Constant Time Montgomery Ladder

Kaushik Nath and Palash Sarkar

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

Abstract

In this work various approaches for constant time conditional branching in Montgomery ladder have been studied. A previous method appearing in a code for implementing X25519 has been formalized algorithmically. This algorithm is based on a conditional select operation. We consider a variant of this algorithm which groups together operations in a more convenient manner. Further, we provide a new implementation of the conditional select operation using the \texttt{cmov} operation such that \texttt{cmov} works only on registers. This provides a better guarantee of constant time behavior.

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

1 Introduction

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 $(x, y) \in \mathbb{F}_{p^i}^2$ satisfying the equation of the curve.

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 [6] 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.

Our Contributions

In the first part of the paper, we provide an overview of various approaches for constant time conditional branching in the Montgomery ladder. One of the algorithms is a formalization of a method used in a code to implement Curve25519. This algorithm is based on a conditional select operation which we denote by \texttt{CSelect}. It turns out that assembly code of \texttt{CSelect} requires a lesser number of data movement operations compared to previous assembly code using a conditional swap operation.

We consider a modified version of the algorithm using conditional select, where the instructions are grouped together in a more convenient manner. The proposed grouping of instructions helps to compute the Montgomery ladder more efficiently. Further, we provide a different assembly implementation of the \texttt{CSelect} operation. Both the new and the previous assembly implementation of \texttt{CSelect} are based on the \texttt{cmov} instruction. In our code, we ensure that \texttt{cmov} works only on registers whereas in the previous code, one of the operands of \texttt{cmov} was a memory location. Arguably, using \texttt{cmov} only on registers provides a better guarantee of constant time execution\footnote{This observation is based on a comment by Daniel J. Bernstein in an email communication}. The new assembly code requires some extra
data movement instructions to load from memory to registers. We consider this to be a small trade-off for achieving better guarantee of constant-time behavior.

2 The Montgomery Ladder

Let $M_{A,B} : By^2 = x^3 + Ax^2 + x$ be a Montgomery curve over a field $\mathbb{F}_p$. Let $P$ be a point on a Montgomery form curve and $n$ be a scalar. The operation of computing $nP$ is called a scalar multiplication. Following the terminology in [5], scalar multiplication on Curve25519 (resp. Curve448) is called X25519 (resp. X448).

Algorithm 1 Montgomery ladder

1: function MontLadder($x_P, n$)
2: input: An $m$-bit scalar $n$ and the $x$-coordinate $x_P$ of a point $P$.
3: output: $(X_{nP}, Z_{nP})$, with $x_{nP} = X_{nP}/Z_{nP}$.
4: $X_1 \leftarrow x_P; Z_2 \leftarrow 1; Z_3 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
5: for $i \leftarrow m - 1$ down to 0 do
6:   if the bit at index $i$ of $n$ is 1 then
7:   $(X_3, Z_3, X_2, Z_2) \leftarrow \text{LadderStep}(X_1, X_3, Z_3, X_2, Z_2)$
8:   else
9:   $(X_2, Z_2, X_3, Z_3) \leftarrow \text{LadderStep}(X_1, X_2, Z_2, X_3, Z_3)$
10: end if
11: end for
12: return $(X_2, Z_2)$
13: end function.

Algorithm 2 Montgomery ladder step

1: function LadderStep($X_1, X_2, Z_2, X_3, Z_3$)
2: $T_1 \leftarrow X_2 + Z_2$
3: $T_2 \leftarrow X_2 - Z_2$
4: $T_3 \leftarrow X_3 + Z_3$
5: $T_4 \leftarrow X_3 - Z_3$
6: $T_5 \leftarrow T_2^1$
7: $T_6 \leftarrow T_2^2$
8: $T_2 \leftarrow T_2 \cdot T_3$
9: $T_1 \leftarrow T_1 \cdot T_4$
10: $T_1 \leftarrow T_1 + T_2$
11: $T_2 \leftarrow T_1 - T_2$
12: $X_3 \leftarrow T_2^2$
13: $X_3 \leftarrow X_2 \cdot X_1$
14: $X_2 \leftarrow T_5 \cdot T_6$
15: $T_5 \leftarrow T_5 - T_6$
16: $T_1 \leftarrow ((A + 2)/4) \cdot T_5$
17: $T_6 \leftarrow T_5 + T_1$
18: $Z_2 \leftarrow T_5 \cdot T_6$
19: return $(X_2, Z_2, X_3, Z_3)$
20: end function.

For Montgomery curves, scalar multiplication is done using the Montgomery ladder. 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 [1], 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 [4]. 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 [6, 3, 4].
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 conditional 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\((X_2, Z_2, X_3, Z_3, \text{swap})\) swaps the pair of field elements \((X_2, Z_2)\) and \((X_3, Z_3)\) if \(\text{swap} = 1\), else not. We mention two methods for implementing CSwap which have been described in the literature. Algorithm 4 describes a method given in [4] whereas Algorithm 5 describes a method given in [3].

Algorithm 3 Constant time Montgomery ladder using conditional swap

```
1: function MontLadderCSwap\((x_P, n)\)
2:   input: An m-bit scalar \(n\) and the x-coordinate \(x_P\) of a point \(P\).
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:   \(\text{prevbit} := 0\)
6:   for \(i \leftarrow m - 1\) down to 0 do
7:     bit \(\leftarrow\) bit at index \(i\) of \(n\)
8:     swap \(\leftarrow\) bit \(\oplus\) prevbit
9:     prevbit \(\leftarrow\) bit
10:    \((X_2, Z_2, X_3, Z_3) \leftarrow\) CSwap\((X_2, Z_2, X_3, Z_3, \text{swap})\)
11:   \((X_2, Z_2, X_3, Z_3) \leftarrow\) LadderStep\((X_1, X_2, Z_2, X_3, Z_3)\)
12: end for
13: return \((X_2, Z_2)\)
14: end function.
```

Algorithm 4 Conditional swap using the operators and and xor

```
1: function CSwap1\((X_2, Z_2, X_3, Z_3, b)\)
2:   input: \(X_2, Z_2, X_3, Z_3\) are field elements encoded as m-bit strings and \(b\) is a bit.
3:   output: The pairs \((X_2, Z_2)\) and \((X_3, Z_3)\) are swapped if \(b = 1\), else not.
4:   mask \(\leftarrow (bb \ldots b)_m\)
5:   \(T_1 \leftarrow\) mask and \((X_2 \text{ xor } X_3)\)
6:   \(T_2 \leftarrow\) mask and \((Z_2 \text{ xor } Z_3)\)
7:   \(T_3 \leftarrow T_1 \text{ xor } X_2\)
8:   \(T_4 \leftarrow T_2 \text{ xor } Z_2\)
9:   \(T_5 \leftarrow T_1 \text{ xor } X_3\)
10:  \(T_6 \leftarrow T_2 \text{ xor } Z_3\)
11:  return \((T_3, T_4, T_5, T_6)\)
12: end function.
```

Conditional selection. Suppose \(X[0..1]\) is an array consisting of two field elements and \(b\) is a bit. Further suppose that the value of \(X[b]\) is required. Bernstein [1] proposed that \(X[b]\) be obtained as \((1 - b)X[0] + bX[1]\). While this method of selecting between \(X[0]\) and \(X[1]\) is more time consuming than simply accessing \(X[b]\), the advantage is that it can be executed in constant time.

We consider a variant of the above problem. Let \(X\) and \(Y\) be two variables and let \(b\) be a bit. Define CSelect\((X, Y, b)\) to be a procedure which performs the following task. If \(b = 0\), then \(X\) retains its value and if \(b = 1\), then \(X\) gets the value of \(Y\). It is possible to rewrite the Montgomery ladder using CSelect. This has been done in the code accompanying [7]. We formalize the method used in the code as Algorithm 6. The correctness of the algorithm is easy to verify. Further, assuming that CSelect can be executed in constant time, the entire ladder algorithm can also be computed in constant time.
Algorithm 5 Conditional swap using the operators $+, -$ and $\cdot$

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

Algorithm 6 Constant time Montgomery ladder using conditional selection

1: function MontLadderCSelect($x_P, n$)
2: input: An $m$-bit scalar $n$ and the $x$-coordinate $x_P$ of a point $P$.
3: output: $(x_{nP}, Z_{nP})$, with $x_{nP} = x_P/\text{znP}$.
4: $X_1 \leftarrow x_P; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
5: prevbit $\leftarrow 0$
6: for $i \leftarrow m - 1$ down to 0 do
7: bit $\leftarrow$ bit at index $i$ of $n$
8: swap $\leftarrow$ bit $\oplus$ prevbit
9: prevbit $\leftarrow$ bit
10: $T_1 \leftarrow X_2 + Z_2$
11: $T_2 \leftarrow X_2 - Z_2$
12: $T_3 \leftarrow X_3 + Z_3$
13: $T_4 \leftarrow X_3 - Z_3$
14: $T_5 \leftarrow T_1 \cdot T_4$
15: $T_6 \leftarrow T_2 \cdot T_3$
16: CSelect(swap, $T_1, T_3$)
17: CSelect(swap, $T_2, T_4$)
18: $T_3 \leftarrow T_1^2$
19: $T_2 \leftarrow T_2^2$
20: $X_3 \leftarrow T_5 + T_6$
21: $Z_3 \leftarrow T_5 - T_6$
22: $X_3 \leftarrow X_3^2$
23: $Z_3 \leftarrow Z_3^2$
24: $X_2 \leftarrow T_2$
25: $Z_2 \leftarrow T_1 - T_2$
26: $T_2 \leftarrow ((A + 2)/4) \cdot Z_2$
27: $T_2 \leftarrow T_2 + X_2$
28: $X_2 \leftarrow X_2 \cdot T_1$
29: $Z_2 \leftarrow Z_2 \cdot T_2$
30: $Z_3 \leftarrow Z_3 \cdot X_1$
31: end for
32: return $(X_2, Z_2)$
33: end function.

Following Bernstein’s suggestion mentioned above, CSelect($X, Y, b$) can be executed in constant time in the following manner. $X \leftarrow (1 - b)X + bY$. Later, we consider the issue of implementing CSelect in constant time using the cmov instruction available on Intel processors.

Remark: There is an implementation\footnote{https://munacl.cryptojedi.org/curve25519-cortexm0.shtml} of constant time conditional branching for micro-controllers which works by swapping the pointers to field elements instead of swapping the field elements themselves. The advantage of this approach is that the number of data movement operations is substantially less. On the other hand, such an approach does not necessarily lead to constant time behavior\footnote{This issue was pointed out to us by Daniel J. Bernstein and Diego Aranha.} on processors.
which have cache memory.

3.1 Assembly Implementations of CSwap and CSelect Using cmov

Intel processors support the cmov instruction. There are a number of variants of this instruction. We mention the manner in which the instruction is relevant in the present context. Suppose, $A$, $B$, $C$ are 64-bit registers or memory locations. Further suppose that $A$ stores the value of a bit $b$. Consider the following sequence of instructions.

```assembly
cmp $1, A
cmov B, C
```

The effect of the above two instructions is the following. If $A$ contains the value 0 (i.e., $b = 0$), then $C$ retains its value, otherwise (i.e., if $b = 1$) the content of $B$ is copied to $C$. So, in effect the two instructions provide an implementation of $CSelect(B, C, b)$. The cmov instruction is supposed to take constant time. We comment on this issue later.

Implementation of CSwap. Consider the CSwap based ladder given in Algorithm MontLadderCSwap. The concrete implementation of CSwap that we discuss here is from the amd64–64 implementation\(^4\) of Curve25519 accompanying the work [2]. For 64-bit implementation, the elements of $\mathbb{F}_{2^{255−19}}$ have 4-limb representations. 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.

$$
X_2 : 0(\%rdi), 8(\%rdi), 16(\%rdi), 24(\%rdi) \\
Z_2 : 32(\%rdi), 40(\%rdi), 48(\%rdi), 56(\%rdi) \\
X_3 : 64(\%rdi), 72(\%rdi), 80(\%rdi), 88(\%rdi) \\
Z_3 : 96(\%rdi), 104(\%rdi), 112(\%rdi), 120(\%rdi)
$$

The assembly instructions for swapping used in the amd64–64 [2] implementation are shown in Figure 1. Except the cmp, the effect of all the other instructions in the first column of Figure 1 is to 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 assembly code in Figure 1 has 32 movq, 8 mov and 16 cmov instructions.

Implementation of CSelect. Consider the CSelect based ladder given in Algorithm MontLadderCSelect. The 64-bit implementation of Curve25519\(^5\) provided with [7] has an implementation of CSelect. The inline assembly code taken from the implementation of [7] 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 $CSelect(\text{swap}, X, Y)$. The register values are conditionally overwritten with the limb values of $Y$ through the cmovnz instruction after the value of $\text{swap}$ is tested using the test instruction.

The assembly code shown in Figure 2 implements one CSelect operation. So, implementation of the two CSelect 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.

Remark. In the 64-bit implementation of Curve448\(^6\) provided with [7], 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.

4 Modified Algorithm for Constant Time Conditional Branching

We rearrange the sequence of steps given in Algorithm MontLadderCSelect and formalize a variant of it, which we denote as MontLadderCSelectNew and is shown in Algorithm 7. The new algorithm has lesser number of temporary variables and the copy statement in Step 24 has been omitted. The correctness of MontLadderCSelectNew follows from the correctness of MontLadderCSelect.


\(^5\)https://github.com/armfazh/rfc7748_precomputed/blob/master/src/x25519_x64.c (accessed on August 5, 2020).

\(^6\)https://github.com/armfazh/rfc7748_precomputed/blob/master/src/x448_x64.c (accessed on August 5, 2020).
cmp $1, %rsi
movq 0(%rdi), %rsi
movq 64(%rdi), %rdx
mov %rsi, %rcx
cmov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 0(%rdi)
movq %rdx, 64(%rdi)
movq 8(%rdi), %rsi
movq 72(%rdi), %rdx
mov %rsi, %rcx
mov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 8(%rdi)
movq %rdx, 72(%rdi)
movq 16(%rdi), %rsi
movq 80(%rdi), %rdx
mov %rsi, %rcx
mov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 16(%rdi)
movq %rdx, 80(%rdi)
movq 24(%rdi), %rsi
movq 88(%rdi), %rdx
mov %rsi, %rcx
mov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 24(%rdi)
movq %rdx, 88(%rdi)
movq 32(%rdi), %rsi
movq 96(%rdi), %rdx
mov %rsi, %rcx
cmov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 32(%rdi)
movq %rdx, 96(%rdi)
movq 40(%rdi), %rsi
movq 104(%rdi), %rdx
mov %rsi, %rcx
mov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 40(%rdi)
movq %rdx, 104(%rdi)
movq 48(%rdi), %rsi
movq 112(%rdi), %rdx
mov %rsi, %rcx
mov %rdx, %rsi
cmov %rcx, %rdx
movq %rsi, 48(%rdi)
movq %rdx, 112(%rdi)
movq 56(%rdi), %rsi
movq 120(%rdi), %rdx
mov %rsi, %rcx
mov %rdx, %rsi
cmov %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 [2].

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 ;" 
                          : "+r"(px[0]), "+r"(px[1]), "+r"(px[2]), 
                            "+r"(px[3])
                          :
                          : "cc"
                          : "%dil, %dil"
                          : "%r8", "%r9"
                          : "%rdx", "%rsi"
                          : "%rdi"
                          : "%r9", 0(%rsi)
                          : %r8, 8(%rsi)
                          : %rcx, 16(%rsi)
                          : %rax, 24(%rsi)
                      );
}

(a) Inline assembly code of CSelect
(b) Generated assembly code of CSelect

Figure 2: Assembly code to implement constant time conditional select for Curve25519.
Taken from the implementation of [7].
Algorithm 7 Constant time Montgomery ladder using conditional selection

1: function MontLadderCSelectNew($x_P$, $n$)
2: input: A scalar $n$ and the $x$-coordinate $x_P$ of a point $P$.
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: prevbit $\leftarrow 0$
6: for $i \leftarrow m - 1$ down to 0 do
7: $T_1 \leftarrow X_2 + Z_2$
8: $T_2 \leftarrow X_2 - Z_2$
9: $T_3 \leftarrow X_3 + Z_3$
10: $T_4 \leftarrow X_3 - Z_3$
11: $Z_3 \leftarrow T_2 \cdot T_3$
12: $X_3 \leftarrow T_1 \cdot T_4$
13: bit $\leftarrow$ bit at index $i$ of $n$
14: select $\leftarrow$ bit $\oplus$ prevbit
15: prevbit $\leftarrow$ bit
16: CSelect($T_1, T_3$, select)
17: CSelect($T_2, T_4$, select)
18: $T_2 \leftarrow T_2^2$
19: $T_1 \leftarrow T_1^2$
20: $T_3 \leftarrow X_3 + Z_3$
21: $Z_3 \leftarrow X_3 - Z_3$
22: $T_3 \leftarrow Z_3^2$
23: $X_3 \leftarrow T_3^2$
24: $T_3 \leftarrow T_1 - T_2$
25: $T_4 \leftarrow ((A + 2)/4) \cdot T_3$
26: $T_4 \leftarrow T_4 + T_2$
27: $Z_2 \leftarrow T_1 \cdot T_2$
28: $T_3 \leftarrow T_3 \cdot T_4$
29: $Z_3 \leftarrow Z_3 \cdot X_1$
30: end for
31: return $(X_2, Z_2)$
32: end function.

New assembly implementation of CSelect. Figure 2 showed the previous assembly implementation of CSelect. We would like to highlight a difference in the manner in which the cmov instructions have been used in Figure 1 and the cmovnz instructions have been used in Figure 2. In Figure 1, all the cmov instructions have their operands to be registers, while in Figure 2 all the cmovnz instructions have one operand to be a memory location while the other is a register. We prefer to have an implementation of CSelect using cmov where both the operands are registers\(^7\). This requires loading an element from the memory to registers before applying cmov. Such a strategy increases the number of mov operations. We consider this to be a small trade-off for increased assurance of constant-time execution of the cmov instruction. The modified assembly implementation of cmov is shown in Figure 3. This requires a total of 24 movq and 8 cmov instructions.

Acknowledgement. We are grateful to Daniel J. Bernstein, Diego Aranha and Adam Langley for comments on a previous work which helped clarify our understanding of constant time execution of Montgomery ladder.

References


\(^7\)In an email communication, Bernstein indicated that there are no known variable-time problems with the cmov instruction that is purely based on registers.
Figure 3: Assembly code to implement \texttt{CSelect} for X25519.


