**
            $(X_3, Z_3, X_2, Z_2) \leftarrow$ LadderStep$(X_1, X_3, Z_3, X_2, Z_2)$
        else
            $(X_2, Z_2, X_3, Z_3) \leftarrow$ LadderStep$(X_1, X_2, Z_2, X_3, Z_3)$
        end if
    end for
    return (X_2, Z_2)
end function.

Algorithm 2 Montgomery ladder step

function LadderStep(X_1, X_2, Z_2, X_3, Z_3)
    $T_1 \leftarrow X_2 + Z_2$
    $T_2 \leftarrow X_2 - Z_2$
    $T_3 \leftarrow X_3 + Z_3$
    $T_4 \leftarrow X_3 - Z_3$
    $T_5 \leftarrow T_1^2$
    $T_6 \leftarrow T_2^2$
    $T_2 \leftarrow T_2 \cdot T_3$
    $T_1 \leftarrow T_1 \cdot T_4$
    $T_1 \leftarrow T_1 + T_2$
    $T_2 \leftarrow T_1 - T_2$
    $X_3 \leftarrow T_1^2$
    $T_2 \leftarrow T_2^2$
    $Z_3 \leftarrow T_2 \cdot X_1$
    $X_2 \leftarrow T_5 \cdot T_6$
    $T_5 \leftarrow T_5 - T_6$
    $T_1 \leftarrow ((A + 2)/4) \cdot T_5$
    $T_6 \leftarrow T_6 + T_1$
    $Z_2 \leftarrow T_5 \cdot T_6$
    return (X_2, Z_2, X_3, Z_3)
end function.

3 Constant Time Montgomery Ladder

As has been noted earlier, the Montgomery ladder has a conditional statement. A secure implementation of the ladder requires a constant time implementation of this conditional statement. This problem is well known in the literature and several methods have been suggested for constant time implementation of the conditional statement. We discuss these below.

Conditional swap. Algorithm 1 can be made to run in constant time by using an idea known as conditionally swapping of field elements. At a top level, a description of the Montgomery ladder which uses the idea is given in Algorithm 3. This algorithm uses a subroutine CSwap which performs a constant time conditional swap as follows: CSwap$(X_2, Z_2, X_3, Z_3, \texttt{swap})$ swaps the pair of field elements (X_2, Z_2) and (X_3, Z_3) if `swap` $= 1$, else not. Two methods for implementing CSwap have been described in the literature. Algorithm 4 describes a method given in [5] whereas Algorithm 5 describes a method given in [4]. Both realizations of CSwap require working with field elements. Depending of the size of the field, a field element would be represented using several 64-bit words (limbs). So, both realizations of CSwap require time which is linear in the number of limbs.

Algorithm 3 Constant time Montgomery ladder using conditional swap

function MontLadderCSwap(x_P, n)
input: A scalar n and the x-coordinate x_P of a point P.
output: (X_{nP}, Z_{nP}), with $x_{nP} = X_{nP}/Z_{nP}$.
    $X_1 \leftarrow x_P; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
    `prevbit` $:= 0$
    for $i \leftarrow m-1$ down to 0 **do**
        `bit` $\leftarrow$ bit at index i of n
        `swap` $\leftarrow$ `bit` $\oplus$ `prevbit`
        `prevbit` $\leftarrow$ `bit`
        $(X_2, Z_2, X_3, Z_3) \leftarrow$ CSwap$(X_2, Z_2, X_3, Z_3, \texttt{swap})$
        $(X_2, Z_2, X_3, Z_3) \leftarrow$ LadderStep$(X_1, X_2, Z_2, X_3, Z_3)$
    end for
    return (X_2, Z_2)
end function.

Algorithm 4 Conditional swap using the operators and and xor

function CSwap1(X_2, Z_2, X_3, Z_3, b)
input: X_2, Z_2, X_3, Z_3 are field elements encoded as m-bit strings and b is a bit.
output: The pairs (X_2, Z_2) and (X_3, Z_3) are swapped if $b = 1$, else not.
    `mask` $\leftarrow (bb \ldots b)_m$
    $T_1 \leftarrow$ `mask` and $(X_2$ xor $X_3)$
    $T_2 \leftarrow$ `mask` and $(Z_2$ xor $Z_3)$
    $T_3 \leftarrow T_1$ xor X_2
    $T_4 \leftarrow T_2$ xor Z_2
    $T_5 \leftarrow T_1$ xor X_3
    $T_6 \leftarrow T_2$ xor Z_3
    return (T_3, T_4, T_5, T_6)
end function.

Algorithm 5 Conditional swap using the operators +, - and $\cdot$

function CSwap2(X_2, Z_2, X_3, Z_3, b)
input: X_2, Z_2, X_3, Z_3 are field elements encoded as m-bit strings and b is a bit.
output: The pairs (X_2, Z_2) and (X_3, Z_3) are swapped if $b = 1$, else not.
    $T_1 \leftarrow b \cdot (X_3 - X_2) + X_2$
    $T_2 \leftarrow b \cdot (Z_3 - Z_2) + Z_2$
    $T_3 \leftarrow (1 - b) \cdot (X_3 - X_2) + X_2$
    $T_4 \leftarrow (1 - b) \cdot (Z_3 - Z_2) + Z_2$
    return (T_1, T_2, T_3, T_4)
end function.

Conditional selection. A different idea, which may be called conditional select, can also be used to make the Algorithm 1 run in constant time. We provide a general formalisation of the idea from the implementation of shared secret computation of Curve25519 accompanying the work [12]. The description of the Montgomery ladder using conditional selection is given in Algorithm 6. This algorithm uses a subroutine CSelect which performs a constant time conditional selection as follows: CSelect(`swap`, X, Y) overwrites the value in X with the value in Y if `swap` $= 1$, else not. The variable X is used for further computation within the ladder-step. So, if `swap` $= 1$, the field element stored in Y is selected, else the field element stored in X is selected. It can be easily verified that Algorithm 6 correctly computes the Montgomery ladder. Using the subroutine CSelect twice within the ladder-step comes out to be beneficial compared to the subroutine CSwap in terms of computation time. We discuss this in further details in the next section with the help of an example.

Algorithm 6 Constant time Montgomery ladder using conditional selection

function MontLadderCSelect(x_P, n)
input: A scalar n and the x-coordinate x_P of a point P.
output: (X_{nP}, Z_{nP}), with $x_{nP} = X_{nP}/Z_{nP}$.
    $X_1 \leftarrow x_P; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
    `prevbit` $\leftarrow 0$
    for $i \leftarrow m-1$ down to 0 **do**
        `bit` $\leftarrow$ bit at index i of n
        `swap` $\leftarrow$ `bit` $\oplus$ `prevbit`
        `prevbit` $\leftarrow$ `bit`
        $T_1 \leftarrow X_2 + Z_2$
        $T_2 \leftarrow X_2 - Z_2$
        $T_3 \leftarrow X_3 + Z_3$
        $T_4 \leftarrow X_3 - Z_3$
        $T_5 \leftarrow T_1 \cdot T_4$
        $T_6 \leftarrow T_2 \cdot T_3$
        CSelect(`swap`, T_1, T_3)
        CSelect(`swap`, T_2, T_4)
        $T_1 \leftarrow T_1^2$
        $T_2 \leftarrow T_2^2$
        $T_7 \leftarrow T_5 + T_6$
        $T_8 \leftarrow T_5 - T_6$
        $X_3 \leftarrow T_7^2$
        $T_7 \leftarrow T_8^2$
        $T_8 \leftarrow T_1 - T_2$
        $T_9 \leftarrow ((A+2)/4) \cdot T_8$
        $T_9 \leftarrow T_9 + T_2$
        $X_2 \leftarrow T_1 \cdot T_2$
        $Z_2 \leftarrow T_8 \cdot T_9$
        $Z_3 \leftarrow T_7 \cdot X_1$
    end for
    return (X_2, Z_2)
end function.

4 Constant Time Implementations of Montgomery Ladder

In this section, we consider prior assembly implementations of Montgomery ladder that runs in constant time. Curve25519 is taken as a concrete example.

Implementation using conditional swap. The example that we discuss here is from the `amd64-64` implementation[1] of Curve25519 accompanying the work [3]. For 64-bit implementation, the elements of $\mathbb{F}_{2^{255}-19}$ have 4-limb representation. Consider the 4 limbs of the field elements X_2, Z_2, X_3, Z_3 to be stored at the memory locations mentioned below. Also, let the register `rsi` hold the value of `swap`.

X_2 : `0(%rdi), 8(%rdi), 16(%rdi), 24(%rdi)`

Z_2 : `32(%rdi), 40(%rdi), 48(%rdi), 56(%rdi)`

X_3 : `64(%rdi), 72(%rdi), 80(%rdi), 88(%rdi)`

[1] https://github.com/floodyberry/supercop/blob/master/crypto_scalarmult/curve25519/amd64-64/work_cswap.s (accessed on November 10, 2019).

Z_3 : `96(%rdi), 104(%rdi), 112(%rdi), 120(%rdi)`

The assembly instructions for swapping used in the `amd64-64` [3] implementation is shown in Figure 1. Except the `cmp`, all other instructions in the first column of Figure 1 perform a conditional swap between X_2 and X_3. Similarly, the instructions in the second column perform a conditional swap between Z_2 and Z_3. The bit value of `swap` is compared with `1` using the `cmp` instruction; if `swap = 1`, then the `cmove` instructions performs the limb-wise swapping of the field elements; else the `cmove` instruction reads the relevant memory locations, but, the elements remain unchanged. From Figure 1 we observe that the constant time implementation of conditional swap involves swapping of two pairs of field elements. The assembly code in Figure 1 has 32 `movq`, 8 `mov` and 16 `cmove` operations.

```
cmp     $1, %rsi

movq    0(%rdi), %rsi
movq    64(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 0(%rdi)
movq    %rdx, 64(%rdi)

movq    8(%rdi), %rsi
movq    72(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 8(%rdi)
movq    %rdx, 72(%rdi)

movq    16(%rdi), %rsi
movq    80(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 16(%rdi)
movq    %rdx, 80(%rdi)

movq    24(%rdi), %rsi
movq    88(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 24(%rdi)
movq    %rdx, 88(%rdi)
```

```
movq    32(%rdi), %rsi
movq    96(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 32(%rdi)
movq    %rdx, 96(%rdi)

movq    40(%rdi), %rsi
movq    104(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 40(%rdi)
movq    %rdx, 104(%rdi)

movq    48(%rdi), %rsi
movq    112(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 48(%rdi)
movq    %rdx, 112(%rdi)

movq    56(%rdi), %rsi
movq    120(%rdi), %rdx
mov     %rsi, %rcx
cmove   %rdx, %rsi
cmove   %rcx, %rdx
movq    %rsi, 56(%rdi)
movq    %rdx, 120(%rdi)
```

Figure 1: Assembly code to implement constant time conditional swap.
Taken from the `amd64-64` implementation of [3].

Implementation using conditional selection. The 64-bit implementation of Curve25519[2] provided with [12] uses conditional selection. As before, here also the elements of $\mathbb{F}_{2^{255}-19}$ have 4-limb representation. The conditional selection in Algorithm 6 between the elements T_1, T_3, and T_2, T_4 are performed using a certain number of `cmovnz` instructions.

The inline assembly code taken from the implementation of [12] is provided in the left column and the generated assembly is shown in the right column of Figure 2. From the generated assembly it can be observed that the registers `r9, r8, rsi, rax` hold the limb value of X for the subroutine $\mathsf{CSelect}(\mathtt{swap}, X, Y)$. The register values are conditionally overwritten with the limb values of Y through the `cmovnz` instruction after the value of `swap` is tested using the `test` instruction. It may be noted that

[2] https://github.com/armfazh/rfc7748_precomputed/blob/master/src/x25519_x64.c (accessed on November 10, 2019).

the functionality of CSelect can also be achieved using the `cmp` and `cmove` instructions without affecting the cost too much.

The assembly code shown in Figure 2 implements one conditional select operation. So, implementation of the two conditional select operations in Algorithm 2 requires a total of 16 `movq` and 8 `cmovnz` operations. It follows that the number of data movement instructions to implement the 2 CSelect operations in Algorithm 6 is significantly smaller than the number of data movement operations to implement the CSwap operation. Nevertheless, both the approaches work on entire field elements and consequently, the number of data movement operations increases linearly with the number of limbs.

Remark. In the 64-bit implementation of Curve448[3] provided with [12], the conditional selection has been implemented using a high level 'C' function. The logic used for the conditional selection is similar to the logic used in Algorithm 4. The generated assembly does not use any conditional move instructions and the number of instructions required to implement the conditional branching is fairly large.

```
static inline void cselect(uint8_t bit,
   uint64_t *const px, uint64_t *const py) {

  __asm__ __volatile__(
    "test %4, %4 ;"
    "cmovnzq %5, %0 ;"
    "cmovnzq %6, %1 ;"
    "cmovnzq %7, %2 ;"
    "cmovnzq %8, %3 ;"
    : "+r"(px[0]), "+r"(px[1]), "+r"(px[2]),
      "+r"(px[3])
    : "r"(bit), "rm"(py[0]), "rm"(py[1]),
      "rm"(py[2]), "rm"(py[3])
    : "cc"
  );
}
```

Inline assembly code of CSelect

```
movq      0(%rsi), %r9
movq      8(%rsi), %r8
movq      16(%rsi), %rcx
movq      24(%rsi), %rax

test      %dil, %dil
cmovnzq   0(%rdx), %r9
cmovnzq   8(%rdx), %r8
cmovnzq   16(%rdx), %rcx
cmovnzq   24(%rdx), %rax

movq      %r9, 0(%rsi)
movq      %r8, 8(%rsi)
movq      %rcx, 16(%rsi)
movq      %rax, 24(%rsi)
```

Generated assembly code of CSelect

Figure 2: Assembly code to implement constant time conditional select for Curve25519. Inline assembly code has been taken from the implementation of [12].

5 New Algorithm for Constant Time Conditional Branching

We propose a new strategy to implement the conditional statement in Steps 6-10 of Algorithm 1. The idea is to work with the addresses of the memory locations storing the field elements instead of the field elements themselves. Assume that the elements X_2, Z_2, X_3, Z_3 are stored in memory. Let $\&X_2, \&Z_2, \&X_3, \&Z_3$ denote the 64-bit addresses of the first bytes of the memory locations storing the elements X_2, Z_2, X_3, Z_3 respectively. Let $U[0..3]$ and $V[0..3]$ denote two arrays of 64-bit quantities which are in memory, each having contiguous 32 bytes of memory. By U (resp. V) we will denote the 64-bit address of the first byte of the memory storing $U[0..3]$ (resp. $V[0..3]$).

To start with, the addresses $\&X_2, \&Z_2, \&X_3, \&Z_3$ are copied to $U[0], U[1], U[2], U[3]$ respectively and the addresses $\&X_3, \&Z_3, \&X_2, \&Z_2$ are copied to $V[0], V[1], V[2], V[3]$ respectively. Within the main loop, first the 64-bit address U is moved to a temporary location X. Let `bit` store the bit at index i of n. Note that the indexes are considered in the order of highest to lowest value. A comparison of `bit` is made to 1. This is followed by the constant time `cmove` operation to move V to X. As discussed previously, the operation `cmove` works as follows: if `bit` equals 1, then V is moved to X, otherwise the locations are read, but, no movement takes place. After the `cmove` operation, if `bit` $= 1$, then X contains the start address of the array $V[0..3]$ while if `bit` $= 0$, then X contains the start address of the array $U[0..3]$. Considering the contents of X to be an address, after the `cmove` operation, $X[0], X[1], X[2], X[3]$ holds either the addresses of X_2, Z_2, X_3, Z_3 or the addresses of X_3, Z_3, X_2, Z_2 according as `bit` $= 0$ or `bit` $= 1$. A second level of indirection provides access to the values required in the i-th iteration. The

[3] https://github.com/armfazh/rfc7748_precomputed/blob/master/src/x448_x64.c (accessed on November 10, 2019).

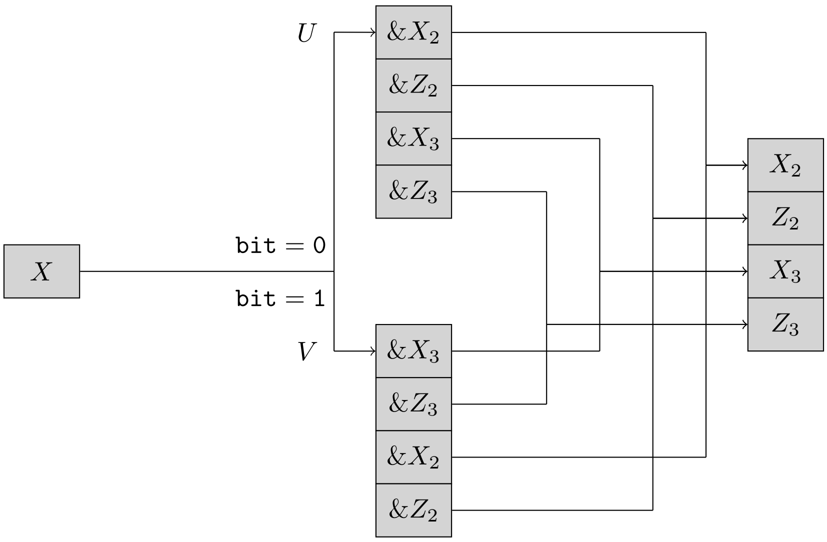

Figure 3: Idea to implement the proposed ladder.

ladder step is computed using contents pointed to by the addresses $X[0], X[1], X[2], X[3]$. Since X points to either U or V depending on whether `bit` $= 1$ or not, the ladder step correctly updates the values of X_2, Z_2, X_3, Z_3.

A diagram explaining the above idea is shown in Figure 3 and Algorithm 7 provides a pseudo-code level description. Note that the inputs to the subroutine LadderStep in MontLadderNew are addresses.

Algorithm 7 Constant time Montgomery ladder proposed through this work

1: **function** MontLadderNew(x_P, n)
2: **input:** A scalar n and the x-coordinate x_P of a point P on the elliptic curve E.
3: **output:** (X_{nP}, Z_{nP}), with $x_{nP} = X_{nP}/Z_{nP}$.
4: $X_1 \leftarrow x_P; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
5: $U[0..3] \leftarrow \{\&X2, \&Z2, \&X3, \&Z3\}$ *// store addresses of $X2, Z2, X3, Z3$ in $U[0..3]$*
6: $V[0..3] \leftarrow \{\&X3, \&Z3, \&X2, \&Z2\}$ *// store addresses of $X3, Z3, X2, Z2$ in $V[0..3]$*
7: **for** $i \leftarrow m-1$ down to 0 **do**
8: `mov` U, X *// $X \leftarrow$ base address of $U[0..3]$*
9: `bit`$\leftarrow$ bit at index i of n
10: `cmp 1, bit`
11: `cmove` V, X *// if `bit` = 1, $X \leftarrow$ base address of $V[0..3]$*
12: LadderStep($\&X_1, X[0], X[1], X[2], X[3]$) *// modifies values at $X[0], X[1], X[2], X[3]$*
13: **end for**
14: $X_2 \leftarrow$ value at $X[0]$; $Z_2 \leftarrow$ value at $X[1]$
15: **return** (X_2, Z_2)
16: **end function**.

We would like to emphasize that no swapping of elements take place in MontLadderNew. In particular, each iteration of Algorithm MontLadderNew requires two 64-bit move operations to implement the conditional branching irrespective of the size of the field. In contrast, previous implementations actually moved two pairs of field elements to achieve the same task.

To provide a concrete example of the new idea, we consider the implementation of Curve25519. Assume that elements X_2, Z_2, X_3, Z_3 are stored at memory locations `0(%rsp)` to `120(%rsp)`. Also, let the register `rcx` holds the bit at index i of n. The relevant assembly instructions for implementing Steps 4-6 and Steps 8-11 of MontLadderNew are shown in Figure 4.

1. The implementation of Steps 4-6 in Figure 4 consists of 4 `leaq` instructions which loads the addresses of X_2, Z_2, X_3, Z_3 to the registers `r11`, `r12`, `r13`, `r14` respectively. The next 8 `movq` instructions move the addresses of X_2, Z_2, X_3, Z_3 to U and the addresses of X_3, Z_3, X_2, Z_2 to V.

2. The implementation of Steps 8-11 in Figure 4 consists of two `leaq` instructions which loads the address of the start location of U to the register `rax` and the address of the start location of V

to the register `rbx`. Based on the result of the `cmp` instruction, a single `cmove` instruction ensures that the register `rax` holds the correct base address.

Note that even though we have considered the example of Curve25519, the codes in Figure 4 implementing Steps 4-6 and Steps 8-11 of MontLadderNew do not depend on the underlying field. As a result, the same codes can be used for implementing Steps 4-6 and Steps 8-11 of MontLadderNew for a Montgomery form curve over a field of any size.

Steps 4-6 of MontLadderNew are outside the main loop and are to be executed once for the entire ladder computation. Steps 8-11 of MontLadderNew are part of the loop and are executed for each bit of the scalar. So, for each iteration, implementing Steps 8-11 of MontLadderNew requires 2 `leaq`, and 1 `cmove` instructions. Additionally, the new method requires 4 `movq` instructions to resolve the first level of indirection for accessing the limb values of X_2, Z_2, X_3, Z_3 before computing $X_2 + Z_2, X_2 - Z_2, X_3 + Z_3, X_3 - Z_3$. Similarly, 4 `movq` instructions are required to resolve the first level of indirection for updating the limb values of X_2, Z_2, X_3, Z_3 at the end of a ladder-step. So, a total of 2 `leaq`, 1 `cmove` and 8 `movq` instructions are required to implement the conditional branching. This is to be contrasted with 32 `movq`, 8 `mov` and 16 `cmove` instructions required to implement CSwap based strategy and 16 `movq` and 8 `cmovnz` required to implement CSelect based strategy. Further, in the new algorithm, the number of data movement operations remain the same irrespective of the field size, whereas in the previous methods this number increases linearly with the number of limbs.

```
leaq    0(%rsp),  %r11    // &X2
leaq    32(%rsp), %r12    // &Z2
leaq    64(%rsp), %r13    // &X3
leaq    96(%rsp), %r14    // &Z3

movq    %r11, 128(%rsp)  // &X2
movq    %r12, 136(%rsp)  // &Z2
movq    %r13, 144(%rsp)  // &X3
movq    %r14, 152(%rsp)  // &Z3

movq    %r13, 160(%rsp)  // &X3
movq    %r14, 168(%rsp)  // &Z3
movq    %r11, 176(%rsp)  // &X2
movq    %r12, 184(%rsp)  // &Z2
```

Assembly code for Steps 4-6

```
leaq    128(%rsp), %rax
leaq    160(%rsp), %rbx
cmp     $1, %rcx
cmove   %rbx, %rax

//  At this point:
//  if %rcx = 0, %rax holds base address
//  of the sequence  (&X2,&Z2,&X3,&Z3)
//  if %rcx = 1, %rax holds base address
//  of the sequence  (&X3,&Z3,&X2,&Z2)
```

Assembly code for Steps 8-11

Figure 4: Assembly code to implement relevant portions of MontLadderNew for Curve25519 and Curve448.

6 Implementation and Timings

For Curve25519 and Curve448, we have carried out 64-bit assembly implementations of the Montgomery ladder using the new idea targeting the Skylake and later generation processors of Intel.

Curve	Field	Security	Skylake	Kaby Lake	Reference
Curve25519	$\mathbb{F}_{2^{255}-19}$	126	118231	113728	[12]
			105135	101564	This work
Curve448	$\mathbb{F}_{2^{448}-2^{224}-1}$	222.5	536362	521934	[12]
			470602	454685	This work

Table 1: CPU-cycle counts on Skylake and Kaby Lake processors for shared secret computation of Curve25519 and Curve448.

As mentioned earlier, the underlying primes for Curve25519 and Curve448 are $p_1 := 2^{255} - 19$ and $p_2 := 2^{448} - 2^{224} - 1$ respectively. Elements of $\mathbb{F}_{p_1}$ are represented using four 64-bit words while elements of $\mathbb{F}_{p_2}$ are represented using seven 64-bit words. The instructions `mulx/adcx/adox` available in the Skylake and later processors allow the use of two independent carry chains for multiplying/squaring two large integers represented using several 64-bit words. A general algorithmic form for multiplication/squaring of 64κ-bit numbers, $\kappa \geq 4$ is available in [10]. We have used these algorithms for implementing the integer multiplication/squaring assemblies. To reduce an element after an integer multiplication/squaring, algorithm reduceSLPMP from [10] has been used while working with $\mathbb{F}_{p_1}$. Implementations of basic field arithmetic for $\mathbb{F}_{p_2}$ have been done following the algorithms given in [11].

Platform specifications. The details of the hardware and software tools used in our software implementations are as follows.

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.

Kaby Lake: Intel®Core™ i7-7700U 4-core CPU @ 3.60GHz. The OS was 64-bit Ubuntu 18.04 LTS and the source code was compiled using GCC version 7.3.0.

Timings. The timing experiments were carried out on a single core of Skylake and Kaby Lake processors. During measurement of the cpu-cycles, turbo-boost and hyper-threading features were turned off. An initial cache warming was done with 25000 iterations and then the median of 100000 iterations was recorded. The time stamp counter TSC was read from the CPU to RAX and RDX registers by RDTSC instruction.

The numbers of cpu-cycles required for variable base scalar multiplication using the new implementations are given in Table 1. For comparison, we also provide the numbers of cpu-cycles required by the previously best known public implementations. The timings of the previous implementations were obtained by downloading the relevant software and measuring the required cycles on the same platforms where the present implementations have been measured. From Table 1, we observe that for Curve25519 about 11% and for Curve448 about 13% speed-ups are achieved.

7 Conclusion

In this work we have provided a simple and novel idea to implement the Montgomery ladder in constant time. The proposed idea has produced significant speed-ups for 64-bit implementations of variable base scalar multiplication of Curve25519 and Curve448 on Skylake and Kaby Lake processors. More generally, the idea can be applied to 64-bit Montgomery ladder computation of other curves.

References

[1] D. J. Bernstein and T. Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to/index.html, accessed on September 23, 2019.

[2] 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.

[3] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[4] 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.

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

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

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

[8] 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.

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

[10] Kaushik Nath and Palash Sarkar. Efficient Arithmetic in (Pseudo-)Mersenne Prime Order Fields. Cryptology ePrint Archive, Report 2018/985, 2018. https://eprint.iacr.org/2018/985.

[11] Kaushik Nath and Palash Sarkar. Reduction modulo $2^{448} - 2^{224} - 1$. Cryptology ePrint Archive, Report 2019/1304, 2019. https://eprint.iacr.org/2019/1304.

[12] 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.

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