Reduction Modulo $2^{448} - 2^{224} - 1$

Kaushik Nath[1,*], Palash Sarkar[1]

[1]Indian Statistical Institute, Kolkata, India

Received: 15th July 2020 | Revised: 4th September 2020 | Accepted: 23rd December 2020

Abstract *An elliptic curve known as Curve448 defined over the finite field $\mathbb{F}_p$, where $p = 2^{448} - 2^{224} - 1$, has been proposed as part of the Transport Layer Security (TLS) protocol, version 1.3. Elements of $\mathbb{F}_p$ can be represented using 7 limbs where each limb is a 64-bit quantity. This paper describes efficient algorithms for reduction modulo p that are required for performing field arithmetic in $\mathbb{F}_p$ using 7-limb representation. A key feature of our work is that we provide the relevant proofs of correctness of the algorithms. We also report efficient constant-time 64-bit assembly implementations for key generation and shared secret computation phases of the Diffie-Hellman key agreement protocol on Curve448. Timings results on the Haswell and Skylake processors demonstrate that the new 64-bit implementations for computing the shared secret and key generation are significantly faster than the previously best known 64-bit implementations.*

Keywords: Curve448, Goldilocks prime, modulo reduction, elliptic curve cryptography, Diffie-Hellman key agreement.

2010 Mathematics Subject Classification: 94A60

1 INTRODUCTION

As part of the Transport Layer Security (TLS) protocol, version 1.3 [21], RFC 7748 [11] specifies the Montgomery form elliptic curve Curve448 and its birationally equivalent Edwards form elliptic curve Edwards448. The curve Edwards448 was originally proposed in [9] where it was named Ed448-Goldilocks. The underlying field for Curve448 and Edwards448 is $\mathbb{F}_p$ where p is the prime $2^{448} - 2^{224} - 1$.

Implementation of elliptic curve operations require arithmetic over the underlying field $\mathbb{F}_p$. Specifically, addition, subtraction, multiplication and squaring are required. Additionally, for implementing Montgomery ladder for Curve448, it is required to implement multiplication by a small constant. All of these operations require reduction modulo p.

For 64-bit architecture, an element of $\mathbb{F}_p$ can be represented using 7 limbs where each limb is a 64-bit quantity. Such a representation can be considered to be a packed or, saturated limb representation of the elements of $\mathbb{F}_p$. Alternatively, elements of $\mathbb{F}_p$ may be represented using 8 limbs where each limb is a 56-bit quantity stored in a 64-bit word. Such a representation can be considered to be a redundant or, unsaturated limb representation. For modern Intel processors such as Skylake and later processors, the implementation of field arithmetic using the saturated limb representation turns out to be faster than that of the unsaturated limb representation.

OUR CONTRIBUTIONS

Algorithms along with their proofs of correctness. In this work, we consider the 7-limb saturated limb representation of elements of $\mathbb{F}_p$. Our focus is on the reduction algorithms which are required to implement field arithmetic operations in $\mathbb{F}_p$.

The main contribution of the paper is to present explicit reduction algorithms along with their proofs of correctness for all the field arithmetic operations required to implement Diffie-Hellman key agreement using Curve448. The algorithms proceed over several iterations successively reducing the size of the input. As part of the proof of correctness, it is required to argue that the algorithms terminate without any overflow. The termination argument has a certain amount of subtlety. To the best of our knowledge, no previous work had considered the issue of proof of correctness. Without a formal argument about termination, a reduction strategy may turn out to be incomplete or may perform redundant operations; we provide a short discussion of these possibilities in Appendix A.

*Corresponding Author: kaushikn_r@isical.ac.in

Efficient constant-time 64-bit assembly implementations of X448. The computation of the Diffie-Hellman key agreement over the curve Curve448 is based on the computation of scalar multiplication over Curve448. This computation has been named as X448 in [11]. Implementation of scalar multiplication requires implementation of field arithmetic over the underlying field. We have implemented field arithmetic over $\mathbb{F}_p$, and based on it we have developed efficient constant-time assembly implementations of the X448 function of Curve448. The performances of our 64-bit implementations for shared secret computation and key generation are faster than the previously best known 64-bit implementations. Timing details are provided later. We have made our software publicly available at the following link.

`https://github.com/kn-cs/x448/tree/master/7limb`

1.1 RELATED WORK

Efficient implementation of elliptic curve cryptography requires efficient implementation of arithmetic over the underlying finite field. Good introductions to implementation of field arithmetic can be found in [10, 12]. Many important elliptic curves have been defined over prime order fields. There have been a number of proposal where the field order is either a Mersenne or a pseudo-Mersenne prime. Examples are the prime $2^{521} - 1$ used for a NIST curve and the prime $2^{255} - 19$ used for the famous Curve25519 [1]. There are a number of works in the literature which explore various aspects of implementation of field arithmetic in the context of implementations of Curve25519 [1, 4, 6, 7, 14, 16]. For a comprehensive description of algorithms for implementing field arithmetic over Mersenne and pseudo-Mersenne primes we refer to [17]. The literature also contains proposals of elliptic curves defined over prime order fields where the prime is not a (pseudo-)Mersenne prime. For such fields, use of Montgomery arithmetic [13, 2, 5] is generally found to be helpful for efficient implementations [3]. An implementation of NIST P-256 based using Montgomery arithmetic has been reported in [8].

2 ARITHMETIC IN $\mathbb{F}_p$

Let $p = 2^{448} - 2^{224} - 1$ and $\theta = 2^{64}$. For $d \geq 0$, define the polynomial

$$f(\theta) \quad = \quad f_0 + f_1\theta + \cdots + f_d\theta^d \tag{1}$$

where $f_0, f_1, \ldots, f_d$ are non-negative integers. Following usual convention, we will call the f_i's to be limbs of $f(\theta)$.

As mentioned above, we consider the 7-limb representation of the elements of $\mathbb{F}_p$. So, elements of $\mathbb{F}_p$ can be represented as a polynomial $f(\theta) = f_0 + f_1\theta + \cdots + f_6\theta^6$ where $0 \leq f_0, f_1, \ldots, f_6 < \theta$. Note that the set of all such $f(\theta)$ is in one-one correspondence with the set of integers $\{0, 1, \ldots, 2^{448} - 1\}$. Since, $p < 2^{448} - 1$, a degree 6 polynomial $f(\theta)$ with $0 \leq f_0, f_1, \ldots, f_6 < \theta$ is not necessarily reduced modulo p. So, some elements of $\mathbb{F}_p$ have non-unique representation. This, however, is a not a problem for intermediate quantities in an elliptic curve computation. It is only the final result that is reduced to have a unique representation modulo p. Avoiding obtaining unique representations for the intermediate quantities leads to an overall faster algorithm for performing the elliptic curve computation. Consequently, given a polynomial $h(\theta) = h_0 + h_1\theta + \cdots + h_d\theta^d$, with $d > 0$, by reduction modulo p, we will denote the task of obtaining a polynomial $f(\theta) = f_0 + f_1\theta + \cdots + f_6\theta^6$ with $0 \leq f_0, f_1, \ldots, f_6 < \theta$ such that $f(\theta) \equiv h(\theta) \bmod p$.

For $i \geq 2$, let x and y be two $64i$-bit integers. Suppose, it is required to compute the integer product $x \cdot y$. If $x = y$, then this corresponds to the squaring operation, while if $x \neq y$, then a general multiplication operation is required. Intel processors from Broadwell (launched in 2014) onwards provide a special set of 64-bit multiplication and addition instructions which allow very fast computation of the product $x \cdot y$. For $i = 4$, the multiplication and squaring algorithms have been illustrated using diagrams in two Intel white papers [19, 20]. Explicit descriptions of the squaring and multiplication algorithms in the general case have been provided in [17].

A field multiplication/squaring in $\mathbb{F}_p$ consists of the following two broad steps. Suppose that $f(\theta)$ and $g(\theta)$ are two 7-limb integers from the set $\{0, 1, \ldots, 2^{448} - 1\}$ representing elements of $\mathbb{F}_p$. In the first step, the integer product of $f(\theta)$ and $g(\theta)$ is obtained in $h(\theta)$. The quantity $h(\theta)$ can be written as a 14-limb quantity $h(\theta) = h_0 + h_1\theta + \cdots + h_{13}\theta^{13}$, where $0 \leq h_0, h_1, \ldots, h_{13} < 2^{64}$. The second step consists of reducing $h(\theta)$ to a 7-limb integer which is congruent to $h(\theta)$ modulo p.

The Montgomery ladder [2, 5, 15] algorithm for Curve448 requires multiplying a 7-limb quantity $f(\theta)$ by the constant $c = 39082$ (note, $2^{15} < c < 2^{16}$ and so c is a 16-bit quantity). The integer product $c \cdot f(\theta)$ can be computed much faster than a general integer multiplication of two 7-limb quantities. The result $c \cdot f(\theta)$ can be written as an 8-limb quantity where all the limbs are 64-bit quantities. A reduction algorithm is to be applied to this 8-limb quantity to reduce it to a 7-limb quantity which represents an element of $\mathbb{F}_p$.

The integer addition of two 7-limb integers $f(\theta)$ and $g(\theta)$ results in an 8-limb integer. In this case, the last limb is a single bit. Nevertheless, the result of the addition has to be reduced to a 7-limb quantity.

Subtraction of two elements $f(\theta)$ and $g(\theta)$ in $\mathbb{F}_p$ is more problematic. The integer operation $f(\theta) - g(\theta)$ can turn out to be negative. To avoid handling negative numbers a suitable multiple of p is added to the result. This creates subtleties in the reduction algorithm.

3 REDUCTION IN $\mathbb{F}_P$

In Section 3.1 below, we describe the method for reducing a 14-limb quantity to a 7-limb quantity. As part of this algorithm, it is required to reduce an 8-limb quantity to a 7-limb quantity. Correspondingly, this part can be used to reduce the result obtained either after multiplication by a 64-bit constant or after addition of two 7-limb quantities. This is pointed out in Section 3.2. The case of subtraction in $\mathbb{F}_p$ is described in Section 3.3.

3.1 REDUCTION FROM 14-LIMB TO 7-LIMB

Let $h(\theta)$ be the 14-limb polynomial which is to be reduced. The polynomial $h(\theta)$ represents an integer z of $2 \cdot 448 = 896$ bits. A formal description of the algorithm to reduce $h(\theta)$ is given in Function reduce448 of Algorithm 1. All the operations in reduce448 can be performed using 64-bit arithmetic instructions available in modern processors. For showing correctness of the algorithm it is required to argue that the output is indeed congruent to the input modulo p. Further, it is also required to argue that the procedure terminates without any overflow.

Let $h^{(0)}(\theta) = h(\theta)$. Function reduce448 takes the 14-limb polynomial $h^{(0)}(\theta)$ as input and reduces it through the intermediate polynomials $h^{(1)}(\theta)$, $h^{(2)}(\theta)$ finally producing the 7-limb output polynomial $h^{(3)}(\theta)$. A summary of the properties of the polynomials $h^{(1)}(\theta)$, $h^{(2)}(\theta)$ and $h^{(3)}(\theta)$ and the different steps of reduce448 that produces these polynomials are as follows:

- $h^{(1)}(\theta)$ has 8 limbs. The last limb is at most 2 bits long. The computation of $h^{(1)}(\theta)$ from $h^{(0)}(\theta)$ is achieved by Steps 4-26.
- $h^{(2)}(\theta)$ has 8 limbs. The last limb is at most 1-bit long and further, if $h_7^{(2)} = 1$, then $h_4^{(2)} = h_5^{(2)} = h_6^{(2)} = 0$. The computation of $h^{(2)}(\theta)$ from $h^{(1)}(\theta)$ is achieved by Steps 27-33.
- $h^{(3)}(\theta)$ has 7 limbs where each limb is a 64-bit quantity. The computation of $h^{(3)}(\theta)$ from $h^{(2)}(\theta)$ is achieved by Steps 34-38.

The properties of $h^{(1)}(\theta)$, $h^{(2)}(\theta)$ and $h^{(3)}(\theta)$ stated above are formally proved in Theorem 1. In particular, we note that the second property stated above is required to argue that the procedure terminates without any overflow in the next iteration.

Theorem 1. *Suppose the input* $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^{13}$ *to* reduce448 *is such that* $0 \le h_i^{(0)} < 2^{64}$ *for* $i = 0, 1, \ldots, 13$. *Then the output* $h^{(3)}(\theta)$ *of* reduce448 *is such that* $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_6^{(3)}(\theta)$ *with* $0 \le h_j^{(3)} < 2^{64}$ *for* $j = 0, 1, \ldots, 6$. *Further,* $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

Proof. Let $\eta = 64$. We have the prime $p = 2^{448} - 2^{224} - 1$ and since $\theta = 2^{64} = 2^{\eta}$, we have

$$2^{448} \;=\; \theta^7 \;\equiv\; 2^{224} + 1 \;=\; 2^{\eta/2}\theta^3 + 1 \mod p. \tag{2}$$

Reduction from $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$. The input $h^{(0)}(\theta)$ to reduce448 can be written as

$$\begin{aligned} h^{(0)}(\theta) &= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)}\theta^7 + h_8^{(0)}\theta^8 + \cdots + h_{13}^{(0)}\theta^{13}), \\ &= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)\theta^7, \\ &\equiv (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)(2^{\eta/2}\theta^3 + 1) \mod p \text{ [using (2)]}, \\ &= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6) + \\ &\quad (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)\theta^3 2^{\eta/2}. \end{aligned} \tag{3}$$

Steps 4-8 add the two polynomials $(h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6)$ and $(h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)$ limb-wise by forwarding the 1-bit carry, producing the polynomial $(r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7)$. Hence, from (3) we write

$$h^{(0)}(\theta) \;\equiv\; \underbrace{(r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7)}_{\text{through Steps 4-8}} + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)\theta^3 2^{\eta/2} \mod p, \tag{4}$$

Algorithm 1 Reduction from 14-limb to 7-limb in $\mathbb{F}_p$. In the algorithm, $\eta = 64$.

function reduce448($h^{(0)}(\theta)$)
input: $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^{13}$ such that $0 \le h_i^{(0)} < 2^\eta$ for $i = 0, 1, \ldots, 13$.
output: $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_6^{(3)}\theta^6$ such that $0 \le h_i^{(3)} < 2^\eta$ for $i = 0, 1, \ldots, 6$ and $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

$t \leftarrow h_0^{(0)} + h_7^{(0)}$; $r_0^{(0)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
for $i \leftarrow 1$ to 6 **do**
  $t \leftarrow h_i^{(0)} + h_{i+7}^{(0)} + \mathsf{carry}$; $r_i^{(0)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
end for
$r_7^{(0)} \leftarrow \mathsf{carry}$
$s_0^{(0)} \leftarrow r_0^{(0)}$; $s_1^{(0)} \leftarrow r_1^{(0)}$; $s_2^{(0)} \leftarrow r_2^{(0)}$
$t \leftarrow r_3^{(0)} + 2^{\eta/2}\lfloor h_{10}^{(0)}/2^{\eta/2} \rfloor$; $s_3^{(0)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
for $i \leftarrow 4$ to 6 **do**
  $t \leftarrow r_i^{(0)} + h_{i+7}^{(0)} + \mathsf{carry}$; $s_i^{(0)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
end for
$s_7^{(0)} \leftarrow r_7^{(0)} + \mathsf{carry}$
for $i \leftarrow 0$ to 2 **do**
  $t_i^{(0)} \leftarrow 2^{\eta/2}(h_{i+11}^{(0)} \bmod 2^{\eta/2}) + \lfloor h_{i+10}^{(0)}/2^{\eta/2} \rfloor$
end for
$t_3^{(0)} \leftarrow 2^{\eta/2}(h_7^{(0)} \bmod 2^{\eta/2}) + \lfloor h_{13}^{(0)}/2^{\eta/2} \rfloor$
for $i \leftarrow 4$ to 6 **do**
  $t_i^{(0)} \leftarrow 2^{\eta/2}(h_{i+4}^{(0)} \bmod 2^{\eta/2}) + \lfloor h_{i+3}^{(0)}/2^{\eta/2} \rfloor$
end for
$t \leftarrow s_0^{(0)} + t_0^{(0)}$; $h_0^{(1)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
for $i \leftarrow 1$ to 6 **do**
  $t \leftarrow s_i^{(0)} + t_i^{(0)} + \mathsf{carry}$; $h_i^{(1)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
end for
$h_7^{(1)} \leftarrow s_7^{(0)} + \mathsf{carry}$

$t \leftarrow h_0^{(1)} + h_7^{(1)}$; $h_0^{(2)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_1^{(1)} + \mathsf{carry}$; $h_1^{(2)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_2^{(1)} + \mathsf{carry}$; $h_2^{(2)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_3^{(1)} + 2^{\eta/2}h_7^{(1)} + \mathsf{carry}$; $h_3^{(2)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_4^{(1)} + \mathsf{carry}$; $h_4^{(2)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_5^{(1)} + \mathsf{carry}$; $h_5^{(2)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_6^{(1)} + \mathsf{carry}$; $h_6^{(2)} \leftarrow t \bmod 2^\eta$; $h_7^{(2)} \leftarrow \lfloor t/2^\eta \rfloor$

$t \leftarrow h_0^{(2)} + h_7^{(2)}$; $h_0^{(3)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_1^{(2)} + \mathsf{carry}$; $h_1^{(3)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$t \leftarrow h_2^{(2)} + \mathsf{carry}$; $h_2^{(3)} \leftarrow t \bmod 2^\eta$; $\mathsf{carry} \leftarrow \lfloor t/2^\eta \rfloor$
$h_3^{(3)} \leftarrow h_3^{(2)} + 2^{\eta/2}h_7^{(2)} + \mathsf{carry}$
$h_4^{(3)} \leftarrow h_4^{(2)}$; $h_5^{(3)} \leftarrow h_5^{(2)}$; $h_6^{(3)} \leftarrow h_6^{(2)}$

return $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_6^{(3)}\theta^6$

end function.

where $0 \le r_0^{(0)}, r_1^{(0)}, \ldots, r_6^{(0)} < 2^\eta$, and $0 \le r_7^{(0)} < 2$.

For $j = 7, 8, \ldots, 13$, define

$$h_j^{(0)} = h_{j,0}^{(0)} + h_{j,1}^{(0)}2^{\eta/2}, \text{ where } h_{j,0}^{(0)} = h_j^{(0)} \bmod 2^{\eta/2}, \text{ and } h_{j,1}^{(0)} = \lfloor h_j^{(0)}/2^{\eta/2} \rfloor. \tag{5}$$

Using (5) for $j = 10$ we can write (4) as

$$\begin{aligned} h^{(0)}(\theta) \quad &\equiv \quad (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + \\ &\quad (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + (h_{10,0}^{(0)} + h_{10,1}^{(0)}2^{\eta/2})\theta^3 + h_{11}^{(0)}\theta^4 + h_{12}^{(0)}\theta^5 + h_{13}^{(0)}\theta^6)\theta^3 2^{\eta/2} \mod p, \end{aligned}$$

which can be further written as

$$\begin{aligned} h^{(0)}(\theta) \quad &\equiv \quad (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2} + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)}2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2})\theta^7 \mod p, \\ &\equiv \quad (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2} + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)}2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2})(\theta^3 2^{\eta/2} + 1) \mod p \text{ [using (2)]}, \\ &= \quad (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (h_{10,1}^{(0)} + h_{11}^{(0)}2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2})\theta^3 2^{\eta/2} + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)}2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2}) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2}, \\ &= \quad (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (2^{\eta/2}h_{10,1}^{(0)}\theta^3 + h_{11}^{(0)}\theta^4 + h_{12}^{(0)}\theta^5 + h_{13}^{(0)}\theta^6) + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)}2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2}) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2}. \end{aligned} \tag{6}$$

Steps 9-14 perform the addition of the polynomials $(r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7)$ and $(2^{\eta/2}h_{10,1}^{(0)}\theta^3 + h_{11}^{(0)}\theta^4 + h_{12}^{(0)}\theta^5 + h_{13}^{(0)}\theta^6)$ to produce the polynomial $(s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7)$. Hence, from (6) we write

$$\begin{aligned} h^{(0)}(\theta) \quad &\equiv \quad \underbrace{(s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7)}_{\text{through Steps 9-14}} + (h_{10,1}^{(0)} + h_{11}^{(0)}2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2}) + \\ &\quad (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2} \mod p, \end{aligned} \tag{7}$$

where $0 \leq s_0^{(0)}, s_1^{(0)}, \cdots, s_6^{(0)} < 2^{\eta}$, and $0 \leq s_7^{(0)} \leq 2$. Using the definitions of (5) we can further write (7) as

$$\begin{aligned} h^{(0)}(\theta) \quad &\equiv \quad (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7) + h_{10,1}^{(0)} + (h_{11,0}^{(0)} + h_{11,1}^{(0)}2^{\eta/2})2^{\eta/2} + (h_{12,0}^{(0)} + h_{12,1}^{(0)}2^{\eta/2})\theta 2^{\eta/2} + \\ &\quad (h_{13,0}^{(0)} + h_{13,1}^{(0)}2^{\eta/2})\theta^2 2^{\eta/2} + (h_{7,0}^{(0)} + h_{7,1}^{(0)}2^{\eta/2})\theta^3 2^{\eta/2} + (h_{8,0}^{(0)} + h_{8,1}^{(0)}2^{\eta/2})\theta^4 2^{\eta/2} + \\ &\quad (h_{9,0}^{(0)} + h_{9,1}^{(0)}2^{\eta/2})\theta^5 2^{\eta/2} + h_{10,0}^{(0)}\theta^6 2^{\eta/2} \mod p, \\ &= \quad (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7) + (h_{10,1}^{(0)} + h_{11,0}^{(0)}2^{\eta/2}) + (h_{11,1}^{(0)} + h_{12,0}^{(0)}2^{\eta/2})\theta + \\ &\quad (h_{12,1}^{(0)} + h_{13,0}^{(0)}2^{\eta/2})\theta^2 + (h_{13,1}^{(0)} + h_{7,0}^{(0)}2^{\eta/2})\theta^3 + (h_{7,1}^{(0)} + (h_{8,0}^{(0)}2^{\eta/2})\theta^4 + \\ &\quad (h_{8,1}^{(0)} + h_{9,0}^{(0)}2^{\eta/2})\theta^5 + (h_{9,1}^{(0)} + h_{10,0}^{(0)}2^{\eta/2})\theta^6, \\ &= \quad (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7) + \underbrace{(t_0^{(0)} + t_1^{(0)}\theta + \cdots + t_6^{(0)}\theta^6)}_{\text{through Steps 15-21}}. \end{aligned} \tag{8}$$

Steps 22-26 add the two polynomials $(s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7)$ and $(t_0^{(0)} + t_1^{(0)}\theta + \cdots + t_6^{(0)}\theta^6)$ limb-wise by forwarding the 1-bit carry, producing the polynomial $(h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_7^{(1)}\theta^7)$. Hence, from (8) we can write

$$\begin{aligned} h^{(0)}(\theta) \quad &\equiv \quad (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^6) + (t_0^{(0)} + t_1^{(0)}\theta + \cdots + t_6^{(0)}\theta^6) \mod p, \\ &= \quad \underbrace{(h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_7^{(1)}\theta^7)}_{\text{through Steps 22-26}} = h^{(1)}(\theta), \end{aligned} \tag{9}$$

where $0 \leq h_0^{(1)}, h_1^{(1)}, \ldots, h_6^{(1)} < 2^{\eta}$, and $0 \leq h_7^{(1)} < 2^2$. In the rest of the proof, we use the looser bound $h_7^{(1)} < 2^{16} = 2^{\eta/4}$. This does not cause any problem. The advantage is that, later we can refer to the subsequent part of the proof to argue about the correctness of the reduction of the quantity obtained after multiplying by the 16-bit curve constant.

Reduction from $h^{(1)}(\theta)$ **to** $h^{(2)}(\theta)$**.** Polynomial $h^{(1)}(\theta)$ can further be written as

$$\begin{aligned} h^{(1)}(\theta) &\equiv h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_6^{(1)}\theta^6 + h_7^{(1)}(2^{\eta/2}\theta^3 + 1) \mod p \text{ [using (2)]}, \\ &= (h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_6^{(1)}\theta^6) + (h_7^{(1)} + 2^{\eta/2}h_7^{(1)}\theta^3). \end{aligned} \tag{10}$$

Steps 27-33 add the polynomial $(2^{\eta/2}\theta^3+1)h_7^{(1)} = (h_7^{(1)}+2^{\eta/2}h_7^{(1)}\theta^3)$ to the polynomial $(h_0^{(1)}+h_1^{(1)}\theta+\cdots+h_6^{(1)}\theta^6)$, which produces $(h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_7^{(2)}\theta^7)$, where $0 \leq h_0^{(2)}, h_1^{(2)}, \cdots, h_6^{(2)} < 2^\eta$, and $0 \leq h_7^{(2)} < 2$. Hence, from (10) we write

$$\begin{aligned} h^{(1)}(\theta) &\equiv (h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_6^{(1)}\theta^6) + (h_7^{(1)} + 2^{\eta/2}h_7^{(1)}\theta^3) \mod p, \\ &= \underbrace{(h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_7^{(2)}\theta^7)}_{\text{through Steps 27-33}} = h^{(2)}(\theta), \end{aligned} \tag{11}$$

where $0 \leq h_0^{(2)}, h_1^{(2)}, \cdots, h_6^{(2)} < 2^\eta$, and $0 \leq h_7^{(2)} < 2$. Note that in Steps 27-33, the value of carry is at most 1. In Step 33, $h_7^{(2)} = 1$ if and only if $h_6^{(1)} = 2^\eta - 1$ and $\mathsf{carry} = 1$ which implies $h_6^{(2)} = t \bmod 2^\eta = 2^\eta \bmod 2^\eta = 0$. Moving one step backward, in Step 32 the output carry is 1 if and only if the conditions $h_5^{(1)} = 2^\eta - 1$ and the input $\mathsf{carry} = 1$ hold, which results in setting $h_5^{(2)}$ to 0. Moving another step backward, in Step 31, the output carry is 1 if and only if the conditions $h_4^{(1)} = 2^\eta - 1$ and the input $\mathsf{carry} = 1$ hold, which results in setting $h_4^{(2)}$ to 0. Moving one more step backward, in Step 30, the output carry is 1 if and only if the conditions $h_4^{(1)} = 2^\eta - 1$ and the input $\mathsf{carry} = 1$ hold, and so the value of $h_3^{(2)}$ is bounded above by $(2^\eta - 1 + 2^{\eta/4} \cdot 2^{\eta/2} + 1) \mod 2^\eta = 2^{3\eta/4}$. Hence, if $h_7^{(2)} = 1$, the conditions

$$h_3^{(2)} < 2^{3\eta/4},\ h_4^{(2)} = h_5^{(2)} = h_6^{(2)} = 0. \tag{12}$$

have to hold.

Reduction from $h^{(2)}(\theta)$ **to** $h^{(3)}(\theta)$**.** Polynomial $h^{(3)}(\theta)$ can further be written as

$$\begin{aligned} h^{(2)}(\theta) &\equiv h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6 + h_7^{(2)}(2^{\eta/2}\theta^3 + 1) \mod p \text{ [using (2)]}, \\ &= (h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6) + (h_7^{(2)} + 2^{\eta/2}h_7^{(2)}\theta^3). \end{aligned} \tag{13}$$

If $h_7^{(2)} = 0$, then after Steps 34-38 we get $h_j^{(3)} = h_j^{(2)}, j = 0, 1, \ldots, 6$; else, if $h_7^{(2)} = 1$, then using (12) we can say that the reduction surely terminates by the addition in Step 37. Using the bound of $h_3^{(2)} < 2^{3\eta/4}$ from (12) the maximum possible value of $h_3^{(3)}$ through Step 37 is $2^{3\eta/4} + 2^{\eta/2} + 1 < 2^\eta$. This implies after Steps 34-38 $0 \leq h_j^{(3)} < 2^\eta, j = 0, 1, 2, 3$, and $h_4^{(3)} = h_5^{(3)} = h_6^{(3)} = 0$. Hence, in any case from (13) it follows that

$$\begin{aligned} h^{(2)}(\theta) &\equiv (h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6) + (h_7^{(2)} + 2^{\eta/2}h_7^{(2)}\theta^3) \mod p, \\ &= \underbrace{(h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_6^{(3)}\theta^6)}_{\text{through Steps 34-38}} = h^{(3)}(\theta), \end{aligned} \tag{14}$$

where $0 \leq h_0^{(3)}, h_1^{(3)}, \cdots, h_6^{(3)} < 2^\eta$. Also, by combining (9), (11) and (14) we have $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$, which proves the theorem. □

Remark. Note that for a non-negative integer x and a positive integer μ, the operation $x \bmod 2^\mu$ extracts the least significant μ bits of x, while the operation $\lfloor x/2^\mu \rfloor$ returns an integer obtained by dropping the least significant μ bits of x. There are efficient ways to implement these operations using assembly instructions. In particular, the operations involved in Steps 15-21 of Algorithm 1 concatenate the least significant 32 bits of a limb with the leading 32 bits of the predecessor limb to create a block of 64 bits. In the assembly implementation, this is fulfilled using the `shrd` instruction. For a detailed understanding of how this is done, we refer to the last two paragraphs of Appendix A.

3.2 REDUCTION FROM 8-LIMB TO 7-LIMB

Integer addition of two field elements in $\mathbb{F}_p$ will produce an 8-limb quantity, the eighth limb of which has a size of at most 1 bit. Multiplying a field element by a field constant will also produce an 8-limb quantity. Considering Curve448, the field constant [9] with which a multiplication of a field element arises in the Montgomery ladder is $(A+2)/4 = (156326+2)/4 = 39082 < 2^{16} = 2^{\eta/4}$. Hence, given an 8-limb quantity, the reduction to 7-limb can be performed as follows. Consider the 8-limb quantity to be $h^{(1)}(\theta)$ and apply the part of reduce448 which reduces $h^{(1)}(\theta)$ to $h^{(3)}(\theta)$. The correctness of the reduction is guaranteed by the part of the proof of Theorem 1 which argues the correctness of the reduction from $h^{(1)}(\theta)$ to $h^{(2)}(\theta)$ and from $h^{(2)}(\theta)$ to $h^{(3)}(\theta)$.

3.3 SUBTRACTION

Let $f(\theta)$ and $g(\theta)$ be 7-limb quantities representing elements of $\mathbb{F}_p$. The requirement is to compute $(f(\theta) - g(\theta)) \bmod p$. Function sub448 of Algorithm 2 performs this computation. The description of sub448 uses the instruction sub which is defined as follows. Let x and y be 64-bit quantities and $\mathfrak{b}_0$ be a bit. The instruction $\mathsf{sub}(x, y, \mathfrak{b}_0)$ produces as output the pair $(z, \mathfrak{b}_1)$ where z is a 64-bit quantity and $\mathfrak{b}_1$ is a bit. The definitions of z and $\mathfrak{b}_1$ are as follows.

$$z = \begin{cases} x - (y + \mathfrak{b}_0) & \text{if } x \geq y + \mathfrak{b}_0, \\ 2^{64} + x - (y + \mathfrak{b}_0) & \text{if } x < y + \mathfrak{b}_0; \end{cases} \tag{15}$$

$$\mathfrak{b}_1 = \begin{cases} 0 & \text{if } x \geq y + \mathfrak{b}_0, \\ 1 & \text{if } x < y + \mathfrak{b}_0. \end{cases} \tag{16}$$

The assembly instruction sub can be used to implement $\mathsf{sub}(x, y, 0)$ while the assembly instruction sbb can be used to implement the more general $\mathsf{sub}(x, y, \mathfrak{b}_0)$.

Algorithm 2 Subtraction in $\mathbb{F}_p$.

function $\mathsf{sub448}((f(\theta), g(\theta)))$
input: 7-limb quantities $f(\theta)$ and $g(\theta)$ such that $0 \leq f_i, g_j < 2^{64}$ for $i, j = 0, 1, \ldots, 6$.
output: $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_1^{(2)}\theta^6$ such that $0 \leq h_i^{(2)} < 2^{64}$ for $i = 0, 1, \ldots, 6$ and $h^{(2)}(\theta) \equiv (f(\theta) - g(\theta)) \bmod p$.

$\mathfrak{b} \leftarrow 0$
for $i \leftarrow 0$ to 6 **do**
$(h_i^{(0)}, \mathfrak{b}) \leftarrow \mathsf{sub}(f_i, g_i, \mathfrak{b})$
end for

$\mathfrak{d} \leftarrow \mathfrak{b}$; $\mathfrak{d}' \leftarrow \mathfrak{d} \ll 32$
$\mathfrak{b} \leftarrow 0$
$(h_0^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_0^{(0)}, \mathfrak{d}, \mathfrak{b})$
$(h_1^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_1^{(0)}, 0, \mathfrak{b})$
$(h_2^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_2^{(0)}, 0, \mathfrak{b})$
$(h_3^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_3^{(0)}, \mathfrak{d}', \mathfrak{b})$
$(h_4^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_4^{(0)}, 0, \mathfrak{b})$
$(h_5^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_5^{(0)}, 0, \mathfrak{b})$
$(h_6^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_6^{(0)}, 0, \mathfrak{b})$

$\mathfrak{d} \leftarrow \mathfrak{b}$; $\mathfrak{d}' \leftarrow \mathfrak{d} \ll 32$
$\mathfrak{b} \leftarrow 0$
$(h_0^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_0^{(1)}, \mathfrak{d}, \mathfrak{b})$
$(h_1^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_1^{(1)}, 0, \mathfrak{b})$
$(h_2^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_2^{(1)}, 0, \mathfrak{b})$
$(h_3^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_3^{(1)}, \mathfrak{d}', \mathfrak{b})$
$h_4^{(2)} \leftarrow h_4^{(1)}$; $h_5^{(2)} \leftarrow h_5^{(1)}$; $h_6^{(2)} \leftarrow h_6^{(1)}$

return $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6$
end function.

The correctness of sub448 is stated in the following theorem.

Theorem 2. *The output* $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6$ *of* sub448 *satisfies* $0 \leq h_i^{(2)} < 2^{64}$ *for* $i = 0, 1, \ldots, 6$ *and* $h^{(2)}(\theta) \equiv (f(\theta) - g(\theta)) \bmod p$.

Proof. The limbs $h_i^{(2)}$, $i = 0, 1, \ldots, 6$ are obtained as the first components of the outputs of some invocations of the sub instruction. Consequently, it follows that all of these are 64-bit quantities. This settles the point about the bounds on these limbs. So, we have to argue two things. First, $h^{(2)}(\theta) = (f(\theta) - g(\theta)) \bmod p$ and second that the procedure terminates without any overflow. The congruency argument is obtained from the following observations.

1. Let $\delta = 2^{224} + 1$. Steps 8-16 of sub448 correspond to the subtraction of δ from the integer represented by $h^{(0)}(\theta)$. Similarly, Steps 17-22 correspond to the subtraction of δ from the integer represented by $h^{(1)}(\theta)$.
2. Suppose $f(\theta) \geq g(\theta)$ (as integers). Then, after Step 7, we have $h^{(0)}(\theta) = f(\theta) - g(\theta)$ and $\mathfrak{b} = 0$. As a consequence of $\mathfrak{b} = 0$ at Step 7, it follows that $h^{(0)}(\theta) = h^{(1)}(\theta) = h^{(2)}(\theta)$ establishing the result for this particular case.
3. In view of the previous point, assume $f(\theta) < g(\theta)$. In this case, after Step 7, we have that $h^{(0)}$ represents the integer $2^{448} + f(\theta) - g(\theta)$ and $\mathfrak{b} = 1$. Steps 10-16 subtract δ from $h^{(0)}(\theta) = 2^{448} + f(\theta) - g(\theta)$.
   (a) If $h^{(0)}(\theta) \geq \delta$, then after Step 16, $h^{(1)}(\theta)$ represents the integer $h^{(0)}(\theta) - \delta = 2^{448} + f(\theta) - g(\theta) - \delta = p + f(\theta) - g(\theta) \equiv (f(\theta) - g(\theta)) \bmod p$ and $\mathfrak{b} = 0$. As a consequence of $\mathfrak{b} = 0$ at Step 16, it follows that $h^{(2)}(\theta) = h^{(1)}(\theta)$ establishing the result for this case.
   (b) If $h^{(0)}(\theta) < \delta$, then after Step 16, $h^{(1)}(\theta)$ represents the integer $2^{448} + h^{(0)}(\theta) - \delta = 2^{448} + 2^{448} + f(\theta) - g(\theta) - \delta = 2^{448} + p + f(\theta) - g(\theta)$ and $\mathfrak{b} = 1$. Steps 19-22 subtract δ from $h^{(1)}(\theta) = 2^{448} + p + f(\theta) - g(\theta)$ to obtain $h^{(2)}(\theta) = h^{(1)}(\theta) - \delta = 2^{448} + p + f(\theta) - g(\theta) - \delta = 2p + f(\theta) - g(\theta) \equiv (f(\theta) - g(\theta)) \bmod p$.

It only remains to argue that $\mathfrak{b}$ produced by the sub instruction in Step 22 is necessarily 0. If the value of $\mathfrak{b}$ in the input of sub in Step 22 is 0, then of course, the value of $\mathfrak{b}$ produced by this sub call is also 0. So, suppose that the value of $\mathfrak{b}$ in the input of sub in Step 22 (which is the output value of $\mathfrak{b}$ in Step 21) is 1. This implies that the value of $\mathfrak{d}'$ in Step 22 is 2^{32}, since otherwise if, $\mathfrak{d}' = 0$ then output of $\mathfrak{b}$ in Step 21 has to be 0, which contradicts the assumption that the output value of $\mathfrak{b}$ in Step 21 is 1. Then the value of $\mathfrak{b}$ produced by the sub call is Step 22 is 0, if and only if $h_3^{(1)} \geq 2^{32} + 1$. The value of $\mathfrak{b}$ in the input of sub in Step 22 is 1, only if the value of $\mathfrak{b}$ produced by the sub call in Step 16 is 1. Arguing backwards, the value of $\mathfrak{b}$ produced by the sub call in Step 13 must be 1. The input to the sub call in Step 13 is $(h_3^{(0)}, \mathfrak{d}', \mathfrak{b})$ and so the value of $\mathfrak{b}$ produced by this sub call is 1 if and only if the value of $\mathfrak{b}$ in the input to this sub call is 1 and $0 \leq h_3^{(0)} < 2^{32} + 1$. This implies $2^{64} - 2^{32} - 1 \leq 2^{64} + h_3^{(0)} - 2^{32} - 1 < 2^{64}$, or, $2^{64} - 1 - 2^{32} \leq h_3^{(1)} < 2^{64}$, from which it follows that $h_3^{(1)} \geq 2^{32} + 1$, as required. □

Remark. All 64-bit operations used to implement Algorithms 1 and 2 on standard processors take time which is independent of the actual values of the operand. The different assembly instructions of the target architecture corresponding to the basic arithmetic operations $*$, $/$, $+$, $-$, mod which are involved in the implementations of the Montgomery ladders are atomic in nature and hence constant-time by default.

4 IMPLEMENTATIONS AND TIMINGS

We present two 7-limb 64-bit implementations for shared secret computation phase of the X448 function. We term these implementations as `mxaa`- and `maax`-type implementations. The implementations based on the instructions `mulx, add, adc` are collectively termed as `mxaa`, and the implementations based on the instructions `mulx, adcx, adox` are collectively termed as `maax`. All the implementations are based on 64-bit assembly instructions targeting the Intel architectures. The `mxaa` type implementations are supported across a wide range of Intel processors. The `maax` type implementations are supported on modern Intel processors such as Skylake, but are not supported on previous generation processors such as Haswell.

Implementation of X448 requires implementation of field arithmetic over $\mathbb{F}_p$. Field multiplication and squaring are done in two steps. The first step multiplies two 7-limb field elements (considered as integers) to obtain a 14-limb integer. The second step reduces the 14-limb integer to a 7-limb integer. For the reduction, we have used Function reduce448 while for the integer multiplication we have used the algorithms given in [17]. Implementations of field addition, subtraction and multiplication by the curve constant are as described in Sections 3.2 and 3.3. Overall, the implementation of X448 requires an implementation of the Montgomery ladder. The shared secret was computed using the left-to-right Montgomery ladder given in Algorithm 7 of [15] and the key generation was computed using the right-to-left Montgomery ladder given in Algorithm 5 of [18]. We have made a careful constant-time assembly implementations of the Montgomery ladder. A major goal of the implementations have been to make efficient use of the available registers so that a minimal number of load/store instructions are required. Below, we provide timing results for the new implementations. The timing experiments were carried out on single cores of the Haswell and Skylake processors. The TurboBoost© and Hyper-Threading© features were turned off while measuring the CPU-cycles.

Platform specifications. The specifications of the hardware and software tools used in our software implementations are given below.

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

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

Operation	Haswell	Skylake	Implementation	Implementation Type
Shared secret	732013	587389	[18]	`mxaa`, inline assembly
	-	530984	[18]	`maax`, inline assembly
	719217	461379	this work	`mxaa`, assembly
	-	434831	this work	`maax`, assembly
Key generation	423703	356113	[18]	`mxaa`, inline assembly
	-	315890	[18]	`maax`, inline assembly
	420453	278743	this work	`mxaa`, assembly
	-	261683	this work	`maax`, assembly

Table 1: CPU-cycle counts on Haswell and Skylake processors for shared secret computation and key generation on Curve448. Computation of key generation has been done using Algorithm 5 of [18].

Timings in the form of CPU-cycles are provided in Table 1. For comparison we have considered the timings of the most efficient (to the best of our knowledge) publicly available 64-bit implementation of Curve448, which is the software implementation corresponding to the work [18][1]. We downloaded the mentioned software and measured the CPU-cycles on the same platforms on which we have measured the CPU-cycles of our implementations. This has been done to keep the comparisons consistent. We summarize the following observations from the timings of Table 1.

- On Skylake, the new implementations are substantially better than the the previous implementations. For shared secret computation a speed-up of about 18% and 22% are obtained for the `maax`-type and `mxaa`-type implementations respectively. For key generation a speed-up of 17% is obtained for the `maax`-type implementation, and a speed-up of about 22% is obtained for the implementation of the `mxaa`-type.
- On Haswell the new `mxaa`-type implementation for computing the shared secret is better than the previous implementation by about 13K CPU-cycles; for key generation the new `mxaa`-type implementation is nominally better. While this are improvements, they are not as substantial the improvements as has been achieved on Skylake.

While the reduction algorithms that we have described avoid certain redundant operations performed by the code corresponding to [18], and consequently, do contribute to the speed improvement, it is not the only reason for the speed-up. A major reason for the speed improvement is a very careful assembly implementation making judicious use of the available registers so that the number of load/store operations is minimal.

5 CONCLUSION

In this work we have presented reduction algorithms and their proofs of correctness required for computation in the field $\mathbb{F}_p$ where $p = 2^{448} - 2^{224} - 1$. Based on these algorithms and other previously known techniques, we have made efficient 64-bit assembly implementations of the X448 function of Curve448 leading to new speed records for 64-bit implementations. While our work has concentrated entirely on the prime $2^{448} - 2^{224} - 1$, we note that the ideas involved can be applied to other primes having a similar form such as the prime $2^{480} - 2^{240} - 1$.

Acknowledgements. We thank the reviewers for their kind comments which have helped in improving the paper.

[1]Program code from `https://github.com/armfazh/rfc7748_precomputed` was accessed on June 25, 2020.

REFERENCES

[1] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *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*. 2006, pp. 207–228. DOI: 10.1007/11745853_14. URL: https://doi.org/10.1007/11745853_14.

[2] Daniel J. Bernstein and Tanja Lange. "Montgomery Curves and the Montgomery Ladder". In: *Topics in Computational Number Theory inspired by Peter L. Montgomery*. Ed. by Joppe W. Bos and Arjen K. Lenstra. Cambridge University Press, 2017, pp. 82–115.

[3] Joppe W. Bos and Peter L. Montgomery. "Montgomery Arithmetic from a Software Perspective". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 1057. URL: http://eprint.iacr.org/2017/1057.

[4] Tung Chou. "Sandy2x: New Curve25519 Speed Records". In: *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*. Code available at https://tungchou.github.io/sandy2x/. 2015, pp. 145–160. DOI: 10.1007/978-3-319-31301-6_8. URL: https://doi.org/10.1007/978-3-319-31301-6_8.

[5] Craig Costello and Benjamin Smith. "Montgomery curves and their arithmetic - The case of large characteristic fields". In: *J. Cryptographic Engineering* 8.3 (2018), pp. 227–240. DOI: 10.1007/s13389-017-0157-6. URL: https://doi.org/10.1007/s13389-017-0157-6.

[6] Michael Düll et al. "High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers". In: *Designs Codes and Cryptography* 77.2-3 (2015), pp. 493–514.

[7] Armando Faz-Hernández and Julio López. "Fast Implementation of Curve25519 Using AVX2". In: *LATINCRYPT*. Vol. 9230. Lecture Notes in Computer Science. Springer, 2015, pp. 329–345.

[8] Shay Gueron and Vlad Krasnov. "Fast prime field elliptic-curve cryptography with 256-bit primes". In: *J. Cryptogr. Eng.* 5.2 (2015), pp. 141–151. DOI: 10.1007/s13389-014-0090-x. URL: https://doi.org/10.1007/s13389-014-0090-x.

[9] Mike Hamburg. "Ed448-Goldilocks, a new elliptic curve". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 625. URL: http://eprint.iacr.org/2015/625.

[10] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2003.

[11] Adam Langley and Mike Hamburg. *Elliptic Curves for Security*. Internet Research Task Force (IRTF), Request for Comments: 7748, https://tools.ietf.org/html/rfc7748. 2016.

[12] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[13] Peter L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44(170) (April 1985), pp. 519–521.

[14] Kaushik Nath and Palash Sarkar. "Security and Efficiency Trade-offs for Elliptic Curve Diffie-Hellman at the 128-bit and 224-bit Security Levels". In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 1259. URL: https://eprint.iacr.org/2019/1259.

[15] Kaushik Nath and Palash Sarkar. "Constant Time Montgomery Ladder". In: *IACR Cryptology ePrint Archive* 2020 (2020), p. 956. URL: https://eprint.iacr.org/2020/956.

[16] Kaushik Nath and Palash Sarkar. "Efficient 4-way Vectorizations of the Montgomery Ladder". In: *IACR Cryptology ePrint Archive* 2020 (2020), p. 378. URL: https://eprint.iacr.org/2020/378.

[17] Kaushik Nath and Palash Sarkar. "Efficient arithmetic in (pseudo-)Mersenne prime order fields". In: *Advances in Mathematics of Communications* (2020). URL: https://www.aimsciences.org/article/doi/10.3934/amc.2020113.

[18] Thomaz Oliveira et al. "How to (Pre-)Compute a Ladder - Improving the Performance of X25519 and X448". In: *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*. 2017, pp. 172–191. DOI: 10.1007/978-3-319-72565-9_9. URL: https://doi.org/10.1007/978-3-319-72565-9_9.

[19] E. Ozturk, J. Guilford, and V. Gopal. *Large integer squaring on Intel architecture processors,* Intel *white paper*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf. 2013.

[20] E. Ozturk et al. *New instructions supporting large integer arithmetic on Intel architecture processors,* Intel *white paper*. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf`. 2012.

[21] Version 1.3 TLS Protocol. *RFC 8446*. `https://datatracker.ietf.org/doc/rfc8446/?include_text=1`. 2018.

A COMPARISON TO THE REDUCTION OF OLIVEIRA ET AL.

Page 17 of [18], provides an abstraction of the reduction strategy used to convert the integer z represented by the 14-limb polynomial $h^{(0)}(\theta)$ to a reduced integer in $\mathbb{F}_p$, which are given as below.

$$
\begin{aligned}
z &\leftarrow (z \bmod 2^{672}) + (2^{448} + 2^{224})\lfloor z/2^{672} \rfloor, && (17)\\
z &\leftarrow (z \bmod 2^{448}) + (2^{224} + 1)\lfloor z/2^{448} \rfloor, && (18)\\
z &\leftarrow (z \bmod 2^{448}) + (2^{224} + 1)\lfloor z/2^{448} \rfloor. && (19)
\end{aligned}
$$

The first two steps (17) and (18) convert the 14-limb input quantity $h^{(0)}(\theta)$ to the 8-limb quantity $h^{(1)}(\theta)$, such that the size of the eighth limb of $h^{(1)}(\theta)$ is at most 2 bits long. Step (17) reduces $h^{(0)}(\theta)$ to an 11-limb polynomial, say $f(\theta) = f_0 + f_1\theta + \cdots + f_{10}\theta^{10}$ and Step (18) reduces $f(\theta)$ to $h^{(1)}(\theta)$. Step (19) further reduces $h^{(1)}(\theta)$ to $h^{(2)}(\theta)$ which is also an 8-limb quantity whose final limb is at most 1 bit long. The final reduction round that converts $h^{(2)}(\theta)$ to $h^{(3)}(\theta)$, which is a 7-limb quantity is missing. As a result, the reduction strategy suggested in [18] is incomplete.

Reduction algorithms used in the code accompanying [18]. We have studied the latest version[2] of the implementation corresponding to [18]. The reduction algorithm used in this code is different from the strategy outlined in the paper. While the strategy suggested in the paper is incomplete, the algorithms implemented in the code are indeed complete. They, however, perform some redundant operations. Recall that in the final round which reduces $h^{(2)}(\theta)$ to $h^{(3)}(\theta)$, Algorithm 1 proceeds only up to the fourth limb. Theorem 1 shows that this is sufficient. The present version of the code corresponding to [18] performs the additions till the last limb. The three extra additions in the final round are redundant. Similar redundancies are also present in addition, subtraction and multiplication by the field constant.

An efficiency issue while reducing $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$. Define $\phi = \theta^3 2^{\eta/2} = 2^{224}$, which implies $\phi^2 \equiv \phi + 1$ using (2). We can also view $h^{(0)}(\theta)$ as an equivalent polynomial $\mathfrak{h}^{(0)}$ in base ϕ defined as $\mathfrak{h}^{(0)}(\phi) = a + b\phi + c\phi^2 + d\phi^3$, where $a, b, c, d < \phi$. Under such a consideration, the reduction from $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$ of Algorithm 1 can be described as a reduction from $\mathfrak{h}^{(0)}(\phi)$ to $\mathfrak{h}^{(1)}(\phi)$ through the following steps.

$$
\begin{aligned}
\mathfrak{h}^{(0)}(\phi) &= a + b\phi + c\phi^2 + d\phi^3 \\
&= (a + b\phi) + (c + d\phi)\phi^2 \\
&\equiv (a + b\phi) + (c + d\phi)(\phi + 1) && \bmod p \\
&= (a + b\phi) + (c + d\phi) + c\phi + d\phi^2 && \bmod p \\
&\equiv (a + b\phi) + (c + d\phi) + c\phi + d\phi + d && \bmod p \\
&= (a + b\phi) + (c + d\phi) + d\phi + (d + c\phi) && \bmod p \\
&= \mathfrak{h}^{(1)}(\phi) && \bmod p.
\end{aligned}
$$

Algorithm 1 computes $\mathfrak{h}^{(1)}(\phi)$ from $\mathfrak{h}^{(0)}(\phi)$ by first adding $(c + d\phi)$ to $(a + b\phi)$ through Steps 4-8. Then it adds $d\phi$ to the result through Steps 9-14. Finally, $(d + c\phi)$ is computed through Steps 15-21 and added to the previous result through Steps 22-26 to produce $\mathfrak{h}^{(1)}(\phi)$. The x86 architecture has 15 64-bit registers (keeping aside the stack pointer register `rsp`) to work with. To store the value of the product $\mathfrak{h}^{(0)}(\phi) = (a + b\phi + c\phi^2 + d\phi^3)$ we need 14 64-bit registers. So, the polynomial $(a + b\phi)$ is stored in 7 registers and $(c + d\phi)$ is stored in another 7. We need d to compute $(d + c\phi)$, so it is better to keep d undisturbed until we compute the value of $(d + c\phi)$. We first add the register values of $(c + d\phi)$ to the registers of $(a + b\phi)$. The register values of $(a + b\phi)$ gets updated to produce the temporary sum and the register values of $(c + d\phi)$ remain unchanged. After that, we only copy the middle limb of $(c + d\phi)$ to a temporary register and mask off its lower 32 bits to achieve the first 32 bits of d. The remaining 192 bits are easily obtained from the last three register values of $(c + d\phi)$ without any extra operations. Now, we have

[2] Program code from `https://github.com/armfazh/rfc7748_precomputed/blob/master/src/fp448_x64.c` was accessed on June 25, 2020.

d and add it to the previous sum to get a modified sum. Finally, we compute $(d + c\phi)$ from $(c + d\phi)$ through `shrd` instructions which works in a circular manner and add the obtained value to the previous sum to get the final value.

An alternative way to compute $\mathfrak{h}^{(1)}(\phi)$ would be to first add $(c + 2d\phi)$ to $(a + b\phi)$ and then add $(d + c\phi)$ to the result. The code corresponding to [18] uses this method. However, depending on the number of available 64-bit registers in the x86 architectures, this is going to be less efficient. This is because, computing $2d$ from d will necessitate extra operations to back up d for computing $(d + c\phi)$ later on. As a result, the number of load/stores will increase.

Inline assembly code of reduction from [18]. We produce below the inline assembly code of reduction after integer multiplication/squaring from the implementation of [18].

```
void red_EltFp448_1w_x64(uint64_t *c, uint64_t *a) {

  __asm__ __volatile__(

      /**
       * (   ,2C13,2C12,2C11,2C10|C10,C9,C8, C7) + (C6,...,C0)
       * (r14, r13, r12, r11,     r10,r9,r8,r15)
       */
      "movq 80(%1),%%rax; movq %%rax,%%r10;"
      "movq $0xffffffff00000000, %%r8;"
      "andq %%r8,%%r10;"

      "movq $0,%%r14;"
      "movq 104(%1),%%r13; shldq $1,%%r13,%%r14;"
      "movq  96(%1),%%r12; shldq $1,%%r12,%%r13;"
      "movq  88(%1),%%r11; shldq $1,%%r11,%%r12;"
      "movq  72(%1), %%r9; shldq $1,%%r10,%%r11;"
      "movq  64(%1), %%r8; shlq  $1,%%r10;"
      "movq $0xffffffff,%%r15; andq %%r15,%%rax; orq %%rax,%%r10;"
      "movq  56(%1),%%r15;"

      "addq  0(%1),%%r15; movq %%r15, 0(%0); movq  56(%1),%%r15;"
      "adcq  8(%1), %%r8; movq  %%r8, 8(%0); movq  64(%1), %%r8;"
      "adcq 16(%1), %%r9; movq  %%r9,16(%0); movq  72(%1), %%r9;"
      "adcq 24(%1),%%r10; movq %%r10,24(%0); movq  80(%1),%%r10;"
      "adcq 32(%1),%%r11; movq %%r11,32(%0); movq  88(%1),%%r11;"
      "adcq 40(%1),%%r12; movq %%r12,40(%0); movq  96(%1),%%r12;"
      "adcq 48(%1),%%r13; movq %%r13,48(%0); movq 104(%1),%%r13;"
      "adcq     $0,%%r14;"

      /**
       * (c10c9,c9c8,c8c7,c7c13,c13c12,c12c11,c11c10) + (c6,...,c0)
       * (   r9,  r8, r15,  r13,   r12,   r11,   r10)
       */
      "movq %%r10, %%rax;"
      "shrdq $32,%%r11,%%r10;"
      "shrdq $32,%%r12,%%r11;"
      "shrdq $32,%%r13,%%r12;"
      "shrdq $32,%%r15,%%r13;"
      "shrdq $32, %%r8,%%r15;"
      "shrdq $32, %%r9, %%r8;"
      "shrdq $32,%%rax, %%r9;"

      "addq  0(%0),%%r10;"
      "adcq  8(%0),%%r11;"
      "adcq 16(%0),%%r12;"
      "adcq 24(%0),%%r13;"
      "adcq 32(%0),%%r15;"
      "adcq 40(%0), %%r8;"
      "adcq 48(%0), %%r9;"
      "adcq     $0,%%r14;"

```

```
53          /**
54           * ( c7) + (c6,...,c0)
55           * (r14)
56           */
57          "movq %%r14,%%rax; shlq $32,%%rax;"
58          "addq %%r14,%%r10; movq  $0,%%r14;"
59          "adcq    $0,%%r11;"
60          "adcq    $0,%%r12;"
61          "adcq %%rax,%%r13;"
62          "adcq    $0,%%r15;"
63          "adcq    $0, %%r8;"
64          "adcq    $0, %%r9;"
65          "adcq    $0,%%r14;"
66
67          "movq %%r14,%%rax; shlq $32,%%rax;"
68          "addq %%r14,%%r10; movq %%r10, 0(%0);"
69          "adcq    $0,%%r11; movq %%r11, 8(%0);"
70          "adcq    $0,%%r12; movq %%r12,16(%0);"
71          "adcq %%rax,%%r13; movq %%r13,24(%0);"
72          "adcq    $0,%%r15; movq %%r15,32(%0);"
73          "adcq    $0, %%r8; movq  %%r8,40(%0);"
74          "adcq    $0, %%r9; movq  %%r9,48(%0);"
75          :
76          : "r"(c), "r"(a)
77          : "memory", "cc", "%rax", "%r8", "%r9", "%r10", "%r11", "%r12", "%r13",
78            "%r14", "%r15");
79  }
```

In the inline assembly code given above, the input operand a refers to the 14-limb input polynomial, and c refers to the 7-limb reduced output polynomial. Within the code the limbs of the input operand c are accessed through the notation `%1` and the limbs of the output operand are accessed though the notation `%0`.

Steps 9-20 of the code computes $(c + 2d\phi)$ and it uses the more costly `shld` instructions for the purpose. Through Step 11 the least significant 32 bits of the middle limb held in `%%r10` is masked off to produce the first 32 bits of d in the leading 32 bits of `%%r10.` Then through the instructions `shld` and `shl` of Steps 14-18, $2d$ is computed in the registers `%%r10, %%r11, %%r12, %%r13, %%r14`. After that in Step 19, the leading 32 bits of the middle limb of $(c + d\phi)$ is concatenated just before $2d$. The first 3 limbs of c are simply read through the `mov` instructions of Steps 17,18 and 20 and we finally have $(c + 2d\phi)$ in the registers `%%r15, %%r8, %%r9, %%r10, %%r11, %%r12, %%r13, %%r14`. Steps 22-29 adds $(c + 2d\phi)$ to $(a + b\phi)$ to produce a temporary sum. Steps 35-42 generates $(d + c\phi)$ from $(c + d\phi)$ through the `shrd` instructions. The polynomial $(d + c\phi)$ is then added to the previous sum through Steps 44-51 to produce an 8-limb polynomial in the registers `%%r10, %%r11, %%r12, %%r13, %%r15, %%r8, %%r9, %%r14`. The eighth limb of this 8-limb polynomial is at most 1 bit long and is stored in the register `%%r14`. Steps 57-74 further reduce the polynomial through the method discussed in Section 3.2. However, the operations in Steps 72, 73 and 74 in the code are redundant according to Theorem 1.

Assembly code of reduction from the implementations of this work. We now provide the assembly code from our implementation which performs the reduction following the steps of Algorithm 1. Please note that here a 64-bit register `r` is accessed using the notation `%r` instead of the notation `%%r` used while writing codes using inline assembly.

The code given below performs the reduction on the 14-limb product polynomial which is held by the 14 registers `%rax, %rbx, %rcx, %rdx, %rbp, %rsi, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15` and is part of a larger assembly function that performs field multiplication/squaring. Steps 1-9 adds the polynomial $(c + d\phi)$ to $(a + b\phi)$ to produce a temporary sum to which d is further added through Steps 16-20 which produces the next temporary sum. The assembly constant `mask32h` holds the 64 bit value `0xffffffff00000000` which is used to mask off the lower 32 bits of the middle limb of $(c + d\phi)$ held by the register `%rax` as a temporary. Steps 22-29 generates $(d + c\phi)$ from $(c + d\phi)$ through the `shrd` instructions. The polynomial $(d + c\phi)$ is then added to the the previous sum through Steps 30-37 to produce the 8-limb polynomial in the registers `%r12, %rbx, %rcx, %rdx, %rbp, %rsi, %r8, %rdi`. After this Steps 39-58 further reduces the polynomial to produce the final 7-limb output polynomial in the registers `%r12, %rbx, %rcx, %rdx, %rbp, %rsi, %r8`.

Our code has only one memory-store operation in Step 11 which is indispensable in the context. We did not find a more efficient way to implement Algorithm 1 using the available 15 registers in assembly. By comparing the two codes of reduction it is easy to see that the number of instructions in our assembly is much lesser than the code of [18].

```
xorq    %rdi,%rdi
addq    %r9, %rax
adcq    %r10, %rbx
adcq    %r11, %rcx
adcq    %r12, %rdx
adcq    %r13, %rbp
adcq    %r14, %rsi
adcq    %r15, %r8
adcq    $0, %rdi

movq    %rax, 536(%rsp)

movq    %r12, %rax
andq    mask32h, %rax

addq    %rax, %rdx
adcq    %r13, %rbp
adcq    %r14, %rsi
adcq    %r15, %r8
adcq    $0, %rdi

movq    %r12,  %rax
shrd    $32, %r13, %r12
shrd    $32, %r14, %r13
shrd    $32, %r15, %r14
shrd    $32, %r9,  %r15
shrd    $32, %r10, %r9
shrd    $32, %r11, %r10
shrd    $32, %rax, %r11
addq    536(%rsp), %r12
adcq    %r13, %rbx
adcq    %r14, %rcx
adcq    %r15, %rdx
adcq    %r9,  %rbp
adcq    %r10, %rsi
adcq    %r11, %r8
adcq    $0, %rdi

movq    %rdi, %r13
shlq    $32, %r13

xorq    %r14, %r14
addq    %rdi, %r12
adcq    $0, %rbx
adcq    $0, %rcx
adcq    %r13, %rdx
adcq    $0, %rbp
adcq    $0, %rsi
adcq    $0, %r8
adcq    $0, %r14

movq    %r14, %r13
shlq    $32, %r13

addq    %r14, %r12
adcq    $0, %rbx
adcq    $0, %rcx
adcq    %r13, %rdx
```