Secure and Efficient Computation of the Diffie-Hellman Protocol Using Montgomery Curves over Prime Order Fields

A thesis submitted to the Indian Statistical Institute
in partial fulfillment of the thesis requirements for the degree of
Doctor of Philosophy in Computer Science

authored by

Kaushik Nath
(Applied Statistics Unit)

under the supervision of

Prof. Palash Sarkar
(Applied Statistics Unit)

Indian Statistical Institute
203, Barrackpore Trunk Road
Kolkata - 700 108
India

November, 2021

I dedicate this effort of mine to the academic spirit of my late father,
Kumud Bihari Nath.

Contents

Acknowledgements

This thesis describe the various research works which I have done towards my Ph.D. in Computer Science under the supervision of Prof. Palash Sarkar at the Applied Statistics Unit (ASU) of Indian Statistical Institute (ISI), Kolkata. I hope the results of this work are found useful to the community.

I completed my M.Tech. in Computer Science from ISI in 2008 and joined the institute again after seven years in pursuit of my Ph.D. degree. It has been a wonderful experience all throughout these years and I have enriched myself to be a better person in many different ways. Now, as I stand to the end of my thesis-writing, I would like to take the opportunity to express my thankfulness to all the people who have helped me in various ways during my Ph.D. days.

First of all, I express my gratitude to my supervisor Prof. Palash Sarkar for putting his faith on me and considering me as one his Ph.D. students. He has been an inspiration to me since the days of my M.Tech. program at ISI. I don't know how much I lived up to his expectations as a student of research. I can only say that I have always tried to give my best effort in the area of work that he has let me embark on. In the beginning, I expressed my interest to work in a possible area of cryptography which demands computer implementations, and I am happy that he mounted me to the area of secure and efficient implementations of software related to Elliptic Curve key Cryptography (ECC). His precise guidance has been very valuable for my learning which produced the different contributions of this thesis. I must say that I have enjoyed enough freedom while working under his supervision which has helped me to explore and further mine the area of my research with passion. The experience was new, exciting and adventurous and I wish to continue working in the area in future. In the context, I wish to thank the various anonymous reviewers and editors whose comments have improved the quality of the different research papers resulting to this thesis.

I would like to thank ISI for the financial and institutional support. I am thankful to the Director's office, the Dean's office and other administrative offices of ISI for their official help on different issues. I wish to thank my course teachers who taught me various courses in the first year of my course work and also during my M.Tech. program back in 2006-2008. I thank all other faculty members of ISI, specially from ASU who have helped me at different stages of my research. I am obliged to the Chairpersons of Research Fellows Advisory Committee (RFAC) of the Applied Statistics Division (ASD) for their kind help which I got on several occasions. I also thank the staff of ASU office for providing the different facilities whenever required. The Computer and Statistical Services Center (CSSC) of ISI provided good support, specially during the pandemic time when I had to work from home. I would like to thank all the CSSC-personnel for extending their support on different issues. I have received quality health services from

the Medical Welfare Unit of ISI regarding different illnesses of mine and eye check-ups. I would like to specially thank Dr. Himajit Debnath for his medical advice and support that I received from him during crucial times. I enjoyed the food of the canteen and I thank the canteen-staff for their services. I had a nice stay at the Research Scholar Hostel of the institute during these years. I want to thank Tapas for helping me to get the hostel room when I joined the institute in 2015. I also thank the different hostel conveners and superintendents for helping me with various issues regarding my stay in the hostel. I have wonderful memories with my hostel room and I will surely miss it for the rest of my life.

I have spent a major amount of time in the Turing lab during my research studies. I am very thankful to my lab-mates Shashank, Butu, Sanjay, Sabyasachi, Sebati, Amit, Aniruddha, Subhadip, Madhurima and Sreyosi for their wonderful company and helping me in different forms at different points of time. I want to specially mention about Sanjay's jovial presence in the Turing lab during the weekends which gave me company and helped to work for long hours during the early days of my research. Aniruddha contacted the CSSC support-staff and helped me in getting my lab-computer fixed during the pandemic time. I thank him for his help and co-operation. I had the pleasure to share good times with other scholars and friends of the institute like Sumit, Atanu, Tapas, Srimanta'da, Ankan, Indranil, Suchismita, Swarup, Mrinmoy, Murthy, Ashwin, Binu, Avik, Nilanajan, Avijit, Diptendu, Amit, Nishant, Prabal, Susanta, Laltu, Subhas, Avinandan, Shion, Jyoti, Avishek, Samir, Pritam, Mostafizar, Soumya, Nayana and many others whom I am missing to mention. My days at ISI would not have been as enjoyable as it was without their presence and I thank them all for their company. A special thanks to Susanta for helping me with the thesis submission process and to Prabal for helping me regarding a personal issue.

The accidental death of Prof. C. A. Murthy was a shocking incident. I had the opportunity to attend his lectures during my M.Tech. program. I often visited the shop of Babu'da with Sanjay to have ghoogni-parota and tea in the afternoon hours. It was sad to hear about his passing away too. I came across to the death news of a few employees of ISI whom I didn't know personally. I offer my condolences to the families of the deceased.

I was lucky to have Sumit again at ISI after my M.Tech. days and I have spent wonderful times with him at different phases. He was of great help, specially while I was doing my the course-works in the first year. I was new to cryptography and the different discussions I had with him, motivated me to opt for cryptography as the area of my research. In particular, the paper-less discussions will remain as the most memorable ones which we had while we went for a stroll around ISI in the dawn after working whole night and during the evening hours. My entry at ISI in 2006 as a student of the M.Tech. program was influenced to a good extent by the interactions on basic mathematics that I had with Atanu. Without those interactions I would have never realized about the basic motivation of the ISI entrance examination. The simple understandings about the various mathematical aspects which I learned from him long back has been one of the major reasons for my entry to ISI again in 2015. I earnestly thank Atanu and Sumit who have inspired me with their selfless academic help and wonderful company.

It has been an absolute pleasure to have Pralay and Plaban as friends since my B.Tech. days. Knowing about them in life has been an inspiration to me in different ways. I sincerely thank both of them for their help and support at different points of time. I

learned a lot from the hard-working nature of Alok'da who has been an inspiration and I sincerely thank him for his support and help while I worked under him. I also thank Dr. Arindam Roy who helped and encouraged me with his good wishes.

I would like to specially remember a few teachers from my school days and undergraduation in the context. I can't ever forget the immense contribution of my mathematics teacher late Raj Kishor Bose from my school who taught me mathematics free for four years after my father passed away. He was a wonderful teacher of mathematics and a nice human being who has always encouraged me whenever I met him after my school days. He would have been very happy today hearing about my research. I would also like to thank Chitta Banik Sir and Basu Sir for their free tuition supports during my school days. Prof. Kashinath Chatterjee was of great help and inspiration while I pursued my B.Sc. in Statistics (Hons.) at Asutosh College in Kolkata. Apart from his remarkable lectures in the classes, I received strong support from him regarding several mathematical problems which I struggled to understand, specially during the third year of the course. I thank him for all the personal time that he gave me during my undergraduation.

I will take the opportunity to express my sincere thanks to all the people who have provided their good suggestions which helped my mother to get the job of the school teacher after my father expired. Those helps were of major support for my family during our struggling times. I am thankful to my paternal uncle, late Binod Bihari Nath who supported us on different family issues even staying far from us. My fathers intimate friend, late Sitanath Mandal was a very kind man and a father figure to me. He has selflessly helped me and my family on various issues. I very much wished to let him know about the completion of my Ph.D. degree. He would have been very happy for me as he has always been, but unfortunately passed away recently. I have known a lot about my father from him and I express my heartfelt thanks for all his help and support. I would also like to extend my thanks to Parimal'da, Boudi, Tapas'da and Lokkhi'di for lending their help to my family at various capacities on different occasions. Sujit'da helped my mother at home with an important issue while I was at ISI and I am thankful to him for that.

My father whom I lost long back during my school days has been a major source of inspiration behind my academics. He would have been very happy today witnessing this academic stage of mine. I have always got unending support from my mother who stood strong behind me all throughout my life. Even being very fragile she has been the lone warrior who took all the pain to uplift our family from the sudden distress after the demise of my father. I wouldn't have been the person today without her contribution and sacrifice and I am thankful for her love, encouragement and continuous support. I would like to specially thank Didi and Sekhar'da for their helps at various times during my research. I also want to extend my thanks to Chordi and Jyotish'da for their love and encouragement. During this time I was blessed with my daughter whose charm and innocence has inspired me to improve as a person in many ways. Her smile has been a source of power which kept my motivation alive in the difficult times of my research. I can't thank enough to my wife who has taken most of the pain till now to take care of our daughter. I also thank my in-laws for their support during my research work.

To end, I wish everyone joy and happiness in life.

Kaushik Nath,
March 24, 2021.

List of Algorithms

List of Figures

List of Tables

CHAPTER 1

Introduction and Overview

Public-key cryptography came into light in 1976 through the seminal work *New Directions in Cryptography* [DH76] by Whitfield Diffie and Martin Hellman. The work introduced the first one-way function which gave birth to the famous *Diffie-Hellman*(DH) key agreement protocol. The function simply exponentiates a positive integer modulo a pre-defined prime number. The inverse of the function is called the *discrete logarithm* and the corresponding problem is known as the *Discrete Logarithm Problem* (DLP). The discrete logarithm problem can also be defined over certain other cryptographically relevant algebraic groups where the problem is believed to be computationally hard. Some of these groups are multiplicative subgroups of finite fields, group of points on elliptic curves [Kob87, Mil85], divisor class groups of degree 0 on hyper-elliptic curves [Kob89]. In 1977, Ron Rivest, Adi Shamir and Leonard Adleman proposed the *RSA cryptosystem* which is based on the hardness of the *integer factorization problem*. The DH protocol was further extended by ElGamal in 1984 to define public-key encryption and signature schemes. The area of work of this thesis is the Diffie-Hellman protocol.

1.1 Motivation

A finite cyclic group G is a group satisfying the following equivalent conditions: (1) it is both finite and cyclic (2) it is isomorphic to the group of integers modulo n for some positive integer n. An algorithmic form of the Diffie-Hellman protocol is provided in Algorithm 1.1. The protocol works on a finite cyclic group G of order n having a generating element $g \in G$. The algorithm has two phases, namely *key generation* and *shared secret computation*. The key generation phase has the operation of the form g^k which exponentiates the fixed element g to the power k. The shared secret computation phase has the operation of the form a^k which exponentiates a variable element $a \in G$ to the power k. At the end of the protocol, both Alice and Bob are in possession of the group element $w = g^{xy}$. The element w is considered as the shared secret key between them. The group G satisfies the necessary condition for making the communication secure if there is not an efficient algorithm for determining g^{xy} given g, g^x, and g^y. This is known as the Diffie Hellman Problem over multiplicative groups and is assumed to be a hard problem in G. If the discrete logarithm problem can be solved then the Diffie-Hellman problem is also solvable, but the reverse is not known to be true.

The Diffie-Hellman protocol can also be instantiated over elliptic curves which gives the *Elliptic Curve Diffie-Hellman* (ECDH) protocol. Our focus in this work is on secure and efficient computation of ECDH. We work with the elliptic curves of *Montgomery form* and consider the underlying fields to be of *prime order*. The primes that we have worked with are of special shape like *Mersenne primes*, *pseudo-Mersenne primes* and the *Solinas trinomial primes*. These primes are taken into consideration because they allow high-speed field arithmetic. We have addressed efficient computation of the ECDH on the Montgomery curves from a *software perspective*, targeting the modern *Intel architectures* at various security levels.

Algorithm 1.1 Diffie-Hellman protocol.

1: **function** DH(n, G, g)
2: **input:** n is a prime, G is a finite multiplicative group of order n and $g \in G$ is the generator of G.
3: **output:** Shared secret between Alice and Bob.

Key Generation

	Alice	**Bob**
4:	Select the private key $x \leftarrow [1, n-1]$	Select the private key $y \leftarrow [1, n-1]$
5:	Compute the public key $u \leftarrow g^x$	Compute the public key $v \leftarrow g^y$

Shared secret computation

	Alice	**Bob**
6:	Send u to Bob	Send v to Alice
7:	Compute $w \leftarrow v^x = (g^y)^x = g^{xy}$	Compute $w \leftarrow u^y = (g^x)^y = g^{xy}$

8: **return** w
9: **end function**.

Security and efficiency. Security and efficiency are two crucial issues that need to be taken care of while an elliptic curve is proposed and implementations are done which can be used in real-world applications. At the design phase the curve parameters need to be chosen such that they satisfy some pre-defined security requirements. At the implementation level an important requirement of cryptographic software is to make them side-channel resistant. For example, the software implementation has to run in constant time to resist timing attacks. Timing attacks have been an active area of research within applied cryptography in which cryptosystem or protocol implementations that do not run in constant time are attacked.

The ECDH protocol has two phases, *key generation* and *shared secret computation*. Both of the phases have a critical component known as the *scalar multiplication* and this has different computational challenges depending on the target architecture of the implementation. In the scalar multiplication operation an elliptic curve point P is multiplied by an n-bit scalar which is defined to be the n-fold addition of P. When the point P is a fixed generator of the cryptographic group then the scalar multiplication is called *fixed-base scalar multiplication*. On the other hand, when the point P is an arbitrary element then the scalar multiplication is called *variable-base scalar multiplication*. In particular, the variable-base scalar multiplication of the shared secret computation phase has more

computational interest as this is relatively the more expensive operation. The key generation is a one-time operation, whereas the shared secret amongst two parties may be computed several times once the key is established. A simple and efficient method to compute the scalar multiplication on Montgomery curves in constant time is known as the Montgomery ladder. Efficiently computing the ladder targeting various platform and architectures has been heavily researched by the cryptographic community over the last two decades. The performance of the associated field and curve arithmetic is a major target of optimization. While implementing the cryptographic software for ECDH, the instruction set of the target architecture is carefully exploited to achieve the best results. Developing code for scalar multiplication is often done by writing the corresponding programs using assembly instructions of the target architecture. The Intel architectures are one of the most heavily used architectures all over the world and many software implementations of ECDH have been developed on the top of it. The architecture provides a rich set of sequential and vectorized instructions which have been exploited to the best possible extent in the software implementations of this thesis.

1.2 Goal of the Thesis

There are several goals which are achieved by this thesis which are highlighted below.

- Provide a variety of efficient 64-bit assembly implementations of field multiplication, squaring and FLT-based inversion for several cryptographically relevant primes covering a wide range of security levels which are suitable for the Haswell-onward Intel architectures. The implementations are done on the basis of various algorithms which we formalize and also prove their correctness rigorously.

- Provide efficient assembly implementations of 4-way field multiplication/squaring using the 4-way vector instructions available in the Haswell-onward Intel architectures. This again has been done for several primes covering various security levels.

- Provide state-of-art implementations of ECDH for the standard curves over prime order fields which have been proposed at various security levels.

- Propose new curves over prime order fields targeting various security levels which are fairly competitive to the existing standards in terms of the shared-secret computation phase of ECDH. On the basis of the new curves proposed and the existing standards, we provide security and efficiency trade-offs targeting the different security levels considered.

In pursuit of the thesis goals mentioned above we have made various research contributions which we summarize next.

1.3 Summary of Contributions

Efficient arithmetic in (pseudo)-Mersenne prime order fields. Elliptic curve cryptography is based on elliptic curves defined over finite fields. Operations over such elliptic curves require arithmetic in the underlying field. In particular, fast implementations of

multiplication and squaring in the finite field are required for performing efficient elliptic curve cryptography. Here we study the problem of obtaining efficient algorithms for field multiplication and squaring. From a theoretical point of view, we present a number of algorithms for multiplication/squaring and reduction which are appropriate for different settings. Our algorithms collect together and generalize ideas which are scattered across various papers and software. At the same time, we also introduce new ideas to improve upon existing works. A key theoretical feature of our work is that we provide formal statements and detailed proofs of correctness of the different reduction algorithms that we describe. On the implementation aspect, a total of fourteen primes are considered, covering all previously proposed cryptographically relevant (pseudo-)Mersenne prime order fields at various security levels. For each of these fields, we provide 64-bit assembly implementations of the relevant multiplication and squaring algorithms targeted towards two different modern Intel architectures. We were able to find previous 64-bit implementations for six of the fourteen primes considered in this work. On the Haswell and Skylake processors of Intel, for all the six primes where previous implementations are available, our implementations outperform such previous implementations.

Reduction modulo $2^{448} - 2^{224} - 1$. 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. Here we describe 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.

Efficient 4-way vectorizations of the Montgomery ladder. We propose two new algorithms for 4-way vectorization of the well known Montgomery ladder over elliptic curves in Montgomery form. The first algorithm is suitable for variable base scalar multiplication. In comparison to the previous work by Hisil et al [HEY20], it eliminates a number of non-multiplication operations at the cost of a single multiplication by a curve constant. Implementation results show this trade-off to be advantageous. The second algorithm is suitable for fixed base scalar multiplication and provides clear speed improvement over a previous vectorization strategy due to Costigan and Schwabe[CS09]. The well known Montgomery curves Curve25519 and Curve448 are part of the TLS protocol, version 1.3. For these two curves, we provide constant time assembly implementations of the new algorithms. Additionally, for the algorithm of Hisil et al [HEY20], we provide improved implementations for Curve25519 and new implementation for Curve448. Timings results on the Haswell and Skylake processors indicate that in practice the new algorithms are to be preferred over previous methods for scalar multiplication on these curves.

Security and efficiency trade-offs for ECDH at the 128-bit, 224-bit and 256-bit security levels. Within the Transport Layer Security (TLS) Protocol Version 1.3, RFC 7748 specifies elliptic curves targeted at the 128-bit and the 224-bit security levels. For the 128-bit security level, the Montgomery curve Curve25519 and its birationally equivalent twisted Edwards curve Ed25519 are specified; for the 224-bit security level, the Montgomery curve Curve448, the Edwards curve Edwards448 (which is isogenous to Curve448) and another Edwards curve which is birationally equivalent to Curve448 are specified. Our first contribution is to provide the presently best known 64-bit assembly implementations of Diffie-Hellman shared secret computation using Curve25519. The main contribution of this work is to propose new pairs of Montgomery-Edwards curves at the 128-bit and the 224-bit and 256-bit security levels. For the new curves we work with the prime $p_1 = 2^{251} - 9$ at 128-bit security level and the prime $p_2 = 2^{444} - 17$ at 224-bit security level. The new curves at at these two security levels are *nice* in the sense that they have very small curve coefficients and base points. Compared to the curves in RFC 7748, the new curves at these two security levels lose two bits of security. The gain is improved efficiency. For Intel processors, we have made different types of implementations of the Diffie-Hellman shared secret computation using the new curves. The new curve at the 128-bit level is faster than Curve25519 for all types of implementations that we considered, while the new curve at the 224-bit level is faster than Curve448 using 64-bit sequential implementation using schoolbook multiplication, but is slower than Curve448 for vectorized implementation using Karatsuba multiplication. Overall, the new curves provide good alternatives to Curve25519 and Curve448.

At the 256-bit security level we work with three primes, namely $p_3 = 2^{506} - 45$, $p_4 = 2^{510} - 75$ and $p_5 = 2^{521} - 1$. While p_5 has been considered earlier in the literature, p_3 and p_4 are new. No Montgomery curves have been proposed in literature at 256-bit security level until now. We define a pair of birationally equivalent Montgomery and Edwards form curves over all the three primes and perform sequential and vectorized computations of the Diffie-Hellman over the proposed curves targeting the modern Intel processors. In Skylake, it has been found that the sequential implementation performs better than the vectorized implementation for p_3 and p_4, but for p_5 the vectorized implementation performs better than the sequential one. We get a comparative idea of security and efficiency of the new Montgomery curves at 256-bit security level with respect to the curves at 128-bit and 224-bit security levels.

1.4 Chapter Organization

In Chapter 2, we provide the background details and discuss the literature related to this work. In Chapter 3, different algorithms to sequentially compute the Montgomery ladder in constant time have been detailed out. After this the thesis is divided in two parts; the first part deals with new techniques for efficient implementations and the second part proposes new curves and security/efficiency trade-offs.

Part I: New techniques for efficient implementations. Part I discusses about the new contributions of efficient arithmetic in prime order fields which are addressed in the Chapters 4, 5, 6 and 7. In Chapter 4 we discuss about the sequential algorithms for multiplication/squaring over Mersenne and pseudo-Mersenne prime order fields. In Chapter

5 we discuss about the sequential algorithms for field arithmetic over the field based on the Solinas trinomial prime $2^{448} - 2^{224} - 1$. In Chapter 6 we provide formalizations of the vectorized field arithmetic that can be implemented using 4-way vector instructions. In Chapter 7 we propose new 4-way vectorizations of the Montgomery ladder.

Part II: New curves and security/efficiency trade-off. Part II discusses about the new curves proposed through this work and the security and efficiency trade-offs in comparison to the existing standards. In Chapter 8 we discuss about the new Montgomery curves proposed through this work and discuss their performances with respect to the existing standards.

Finally, in Chapter 9 we draw the conclusion where we summarize the entire work of this thesis and briefly comment on possible future works.

1.5 List of Software

Following is the list of softwares which have been contributed through this thesis. The implementations are state-of-art and have been used by researchers with acknowledgement. All our software are publicly available at the link

`https://github.com/kn-cs`.

1. **Chapter 4** provides various 64-bit implementations of field arithmetic for 14 different (pseudo)-Mersenne primes covering a wide range of security levels.

2. **Chapter 5** provides various 64-bit implementations of field arithmetic for the Solinas trinomial prime $2^{448} - 2^{224} - 1$. It also provides efficient 64-bit implementations of ECDH using Curve448.

3. **Chapter 6** provides various 4-way vectorized implementations of field arithmetic for 8 (pseudo)-Mersenne primes and the Solinas trinomial prime $2^{448} - 2^{224} - 1$.

4. **Chapter 7** provides various 4-way vectorized implementations of ECDH for the standard curves Curve25519 and Curve448.

5. **Chapter 8** provides various 64-bit implementations of ECDH for the standard curve Curve25519. It also provides various 64-bit and 4-way vectorized implementations for 5 new Montgomery curves at the 128-bit, 224-bit and 256-bit security levels.

1.6 List of Publications

This different chapters of the thesis have been written based on the following publications.

1. Kaushik Nath and Palash Sarkar.: Security and Efficiency Trade-offs for Elliptic Curve Diffie-Hellman at the 128- and 224-bit Security Levels, **Journal of Cryptographic Engineering**, March 2021. Available at https://link.springer.com/article/10.1007/s13389-021-00261-y.

2. Kaushik Nath and Palash Sarkar. Efficient 4-way Vectorizations of the Montgomery Ladder. **IEEE Transactions on Computers**, Feb. 2021. Available at https://ieeexplore.ieee.org/document/9359500.

3. Kaushik Nath and Palash Sarkar. Reduction Modulo $2^{448} - 2^{224} - 1$. **Mathematical Cryptology**, 1(1):821, Jan. 2021. Available at https://journals.flvc.org/mathcryptology/article/view/123700.

4. Kaushik Nath and Palash Sarkar. **Efficient Arithmetic in (Pseudo-)Mersenne Prime Order Fields**. Advances in Mathematics of Communications, 2020. Available at https://www.aimsciences.org/article/doi/10.3934/amc.2020113.

5. Kaushik Nath and Palash Sarkar. **Efficient Elliptic Curve Diffie-Hellman Computation at the 256-bit Security Level**. IET Information Security, 14(6):633640, 2020. Available at https://doi.org/10.1049/iet-ifs.2019.0620.

6. Kaushik Nath and Palash Sarkar. Constant Time Montgomery Ladder, 2020. **International Association for Cryptologic Research, Cryptology ePrint Archive**, Report 2020/956. Available at https://eprint.iacr.org/2020/956.

CHAPTER 2

Background and Related Work

Elliptic Curve Cryptography (ECC) is an approach to public-key cryptography which is based on the algebraic structure of elliptic curves over finite fields. Use of elliptic curves for cryptography was suggested independently by Koblitz [Kob87] and Miller [Mil85] in 1985 as a mechanism to implement public-key cryptography. Some good texts on elliptic curves and elliptic curve cryptography can be found at [Was08, Sil86, MvOV96, CFA+05].

Two basic applications of ECC are key agreement and signature schemes. There are several approaches for choosing a suitable curve on which a cryptographic primitive is computed. The shared-secret component of ECDH can be efficiently computed over a special form elliptic curve which are known as Montgomery form elliptic curves. While sequential implementations of shared secret over Montgomery curves have been common, very recently it was found that 4-way vector instructions can also be efficiently used for this computation [HEY20].

Finite field arithmetic constructs the fundamental base of a curve-based cryptosystem, the operation of which are used to construct the point arithmetic formulas of the curve. The point addition and doubling formulas exist right on the top of the field arithmetic layer and these formulas are used to compute the scalar multiplication on the curve. Apart from ECDH, scalar multiplication is also used in the signature schemes where a fixed-base scalar multiplication is needed for signature generation and a double-base scalar multiplication is needed for signature verification. While the key needed for ECDH can be computed on Montgomery curves, it has been found that computing the key is usually faster on the related (twisted) Edward form elliptic curves. Nevertheless, computing the key efficiently on Montgomery form curves have also been investigated and hence bears certain research interests. Efficiently computing the ECC primitives in hardware and software are both considered as major challenges in cryptographic research keeping in mind the huge practical importance of the corresponding security applications.

2.1 Elliptic Curves

We consider elliptic curves over prime order fields. Let $p \notin \{2, 3\}$ be a prime and $\mathbb{F}_p$ be the finite field of cardinality p. The algebraic closure of $\mathbb{F}_p$ is an algebraic extension of

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

Scalar multiplication. The ECDH protocol has two phases, *key generation* and *shared secret computation*. The major operation involved in the two phases is *scalar multiplication*. Given a point P on E and a non-negative integer n, the point nP is the n-fold addition of P. The operation of computing nP is called *scalar multiplication*. We will be interested in the case, where P is an $\mathbb{F}_p$-rational point of E. The key generation phase has the operation of the form $Q = dG$ where G is the generator of the cryptographic group and $d \in \mathbb{F}_p$ is the scalar. This is known as *fixed-base scalar multiplication*. The *shared secret computation* phase has the operation of the form $Q' = dQ$ where the input point Q is an arbitrary element in which case it is known as *variable-base scalar multiplication*.

2.1.1 Elliptic Curve Diffie-Hellman

The Diffie-Hellman protocol on elliptic curves is known as *Elliptic Curve Diffie-Hellman* (ECDH) which is defined over additive group of points on elliptic curves. The idea was suggested independently by Koblitz [Kob87] and Miller [Mil85] in 1985. The related hard problem in the elliptic curve group of points is known as *Elliptic Curve Discrete Logarithm Problem* (ECDLP). We illustrate how a shared key is established among two parties using an elliptic curve defined over prime order fields with the following example.

Key establishment protocol. An algorithmic form of the ECDH protocol is provided in Algorithm 2.1. Suppose, Alice and Bob are two parties who wish to establish a shared key amongst themselves through an insecure channel which may be eavesdropped by a third party, say Eve. Initially, the domain parameters $\langle p, a, b, G, \ell, h \rangle$ have to be agreed upon. Here, p is the prime specifying the base field, a, b are elements in $\mathbb{F}_p$ which denote the coefficient of the corresponding elliptic curve equation, $G = (x, y)$ is the base point, ℓ is the prime order of the group generated by G and h is the cofactor of G. Both Alice and Bob must have a key pair suitable for elliptic curve cryptography, consisting of a private key d which is a randomly selected integer in the interval $[1, \ell - 1]$ and a public key represented by a point Q where $Q = dG$. Let Alice's key pair be (d_A, Q_A) and Bob's key pair be (d_B, Q_B). Each party must know the other party's public key prior to execution of the protocol. Alice computes the point $(u, v) = d_A Q_B$ and Bob computes the point $(u, v) = d_B Q_A$. Then the x-coordinate of the computed point, u is considered as the shared secret between Alice and Bob. A symmetric key from u is derived using some hash-based key derivation function by most standardized protocols which are based on ECDH. The shared secrets calculated by Alice and Bob are equal because $d_A Q_B =$

$d_A(d_BG) = d_B(d_AG) = d_BQ_A$. The only information about her key that Alice initially exposes is her public key. So, no third party can determine Alice's private key unless that party can solve the ECDLP. Similarly, Bob's private key is also secure. No third can compute the secret which is shared amongst Alice and Bob unless that party can solve the ECDH problem. While the shared secret may be directly used as a key, it may be desirable to hash the secret to remove weak bits due to the Diffie-Hellman exchange.

Algorithm 2.1 Elliptic curve Diffie-Hellman protocol.

1: **function** ECDH(p, $E(\mathbb{F}_p)$, $G = (x, y)$)
2: **input:** p is a prime, $E(\mathbb{F}_p)$ is an elliptic curve, $G \in E(\mathbb{F}_p)$ is a point of order ℓ.
3: **output:** Shared secret between Alice and Bob.

Key Generation

	Alice	**Bob**
4:	Select the private key $d_A \leftarrow [1, \ell - 1]$	Select the private key $d_B \leftarrow [1, \ell - 1]$
5:	Compute the public key $Q_A \leftarrow d_AG$	Compute the public key $Q_B \leftarrow d_BG$

Shared secret computation

	Alice	**Bob**
6:	Send Q_A to Bob	Send Q_B to Alice
7:	Compute $P = (u, v) \leftarrow d_AQ_B$	Compute $P = (u, v) \leftarrow d_BQ_A$

8: **return** u
9: **end function**.

2.2 Efficiency and Security Requirements in ECC

The two major issues related to implementations of cryptographic software are *security* and *efficiency*. The security requirement of an implementation is the first priority and then appropriate techniques need to be applied to improve the overall performance of the software. We discuss these issues in moderate detail in the context of our work.

2.2.1 Security Requirements

ECDLP security and ECC security. There is a gap between the difficulty of ECDLP and desired security of ECC. There exist many attacks which are capable of breaking real-world ECC without solving ECDLP. There can be issues while implementing standard curves which can make the curve insecure. For example, the implementations may produce incorrect results for some rare curve points. The implementations may also leak secret data when the input is not a curve point or through branch/cache timing. These problems can be exploited by real attackers by taking advantage of the gap between ECDLP and real-world ECC. The *invalid curve attack* [BMM00] exploits ECDH implementations which validate received public keys inappropriately. This kind of attack belongs to a larger class of attacks known as *small subgroup key recovery attacks*. This class of attacks utilizes small subgroups of finite groups in order to extract non-ephemeral secret information. The MOV attack [MOV93] which is based on multiplicative transfers relies on the elliptic curves which have low embedding degrees and the Smart-ASS

attack [Sem98, SA98, Sma99] is based on additive transfers. We discuss the different security requirements below in brief. Details about the different security aspects can be found at [BLb, BCC+15, BCLN16, CLN15, BHH+14].

Our consideration of security is based on the recommendations provided in [BLb]. Let E be an elliptic curve over $\mathbb{F}_p$, where p is a prime. Let $n = \#E(\mathbb{F}_p)$ and $n_T = 2(p+1) - \#E(\mathbb{F}_p)$, i.e., n and n_T are the orders of $E(\mathbb{F}_p)$ and its twist. A twist is another elliptic curve which is isomorphic to E over an algebraic closure of $\mathbb{F}_p$. Let ℓ (resp. ℓ_T) be a prime such that $n = h \cdot \ell$ (resp. $n_T = h_T \cdot \ell_T$). Cryptography is done over a subgroup of $E(\mathbb{F}_p)$ of size ℓ. The parameters h and h_T are the co-factors of $E(\mathbb{F}_p)$ and its twist respectively. For a Montgomery curve, the curve order n is a multiple of 4. Using this fact along with $n + n_T = 2(p+1)$, it is easy to argue that if $p \equiv 3 \bmod 4$, then the minimum value of (h, h_T) is $(4,4)$, while if $p \equiv 1 \bmod 4$, then the minimum value of (h, h_T) is either $(8,4)$ or $(4,8)$. Let k (resp. k_T) be the smallest positive integer such that $\ell | p^k - 1$ (resp. $\ell_T | p^{k_T} - 1$). The parameters k and k_T are the embedding degrees of the curve and its twist respectively. The complex multiplication field discriminant D of E is defined in the following manner. Let $t = p + 1 - n$. By Hasse's theorem [Has36], $|t| \leq 2\sqrt{p}$ and in the cases that we considered $|t| < 2\sqrt{p}$, so that $t^2 - 4p$ is a negative integer; let s^2 be the largest square dividing $t^2 - 4p$; define $D = (t^2 - 4p)/s^2$, if $t^2 - 4p \bmod 4 = 1$ and $D = 4(t^2 - 4p)/s^2$, otherwise. Recommendations in [BLb] suggest choosing curves such that both h and h_T are small, k and k_T are large and also $|D|$ is large to ensure security against various known attacks. Note that if h and h_T are small, this implies that ℓ and ℓ_T are large. Considering twist security, the security level of a curve in terms of bits is defined to be $\frac{1}{2}\min(\log_2 \ell, \log_2 \ell_T)$.

Timing attacks. One of the major threat to the security of a cryptographic implementation are the so called timing attacks. A timing attack is a side channel attack through which the attacker tries to discover vulnerabilities in a cryptosystem by inspecting the time consumed by the implementations of different cryptographic algorithms in a computer. It is a kind of attack that exploits the data-dependent behavioral characteristics of the implementation of the cryptographic algorithm. Computers take different amounts of time to process different inputs and hence timing characteristics vary depending on the encryption key. The important factors causing time variability in implementations are compiler optimizations, branching and conditional statements, processor instructions, RAM and cache hits. A timing attack studies the various computation times of an implementation in a computer system and then makes use of statistical analysis to identify the right decryption key to gain access.

Timing attacks are usually overlooked in the design phase as they are very much dependent on the final implementation and can inadvertently creep in through different optimizations incorporated by the compiler. We can get rid of the attacks by designing constant-time functions in the target implementation and carefully test the final executable code.

Constant time implementations. Cryptographic algorithms can be implemented in a way which eliminates data dependent timing information of the implementation. Then the implementations are said to run in constant time on all secret inputs and the implementations are termed as *constant-time implementations*. Let us consider an implementation in which every call to a subroutine always takes exactly t seconds to execute, where

t is the maximum time it ever takes to execute that routine on every possible authorized input. In such an implementation, the timing of the algorithm leaks no information about the data supplied to that invocation. The drawback of this approach is that the time used for all executions becomes that of the worst-case performance of the function. Software implementations are considered to be constant-time in the following sense as stated in [Ber06b]: "*avoids all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings.*" The Github page [Aum] lists coding rules for implementations of cryptographic operations, and more generally for operations involving secret or sensitive values. A useful resource specifying guidelines to mitigate timing attacks against cryptographic implementations is [Zon].

Formal verification. In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. In cryptography formal verification is used to verify the security services of a cryptographic algorithm or protocol. It uses specific high level modeling specification to specify security solution and uses a back end formal verification tools to find out if there are any security breaches. The outcome of the formal verification provides information if the algorithm/protocol is safe or unsafe to use.

2.2.2 Efficiency Requirements

Field arithmetic. Since field multiplication and squaring are the most important operations, most of the developments target to optimize these two operations in the implementations. The schoolbook method is the most common method for integer multiplication and there are different ways of tackling it depending on the representation of the field elements and considering the underlying multiplier within the target architecture. If the field element is represented using a smaller base then strategies based on Karatsuba multiplication [K63] can be applied to speed-up the integer multiplication/squaring. However, deciding on Karatsuba over schoolbook and selecting an appropriate Karatsuba strategy needs a good amount of analysis before diving for a final implementation of the cryptographic primitive.

Field reduction is an important part of field multiplication/squaring which plays a crucial role in the efficiency of implementations. Reduction on the NIST recommended prime order fields are mainly based on the ideas proposed by Solinas [Sol]. A detailed study on the software implementations of the NIST curves has been addressed in [BHLM01]. Mersenne and pseudo-Mersenne primes help with fast and relatively simple methods to implement. They support compact and scalable implementations across different security levels and their performance scales well with the bit-length as the security level increases. The primes of the form $2^m \pm 2^n \pm 1$ such that $m > n$ are known as Solinas trinomial primes. Additionally, when the condition $m = 2n$ satisfies for such primes, the use of 2-way Karatsuba technique promotes efficient field multiplication/squaring for some large primes at high security level. The possible efficiency gain of the strategy depends on representations of field elements which have an even number of limbs. This condition may not be met with for the implementations that use saturated-limb representations or may necessitate the use of an extra limb and deal with unsaturated limb representation in some platforms. These primes are relatively rare in compar-

ison to the pseudo-Mersenne primes. Also, if the values of m, n are not aligned suitably to the computer word size, then saturated limb implementations with these primes may not be simple. As a consequence, the Solinas trinomial primes become quite suitable for some platforms, but not for all. This makes uniform selection of Solinas primes for ECC difficult. There are other primes of the form $2^m(2^n - r) - 1$ such that m, n, r are positive integers which are suitable for Montgomery multiplication [Mon85]. These primes are called Montgomery-friendly primes and efficient field arithmetic based on these primes can be implemented on various architectures. At the 128-bit security level Montgomery-friendly primes are very efficient, but they lose relative efficiency compared to pseudo-Mersenne primes at higher security levels. Efficient modular arithmetic using vector instructions is also challenging while working with the Montgomery-friendly primes. Some random primes have also been used by Brainpool curves [Brab]. But, software implementations of these curves are slower than the curves that use primes of special form. A detailed study on various primes can be found at [CLN15].

Curve arithmetic. Tackling the curve arithmetic properly also plays an important role in the efficiency gain of the implementations. The algorithms to compute the cryptographic schemes are essentially sequential in nature. Scheduling the operations within the algorithm without the affecting the final output and optimizing the number of basic instructions for the implementations provide noticeable benefits. In some cases the vector instructions of the target architecture can be intelligently exploited to vectorize the curve arithmetic accordingly and achieve a major speed-up. The combined curve arithmetic involved within the algorithm of scalar multiplication over Montgomery curves is known as the Montgomery ladder. This ladder computes the shared secret and has been attempted with 2-way/4-way vector instructions in several implementations of Curve25519 [CS09, BS12, Cho15, FL15, FHD19, HEY20]. Along with the Diffie-Hellman some of these implementations also addresses vector implementations of the Ed25519 signatures.

Efficient software. Cryptographic software is used by several applications for providing security to the corresponding users. These applications can run on different devices, each having different hardware architectures. While the underlying hardware is a factor behind the performance of these applications, the efficiency of the software used by them is also an important factor which contributes to the overall performance of the applications. Efficient software implementation of cryptographic schemes is an important area of research which has been heavily cultured for several years now. Generic implementation of software using a certain high-level language usually provide sub-optimal performance in terms of CPU-cycles consumed by the software while executed and the performance loss depends on the choice of language used. Considering this effect, it is preferable to use the instruction set of the target architecture directly bypassing the compiler to write programs that implement the required cryptographic primitives. Standard mnemonics of the target architecture can be used to fulfill this purpose leading to implementations that use assembly language. There are interfaces to access assembly instructions while developing programs using certain high-level languages, for example, the interface to write *inline assembly instructions* on the top of a C compiler. On the other hand there are interfaces like *intrinsic functions* provided by Intel which can almost access the assembly level instructions from high-level using C. But, on Intel architectures we

have found that, software developed using both these approaches under-perform compared to software which is developed using hand-written assembly instructions and the performance gain achieved by hand-written assembly instructions is quite significant for modern Intel architectures. Writing assembly programs by hand provides complete control to the developer over the set of registers available in the target machine and largely helps to efficiently schedule instructions implementing the required high level algorithms. However, working with registers directly to write efficient assembly programs is challenging and debugging turns out to be quite difficult. To develop a correct code one requires good knowledge of the target architecture and the corresponding assembly instructions.

2.3 Edwards and Twisted Edwards Curves

In 2007, Edwards [Edw07] introduced the family of elliptic curves known as Edwards curves. Applications of Edwards curves to cryptography were developed by Bernstein and Lange in 2008 [BL07b]. Twisted Edwards curves are a generalized form of the Edwards curves which was introduced by Bernstein, Birkner, Joye, Lange and Peters in 2008 [BBJ+08].

In [BL07b], Bernstein and Lange introduced a form of Edwards curves defined by $u^2 + v^2 = c^2(1 + du^2v^2)$ where $c, d \in \mathbb{F}_p$ with $cd(1 - dc^4) \neq 0$. In [BBJ+08], this form is generalized to twisted Edwards form defined by $E_{E,a,d} : au^2 + v^2 = 1 + du^2v^2$ where $a, d \in \mathbb{F}_p$ with $ad(a - d) \neq 0$. Edwards curves are then a special case of twisted Edwards curve where a can be rescaled to 1.

The fixed-base scalar multiplication which is used for key generation can be efficiently computed over Edwards curves. A double-base scalar multiplication is used while computing signatures and this can also be efficiently done using Edwards curves. The most efficient explicit formula for arithmetic on Edwards curve is due to [HWCD08]. As we have completely focused on Montgomery curves in this work we keep the discussion about Edwards curves brief.

2.4 Montgomery Curves and Montgomery Ladder

Let $A, B \in \mathbb{F}_p$ such that $B(A^2 - 4) \neq 0$. The Montgomery form elliptic curve $E_{M,A,B}$ is the set of all $(x, y) \in \overline{\mathbb{F}}_p \times \overline{\mathbb{F}}_p$ satisfying the equation $By^2 = x(x^2 + Ax + 1)$ along with the point at infinity denoted as ∞. This is called the affine form of the curve. The set of all $\mathbb{F}_p$-rational points of $E_{M,A,B}$, denoted as $E_{M,A,B}(\mathbb{F}_p)$ is the set of all $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying $By^2 = x(x^2 + Ax + 1)$ along with ∞. Under a suitably defined addition operation, $E_{M,A,B}(\mathbb{F}_p)$ is a group with ∞ as the identity element. It is known that the order of this group is a multiple of 4. In fact, it is usually possible to obtain A and B such that the order of $E_{M,A,B}$ is $4q$ for a prime q. For more extensive discussions of Montgomery curves and their arithmetic, we refer to [BL17, CS18, Mon87].

The most famous example of Montgomery curve is Curve25519 which was introduced by Bernstein [Ber06b]. For Curve25519, $p = 2^{255} - 19$, $A = 486662$ and $B = 1$. The other Montgomery curve which is part of TLS 1.3 is Curve448 which was introduced by Hamburg [Ham15]. For Curve448, $p = 2^{448} - 2^{224} - 1$, $A = 156326$ and $B = 1$. Apart from these two, other proposals of Montgomery curves can be found at [BLb].

2.4.1 Diffie-Hellman Key Agreement on Montgomery Curves

For computational efficiency, it is preferable to work with projective coordinates. The projective form of the Montgomery curve $E_{M,A,B}$ is $BY^2Z = X(X^2 + AXZ + Z^2)$.

For a point $P = [X : Y : Z]$ on $E_{M,A,B}$, the x-coordinate map $\mathbf{x}$ is the following [CS18]: $\mathbf{x}(P) = [X : Z]$ if $Z \neq 0$ and $\mathbf{x}(P) = [1 : 0]$ if $P = [0 : 1 : 0]$. Bernstein [Ber06a, Ber06b] introduced the map $\mathbf{x}_0$ as follows: $\mathbf{x}_0(X : Z) = XZ^{p-2}$ which is defined for all values of X and Z in $\mathbb{F}_p$.

Following Miller [Mil85], Montgomery [Mon87] and Bernstein [Ber06b], the Diffie-Hellman key agreement can be carried out on a Montgomery curve as follows. Let Q be a generator of a prime order subgroup of $E_{M,A,B}(\mathbb{F}_p)$. Alice chooses a secret key s and has public key $\mathbf{x}_0(sQ)$; Bob chooses a secret key t and has public key $\mathbf{x}_0(tQ)$. The shared secret key of Alice and Bob is $\mathbf{x}_0(stQ)$. Using classical computers, the best known method of obtaining $\mathbf{x}_0(stQ)$ from Q, $\mathbf{x}_0(sQ)$ and $\mathbf{x}_0(tQ)$ requires about $O(p^{1/2})$ time using the Pollards rho algorithm [Pol78].

2.4.2 Shared Secret Computation

The shared secret computation of both Alice and Bob is the following. Given $[X_1 : Z_1]$ corresponding to a point $P = [X_1 : Y_1 : Z_1]$ and a non-negative integer n, obtain $\mathbf{x}_0(nP)$. Montgomery [Mon87] introduced a variant of the usual double-and-add algorithm for the purpose of computing $\mathbf{x}_0(nP)$. Let $R = [X_2 : Y_2 : Z_2]$ and $S = [X_3 : Y_3 : Z_3]$ be such that $R - S = P$. Let $2R = [X_2' : Y_2' : Z_2']$ and $R + S = [X_3' : Y_3' : Z_3']$. Doubling corresponds to obtaining $[X_2', Z_2']$ from $[X_2 : Z_2]$ while differential addition corresponds to obtaining obtaining $[X_3' : Z_3']$ from $[X_1 : Z_1]$, $[X_2 : Z_2]$ and $[X_3 : Z_3]$. Based on Theorems B.1 and B.2 of [Ber06b], Montgomery's formulas for differential-add and doubling are as given in (2.1).

$$\left.\begin{array}{rcl} X_3' &=& Z_1((X_2 - Z_2)(X_3 + Z_3) + (X_2 + Z_2)(X_3 - Z_3))^2 \\ Z_3' &=& X_1((X_2 - Z_2)(X_3 + Z_3) - (X_2 + Z_2)(X_3 - Z_3))^2 \\ X_2' &=& (X_2 + Z_2)^2(X_2 - Z_2)^2 \\ Z_2' &=& 4X_2Z_2((X_2 - Z_2)^2 + \frac{A+2}{4}(4X_2Z_2)). \end{array}\right\} \tag{2.1}$$

Algorithm 2.2 Differential addition operation on Montgomery curve $E_{M,A,1}$

1: **function** DIFF-ADD($[X_1 : Z_1], [X_2 : Z_2], [X_3 : Z_3]$)
2: **input:** $[X_1 : Z_1], [X_2 : Z_2], [X_3, Z_3] \in E_{M,A,1}$.
3: **output:** $[X_3' : Z_3'] \in E_{M,A,1}$.
4: $R \leftarrow (X_2 - Z_2) + (X_3 + Z_3)$
5: $S \leftarrow (X_2 - Z_2) - (X_3 + Z_3)$
6: $X_3' \leftarrow Z_1(R + S)^2$
7: $Z_3' \leftarrow X_1(R - S)^2$
8: **return** $[X_3' : Z_3']$
9: **end function**.

Note that the parameter B is not required in (2.1). Assume $Z_1 = 1$. The quantity $4X_2Z_2$ in (2.1) is to be computed as $4X_2Z_2 = (X_2 + Z_2)^2 - (X_2 - Z_2)^2$. As a result,

the formulas in (2.1) require 5 multiplications, 4 squarings and 1 multiplication by the field constant $(A+2)/4$. The operations differential addition and doubling respectively denoted as DIFF-ADD(R, S, P) and DOUBLE(R) are described in Algorithms 2.2 and 2.3. In Algorithm 2.3, the constant a_{24} is equal to $(A+2)/4$. The operations DIFF-ADD and DOUBLE do not involve the Y-coordinate and the parameter B is not required in the computation.

Algorithm 2.3 Double operation on Montgomery curve $E_{M,A,1}$

function DOUBLE($[X_2 : Z_2]$)
input: $[X_2 : Z_2] \in E_{M,A,1}$.
output: $[X_2' : Z_2'] \in E_{M,A,1}$.
    $R \leftarrow (X_2 + Z_2)$
    $S \leftarrow (X_2 - Z_2)$
    $X_2' \leftarrow R^2 \cdot S^2$
    $Z_2' \leftarrow (R^2 - S^2)(S^2 + a_{24}(R^2 - S^2))$
    return $[X_2' : Z_2']$
end function.

Algorithm 2.4 Montgomery ladder

function MONTGOMERY-LADDER(P, n)
input: P is a projective point $[X_1 : Z_1] \in E_{M,A,B}$, n is an ℓ-bit scalar such that $n = (n_{\ell-1} n_{\ell-2} \ldots n_0)$.
output: x-coordinate of nP, the n-times scalar multiple of P.
    $R \leftarrow [1 : 0]$; $S \leftarrow P$
    for $i \leftarrow \ell - 1$ down to 0 **do**
        $(R, S) \leftarrow$ LADDER-STEP(R, S, P, n_i)
    end for
    let $R = [X : Z]$
    return XZ^{p-2}
end function.

Algorithm 2.5 Single ladder-step based on the differential add and double operations

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

2.4.3 Montgomery Ladder

The combined formula due to Montgomery (2.1) is known as the Montgomery ladder. which is built using the differential addition and doubling operations. The ladder is described in Algorithm 2.4 which uses Algorithm 2.5 as a sub-routine. In the ladder, Algorithm 2.2 is used to implement the operation DIFF-ADD and Algorithm 2.3 is used to implement the operation DOUBLE. The value of Z_1 in Algorithm 2.2 is considered as 1. Hence, the call LADDER-STEP(R, S, P) in Step 6 of Algorithm 2.4 is essentially LADDER-STEP(R, S, X_1).

2.5 Complete Base Point Coordinates

For Montgomery curves, given the x-coordinate, the two possible y-coordinates are given by $\pm\sqrt{x^3 + Ax^2 + x}$. For Edward curves, given the v-coordinate, the two possible u-coordinates are given by $\pm\sqrt{(1 - v^2)/(1 - dv^2)}$. In the two formulas the square-root computations are done over the underlying field.

2.6 Related Work

Finite field computations lie at the base of ECC operations and protocols. The underlying finite fields can be binary extension fields, prime fields, or prime extension fields. Fields of prime order are well suited for cryptographic applications and are primarily used in practice. For efficiency reasons primes of special shapes are considered while choosing a finite field instead of random primes.

In this thesis we work with elliptic curves defined over fields of prime order. To be more specific we concentrate on curves which are defined over fields based on Mersenne and pseudo-Mersenne primes. We also work with Curve448 which is defined over the field based on the Solinas trinomial prime $2^{448} - 2^{224} - 1$. Elliptic curves which are defined over other types of fields, like binary extension fields and quadratic extension fields have also been studied in the context of cryptography. Some examples of these kind of curves can be found at [HKM09, OLAR13, OLAR14, LS14, CHS14, CL15, FLS15, OLR16]. We do not get into the details of these curves since they are not related to our work. As mentioned in [BLb] several standard bodies have proposed various curves for use in elliptic curve cryptography. Some of these curves which are relevant in the context of our work are discussed below.

2.6.1 NIST Curves

In FIPS 186-4 standard [fSTa], *National Institute of Standards and Technology* (NIST) recommended ten elliptic curves for use in the *Elliptic Curve Digital Signature Algorithm* (ECDSA) targeting a variety of security levels. The recommendation contained a total of five prime curves and ten binary curves which were apparently chosen keeping *optimal security* and *implementation efficiency* in focus. Due to the progress of solving the ECDLP over binary curves [FPPR12], considerable doubts raised on the use of binary curves due to which prime curves became popular for deployment. The five primes of the NIST prime curves are generalized Mersenne primes of specific shapes which are defined as $p_{192} = 2^{192} - 2^{64} - 1$, $p_{224} = 2^{224} - 2^{96} + 1$, $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$,

$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ and $p_{521} = 2^{521} - 1$. The respective curves are called P-192, P-224, P-256, P-384, and P-521. They are also known as `nistp192`, `nistp224`, `nistp256`, `nistp384` and `nistp521`.

2.6.2 Curves Proposed by Certicom Research

In 2000, *Certicom Research* specified a set of elliptic curves [Res10] as standards in which these curves were also included and named as `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1` and `secp521r1`. In [Res10] a curve named `secp256k1` over the prime $2^{256} - 2^{32} - 977$ has also been defined which is used to generate the Bitcoin signatures.

2.6.3 Curve25519

In 2006, the famous Curve25519 targeting the 128-bit security level was proposed by Daniel J. Bernstein which was defined over the field corresponding to the prime $2^{255} - 19$. At the same security level the existing NIST standard is P-256 which is defined against the prime p_{256}. A major performance difference of Curve25519 over P-256 comes from the fact that $2^{255} - 19$ is a more structured prime in comparison to p_{256}, for which field arithmetic turns out to be substantially faster when done with $2^{255} - 19$. In particular, the field reduction becomes much faster with $2^{255} - 19$. Apart from field arithmetic there are other crucial reasons for which ECDH is reasonably faster when done with Curve25519 rather than NIST P-256. The same is true while computing the other cryptographic schemes with the related curves. Later on, Bernstein and Lange quoted in [BLa], "*NISTs ECC standards create (1) unnecessary losses of simplicity, security, and speed in ECC implementations and (2) unnecessary tensions between simplicity, security, and speed in ECC implementations.*" and discussed the potential problems with the NIST's ECC standards in detail. Curve25519 has been included in the Transport Layer Security(TLS) protocol. It has also been extensively deployed for various security applications [Cur].

2.6.4 Curves Based on Mersenne and Pseudo-Mersenne Primes

A prime is called Mersenne if it is of the form $2^m - 1$ for some $m > 0$. A prime is called pseudo-Mersenne if it is of the form $2^m - \delta$ for some $m > 0$ such that δ is relatively much small in comparison to 2^m. While pseudo-Mersenne primes are pretty much available, Mersenne primes are rare within a range of cryptographic interest. The first Mersenne prime which has been considered in ECC is $2^{127} - 1$ and the next is $2^{521} - 1$. It was first highlighted by Curve25519 that choosing a pseudo-Mersenne prime to define an elliptic curve provides substantial speed-up to compute the ECDH at 128-bit security level. Later on Bernstein et al [BDL+12] showed that a twisted Edwards curve (Ed25519) which is birationally equivalent to Curve25519 can be used to efficiently generate and verify elliptic curve signatures. Several other curves over pseudo-Mersenne primes have been proposed after Curve25519. In 2013, Bernstein et al proposed the Curve1174 [BHKL13] based on the prime $2^{251} - 9$ and Curve41417 [BCL14] at a higher security level which is based on the prime $2^{414} - 17$. In [ABGR13] Aranha et al described some general-purpose, high-efficiency elliptic curves tailored for security levels beyond 2^{128} based on (pseudo-)Mersenne primes. The *SafeCurves* project [BLb] of Bernstein and Lange analyzes and mentions about many such secure curves covering a wide range of security levels.

2.6.5 Curve448

As part of the Transport Layer Security (TLS) protocol, version 1.3 [TP18], the document RFC 7748 [LH16] specifies the Montgomery form elliptic curve Curve448 and its birationally equivalent Edwards form elliptic curve Edwards448. The curve Edwards448 was originally proposed in [Ham15] where it was named Ed448-Goldilocks. The underlying field for Curve448 and Edwards448 is $\mathbb{F}_p$ where p is the Solinas trinomial prime $2^{448} - 2^{224} - 1$ which promotes efficient Karatsuba multiplication.

2.7 Issues Related to Software Implementation

Implementations of cryptographic software is an important part of cryptography. We have worked with the modern Intel architectures. Here we discuss in brief about the issues which are related to software implementation.

2.7.1 Relevant Instructions of Modern Intel Architectures

Multiplication is considered fundamentally the most important and critical operation among all field operations while implementing the ECDH cryptographic protocol. The sequential or the vectorized multiplier available in the modern Intel architectures can be used to implement the multiplication operation over a field and hence the corresponding assembly instructions play a major role in implementing the basic primitives of the ECDH protocol. Nevertheless, there are also other assembly instructions which are also involved in the implementations. Below we briefly summarize the instructions available in modern Intel architectures which are most relevant to the various implementations of this thesis.

Sequential instructions. The 64-bit architecture of the Intel x86 processors has sixteen 64-bit registers, namely rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8, r9, r10, r11, r12, r13, r14, r15. Except rsp (which is the stack pointer), all other registers can be used for storing data and operating on them. There is a register named FLAGS, which consists of various available flags. We note two of these flags. Bit 0 of FLAGS is the carry flag CF and bit 11 of FLAGS is the overflow flag OF. Integer addition and multiplication affect the states of these two flags and are relevant to our work.

The basic 64-bit arithmetic operations in the x86 processors are `mul`, `imul`, `add` and `adc`. From the Broadwell processor onwards, Intel also provides another set of arithmetic instructions, namely, `mulx`, `adcx` and `adox`. The structure of multiplication and addition instructions and their operations are as follows.

`mul` src_2;	$\mathsf{rdx} : \mathsf{rax} \leftarrow \mathsf{src}_2 \cdot \mathsf{rax}$.
`imul` src_1, src_2, dst;	$\mathsf{dst} \leftarrow \mathsf{lsb}_{64}(\mathsf{src}_1 \cdot \mathsf{src}_2)$.
`add` src, dst;	$\mathsf{dst} \leftarrow \mathsf{src} + \mathsf{dst}$.
`adc` src, dst;	$\mathsf{dst} \leftarrow \mathsf{src} + \mathsf{dst} + \mathsf{CF}$.
`mulx` src_1, dst_ℓ, dst_h;	$\mathsf{dst}_h : \mathsf{dst}_\ell \leftarrow \mathsf{src}_1 \cdot \mathsf{rax}$.
`adcx` src, dst;	$\mathsf{dst} \leftarrow \mathsf{src} + \mathsf{dst} + \mathsf{CF}$.
`adox` src, dst;	$\mathsf{dst} \leftarrow \mathsf{src} + \mathsf{dst} + \mathsf{OF}$.

The operation `mulx` is available from the Haswell processor onwards; `adcx` and `adox` are available from the Broadwell processor onwards. Processors previous to Haswell had only `mul`, `imul`, `add` and `adc`.

The effect on the carry and the overflow flags for the above mentioned arithmetic operations are the following.

- `mul, imul, add` and `adc` affect *both* CF and OF;
- `mulx` affects neither CF nor OF;
- `adcx` affects only CF but not OF;
- `adox` affects only OF but not CF.

Suppose, there is an interleaved sequence of multiplications and additions to be performed. The additions generate carries which need to be taken into consideration for subsequent additions. The `mul` and `imul` instructions affect the carry flag and so the carry out of the previous addition gets lost. On the other hand, a sequence of `mulx` and `adc` instructions can efficiently perform such an interleaved sequence of multiplications and additions. The `mulx` instruction does not affect the carry flag and so the sequence of `adc` instructions can carry out the instructions using a *single carry chain*.

The combination of `mulx`, `adcx` and `adox` provides a more powerful tool. As mentioned above, the `mulx` instruction does not affect either CF or OF. A sequence of `adcx` instructions proceeds by using a carry chain using only CF, while a sequence of `adox` instructions proceeds by using a carry chain using only OF. So, in effect, it is possible to use two *independent* carry chains which we call *double carry chain*. This greatly facilitates arithmetic computations as we will see later.

Vector instructions. Modern day processor architectures, along with the general purpose registers provide a limited set of vector registers, in which multiple values of the same type having same bit-length can be stored in juxtaposition. For example, we can keep eight 32-bit or four 64-bit integers in a 256-bit vector register. Arithmetic and other kind of operations can be carried out on all these values simultaneously by the so called SIMD instructions. The Intel x86-64 architecture supports instructions on vectors of different data types, typically 8-bit, 16-bit, 32-bit and 64-bit integers, and single-precision and double-precision floating-point values. For Intel architectures, the 4-way vectorized instruction set is known as the `AVX2` instruction set. The modern Intel architectures like Haswell, Skylake include sixteen 256-bit registers, which are named `YMM0-YMM15`. It is possible to pack four 64-bit words into a single 256-bit `YMM` register and then use SIMD instructions to simultaneously work on the four 64-bit words. Exploiting such a facility, simultaneous computations of different cryptographic operations can be performed efficiently on the top of Intel processors.

The basic 4-way vectorized instructions in the x86 processors are `vpmuludq` and `vpaddq`. The synopsis of the instructions are as follows.

`vpmuludq` $\mathsf{ymm}_1, \mathsf{ymm}_2, \mathsf{ymm}_3$;
`vpaddq` $\mathsf{ymm}_1, \mathsf{ymm}_2, \mathsf{ymm}_3$;

The instruction `vpmuludq` multiplies the low unsigned 32-bit integers from each packed 64-bit element in ymm_1 and ymm_2, and stores the unsigned 64-bit results in ymm_3. Similarly, the instruction `vpaddq` adds packed 64-bit integers in ymm_1 and ymm_2, and stores the unsigned 64-bit results in ymm_3. The overflowed carry bits, if any, are lost due to the operations.

2.7.2 Implementation Types

All the implementations of this work have been developed using the Intel x86 64-bit assembly language. Depending on the kind instructions used in the implementations we will categorize them as follows.

`maax`-type implementations[1]: Primarily uses the instructions `mulx`, `adcx` and `adox`. It can also use the instructions `mul`, `add` and `adc`.

`mxaa`-type implementations: Primarily uses the instructions `mulx`, `add` and `adc`. It doesn't use the instructions `adcx` and `adox`.

`maa`-type implementations: Primarily uses the instructions `mul`, `add` and `adc`. It doesn't use the instructions `mulx`, `adcx` and `adox`.

`AVX2`-type implementations: Primarily uses the 4-way vector instructions `vpmuludq`, `vpaddq`, `vpsllq` and `vpsrlq`. It can also use the sequential instructions if needed.

2.7.3 Performance Measurement

The timing experiments in this thesis were carried out on a single core of Haswell and Skylake processors. During measurement of the CPU cycles, TurboBoost© and Hyper-Threading© features were turned off. The time stamp counter TSC was read from the CPU to RAX and RDX registers by the RDTSC instruction.

Platform specifications. The details of the hardware and software tools used in our software implementations are as follows. The compiler version essentially does not matter other than benchmarking and testing code, since it is being used essentially as an assembler.

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.

Measurement. The timings reported in the different tables across different chapters are the numbers of CPU cycles. For comparison, we provide the timings of the most efficient (to the best of our knowledge) and publicly available previous implementations. The timings of the previous implementations were obtained by downloading the relevant software and measuring the required CPU cycles on the same platforms where the present implementations have been measured. The complier flags with which we have complied these codes are identical to the ones with which we have compiled our code. In some cases these values are different from the timings reported in the original papers. Among the reasons for such differences are the possible differences in the micro-architectures of the same family of processors along with the difference in the (version of

[1]These implementations are not applicable for the Haswell architecture. Corresponding timing entries are marked with the "-" sign in different performance tables of this thesis.

the) compiler. Different methodologies for measuring timings could also lead to a difference, but two reasonable methodologies are expected to provide similar timings and so this issue would not lead to significant difference in timings. The speed-up percentage have been computed using the following formula.

$$\mathsf{sup} = 100 \times \frac{(\text{previous cycle count} - \text{present cycle count})}{\text{previous cycle count}}.$$

CHAPTER 3

Constant Time Montgomery Ladder

The Montgomery ladder described in Algorithm 2.4 which calls Algorithm 2.5 has a conditional branching 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. We discuss these methods below.

Algorithm 3.1 Constant time Montgomery ladder using conditional swap

1: **function** MONTGOMERY-LADDER-CSWAP(x_P, n)
2: **input:** An ℓ-bit scalar n and the x-coordinate x_P of a point P.
3: **output:** x-coordinate of nP, the n-times scalar multiple of P.
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` := 0
6: **for** $i \leftarrow \ell - 1$ down to 0 **do**
7: `bit` $\leftarrow$ bit at index i of n
8: $b \leftarrow$ `bit` $\oplus$ `prevbit`
9: `prevbit` $\leftarrow$ `bit`
10: $(\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle) \leftarrow$ CSWAP$(\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle, b)$
11: $(\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle) \leftarrow$ LADDER-STEP$(\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle, X_1)$
12: **end for**
13: **return** $X_2 Z_2^{p-2}$
14: **end function**.

3.1 Conditional Swap

Algorithm 2.4 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 3.1. This algorithm calls Algorithm 3.2 which contains the combined field operations of the differential add and doubling operations involved in a single step of the Montgomery ladder. It should be noted that y-coordinates are not used in Algorithm 3.1 because the Montgomery ladder is computed using only the x-coordinates. The z-coordinates are participating in the algorithm

because the computation is done using projective coordinates.

Let the x-coordinate of the point P on which scalar multiplication has to be performed be x_P. The z-coordinate of the point P is conventionally considered to be 1 for computational efficiency. Hence, we have $P = (X_1 : Z_1) = (x_P : 1)$. Since $Z_1 = 1$ the participation of Z_1 in the ladder computation is nullified according to (2.2) and so the variable is not explicitly used in Algorithm 3.1. According to Step 4 of Algorithm 2.4 the variable $R = (X_2 : Z_2)$ is initialized to $\mathbf{x}(\infty) = (1 : 0)$ and the variable $S = (X_3 : Z_3)$ is initialized to $\mathbf{x}(P) = (x_P : 1)$. This justifies the initializations done in Step 4 of Algorithm 3.1.

Algorithm 3.2 Ladder-step combined with the differential add and doubling operations

1: **function** LADDER-STEP($\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle, X_1$)
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: $T_5 \leftarrow T_1^2$
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: $X_3 \leftarrow T_1^2$
13: $T_2 \leftarrow T_2^2$
14: $Z_3 \leftarrow T_2 \cdot X_1$
15: $X_2 \leftarrow T_5 \cdot T_6$
16: $T_5 \leftarrow T_5 - T_6$
17: $T_1 \leftarrow a_{24} \cdot T_5$
18: $T_6 \leftarrow T_6 + T_1$
19: $Z_2 \leftarrow T_5 \cdot T_6$
20: **return** $(\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle)$
21: **end function**.

Algorithm 3.1 uses a subroutine CSWAP which performs a constant time conditional swap as follows: Function CSWAP($\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle, b$) swaps the pair of field elements $\langle X_2, Z_2\rangle$ and $\langle X_3, Z_3\rangle$ if $b = 1$, else not. We mention two methods for implementing CSWAP which have been described in the literature. Algorithm 3.3 describes a method given in [CS18] whereas Algorithm 3.4 describes a method given in [BL17].

3.2 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 [Ber06b] 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 [OLH+17]. We formalize the method used in the code as Algorithm 3.5. 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 3.3 Conditional swap using the operators and and xor

function CSWAP1($\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle, b$)
input: X_2, Z_2, X_3, Z_3 are field elements encoded as ℓ-bit strings and b is a bit.
output: The pairs $\langle X_2, Z_2\rangle$ and $\langle X_3, Z_3\rangle$ are swapped if $b = 1$, else not.
    mask $\leftarrow (bb\ldots b)_\ell$
    $T_1 \leftarrow$ 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 $(\langle T_3, T_4\rangle, \langle T_5, T_6\rangle)$
end function.

Algorithm 3.4 Conditional swap using the operators +, - and ·

function CSWAP2($\langle X_2, Z_2\rangle, \langle X_3, Z_3\rangle, b$)
input: X_2, Z_2, X_3, Z_3 are field elements encoded as ℓ-bit strings and b is a bit.
output: The pairs $\langle X_2, Z_2\rangle$ and $\langle X_3, Z_3\rangle$ 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 $(\langle T_1, T_2\rangle, \langle T_3, T_4\rangle)$
end function.

Following Bernstein's suggestion mentioned above, CSELECT(X, Y, b) can be executed in constant time using the substitution rule

$$X \leftarrow (1 - b)X + bY.$$

Later, we consider the issue of implementing CSELECT in constant time using the `cmov` instruction available on Intel processors.

Remark 3.1. *There is an implementation*[1] *of constant time conditional branching for microcontrollers which works by swapping the pointers to field elements instead of swapping the field*

[1] https://munacl.cryptojedi.org/curve25519-cortexm0.shtml

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[2] *on processors which have cache memory or non-constant time memory access.*

Algorithm 3.5 Constant time Montgomery ladder using conditional selection

function MONTGOMERY-LADDER-CSELECT(x_P, n)
input: An ℓ-bit scalar n and the x-coordinate x_P of a point P.
output: x-coordinate of nP, the n-times scalar multiple of P.
    $X_1 \leftarrow x_P; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
    `prevbit` $\leftarrow 0$
    for $i \leftarrow \ell - 1$ down to 0 **do**
        `bit` $\leftarrow$ bit at index i of n
        $b \leftarrow$ `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(T_1, T_3, b)
        CSELECT(T_2, T_4, b)
        $T_1 \leftarrow T_1^2$
        $T_2 \leftarrow T_2^2$
        $X_3 \leftarrow T_5 + T_6$
        $Z_3 \leftarrow T_5 - T_6$
        $X_3 \leftarrow X_3^2$
        $Z_3 \leftarrow Z_3^2$
        $X_2 \leftarrow T_2$
        $Z_2 \leftarrow T_1 - T_2$
        $T_2 \leftarrow ((A + 2)/4) \cdot Z_2$
        $T_2 \leftarrow T_2 + X_2$
        $X_2 \leftarrow X_2 \cdot T_1$
        $Z_2 \leftarrow Z_2 \cdot T_2$
        $Z_3 \leftarrow Z_3 \cdot X_1$
    end for
    return $X_2 Z_2^{p-2}$
end function.

3.2.1 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

[2]This issue was pointed out to us independently by Daniel J. Bernstein and Diego Aranha.

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.

```
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.

```
cmp   $1, %rsi

movq  0(%rdi), %rsi                 movq  32(%rdi), %rsi
movq  64(%rdi), %rdx                movq  96(%rdi), %rdx
mov   %rsi, %rcx                    mov   %rsi, %rcx
cmov  %rdx, %rsi                    cmov  %rdx, %rsi
cmov  %rcx, %rdx                    cmov  %rcx, %rdx
movq  %rsi, 0(%rdi)                 movq  %rsi, 32(%rdi)
movq  %rdx, 64(%rdi)                movq  %rdx, 96(%rdi)

movq  8(%rdi), %rsi                 movq  40(%rdi), %rsi
movq  72(%rdi), %rdx                movq  104(%rdi), %rdx
mov   %rsi, %rcx                    mov   %rsi, %rcx
cmov  %rdx, %rsi                    cmov  %rdx, %rsi
cmov  %rcx, %rdx                    cmov  %rcx, %rdx
movq  %rsi, 8(%rdi)                 movq  %rsi, 40(%rdi)
movq  %rdx, 72(%rdi)                movq  %rdx, 104(%rdi)

movq  16(%rdi), %rsi                movq  48(%rdi), %rsi
movq  80(%rdi), %rdx                movq  112(%rdi), %rdx
mov   %rsi, %rcx                    mov   %rsi, %rcx
cmov  %rdx, %rsi                    cmov  %rdx, %rsi
cmov  %rcx, %rdx                    cmov  %rcx, %rdx
movq  %rsi, 16(%rdi)                movq  %rsi, 48(%rdi)
movq  %rdx, 80(%rdi)                movq  %rdx, 112(%rdi)

movq  24(%rdi), %rsi                movq  56(%rdi), %rsi
movq  88(%rdi), %rdx                movq  120(%rdi), %rdx
mov   %rsi, %rcx                    mov   %rsi, %rcx
cmov  %rdx, %rsi                    cmov  %rdx, %rsi
cmov  %rcx, %rdx                    cmov  %rcx, %rdx
movq  %rsi, 24(%rdi)                movq  %rsi, 56(%rdi)
movq  %rdx, 88(%rdi)                movq  %rdx, 120(%rdi)
```

Figure 3.1: Assembly code to implement constant time conditional swap. Taken from the amd64-64 implementation of [BDL+12].

Implementation of CSWAP. Consider the CSWAP based ladder given in Algorithm 3.1. The concrete implementation of CSWAP that we discuss here is from the `amd64-64` implementation[3] of Curve25519 accompanying the work [BDL+12].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` [BDL+12] implementation are shown in Figure 3.1. Except the `cmp`, the effect of all the other instructions in the first column of Figure 3.1 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 3.1 has 32 `movq`, 8 `mov` and 16 `cmov` operations.

```
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"
  );
}
```

(a) 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)
```

(b) Generated assembly code of CSelect

Figure 3.2: Assembly code to implement constant time conditional select for Curve25519 taken from the implementation of [OLH+17].

[3] https://github.com/floodyberry/supercop/blob/master/crypto_scalarmult/curve25519/amd64-64/work_cswap.s (accessed on August 5, 2020).

Implementation of CSELECT. Consider the CSELECT based Montgomery ladder given in Algorithm 3.5. The 64-bit implementation of Curve25519[4] provided with [OLH+17] has an implementation of CSELECT. The inline assembly code taken from the implementation of [OLH+17] is provided in the left column and the generated assembly is shown in the right column of Figure 3.2. 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$(X, Y,$ `select,`$)$. 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.

```
cmp   $1, %rcx

movq  0(%rsp), %r8
movq  8(%rsp), %r9
movq  16(%rsp), %r10
movq  24(%rsp), %r11

movq  64(%rsp), %r12
movq  72(%rsp), %r13
movq  80(%rsp), %r14
movq  88(%rsp), %r15

cmov  %r12, %r8
cmov  %r13, %r9
cmov  %r14, %r10
cmov  %r15, %r11

movq  %r8, 0(%rsp)
movq  %r9, 8(%rsp)
movq  %r10, 16(%rsp)
movq  %r11, 24(%rsp)
```

```
movq  32(%rsp), %r8
movq  40(%rsp), %r9
movq  48(%rsp), %r10
movq  56(%rsp), %r11

movq  96(%rsp), %r12
movq  104(%rsp), %r13
movq  112(%rsp), %r14
movq  120(%rsp), %r15

cmov  %r12, %r8
cmov  %r13, %r9
cmov  %r14, %r10
cmov  %r15, %r11

movq  %r8, 32(%rsp)
movq  %r9, 40(%rsp)
movq  %r10, 48(%rsp)
movq  %r11, 56(%rsp)
```

Figure 3.3: Assembly code to implement CSELECT for X25519.

The assembly code shown in Figure 3.2 implements one CSELECT operation. So, implementation of the two CSELECT operations in Algorithm 3.2 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 3.5 is significantly smaller than the number of data movement operations to implement the CSWAP operation.

Remark 3.2. *In the 64-bit implementation of Curve448[5] provided with [OLH+17], 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 3.3. The generated assembly does*

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

[5] https://github.com/armfazh/rfc7748_precomputed/blob/master/src/x448_x64.c (accessed on August 5, 2020).

not use any conditional move instructions and the number of instructions required to implement the conditional branching is fairly large.

Algorithm 3.6 Constant time Montgomery ladder using conditional selection

1: **function** MONTGOMERY-LADDER-CSELECT-NEW(x_P, n)
2: **input:** An ℓ-bit scalar n and the x-coordinate x_P of a point P.
3: **output:** x-coordinate of nP, the n-times scalar multiple of P.
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** $i \leftarrow \ell - 1$ 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: $b \leftarrow$ `bit` $\oplus$ `prevbit`
15: `prevbit` $\leftarrow$ `bit`
16: CSELECT(T_1, T_3, b)
17: CSELECT(T_2, T_4, b)
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: $Z_3 \leftarrow Z_3^2$
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: $X_2 \leftarrow T_1 \cdot T_2$
28: $Z_2 \leftarrow T_3 \cdot T_4$
29: $Z_3 \leftarrow Z_3 \cdot X_1$
30: **end for**
31: **return** $X_2 Z_2^{p-2}$
32: **end function**.

3.3 Modified Constant Time Conditional Branching

We rearrange the sequence of steps given in Algorithm 3.5 and formalize a variant of it, which we denote as MONT-LADDER-CSELECT-NEW and is shown in Algorithm 3.6. The new algorithm needs fewer number of temporary variables and the copy statement in Step 24 has been omitted. The correctness of MONT-LADDER-CSELECT-NEW follows from the correctness of MONT-LADDER-CSELECT.

3.4 New Assembly Implementation of CSELECT

Figure 3.2 described the previous assembly implementation of CSELECT. We would like to highlight a difference in the manner by which the `cmov` instructions have been used in Figure 3.1 and the `cmovnz` instructions have been used in Figure 3.2. In Figure 3.1 all the `cmov` instructions have their operands to be registers, while in Figure 3.2 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[6]. 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 3.3. This require a total of 24 `movq` and 8 `cmov` instructions.

[6]In an email communication, Bernstein indicated that there are no known variable-time problems with the `cmov` instruction that is purely based on registers. Latencies of the `cmov` instructions whose one operand is a memory location are missing in the Agner Fog's instruction tables. We communicated with Agner Fog regarding this and he commented that the corresponding timing is constant in case of a cache hit.

Part I

New Techniques for Efficient Implementations

CHAPTER 4

Efficient Arithmetic in (Pseudo-)Mersenne Prime Order Fields

4.1 Introduction

Two basic applications of curve-based cryptography are key agreement and signature schemes. Both of these schemes require scalar multiplications. The computation consists of two steps. In the first step, the scalar multiplication is computed using projective coordinates and in the second step, an inversion in the underlying finite field is required to convert the output to affine coordinates. Computing scalar multiplication using projective coordinates requires efficient algorithms for finite field arithmetic, especially multiplication and squaring. For performing an inversion, there are two approaches. Using Fermat's little theorem (FLT), an inversion can be computed using an exponentiation which again requires efficient algorithms for multiplication and squaring. Alternatively, inversion can be computed using Extended Euclidean algorithm which requires additions and subtractions.

In this chapter we carry out a comprehensive study of multiplication and squaring algorithms over fields whose order is either a Mersenne or a pseudo-Mersenne prime. We concentrate on single multiplication and squaring algorithms and so the aspect of simultaneous multiplications using single instruction multiple data (SIMD) instructions is not considered in the present work.

Field multiplication and squaring have two broad phases, namely, a multiplication phase and a reduction phase. Let the prime $p = 2^m - \delta$. Elements of $\mathbb{F}_p$ fit within an m-bit string. Such an m-bit string is formatted into κ binary strings where the first $(\kappa - 1)$ strings are each η bits long and the last string is ν bits long with $0 < \nu \leq \eta$. Following the usual convention, we call each of the individual κ binary strings as limbs. For 64-bit arithmetic, each limb fits into a 64-bit word. Two kinds of representations have been considered in the literature. In the first kind of representation, $\eta = 64$, and so the limbs (except possibly for the last one) are packed tightly into 64-bit words. In the second kind of representation, $\eta < 64$, and so the 64-bit words containing the limbs have some free or redundant bits. We call the first kind of representation to be *saturated limb representation* and the second kind to be *unsaturated limb representation*.

We provide various algorithms for multiplication/squaring and reduction using both the saturated and the unsaturated limb representations. A brief summary of these con-

tributions is as follows.

4.1.1 Multiplication/Squaring for Saturated Limb Representation

We describe two sets of algorithms with each set consisting of an algorithm for multiplication and one for squaring. The first set of algorithms (which we call mulSLDCC/sqrSLDCC) generalizes the multiplication/squaring algorithms in the Intel white papers [OGG13, OGGF12] to work for $64i$-bit integers for any $i \geq 2$. These algorithms use two independent carry chains and can be implemented in the newer generation of processors. The second set of algorithms (which we call mulSLa/sqrSLa) do not use double carry chains and can be implemented across all generation of processors. These algorithms combine an initial step of the reduction with the multiplication. The idea behind mulSLa/sqrSLa has not appeared earlier in the literature.

4.1.2 Multiplication/Squaring for Unsaturated Limb Representation

We describe two sets of algorithms. The first set of algorithms (which we call mulUSL/sqrUSL) generalize the ideas used in [BDL+12] for the prime $2^{255} - 19$. These algorithms, however, lead to overflow for certain primes such as the Bitcoin prime $2^{256} - 2^{32} - 977$. To handle such overflow issues, we describe a second set of algorithms (which we call mulUSLa/sqrUSLa) which have not appeared earlier in the literature.

4.1.3 Reduction for Saturated Limb Representation

We describe four reduction algorithms, namely reduceSLMP, reduceSLPMP, reduceSLPMPa and reduceSL. Algorithms reduceSLMP, reduceSLPMP and reduceSLPMPa reduce the outputs of mulSLDCC/sqrSLDCC. Specifically, reduceSLMP works for all Mersenne primes and is a generalization of the ideas used in [BCLS14] for the prime $2^{127} - 1$. Algorithm reduceSLPMP works for a large class of pseudo-Mersenne primes and has not appeared earlier. Algorithm reduceSLPMPa works for a large class of pseudo-Mersenne primes and is a generalization of the ideas for 4-limb representation used in [BDL+12] for the prime $2^{255} - 19$. Algorithm reduceSL reduces the output mulSLa/sqrSLa and has not appeared earlier in the literature.

4.1.4 Reduction for Unsaturated Limb Representation

We describe three reduction algorithms, namely, reduceUSL, reduceUSLA and reduceUSLB. Algorithm reduceUSL works for a large class of pseudo-Mersenne primes and is a generalization of the ideas for 5-limb representation used in [Cho15][1] for the prime $2^{255} - 19$. For certain primes, reduceUSLA is more efficient than reduceUSL and generalizes ideas used in [BDL+12] for the prime $2^{255} - 19$. For certain other primes, reduceUSLB is more efficient than both reduceUSL and reduceUSLA. Compared to reduceUSLA, Algorithm reduceUSLB leaves an extra bit in most of the limbs of the partially reduced output. The idea behind reduceUSLB has not appeared earlier in the literature.

In Table 4.1, for each algorithm presented in this work, we state whether it is new, or the earlier work that it generalizes.

[1] See also https://github.com/floodyberry/supercop/tree/master/crypto_scalarmult/curve25519/amd64-51.

There are two key theoretical features of our work.

1. While previous works have developed code for a single prime, we describe the algorithms in their full generality.

2. For each reduction algorithm, we state precise theorems about their correctness and provide detailed proofs of correctness.

Works on implementation of Curve25519 provide reduction methods for the prime $2^{255} - 19$ [Ber06b, Cho15, DHH+15, FL15], though without proofs of correctness. We note that there are excellent discussions on Barrett and Montgomery reductions available in the literature [CFA+05, MvOV96, HMV03, BD20] which also describe reduction methods for specific primes. On the other hand, to the best of our knowledge, the reduction algorithms and the proofs that we describe do not appear in the literature.

The second aspect of our work is in the efficient implementation of the various multiplication and squaring algorithms.

4.1.5 Assembly Implementations

All the algorithms described in this paper have been implemented in 64-bit assembly for Intel processors. The implementations are divided into two groups, namely maa and maax. For implementations in the maa group, the only arithmetic instructions used are `mul, imul, add` and `adc`, while for implementations in the maax group, the arithmetic instructions `mulx, adcx` and `adox` are also used. These second set of instructions are available from the Broadwell processor onwards.

4.1.6 Library for Field Arithmetic

Through our efficient 64-bit assembly implementations, we provide a library of field multiplication and squarings in cryptographically relevant prime order fields targeting the modern Intel processors. The efficient field arithmetic library can be used for the development of fast projective-coordinate-based scalar multiplication over appropriate elliptic curves. The implementations of the library can also be used to develop other efficient applications of cryptography where the prime order fields of this work are relevant.

We have considered a total of fourteen primes which include all previously proposed cryptographically relevant (pseudo-)Mersenne primes at various security levels. These primes are shown in Table 4.2. For the prime $2^{255} - 19$, we have found earlier implementations of both maa and maax types and for five of the other primes, we have found implementations of maa type. So, for eight of the fourteen primes, we provide the first maa type implementation and for thirteen of the fourteen primes, we provide the first maax type implementations.

Timings of the field operations for the new implementations and the existing implementations have been measured on the Haswell and Skylake processors. For each prime where a previous implementation is available, our implementation improves upon such previous implementations. A summary of the various speed-ups that were observed is as follows. Further details are provided later.

maa type implementations:

algorithm	feature
mulSLDCC/sqrSLDCC	generalizes [OGG13, OGGF12]
mulSLa/sqrSLa	new
mulUSL/sqrUSL	generalizes [BDL+12]
mulUSLa/sqrUSLa	new
reduceSLMP	generalizes [BCLS14]
reduceSLPMP	new
reduceSLPMPa	generalizes [BDL+12, 4-limb]
reduceSL	new
reduceUSL	generalizes [Cho15, 5-limb]
reduceUSLA	generalizes [BDL+12, 5-limb]
reduceUSLB	new

Table 4.1: The various algorithms for multiplication/squaring and reduction described in this paper.

On Haswell: Speed-ups of about 10%, 3%, 4%, 36%, 2% and 18% were observed for the primes $2^{127}-1$, $2^{251}-9$, $2^{255}-19$, $2^{256}-2^{32}-977$, $2^{266}-3$ and $2^{521}-1$ respectively.

On Skylake: Speed-ups of about 10%, 12%, 4%, 28%, 8% and 16% were observed for the primes $2^{127}-1$, $2^{251}-9$, $2^{255}-19$, $2^{256}-2^{32}-977$, $2^{266}-3$ and $2^{521}-1$ respectively.

`maax` type implementations: On the Skylake processor, a speed-up of about 26% was observed for the prime $2^{255}-19$.

The speed-ups obtained for $2^{255}-19$ and $2^{256}-2^{32}-977$ are particularly important. The prime $2^{255}-19$ defines the underlying field for the famous Curve25519 while the prime $2^{256}-2^{32}-977$ defines the underlying field for the curve secp256k1 which is used in the Bitcoin protocol. The above mentioned improvements arises from use of new reduction algorithms as well as hand optimizations of the corresponding assembly codes.

Source codes of all our implementations are publicly available at the link

`https://github.com/kn-cs/pmp-farith`.

Remark 4.1. *Let x be an ℓ-bit non-negative integer and $\eta \leq \ell$. The operation $x \bmod 2^{\eta}$ returns $\mathsf{lsb}_{\eta}(x)$, i.e., the η least significant bits of x, whereas the operation $\lfloor x/2^{\eta} \rfloor$ returns the $\ell-\eta$ most significant bits of x. It will be helpful to keep this simple observation in mind while going through the various algorithms given later.*

4.2 Representation of Elements in $\mathbb{F}_p$

Let η be a positive integer and $\theta = 2^{\eta}$. Consider the expression

$$h(\theta) \quad = \quad h_0 + h_1\theta + \cdots + h_{k-1}\theta^{k-1} \tag{4.1}$$

prime	curve(s)
$2^{127}-1$	Kummer_2 [BCLS14], FourQ [CL15]
$2^{221}-3$	M-221 [ABGR13]
$2^{222}-117$	E-222 [ABGR13]
$2^{251}-9$	Curve1174 [ABGR13], KL2519(81,20) [KS20]
$2^{255}-19$	Curve25519 [Ber06b], KL25519(82,77) [KS20]
$2^{256}-2^{32}-977$	secp256k1 [Res10]
$2^{266}-3$	KL2663(260,139) [KS20]
$2^{382}-105$	E-382 [ABGR13]
$2^{383}-187$	M-383 [ABGR13]
$2^{414}-17$	Curve41417 [BCL14]
$2^{511}-187$	M-511 [ABGR13]
$2^{512}-569$	-
$2^{521}-1$	P-521 [fSTb], E-521 [ABGR13]
$2^{607}-1$	-

Table 4.2: The primes considered in this work.

where $h_0, h_1, \ldots, h_{k-1}$ are non-negative integers. The polynomial $h(\theta)$ is given by the vector of coefficients $\langle h_0, h_1, \ldots, h_{k-1}\rangle$. We will call these coefficients to be the limbs of the polynomial. Note that we do not insist that the coefficients are less than 2^η; in fact, at intermediate steps, the coefficients will not necessarily be less than 2^η.

Given an m-bit integer, by a (κ, η, ν)-representation we will mean a κ-limb representation, where the first $\kappa - 1$ limbs are η bits long and the last limb (least significant limb) is ν bits long.

Example 4.1. *For the prime $p = 2^{251} - 9$, the elements in the underlying field in base $\theta = 2^{51}$ have the (κ, η, ν)-representation such that $\kappa = 5, \eta = 51, \nu = 47$.*

Proposition 4.1. *Let x and y be two m-bit integers both having a (κ, η, ν)-representation and let $z = x \cdot y$. Then z has a (κ', η, ν')-representation where*

$$\kappa' = 2\kappa - 1,\ \nu' = 2\nu \text{ if } 0 < \nu \leq \eta/2; \quad \text{and} \quad \kappa' = 2\kappa,\ \nu' = 2\nu - \eta \text{ if } \eta/2 < \nu \leq \eta.$$

Proof. We have $m = \eta(\kappa - 1) + \nu$. The number of bits in z is at most $2m$ and we may write $2m = \eta(2\kappa - 2) + 2\nu$. If $0 < \nu \leq \eta/2$, then z has a $(2\kappa - 1)$-limb representation where the first $2\kappa - 2$ limbs are each η bits long and the last limb is $\nu' = 2\nu$ bits long. On the other hand, if $\eta/2 < \nu \leq \eta$, then we may write $2m = \eta(2\kappa - 1) + 2\nu - \eta$ and so z has a 2κ-limb representation where the first $2\kappa - 1$ limbs are each η bits long and the last limb is $2\nu - \eta$ bits long. (Note that $\eta/2 < \nu \leq \eta$ implies $0 < 2\nu - \eta \leq \eta$.) □

Consider a (κ, η, ν)-representation of an m-bit integer w. Suppose that ω-bit arithmetic will be used for implementation. Using ω-bit arithmetic, w will be represented by κ ω-bit words $w_0, w_1 \ldots, w_{\kappa-1}$ such that the binary representation of w is given by

$$\mathsf{lsb}_\nu(w_{\kappa-1})||\mathsf{lsb}_\eta(w_{\kappa-2})||\cdots||\mathsf{lsb}_\eta(w_0). \tag{4.2}$$

Here $\mathsf{lsb}_i(x)$ denotes the i least significant bits of the binary string x and $0 < \nu \leq \eta \leq \omega$. Depending on the value of η, we identify two kinds of representations.

Saturated limb representation. In this case $\eta = \omega$. So, each of the ω-bit words $w_0, w_1, \ldots, w_{\kappa-2}$ are "saturated" in the sense that there are no leading redundant bits in these words. The ω-bit word $w_{\kappa-1}$ is saturated or unsaturated depending on whether $\nu = \eta$ or $\nu < \eta$ respectively.

Unsaturated limb representation. In this case $\eta < \omega$. So, each of the ω-bit words $w_0, w_1, \ldots, w_{\kappa-1}$ are "unsaturated" in the sense that they contain some leading redundant bits. The word $w_{\kappa-1}$ contains the same or more leading redundant bits according to whether $\nu = \eta$ or $\nu < \eta$ respectively.

Remark 4.2. In this work, we will consider 64-bit arithmetic and so $\omega = 64$. *The general ideas of the algorithms apply to arbitrary values of ω. The actual value of $\omega = 64$ is used at some places in the (non-)overflow analysis of the correctness proofs.*

The primes p that we consider are of the form

$$p \quad = \quad 2^m - \delta, \tag{4.3}$$

where δ is sufficiently small. Given (κ, η, ν)-representation of m-bit integers, we have

$$2^{\eta(\kappa-1)+\nu} \quad = \quad 2^m \equiv \delta \mod p. \tag{4.4}$$

For future reference, we define

$$c_p \quad = \quad 2^{\eta-\nu}\delta. \tag{4.5}$$

The different values of m, δ, κ, η and ν for the various primes p considered in this work are given in Table 4.3 [2]. For each prime, two sets of values of κ, η and ν are provided, one using saturated limb representation and the other using unsaturated limb representation.

4.2.1 Saturated versus Unsaturated Limb Representation

For processors which provide support for `maax` instructions, field arithmetic using the saturated limb representation is more efficient than using unsaturated limb representation. On the other hand, for older processors which have support for only `maa` type instructions, field arithmetic using saturated limb representation becomes quite complex due to the problem of handling carries. For such processors, in general, using the unsaturated limb representation turns out to be more efficient.

4.2.2 Representation of Primes

Suppose m-bit integers have a (κ, η, ν)-representation and $\theta = 2^\eta$. The prime $p = 2^m - \delta$ is represented as the polynomial $\mathfrak{p}(\theta)$, defined as

$$\mathfrak{p}(\theta) \quad = \quad p_0 + p_1\theta + \cdots + p_{\kappa-1}\theta^{\kappa-1}, \tag{4.6}$$

where $p_0 = 2^\eta - \delta (\delta < 2^\eta)$, $p_1 = p_2 = \cdots = p_{\kappa-2} = 2^\eta - 1$, and $p_{\kappa-1} = 2^\nu - 1$.

[2] For the prime $2^{414} - 17$ an unsaturated limb representation with $(\kappa = 7, \eta = 60, \nu = 54)$ is also possible. The corresponding multiplication, squaring and inversion algorithms are then mulUSLa, sqrUSLa and invUSLa respectively. We discuss this idea in Section 4.7.1.

prime	m	δ	unsaturated limb			saturated limb		
			κ	η	ν	κ	η	ν
$2^{127}-1$	127	1	3	43	41	2	64	63
$2^{221}-3$	221	3	4	56	53	4	64	29
$2^{222}-117$	222	117	4	56	54	4	64	30
$2^{251}-9$	251	9	5	51	47	4	64	59
$2^{255}-19$	255	19	5	51	51	4	64	63
$2^{256}-2^{32}-977$	256	$2^{32}+977$	5	52	48	4	64	64
$2^{266}-3$	266	3	5	54	50	5	64	10
$2^{382}-105$	382	105	7	55	52	6	64	62
$2^{383}-187$	383	187	7	55	53	6	64	63
$2^{414}-17$	414	17	8	52	50	7	64	30
$2^{511}-187$	511	187	9	57	55	8	64	63
$2^{512}-569$	512	569	9	57	56	8	64	64
$2^{521}-1$	521	1	9	58	57	9	64	9
$2^{607}-1$	607	1	10	61	58	10	64	31

Table 4.3: The primes considered in this work and their saturated and unsaturated limb representations.

An element in $\mathbb{F}_p$ is represented as a polynomial $f(\theta)$ of degree at most $\kappa-1$, defined as

$$f(\theta) \quad = \quad f_0 + f_1\theta + \cdots + f_{\kappa-1}\theta^{\kappa-1}, \tag{4.7}$$

where $0 \leq f_0, f_1, \ldots, f_{\kappa-1} < 2^{\eta}$ and $0 \leq f_{\kappa-1} < 2^{\nu}$.

4.2.3 Unique Representation of Field Elements

It should be noted that the expressions $f(\theta)$ are in one-one correspondence with the integers $0, 1, \ldots, 2^m-1$. This leads to a non-unique representation of δ elements in $\mathbb{F}_p$, i.e., the elements $0, 1, \ldots, \delta-1$ are also represented as $2^m-\delta, 2^m-\delta+1, \ldots, 2^m-1$. The non-unique representation does not affect the correctness of the computations. At the end of the computation, the final result is converted to a unique representation by subtracting the prime p from the field element and discarding the result if a carry is generated after the subtraction. This conditional subtraction has been implemented in constant time.

4.2.4 Inversion in $\mathbb{F}_p$

Fermat's little theorem states that for a prime p and any non-zero $a \in \mathbb{F}_p$, $a^{(p-1)} \equiv 1 \bmod p$. So, $a^{(p-2)}$ is the inverse of a in $\mathbb{F}_p$. Thus, the computation of inverse of any non-zero element in $\mathbb{F}_p$ reduces to the problem of exponentiating a to the power $(p-2)$. The standard way to compute this exponentiation is to use the square-and-multiply algorithm which can be implemented through a addition chain. Usually, we can design better addition chain algorithms than the default one by careful inspections. Since, the value $(p-2)$ is fixed, the numbers of squarings and multiplications are fixed and do

not depend on the value of a. So, if squaring and multiplication in $\mathbb{F}_p$ are constant time algorithms, then the exponentiation based inversion is also a constant time algorithm.

For (pseudo-)Mersenne primes, the binary representation of $(p-2)$ is sparse. So, only a few multiplications are required in the square-and-multiply algorithm to compute $a^{(p-2)}$. It is due to this reason that the FLT-based inversion is attractive for (pseudo-)Mersenne primes. For dense primes, the number of multiplications can be reduced by classical recoding techniques which only leak the exponent $(p-2)$, that is already public.

Till recently, FLT based inversion was considered to be the more efficient of the two methods for inversion in (pseudo-)Mersenne prime order fields. On the face of it, this seems counter-intuitive since the Fermat based approach uses multiplications and squarings, whereas the Euclid based approach uses only additions and logical shifts. The reason for Fermat based approach being faster for (pseudo-)Mersenne primes seems to be based on two factors, namely, the number of iterations in the Euclid based approach is higher and the availability of very fast multiplication instructions in modern processors. Defying this reasoning, a recent work by Bernstein and Yang [BY19] showed that the Euclid based approach could indeed be faster for (pseudo-)Mersenne primes.

Remark 4.3. *The integer multiplication and squaring algorithms considered in this paper are based on the schoolbook method. For larger primes, it may be more efficient to consider Karatsuba's algorithm to implement integer multiplication and squaring. Deciding upon a Karatsuba strategy would require determining how to divide the operand and the recursion depth after which schoolbook would be applied. So, determining the best Karatsuba strategy would depend on the specific prime and also the target architecture. We note, on the other hand, that the reduction algorithms that we describe do not depend on whether schoolbook or Karatsuba strategy is used for integer multiplication and squaring.*

As a case study, we have implemented the field operations for the prime $p = 2^{607} - 1$ following a $(5+5)$-Karatsuba decomposition using the `maax` *instructions in Skylake; for the subproblems of size 5 we apply directly the schoolbook method. The obtained timings for field multiplication, squaring and inversion are* $(175, 142, 86292)$. *In contrast, the timings* $(159, 129, 74442)$ *obtained using a schoolbook method over 10 limbs are better. So, it is not necessarily true that a Karatsuba strategy will be faster for larger primes.*

In the rest of the paper, we will focus entirely on field multiplication and squaring. In comparison, field addition, negation and subtraction are much faster. We note one important difference in these operations which arises from the representation of the elements. For saturated limb representations, field addition/negation/subtraction can be implemented using `add/adc/sub/sbb`. *On the other hand, for unsaturated limb representations, implementation of these operations also require shift operations.*

4.3 Overview of the Algorithms

All algorithms in this work are described keeping 64-bit arithmetic in mind.

4.3.1 Meanings of Various Abbreviations

SL : saturated limb;
USL : unsaturated limb;
SCC : single carry chain;

DCC : double (independent) carry chains;
MP : Mersenne prime;
PMP : pseudo-Mersenne prime.

Brief descriptions of the tasks of the different algorithms that we consider are given below.

4.3.2 Algorithms for the Saturated Limb Representation

mulSCC: Multiply a word whose value is less than 2^{64} to an integer given by a saturated limb representation using a single carry chain.

mulSLDCC: Multiply two integers given in saturated limb representations using double (independent) carry chains.

sqrSLDCC: Square an integer given in saturated limb representation using double carry chains.

reduceSLMP: Reduction algorithm to be applied to the outputs of mulSLDCC or sqrSLDCC when the underlying prime is a Mersenne prime.

reduceSLPMP: Reduction algorithm to be applied to the outputs of mulSLDCC or sqrSLDCC when the underlying prime is a pseudo-Mersenne prime.

reduceSLPMPa: A partial reduction algorithm to be applied to the outputs of mulSLDCC or sqrSLDCC when the underlying prime is a pseudo-Mersenne prime.

mulSLa: Multiply two integers given in saturated limb representations and perform an initial step of the reduction.

sqrSLa: Square an integer given in saturated limb representation and perform an initial step of the reduction.

reduceSL: A generic reduction algorithm to be applied to the outputs of mulSLa/sqrSLa.

farith-SLa: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulSLa, sqrSLa and reduceSL.

farith-SLMP: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulSL, sqrSL and reduceSLMP. See the remark below for mulSL and sqrSL.

farithx-SLMP: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulSLDCC, sqrSLDCC and reduceSLMP.

farithx-SLPMP: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulSLDCC, sqrSLDCC and reduceSLPMP.

Remark 4.4. *The output of* mulSLDCC *is the product of the two integers and the output of* sqrSLDCC *is the square of an integer. Algorithms* mulSLDCC/sqrSLDCC *utilize double carry chains to perform the computations. The product of two integers in the saturated limb representation can also be performed without using double carry chains and similarly, the square of an*

integer in the saturated limb representation can be performed without using double carry chains. For the prime $2^{255} - 19$*, the 4-limb algorithms in [BDL+12] perform such computations. The 4-limb algorithms in [BDL+12] can be extended to work for arbitrary limb representations. We will denote the resulting multiplication and squaring algorithms by* mulSL *and* sqrSL*. Note that* mulSL/sqrSL *are different from* mulSLa/sqrSLa *since* mulSLa/sqrSLa *also perform an initial step of reduction while this is not done by* mulSL/sqrSL*.*

4.3.3 Algorithms for the Unsaturated Limb Representation

mulUSL: Multiply two integers given in unsaturated limb representations and perform an initial step of the reduction.

sqrUSL: Square an integer given in unsaturated limb representation and perform an initial step of the reduction.

mulUSLa: Multiply two integers given in unsaturated limb representations and perform an initial step of the reduction. This is a variant of mulUSL which is to be used when mulUSL leads to overflows.

sqrUSLa: Square an integer given in unsaturated limb representation and perform an initial step of the reduction. This is a variant of sqrUSL which is to be used when sqrUSL leads to overflows.

reduceUSL: A generic reduction algorithm to be applied to the outputs of mulUSL/sqrUSL or mulUSLa/sqrUSLa.

reduceUSLA: An algorithm to be applied to the outputs of mulUSL/sqrUSL or mulUSLa/sqrUSLa when the prime is of type A. For such primes, reduceUSLA is more efficient than reduceUSL.

reduceUSLB: An algorithm to be applied to the outputs of mulUSL/sqrUSL or mulUSLa/sqrUSLa when the prime is of type B. For such primes, reduceUSLB is more efficient than reduceUSL or reduceUSLA.

farith-USL: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulUSL, sqrUSL and reduceUSL.

farith-USLA: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulUSLA, sqrSLA and reduceUSLA.

farithx-USLB: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulUSLB, sqrUSLB and reduceUSLB.

farithx-USLa: Denotes the algorithm triplet which computes a field multiplication, squaring and inverse using mulUSLa, sqrUSLa and reduceUSLA.

The implementations of the various algorithms are divided into two groups.

Algorithms in the maa **setting.** The algorithms farith-SLa, farith-USL, farith-USLA, farith-USLB and farith-USLa have been implemented in assembly using only the instructions `mul, imul, add` and `adc` to do arithmetic. These implementations are downward compatible with previous generations of Intel processors.

Algorithms in the maax **setting.** The implementations of the algorithms farithx-SLMP and farithx-SLPMP also use the instructions `mulx`, `adcx` and `adox` for doing arithmetic. These implementations work on the Broadwell and later generation processors.

4.3.4 Descriptions of the Algorithms

We describe a number of algorithms. The descriptions of the algorithms are at a fairly high level. They are provided in a form which make it easy to understand the algorithms and present the proofs of correctness. For the various reduction algorithms, the input is considered to be a polynomial $h^{(0)}(\theta)$, with $\theta = 2^\eta$, and the output is $h^{(k)}(\theta)$ for some $k \geq 1$, such that

$$h^{(0)}(\theta) \equiv h^{(1)}(\theta) \equiv \cdots \equiv h^{(k)}(\theta) \bmod p.$$

Conceptually, the algorithm proceeds in stages where the i-th stage computes $h^{(i)}(\theta)$ from $h^{(i-1)}(\theta)$ for $i = 1, 2, \ldots, k$. The proofs of correctness show that $h^{(i)}(\theta) \equiv h^{(i-1)}(\theta) \bmod p$ and also provide precise bounds on the coefficients of $h^{(i)}(\theta)$. In order to define the polynomials $h^{(i)}(\theta)$, the algorithms use certain statements which simply copy some of the coefficients of $h^{(i-1)}(\theta)$ to $h^{(i)}(\theta)$. Also, for ease of reference in the proofs, certain temporary variables are indexed by the loop counter creating the impression that a number of such variables are required, whereas in actual implementation one variable is sufficient.

For actual assembly implementation, it is desirable to use the registers as much as possible and also to avoid using load/store instructions to the extent possible. As such, the strict distinction between the various stages of the algorithm is not maintained so that some of the copy statements become redundant and are not implemented. Also, the use of temporary variables is minimized as much as possible and such variables are reused whenever feasible. Modulo such routine simplifications, the implementations follow the general flow of the algorithms. For each of the algorithms, we provide efficient assembly implementations for a number of primes. Studying the code together with the algorithm descriptions will make the associations between them clear and lead to a better understanding of the code.

4.4 Integer Multiplication/Squaring for Saturated Limb Representation Using Independent Carry Chains

Let c be an η-bit constant, $\theta = 2^\eta$ and $f(\theta)$ be a polynomial in θ of degree at most $d-1$ whose coefficients are from $\mathbb{Z}_\theta$. A basic step in the multiplication and squaring algorithms is the computation $c \cdot f(\theta)$. The result is a polynomial $h(\theta)$ of degree at most d and whose coefficients are from $\mathbb{Z}_\theta$. Function mulSCC given in Algorithm 4.1 performs this computation.

The multiplication in Step 4 of mulSCC can be completed using a single `mulx` operation. The **for loop** in Steps 6-9 uses an interleaved sequence of multiplications and additions. The additions involve a carry propagation through the variable $\mathfrak{c}$. Step 7 can be completed using a single `mulx` instruction while Step 8 can be completed using an `adc` instruction. The single bit value of the carry variable $\mathfrak{c}$ is carried through CF. Note that `mulx` does not affect CF and so it is possible to use `adc` instructions to implement the

Algorithm 4.1 Multiply $f(\theta)$ with an η-bit constant c; $\theta = 2^{\eta}, \eta = 64$.

1: **function** mulSCC($f(\theta), c$)
2: **input:** $f(\theta) = f_0 + f_1\theta + \cdots + f_{d-1}\theta^{d-1}$, c, where $0 \leq c, f_0, f_1, \ldots, f_{d-1} < 2^{\eta}$ and $d \geq 1$.
3: **output:** $h(\theta) = h_0 + h_1\theta + \cdots + h_d\theta^d = c \cdot f$, where $0 \leq h_0, h_1, \ldots, h_d < 2^{\eta}$.
4: $t \leftarrow c \cdot f_0$; $h_0 \leftarrow t \bmod 2^{\eta}$; $h_1 \leftarrow \lfloor t/2^{\eta} \rfloor$
5: $\mathfrak{c} \leftarrow 0$
6: **for** $i \leftarrow 1$ to $d-1$ **do**
7: $t \leftarrow c \cdot f_i$; $h_{i+1} \leftarrow \lfloor t/2^{\eta} \rfloor$; $\ell \leftarrow t \bmod 2^{\eta}$
8: $t \leftarrow h_i + \ell + \mathfrak{c}$; $h_i \leftarrow t \bmod 2^{\eta}$; $\mathfrak{c} \leftarrow \lfloor t/2^{\eta} \rfloor$
9: **end for**
10: $h_d \leftarrow h_d + \mathfrak{c}$
11: **return** $h(\theta) = h_0 + h_1\theta + \cdots + h_d\theta^d$
12: **end function**.

carry chain. Since the `mul` instruction affects CF, using `mul` instead of `mulx` would not have allowed an efficient implementation of the carry chain using `adc` instructions.

The single carry chain of mulSCC is pictorially depicted in Figure 4.1. The horizontal rectangular boxes denote the two η-bit quantities arising out of the multiplication shown at the left end of the corresponding row. The vertical oval shape encapsulates the quantities that are added using the `adc` instruction. These consist of two η-bit quantities and the carry $\mathfrak{c}$ whose value is available in the CF flag.

In general, it is required to multiply two integers written as polynomials $f(\theta)$ and $g(\theta)$ having degrees d and e respectively. This is performed using Function mulSLDCC given in Algorithm 4.2. The algorithm is written in a manner so that there are two independent carry chains in action which can exploit the two different flags CF and OF of the Intel architecture through the `adcx` and `adox` instructions. This is illustrated in Figure 4.2.

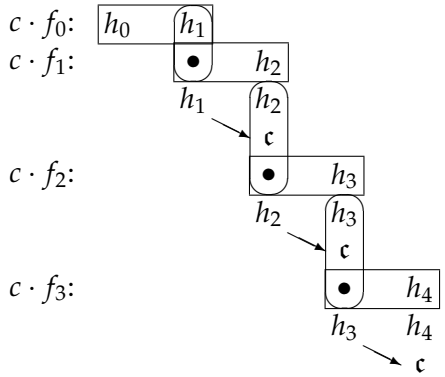

Figure 4.1: Single carry chain for mulSCC.

The multiplications in mulSLDCC are independent and can be performed simultaneously. The two additions are also independent and can be performed simultaneously. The additions, however, depend on the result of the previous multiplication. The `mulx`

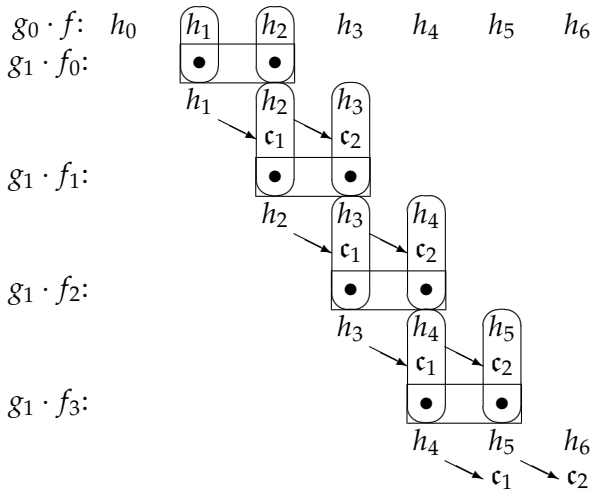

Figure 4.2: Two independent carry chains for mulSLDCC.

Algorithm 4.2 Multiply $f(\theta)$ and $g(\theta)$; $\theta = 2^\eta, \eta = 64$.

1: **function** mulSLDCC($f(\theta), g(\theta)$)
2: **input:** $f(\theta) = f_0 + f_1\theta + \cdots + f_{d-1}\theta^{d-1}$ and $g(\theta) = g_0 + g_1\theta + \cdots + g_{e-1}\theta^{e-1}$, where $0 \leq f_0, f_1, \ldots, f_{d-1}, g_0, g_1, \ldots, g_{e-1} < 2^\eta$, and $d \geq e \geq 2$.
3: **output:** $h(\theta) = h_0 + h_1\theta + \cdots + h_{d+e-1}\theta^{d+e-1} = f \cdot g$, where $0 \leq h_0, h_1, \ldots, h_{d+e-1} < 2^\eta$.
4: $h_0 + h_1\theta + \cdots + h_d\theta^d \leftarrow$ mulSCC($f(\theta), g_0$)
5: **for** $i \leftarrow 1$ to $e - 1$ **do**
6: $c_1 \leftarrow 0$; $c_2 \leftarrow 0$
7: **for** $j \leftarrow 0$ to $d - 1$ **do**
8: $t \leftarrow g_i \cdot f_j$
9: $r \leftarrow h_{i+j} + (t \bmod 2^\eta) + c_1$
10: $s \leftarrow h_{i+j+1} + \lfloor t/2^\eta \rfloor + c_2$
11: $h_{i+j} \leftarrow r \bmod 2^\eta$; $c_1 \leftarrow \lfloor r/2^\eta \rfloor$
12: $h_{i+j+1} \leftarrow s \bmod 2^\eta$; $c_2 \leftarrow \lfloor s/2^\eta \rfloor$
13: **end for**
14: $h_{i+j+1} \leftarrow h_{i+j+1} + c_1$
15: **end for**
16: **return** $h(\theta) = h_0 + h_1\theta + \cdots + h_{d+e-1}\theta^{d+e-1}$
17: **end function**.

instruction is used to perform the multiplications. This instruction does not affect either CF or OF. The two independent carry chains arising in Function mulSLDCC (and as illustrated in Figure 4.2) are implemented using a sequence of adcx and adox instructions.

The adcx instruction uses CF to propagate the carry while the adox instruction uses OF to propagate the carry.

Algorithm 4.3 Square $f(\theta); \theta = 2^{\eta}, \eta = 64$.

1: **function** sqrSLDCC($f(\theta)$)
2: **input:** $f(\theta) = f_0 + f_1\theta + \cdots + f_{d-1}\theta^{d-1}$ such that $0 \leq f_0, f_1, \ldots, f_{d-1} < 2^{\eta}$.
3: **output:** $h(\theta) = h_0 + h_1\theta + \cdots + h_{2d-1}\theta^{2d-1} = f^2$ such that $0 \leq h_0, h_1, \ldots, h_{2d-1} < 2^{\eta}$.
4: $h_1 + h_2\theta + \cdots + h_d\theta^d \leftarrow$ mulSCC$(f_1 + f_2\theta + \cdots + f_{d-1}\theta^{d-2}, f_0)$
5: **for** $i \leftarrow 1$ to $d-3$ **do**
6: $h_{d+i} \leftarrow 0$
7: **end for**
8: **for** $i \leftarrow 1$ to $d-3$ **do**
9: $\mathfrak{c}_1 \leftarrow 0;\ \mathfrak{c}_2 \leftarrow 0$
10: **for** $j \leftarrow i+1$ to $d-1$ **do**
11: $t \leftarrow f_i \cdot f_j$
12: $r \leftarrow h_{i+j} + (t \bmod 2^{\eta}) + \mathfrak{c}_1; s \leftarrow h_{i+j+1} + \lfloor t/2^{\eta} \rfloor + \mathfrak{c}_2$
13: $h_{i+j} \leftarrow r \bmod 2^{\eta}; \quad \mathfrak{c}_1 \leftarrow \lfloor r/2^{\eta} \rfloor$
14: $h_{i+j+1} \leftarrow s \bmod 2^{\eta};\ \mathfrak{c}_2 \leftarrow \lfloor s/2^{\eta} \rfloor$
15: **end for**
16: $h_{i+j+1} \leftarrow h_{i+j+1} + \mathfrak{c}_1$
17: **end for**
18: $t \leftarrow f_{d-1} \cdot f_{d-2}$
19: $r \leftarrow h_{2d-3} + (t \bmod 2^{\eta}); h_{2d-3} \leftarrow r \bmod 2^{\eta}$
20: $\mathfrak{c} \leftarrow \lfloor r/2^{\eta} \rfloor; h_{2d-2} \leftarrow \lfloor t/2^{\eta} \rfloor + \mathfrak{c}$
21: $h_{2d-1} \leftarrow \lfloor h_{2d-2}/2^{\eta-1} \rfloor$
22: **for** $i \leftarrow 2d-1$ down to 2 **do**
23: $h_i \leftarrow (2h_i \bmod 2^{\eta}) + \lfloor h_{i-1}/2^{\eta-1} \rfloor$
24: **end for**
25: $h_1 \leftarrow 2h_1 \bmod 2^{\eta}$
26: $t \leftarrow f_0 \cdot f_0$
27: $h_0 \leftarrow t \bmod 2^{\eta}; t \leftarrow h_1 + \lfloor t/2^{\eta} \rfloor$
28: $h_1 \leftarrow t \bmod 2^{\eta}; \mathfrak{c} \leftarrow \lfloor t/2^{\eta} \rfloor$
29: **for** $i \leftarrow 1$ to $d-1$ **do**
30: $t \leftarrow f_i \cdot f_i$
31: $r \leftarrow h_{2i} + (t \bmod 2^{\eta}) + \mathfrak{c}; h_{2i} \leftarrow r \bmod 2^{\eta}; \mathfrak{c} \leftarrow \lfloor r/2^{\eta} \rfloor$
32: $r \leftarrow h_{2i+1} + \lfloor t/2^{\eta} \rfloor + \mathfrak{c}; h_{2i+1} \leftarrow r \bmod 2^{\eta}; \mathfrak{c} \leftarrow \lfloor r/2^{\eta} \rfloor$
33: **end for**
34: **return** $h(\theta) = h_0 + h_1\theta + \ldots + h_{2d-1}\theta^{2d-1}$
35: **end function**.

Intel processors have multiple ALUs. So, the independent additions can be simultaneously executed on two separate ALUs. Further, subject to availability, the independent multiplications can be scheduled on separate ALUs and the multiplications and additions can be scheduled in a pipelined manner on separate ALUs such that the time for addition does not cause any delay in the overall computation.

Squaring an integer of the form $f(\theta)$ can be performed by setting both inputs in mulSLDCC to be equal to $f(\theta)$. On the other hand, it is possible to reduce the number

of multiplications. Function sqrSLDCC given in Algorithm 4.3 squares $f(\theta)$. It consists of three phases. In the first phase, the cross product terms are computed; in the second phase, these are multiplied by 2 (which is a doubling operation); and in the third phase, the squares of the coefficients of $f(\theta)$ are computed. Multiplications are performed in the first and the third phase. In the first phase, two independent carry chains arise in a manner similar to that of mulSLDCC. These two chains are implemented using the instructions `mulx`, `adcx` and `adox`. In the third phase, there is a single carry chain which is implemented using the instructions `mulx`, `add` and `adc` in a manner similar to that used in mulSCC.

4.5 Reduction Using Saturated Limb Representation

For $p = 2^m - \delta$, elements of $\mathbb{F}_p$ are m-bit integers and have a (κ, η, ν)-representation. In this section we consider saturated limb representation and so $\eta = 64$. As mentioned earlier, a multiplication/squaring in $\mathbb{F}_p$ consists of an integer multiplication/squaring followed by a reduction. The integer multiplication and squaring operations are respectively performed by the functions mulSLDCC and sqrSLDCC described in Section 4.4. In both cases two m-bit integers having (κ, η, ν)-representations are multiplied and the product is a $2m$-bit integer having (κ', η, ν')-representation where the values of κ' and ν' are given by Proposition 4.1. The task of the reduction is to reduce the product modulo p to an m-bit integer which again has a (κ, η, ν)-representation. Such a reduction maintains the bit-sizes of all the limbs of the reduced field element and would be termed as full reduction. It has to be noted that the elements might not be unique after full reduction and should be converted to a unique representation as discussed in Section 4.2.3.

We provide two reduction algorithms using the saturated limb representation, namely reduceSLMP which works for Mersenne primes and reduceSLPMP which works for pseudo-Mersenne primes. A Mersenne prime is also a pseudo-Mersenne prime and so reduceSLPMP also works for Mersenne primes, but for such primes it will be slower than reduceSLMP. On the other hand, reduceSLMP does not work for pseudo-Mersenne primes.

4.5.1 Mersenne Primes

Let $p = 2^m - 1$ and suppose m-bit integers have a (κ, η, ν)-representation. Function reduceSLMP given in Algorithm 4.4 takes as input the output of either mulSLDCC or sqrSLDCC and outputs an m-bit integer in an (κ, η, ν)-representation which is congruent to the input modulo p.

The following result states the correctness of reduceSLMP.

Theorem 4.1. *Let $p = 2^m - 1$ be a Mersenne prime and let $\kappa \geq 2$, η and ν be such that m-bit integers have a (κ, η, ν)-representation. Suppose that the input $h^{(0)}(\theta)$ to* reduceSLMP *is the output of either* mulSLDCC$(f(\theta), g(\theta))$ *or* sqrSLDCC$(f(\theta))$ *where $f(\theta)$ and $g(\theta)$ represent m-bit integers having (κ, η, ν)-representations. Then the output $h^{(3)}(\theta)$ of* reduceSLMP *has a (κ, η, ν)-representation and $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

Proof. Since m-bit integers have a (κ, η, ν)-representation, we have $m = \eta(\kappa - 1) + \nu$ with $0 < \nu \leq \eta$. If $\nu = \eta$ then $m = \kappa\eta$ and so $p = 2^m - 1 = 2^{\kappa\eta} - 1 = (2^\kappa)^\eta - 1$, which has a factor $2^\kappa - 1$ contradicting that p is a prime. So, if p is a Mersenne prime then it necessarily follows that $\nu < \eta$.

Algorithm 4.4 Reduction for saturated limb representation. Performs reduction modulo p, where $p = 2^m - 1$ is a Mersenne prime; $\theta = 2^\eta$.

function reduceSLMP($h^{(0)}(\theta)$)
input: $h^{(0)}(\theta)$.
output: $h^{(3)}(\theta)$.
  for $i \leftarrow 2\kappa - 1$ down to κ **do**
    $h_i^{(1)} \leftarrow (2^{\eta-\nu} h_i^{(0)}) \bmod 2^\eta + \lfloor h_{i-1}^{(0)} / 2^\nu \rfloor$; $h_{i-\kappa}^{(1)} \leftarrow h_{i-\kappa}^{(0)}$
  end for
  $h_{\kappa-1}^{(1)} \leftarrow h_{\kappa-1}^{(1)} \bmod 2^\nu$
  $t \leftarrow h_0^{(1)} + h_\kappa^{(1)}$; $h_0^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
  for $i \leftarrow 1$ to $\kappa - 2$ **do**
    $t \leftarrow h_i^{(1)} + h_{\kappa+i}^{(1)} + \mathfrak{c}$; $h_i^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
  end for
  $h_{\kappa-1}^{(2)} \leftarrow h_{\kappa-1}^{(1)} + h_{2\kappa-1}^{(1)} + \mathfrak{c}$
  $t \leftarrow h_0^{(2)} + \lfloor h_{\kappa-1}^{(2)} / 2^\nu \rfloor$; $h_0^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
  for $i \leftarrow 1$ to $\kappa - 2$ **do**
    $t \leftarrow h_i^{(2)} + \mathfrak{c}$; $h_i^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
  end for
  $h_{\kappa-1}^{(3)} \leftarrow h_{\kappa-1}^{(2)} \bmod 2^\nu + \mathfrak{c}$
  FULL REDUCTION: **return** $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1}^{(3)}\theta^{\kappa-1}$
end function.

The input $h^{(0)}(\theta)$ to reduceSLMP is the product of two m-bit integers each having a (κ, η, ν)-representation. From Proposition 4.1, the $2m$-bit integer $h^{(0)}(\theta)$ has a (κ', η, ν')-representation where the values of κ' and ν' are given by Proposition 4.1. Using these values we have the following bounds on the coefficients of $h^{(0)}(\theta)$.

$$\begin{array}{lll} 0 \le h_0^{(0)}, h_1^{(0)}, \ldots, h_{2\kappa-3}^{(0)} < 2^\eta; \text{ and} & & \\ 0 \le h_{2\kappa-2}^{(0)} < 2^{2\nu},\ h_{2\kappa-1}^{(0)} = 0 & \text{if} \quad 0 < \nu \le \eta/2; & \\ 0 \le h_{2\kappa-2}^{(0)} < 2^\eta,\ 0 \le h_{2\kappa-1}^{(0)} < 2^{2\nu-\eta} & \text{if} \quad \eta/2 < \nu < \eta. & \end{array} \tag{4.8}$$

The input $h^{(0)}(\theta)$ can be written as

$$\begin{aligned} h^{(0)}(\theta) &= h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} + h_\kappa^{(0)}\theta^\kappa + \\ &\quad h_{\kappa+1}^{(0)}\theta^{\kappa+1} + \cdots + h_{2\kappa-1}^{(0)}\theta^{2\kappa-1}, \\ &= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}) + \\ &\quad (h_\kappa^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1})\theta^\kappa, \\ &\equiv (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}) + \\ &\quad (h_\kappa^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1})2^{\eta-\nu} \mod p, \end{aligned} \tag{4.9}$$

since $\delta = 1$ and using (4.4) we have $\theta^\kappa = 2^{\kappa\eta} = 2^{(\kappa-1)\eta+\nu} \cdot 2^{\eta-\nu} = 2^m \cdot 2^{\eta-\nu} \equiv 2^{\eta-\nu} \bmod p$. For $j = \kappa - 1, \kappa, \ldots, 2\kappa - 2$, define

$$h_j^{(0)} = h_{j,0}^{(0)} + h_{j,1}^{(0)} 2^\nu, \text{ where } h_{j,0}^{(0)} = h_j^{(0)} \bmod 2^\nu, \text{ and } h_{j,1}^{(0)} = \lfloor h_j^{(0)} / 2^\nu \rfloor. \tag{4.10}$$

Using (4.8), we have the following bounds on $h_{j,0}^{(0)}$ and $h_{j,1}^{(0)}$.

Claim 4.1. $0 \leq h_{j,0}^{(0)} < 2^{\nu}$ *for* $j = \kappa-1, \kappa, \ldots, 2\kappa-2$; $0 \leq h_{j,1}^{(0)} < 2^{\eta-\nu}$ *for* $j = \kappa-1, \kappa, \ldots, 2\kappa-3$; *and* $0 \leq h_{2\kappa-2,1}^{(0)} < 2^{\nu}$ *if* $0 < \nu \leq \eta/2$; $0 \leq h_{2\kappa-2,1}^{(0)} < 2^{\eta-\nu}$ *if* $\eta/2 < \nu < \eta$.

Substituting (4.10) in (4.9) we obtain

$$
\begin{aligned}
& h^{(0)}(\theta) \\
& \equiv \; (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-2}^{(0)}\theta^{\kappa-2} + (h_{\kappa-1,0}^{(0)} + h_{\kappa-1,1}^{(0)}2^{\nu})\theta^{\kappa-1}) \; + \\
& \quad ((h_{\kappa,0}^{(0)} + h_{\kappa,1}^{(0)}2^{\nu}) + (h_{\kappa+1,0}^{(0)} + h_{\kappa+1,1}^{(0)}2^{\nu})\theta + \cdots + (h_{2\kappa-2,0}^{(0)} + h_{2\kappa-2,1}^{(0)}2^{\nu})\theta^{\kappa-2} \; + \\
& \quad h_{2\kappa-1}^{(0)}\theta^{\kappa-1})2^{\eta-\nu} \mod p \\
& = \; (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-2}^{(0)}\theta^{\kappa-2} + h_{\kappa-1,0}^{(0)}\theta^{\kappa-1}) + \\
& \quad (h_{\kappa-1,1}^{(0)}2^{((\kappa-1)\eta+\nu)} + h_{\kappa,0}^{(0)}2^{\eta-\nu} + h_{\kappa,1}^{(0)}2^{\eta} + h_{\kappa+1,0}^{(0)}2^{2\eta-\nu} + h_{\kappa+1,1}^{(0)}2^{2\eta} + \cdots + \\
& \quad h_{2\kappa-2,0}^{(0)}2^{(\kappa-1)\eta-\nu} + h_{2\kappa-2,1}^{(0)}2^{(\kappa-1)\eta} + h_{2\kappa-1}^{(0)}2^{\kappa\eta-\nu}) \text{ [using } \theta = 2^{\eta}\text{]} \qquad (4.11) \\
& \equiv \; (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-2}^{(0)}\theta^{\kappa-2} + h_{\kappa-1,0}^{(0)}\theta^{\kappa-1}) \; + \\
& \quad (h_{\kappa-1,1}^{(0)} + 2^{\eta-\nu}h_{\kappa,0}^{(0)}) + (h_{\kappa,1}^{(0)} + 2^{\eta-\nu}h_{\kappa+1,0}^{(0)})\theta + (h_{\kappa+1,1}^{(0)} + 2^{\eta-\nu}h_{\kappa+2,0}^{(0)})\theta^2 + \cdots + \\
& \quad (h_{2\kappa-3,1}^{(0)} + 2^{\eta-\nu}h_{2\kappa-2,0}^{(0)})\theta^{\kappa-2} + (h_{2\kappa-2,1}^{(0)} + 2^{\eta-\nu}h_{2\kappa-1}^{(0)})\theta^{\kappa-1} \mod p \\
& \qquad\qquad \text{[using (4.4) and } \delta = 1\text{]}. \qquad (4.12)
\end{aligned}
$$

Steps 4-7 of reduceSLMP perform the computations in (4.12) giving us

$$
\begin{aligned}
h^{(0)}(\theta) \; &\equiv \; \underbrace{(h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1}) + (h_{\kappa}^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa-1}^{(1)}\theta^{\kappa-1})}_{\text{through Steps 4-7}} \mod p \\
&= \; h^{(1)}(\theta), \qquad (4.13)
\end{aligned}
$$

where $h_j^{(1)} = h_j^{(0)}$ for $j = 1, 2, \ldots, \kappa-2$, $h_{\kappa-1}^{(1)} = h_{\kappa-1,0}^{(0)}$, $h_j^{(1)} = 2^{\eta-\nu}h_{j,0}^{(0)} + h_{j-1,1}^{(0)}$ for $j = \kappa, \kappa+1, \ldots, 2\kappa-2$ and $h_{2\kappa-1}^{(1)} = 2^{\eta-\nu}h_{2\kappa-1,0}^{(0)} + h_{2\kappa-2,1}^{(0)}$.

In (4.13) it directly follows that $0 \leq h_0^{(1)}, h_1^{(1)}, \ldots, h_{\kappa-2}^{(1)} < 2^{\eta}$ and $h_{\kappa-1}^{(1)} < 2^{\nu}$. The bounds on $h_{\kappa}^{(1)}, h_{\kappa+1}^{(1)}, \ldots, h_{2\kappa-1}^{(1)}$ are given by the following result.

Claim 4.2. $0 \leq h_{\kappa}^{(1)}, h_{\kappa+1}^{(1)}, \ldots, h_{2\kappa-2}^{(1)} < 2^{\eta}$ *and* $0 \leq h_{2\kappa-1}^{(1)} < 2^{\nu}$.

Proof. Using Claim 4.1, for $j = \kappa, \kappa+1, \ldots, 2\kappa-2$ we have

$$
\begin{aligned}
0 \leq h_j^{(1)} \; &= \; \lfloor h_{j-1}^{(0)}/2^{\nu} \rfloor + (2^{\eta-\nu}h_j^{(0)}) \bmod 2^{\eta} \\
&= \; h_{j-1,1}^{(0)} + (2^{\eta-\nu}h_{j,0}^{(0)} + 2^{\eta}h_{j,1}^{(0)}) \bmod 2^{\eta} \\
&= \; h_{j-1,1}^{(0)} + 2^{\eta-\nu}h_{j,0}^{(0)},
\end{aligned}
$$

which implies $0 \leq h_j^{(1)} < 2^{\eta-\nu} + 2^{\eta-\nu}(2^{\nu}-1) = 2^{\eta}$. The argument for the bounds on $h_{2\kappa-1}^{(1)}$ is in two cases.

Case 1. $0 < \nu \leq \eta/2$. From (4.8) and Claim 4.1, $h^{(1)}_{2\kappa-1} = 2^{\eta-\nu}h^{(0)}_{2\kappa-1} + h^{(0)}_{2\kappa-2,1} < 2^{\nu}$.

Case 2. $\eta/2 < \nu < \eta$. From (4.8) and Claim 4.1, $h^{(1)}_{2\kappa-1} = 2^{\eta-\nu}h^{(0)}_{2\kappa-1} + h^{(0)}_{2\kappa-2,1} < 2^{\eta-\nu}(2^{2\nu-\eta}-1) + 2^{\eta-\nu} = 2^{\nu}$. □

In Steps 8-12 we pairwise add the coefficients of $\theta^0, \theta^1, \ldots, \theta^{\kappa-1}$ sequentially in (4.13) by forwarding the 1-bit carry to the subsequent pair to get an intermediate κ-limb polynomial $h^{(2)}$ as

$$\begin{aligned} h^{(1)}(\theta) &= (h^{(1)}_0 + h^{(1)}_{\kappa}) + (h^{(1)}_1 + h^{(1)}_{\kappa+1})\theta + \cdots + (h^{(1)}_{\kappa-1} + h^{(1)}_{2\kappa-1})\theta^{\kappa-1}, \\ &= \underbrace{h^{(2)}_0 + h^{(2)}_1\theta + \cdots + h^{(2)}_{\kappa-1}\theta^{\kappa-1}}_{\text{through Steps 8-12}} = h^{(2)}(\theta). \end{aligned} \tag{4.14}$$

From the computation done in the Steps 8-11 it follows that

$$0 \leq h^{(2)}_0, h^{(2)}_1, \ldots, h^{(2)}_{\kappa-2} < 2^{\eta}. \tag{4.15}$$

Also, since $0 \leq h^{(1)}_{\kappa-1}, h^{(1)}_{2\kappa-1} \leq 2^{\nu}-1$ and $0 \leq \mathfrak{c} \leq 1$, we have

$$0 \leq h^{(2)}_{\kappa-1} = h^{(1)}_{\kappa-1} + h^{(1)}_{2\kappa-1} + \mathfrak{c} \leq 2^{\nu+1} - 1 \tag{4.16}$$

$$\leq 2^{\eta} - 1 \text{ (using } \nu < \eta). \tag{4.17}$$

From (4.13), (4.14), (4.15) and (4.16), we have $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$ and $h^{(2)}(\theta)$ has a $(\kappa, \eta, \nu+1)$-representation.

Equation (4.17) proves that Step 12 does not lead to an overflow. Define

$$\begin{aligned} &h^{(2)}_{\kappa-1} = h^{(2)}_{\kappa-1,0} + h^{(2)}_{\kappa-1,1}2^{\nu}, \text{ where} \\ &h^{(2)}_{\kappa-1,0} = h^{(2)}_{\kappa-1} \bmod 2^{\nu} \text{ and } h^{(2)}_{\kappa-1,1} = \lfloor h^{(2)}_{\kappa-1}/2^{\nu} \rfloor. \end{aligned} \tag{4.18}$$

From (4.16) it follows that $0 \leq h^{(2)}_{\kappa-1,0} < 2^{\nu}$ and $0 \leq h^{(2)}_{\kappa-1,1} \leq 1$. We write

$$\begin{aligned} h^{(2)}(\theta) &= h^{(2)}_0 + h^{(2)}_1\theta + \cdots + h^{(2)}_{\kappa-2}\theta^{\kappa-2} + (h^{(2)}_{\kappa-1,0} + h^{(2)}_{\kappa-1,1}2^{\nu})\theta^{\kappa-1} \\ &= h^{(2)}_0 + h_1\theta + \cdots + h^{(2)}_{\kappa-2}\theta^{\kappa-2} + h^{(2)}_{\kappa-1,0}\theta^{\kappa-1} + h^{(2)}_{\kappa-1,1}2^{(\kappa-1)\eta+\nu} \\ &\equiv (h^{(2)}_0 + h^{(2)}_{\kappa-1,1}) + h^{(2)}_1\theta + \cdots + h^{(2)}_{\kappa-2}\theta^{\kappa-2} + h^{(2)}_{\kappa-1,0}\theta^{\kappa-1} \bmod p \\ &\qquad \text{[using (4.4) and } \delta = 1\text{]}. \end{aligned} \tag{4.19}$$

The following result is crucial in arguing that the carry will be absorbed at some point in the computation.

Claim 4.3. *If $h^{(2)}_{\kappa-1,1} = 1$, then it is impossible to simultaneously have $h^{(2)}_0 = h^{(2)}_1 = \cdots = h^{(2)}_{\kappa-2} = 2^{\eta}-1$ and $h^{(2)}_{\kappa-1,0} = 2^{\nu}-1$.*

Proof. Suppose $h^{(2)}_{\kappa-1,1} = 1$ and if possible, let $h^{(2)}_0 = h^{(2)}_1 = \cdots = h^{(2)}_{\kappa-2} = 2^\eta - 1$, $h^{(2)}_{\kappa-1,0} = 2^\nu - 1$. So, from (4.18) we have $h^{(2)}_{\kappa-1} = h^{(2)}_{\kappa-1,0} + h^{(2)}_{\kappa-1,1}2^\nu = 2^{\nu+1} - 1$. In this case the polynomial $h^{(2)}(\theta)$ is given as follows.

$$\begin{aligned} h^{(2)}(\theta) &= h^{(2)}_0 + h^{(2)}_1\theta + \cdots + h^{(2)}_{\kappa-1}\theta^{\kappa-1} \\ &= (2^\eta - 1) + (2^\eta - 1)2^\eta + \cdots + (2^{\nu+1} - 1)2^{(\kappa-1)\eta} \\ &= 2^{(\kappa-1)\eta+\nu+1} - 1 \\ &= 2^{m+1} - 1 \text{ [using (4.4)].} \end{aligned} \tag{4.20}$$

From (4.14) $h^{(2)}(\theta)$ is obtained by adding the polynomials $(h^{(1)}_0 + h^{(1)}_1\theta + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1})$ and $(h^{(1)}_\kappa + h^{(1)}_{\kappa+1}\theta + \cdots + h^{(1)}_{2\kappa-1}\theta^{\kappa-1})$, where $0 \le h^{(1)}_0, h^{(1)}_1, \ldots, h^{(1)}_{\kappa-2}, h^{(1)}_\kappa, \ldots, h^{(1)}_{2\kappa-2} < 2^\eta$ and $0 \le h^{(1)}_{\kappa-1}, h^{(1)}_{2\kappa-1} < 2^\nu$. So, the maximum possible value of each of the polynomials is $2^m - 1$ and hence the bounds of $h^{(2)}(\theta)$ should be $0 \le h^{(2)}(\theta) < 2^{m+1} - 1$, which contradicts what is obtained in (4.20). Hence the result. □

The computation $h^{(3)}_0 = (h^{(2)}_0 + \lfloor h^{(2)}_{\kappa-1}/2^\nu \rfloor) \bmod 2^\eta$ in (4.19) is performed in Step 13 and the 1-bit $\mathfrak{c}$ is forwarded to the subsequent terms for addition, which are performed in Steps 14-17 producing the values of $h^{(3)}_1, h^{(3)}_2, \ldots, h^{(3)}_{\kappa-1}$. Hence, (4.19) can be written as

$$h^{(2)}(\theta) \equiv \underbrace{h^{(3)}_0 + h^{(3)}_1\theta + \cdots + h^{(3)}_{\kappa-1}\theta^{\kappa-1}}_{\text{through Steps 14-17}} \bmod p = h^{(3)}(\theta). \tag{4.21}$$

We now argue that either the $\mathfrak{c}$ out of Step 13 is 0 or it is absorbed in one of the subsequent additions in Step 15 or in the addition of Step 17. If $h^{(2)}_{\kappa-1,1} = 0$ then the $\mathfrak{c}$ out of Step 13 itself is 0. So, suppose that $h^{(2)}_{\kappa-1,1} = 1$. From Claim 4.3 it follows that either there is a $j \in \{0, 1, \ldots, \kappa-2\}$ such that $h^{(2)}_j < 2^\eta - 1$ or $h^{(2)}_{\kappa-1,0} < 2^\nu - 1$. In the former case, $\mathfrak{c}$ is absorbed by one of the additions in Step 15; if this does not happen, then the later case arises and the carry is absorbed by the addition in Step 17.

This shows that the algorithm terminates without any overflow and at the end of the algorithm we have $0 \le h^{(3)}_0, h^{(3)}_1, \ldots, h^{(3)}_{\kappa-2} < 2^\eta$ and $0 \le h^{(3)}_{\kappa-1} < 2^\nu$ and so $h^{(3)}(\theta)$ has a (κ, η, ν)-representation. Combining (4.13), (4.14), (4.19) and (4.21) we have $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$ which proves the statement of the theorem on full reduction. □

4.5.2 Pseudo-Mersenne Primes

Let $p = 2^m - \delta$ and suppose m-bit integers have a (κ, η, ν)-representation. Function reduceSLPMP given in Algorithm 4.5 takes as input the output of either mulSLDCC or sqrSLDCC and outputs an m-bit integer in an (κ, η, ν)-representation which is congruent to the input modulo p.

As in the case of reduceSLMP, for the correctness of reduceSLPMP, it is not required to have $\eta = 64$. The value of $\eta = 64$ is used for 64-bit implementation and the algorithm can equally well be used with η-bit arithmetic for any value of η (say $\eta = 32$ or $\eta = 128$).

We note that reduceSLMP does not work if $\delta > 1$. This may not be immediately obvious from the description of reduceSLMP. To see that reduceSLMP does not work when

Algorithm 4.5 Reduction for saturated limb representation. Performs reduction modulo p, where $p = 2^m - \delta$ is a pseudo-Mersenne prime; $c_p = 2^{\eta-\nu}\delta$, $2^{\alpha-1} \leq \delta < 2^\alpha$, $\nu' = 2(1 - \lfloor \nu/\eta \rfloor)$ and $\theta = 2^\eta$.

1: **function** reduceSLPMP($h_0^{(0)}(\theta)$)
2: **input**: $h^{(0)}(\theta)$.
3: **output**: $h^{(3)}(\theta)$ or $h^{(4)}(\theta)$.
4: **for** $i \leftarrow 0$ to $\kappa - 1$ **do**
5: $h_i^{(1)} \leftarrow h_i^{(0)}$
6: **end for**
7: $h_\kappa^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa}^{(1)}\theta^\kappa \leftarrow \mathsf{mulSCC}(h_\kappa^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1}, c_p)$
8: $t \leftarrow h_0^{(1)} + h_\kappa^{(1)}$; $h_0^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
9: **for** $i \leftarrow 1$ to $\kappa - 1$ **do**
10: $t \leftarrow h_i^{(1)} + h_{\kappa+i}^{(1)} + \mathfrak{c}$; $h_i^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
11: **end for**
12: $h_\kappa^{(2)} \leftarrow h_{2\kappa}^{(1)} + \mathfrak{c}$
13: $r \leftarrow 2^{\eta-\nu}h_\kappa^{(2)} + \lfloor h_{\kappa-1}^{(2)}/2^\nu \rfloor$; $h_{\kappa-1}^{(2)} \leftarrow h_{\kappa-1}^{(2)} \bmod 2^\nu$
14: $u \leftarrow h_0^{(2)} + \delta r$; $h_0^{(3)} \leftarrow u \bmod 2^\eta$; $q \leftarrow \lfloor u/2^\eta \rfloor$
15: $v \leftarrow h_1^{(2)} + q$; $h_1^{(3)} \leftarrow v \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor v/2^\eta \rfloor$
16: **for** $i \leftarrow 2$ to $\kappa - 2$ **do**
17: $t \leftarrow h_i^{(2)} + \mathfrak{c}$; $h_i^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
18: **end for**
19: $h_{\kappa-1}^{(3)} \leftarrow h_{\kappa-1}^{(2)} + \mathfrak{c}$
20: PARTIAL REDUCTION: **return** $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1}^{(3)}\theta^{\kappa-1}$
21: $t \leftarrow h_{\kappa-1}^{(3)}$; $h_{\kappa-1}^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$; $h_\kappa^{(3)} \leftarrow \mathfrak{c}$
22: $s \leftarrow 2^{\eta-\nu}h_\kappa^{(3)} + \lfloor h_{\kappa-1}^{(3)}/2^\nu \rfloor$; $h_{\kappa-1}^{(3)} \leftarrow h_{\kappa-1}^{(3)} \bmod 2^\nu$
23: $z = h_0^{(3)} + \delta s$
24: **if** $\max(2^{\eta-\nu+\alpha}, 2^{2\alpha+\nu'}) + 2^{\eta-\nu+\alpha} - 2^\alpha \leq 2^{\eta-1}$ **then**
25: $h_0^{(4)} \leftarrow z$; $h_1^{(4)} \leftarrow h_1^{(3)}$
26: **else**
27: $h_0^{(4)} \leftarrow z \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor z/2^\eta \rfloor$; $h_1^{(4)} = h_1^{(3)} + \mathfrak{c}$
28: **end if**
29: **for** $i \leftarrow 2$ to $\kappa - 1$ **do**
30: $h_i^{(4)} \leftarrow h_i^{(3)}$
31: **end for**
32: FULL REDUCTION: **return** $h^{(4)}(\theta) = h_0^{(4)} + h_1^{(4)}\theta + \cdots + h_{\kappa-1}^{(4)}\theta^{\kappa-1}$
33: **end function**.

$\delta > 1$, one needs to consider the proof of correctness of the algorithm. In the proof of Theorem 4.1, the step from (4.11) to (4.12) uses $2^{(\kappa-1)\eta+\nu} = 2^m \equiv \delta \bmod p$. In the case of Mersenne primes, $\delta = 1$ and so the step from (4.11) to (4.12) works; for $\delta > 1$, this step does not work. Instead, we consider a multiplication of the upper half of the input by $c_p = 2^{\eta-\nu}\delta$ at the very beginning and then the resulting polynomial is reduced in several

steps. Due to this multiplication, the number of iterations required to obtain the complete reduction in reduceSLPMP is one more than that required in reduceSLMP. Also, the termination argument (that after a certain stage there is no carry) is more complicated.

Remark 4.5. *The boolean condition in Step 24 of* reduceSLPMP *does not depend on the input* $h^{(0)}(\theta)$ *and is determined entirely by* η, ν *and* α. *So, once the prime and the values of* η *and* ν *are fixed, either the 'then' part of the 'if' statement will be required or, the 'else' part of the 'if' statement will be required. Among the primes considered in Table 4.3, the 'else' part is required only for the prime* $2^{256} - 2^{32} - 977$.

We state a simple result which will be useful in arguing about the termination of reduceSLPMP.

Lemma 4.1. *Let* x, y_1 *and* y_2 *be two integers such that* $0 \leq x < 2^\eta$ *and* $0 \leq y_1, y_2 \leq 2^{\eta-1}$. *Then either* $x + y_1 < 2^\eta$ *or* $y_2 + (x + y_1 \bmod 2^\eta) < 2^\eta$.

Proof. If $0 \leq x < 2^\eta - y_1$, then $x + y_1 < 2^\eta$ and so the result holds. Otherwise, assume that $2^\eta - y_1 \leq x < 2^\eta$. In this case, $2^\eta \leq x + y_1 < 2^\eta + y_1$. So, $0 \leq x + y_1 \bmod 2^\eta < y_1 \leq 2^{\eta-1}$. Consequently, $y_2 \leq y_2 + (x + y_1 \bmod 2^\eta) < y_2 + 2^{\eta-1} \leq 2^\eta$, which proves the result. □

The following result states the correctness of reduceSLPMP.

Theorem 4.2. *Let* $p = 2^m - \delta$ *be a prime and let* $\kappa \geq 2$, η *and* ν *be such that m-bit integers have a* (κ, η, ν)*-representation. Let* α *be such that* $2^{\alpha-1} \leq \delta < 2^\alpha$ *and* $\alpha < \min(\nu + 1, \eta - 2(1 - \lfloor \nu/\eta \rfloor))$. *Suppose that the input* $h^{(0)}(\theta)$ *to* reduceSLPMP *is the output of either* mulSLDCC$(f(\theta), g(\theta))$ *or* sqrSLDCC$(f(\theta))$ *where*

- $f(\theta)$ *and* $g(\theta)$ *are m-bit integers having* (κ, η, ν)*-representations, if* $\nu = \eta$;
- $f(\theta)$ *and* $g(\theta)$ *are* $(m+1)$*-bit integers having* $(\kappa, \eta, \nu + 1)$*-representations, if* $\nu < \eta$.

Then the following holds.

1. *In the case of partial reduction for* $\nu < \eta$, *the output* $h^{(3)}(\theta)$ *of* reduceSLPMP *has a* $(\kappa, \eta, \nu + 1)$*-representation and* $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.
2. *In the case of full reduction, the output* $h^{(4)}(\theta)$ *of* reduceSLPMP *has a* (κ, η, ν)*-representation and* $h^{(4)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

Proof. Note that since $p = 2^m - \delta$ is a prime, for $\delta > 1$, δ cannot be a power of 2. Let $\nu' = 2(1 - \lfloor \nu/\eta \rfloor)$ and so $\nu' = 0$ if $\nu = \eta$ and $\nu' = 2$ for $0 < \nu < \eta$. From $\alpha < \min(\nu + 1, \eta - \nu')$, we have

$$\alpha \leq \alpha + \nu' \leq \eta - 1 < \eta. \tag{4.22}$$

Also, since $\delta < 2^\alpha$ and $\alpha \leq \nu$, we have $\delta < 2^\nu$. Using $2^{\alpha-1} \leq \delta < 2^\alpha$,

$$\begin{aligned} 2^{\eta-\nu+\alpha-1} \leq c_p = 2^{\eta-\nu}\delta &< 2^{\eta-\nu+\alpha} \\ &\leq 2^\eta \text{ (since } \alpha \leq \nu) \end{aligned} \tag{4.23}$$

So, $c_p < 2^\eta$ and hence can be considered to be an η-bit word.

The input $h^{(0)}(\theta)$ is the product of two m-bit integers each having a (κ, η, ν)-representation. As in the proof of Theorem 4.1, using Proposition 4.1 we have the following bounds on the coefficients of $h^{(0)}(\theta)$.

Case 1. $\nu < \eta$.

$$\begin{array}{ll} 0 \leq h_0^{(0)}, h_1^{(0)}, \ldots, h_{2\kappa-3}^{(0)} < 2^{\eta}; \text{ and} & \\ 0 \leq h_{2\kappa-2}^{(0)} < 2^{2(\nu+1)},\ h_{2\kappa-1}^{(0)} = 0 & \text{if} \quad 1 < \nu+1 \leq \eta/2; \\ 0 \leq h_{2\kappa-2}^{(0)} < 2^{\eta},\ 0 \leq h_{2\kappa-1}^{(0)} < 2^{2(\nu+1)-\eta} & \text{if} \quad \eta/2 < \nu+1 \leq \eta. \end{array}$$

Case 2. $\nu = \eta$.

$$0 \leq h_0^{(0)}, h_1^{(0)}, \ldots, h_{2\kappa-1}^{(0)} < 2^{\eta}.$$

For the case $0 < \nu + 1 \leq \eta/2$, we have $0 \leq h_{2\kappa-2}^{(0)} < 2^{2\nu+2} \leq 2^{\eta}$. The above cases can be merged and the following bounds can be stated for all $0 < \nu \leq \eta$.

$$0 \leq h_0^{(0)}, h_1^{(0)}, \ldots, h_{2\kappa-2}^{(0)} < 2^{\eta};\ 0 \leq h_{2\kappa-1}^{(0)} < \max(1, 2^{2\nu-\eta+\nu'}). \tag{4.24}$$

Using $\theta = 2^{\eta}$ and $p = 2^m - \delta$ we have

$$\theta^{\kappa} = 2^{\kappa\eta} = 2^{(\kappa-1)\eta+\nu} \cdot 2^{\eta-\nu} = 2^m \cdot 2^{\eta-\nu} \equiv 2^{\eta-\nu}\delta \bmod p = c_p. \tag{4.25}$$

The input $h^{(0)}$ to reduceSLPMP can be written as

$$\begin{aligned} h^{(0)}(\theta) &= h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} + h_{\kappa}^{(0)}\theta^{\kappa} + \\ &\quad h_{\kappa+1}^{(0)}\theta^{(\kappa+1)} + \cdots + h_{2\kappa-1}^{(0)}\theta^{(2\kappa-1)} \\ &= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}) + \\ &\quad (h_{\kappa}^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1})\theta^{\kappa} \\ &\equiv (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}) + \\ &\quad (h_{\kappa}^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1})c_p \mod p. \end{aligned} \tag{4.26}$$

Step 7 computes the product $(h_{\kappa}^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1})c_p$ of (4.26) using mulSCC, obtaining the output as $(h_{\kappa}^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa}^{(1)}\theta^{\kappa})$. Steps 4-6 simply copies the values of $h_i^{(0)}$ to $h_i^{(1)}$, for $i = 0, 1, \ldots, \kappa-1$. This defines the polynomial $h^{(1)}(\theta)$ and from (4.26) we have

$$\begin{aligned} h^{(0)}(\theta) &\equiv \underbrace{(h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1})}_{\text{through Steps 4-6}} + \underbrace{(h_{\kappa}^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa}^{(1)}\theta^{\kappa})}_{\text{through Step 7}} \mod p \\ &= h^{(1)}(\theta). \end{aligned} \tag{4.27}$$

Limb bounds of $h^{(1)}(\theta)$. The bounds on $h_j^{(0)}$ are given in (4.24). So, by Steps 4-6, $0 \leq h_j^{(1)} < 2^{\eta}, j = 0, 1, \ldots, \kappa-1$. Let $X(\theta) = h_{\kappa}^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1}$ and $Y(\theta) = h_{\kappa}^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa}^{(1)}\theta^{\kappa}$. Using (4.24), the size of the integer represented by $X(\theta)$ is at most $(\kappa-1)\eta + 2\nu - \eta + \nu'$ bits. The integer represented by $Y(\theta)$ is obtained by multiplying

$X(\theta)$ by the constant c_p. From (4.23), the size of c_p is at most $(\eta - \nu + \alpha)$ bits. Hence, the number of bits in the integer represented by $Y(\theta)$ is at most

$$(\kappa-1)\eta + 2\nu - \eta + \nu' + (\eta - \nu + \alpha) \;=\; (\kappa-1)\eta + \nu + \alpha + \nu'.$$

If $\nu + \alpha + \nu' \leq \eta$ then $Y(\theta)$ has a $(\kappa, \eta, \nu + \alpha + \nu')$-representation. Suppose that $\nu + \alpha + \nu' > \eta$. Since $0 < \nu \leq \eta$ and from (4.22) $\alpha + \nu' < \eta$, we have $\alpha + \nu' < \nu + \alpha + \nu' < 2\eta$. Writing $(\kappa-1)\eta + (\nu + \alpha + \nu') = \kappa\eta + (\nu + \alpha + \nu' - \eta)$, in this case, $Y(\theta)$ has a $(\kappa+1, \eta, \nu + \alpha + \nu' - \eta)$-representation. Combining the two cases the limb bounds for $Y(\theta) = (h_\kappa^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa}^{(1)}\theta^\kappa)$ are $0 \leq h_j^{(1)} < 2^\eta, j = \kappa, \kappa+1, \ldots 2\kappa-1, 0 \leq h_{2\kappa}^{(1)} < \max(1, 2^{\nu+\alpha+\nu'-\eta})$. Hence, the limb bounds of $h^{(1)}(\theta)$ can be stated as

$$0 \leq h_0^{(1)}, h_1^{(1)}, \ldots, h_{2\kappa-1}^{(1)} < 2^\eta,\ 0 \leq h_{2\kappa}^{(1)} < \max(1, 2^{\nu+\alpha+\nu'-\eta}). \tag{4.28}$$

Through Steps 8-12, we pairwise add the coefficients of $\theta^0, \theta^1, \ldots, \theta^{\kappa-1}$ given in (4.27) sequentially by forwarding the 1-bit carry, and in Step 12 we add the last carry to $h_{2\kappa}^{(1)}$ producing the $(\kappa+1)$-limb polynomial $h^{(2)}(\theta)$. Hence, from (4.27) we have

$$\begin{aligned} h^{(1)}(\theta) &= (h_0^{(1)} + h_\kappa^{(1)}) + (h_1^{(1)} + h_{\kappa+1}^{(1)})\theta + \cdots + (h_{\kappa-1}^{(1)} + h_{2\kappa-1}^{(1)})\theta^{\kappa-1} + h_{2\kappa}^{(1)}\theta^\kappa, \\ &= \underbrace{h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1} + h_\kappa^{(2)}\theta^\kappa}_{\text{through Steps 8-12}} = h^{(2)}(\theta). \end{aligned} \tag{4.29}$$

Limb bounds of $h^{(2)}(\theta)$. Using the bounds in (4.28) the bounds on the limbs of $h^{(2)}(\theta)$ defined in (4.29) are given by

$$0 \leq h_0^{(2)}, h_1^{(2)}, \ldots, h_{\kappa-1}^{(2)} < 2^\eta;\ 0 \leq h_\kappa^{(2)} \leq \max(1, 2^{\nu+\alpha+\nu'-\eta}). \tag{4.30}$$

In Step 13 $r = 2^{\eta-\nu}h_\kappa^{(2)} + \lfloor h_{\kappa-1}^{(2)}/2^\nu \rfloor$ is computed and the product δr is used in Step 14. The bounds on r and δr are obtained as follows.

Bounds on r and δr. From (4.30) we have $0 \leq h_{\kappa-1}^{(2)} < 2^\eta$ and so $\lfloor h_{\kappa-1}^{(2)}/2^\nu \rfloor < 2^{\eta-\nu}$ i.e., $\lfloor h_{\kappa-1}^{(2)}/2^\nu \rfloor \leq 2^{\eta-\nu} - 1$. Also, from (4.30) we have $0 \leq h_\kappa^{(2)} \leq \max(1, 2^{\nu+\alpha+\nu'-\eta})$. From the definition of r we obtain

$$\begin{aligned} 0 \;&\leq\; r \;\leq\; \max(2^{\eta-\nu}, 2^{\alpha+\nu'}) + 2^{\eta-\nu} - 1 \\ \Rightarrow\; 0 \;&\leq\; \delta r \;<\; \max(2^{\eta-\nu+\alpha}, 2^{2\alpha+\nu'}) + 2^{\eta-\nu+\alpha} - 2^\alpha \text{ [since } \delta < 2^\alpha\text{]}. \end{aligned} \tag{4.31}$$

If the boolean condition in Step 24 holds, then we have

$$0 \leq \delta r < 2^{\eta-1}. \tag{4.32}$$

Otherwise, a bound on δr is obtained by continuing the computation of (4.31) as follows.

$$\begin{aligned} \Rightarrow\; 0 \;&\leq\; \delta r \;<\; \max(2^{2\eta-\nu-1}, 2^{2\eta-2}) + 2^{2\eta-\nu-1} - 2^{\eta-1} \\ &\qquad\qquad \text{[since from (4.22) we have } \alpha, \alpha+\nu' \leq \eta - 1\text{]} \\ \Rightarrow\; 0 \;&\leq\; \delta r \;<\; 2^{2\eta-2} + 2^{2\eta-\nu-1} - 2^{\eta-1} \text{ [since } \nu \geq 1\text{]} \\ \Rightarrow\; 0 \;&\leq\; \delta r \;<\; 2^{2\eta-1} - 2^{\eta-1} \text{ [again since } \nu \geq 1\text{]}. \end{aligned} \tag{4.33}$$

So, (4.33) holds irrespective of whether the boolean condition in Step 24 holds or not. The variable u is defined in Step 14. From (4.30) and (4.33) an upper bound on u is as follows.

$$u \;=\; h_0^{(2)} + \delta r \;<\; 2^{\eta} - 1 + 2^{2\eta-1} - 2^{\eta-1}. \tag{4.34}$$

Define $h_{\kappa-1}^{(2)} = h_{\kappa-1,0}^{(2)} + h_{\kappa-1,1}^{(2)} 2^{\nu}$ where $h_{\kappa-1,0}^{(2)} = h_{\kappa-1}^{(2)} \bmod 2^{\nu}$ and $h_{\kappa-1,1}^{(2)} = \lfloor h_{\kappa-1}^{(2)}/2^{\nu} \rfloor$. Then we have

$$\begin{aligned}
h^{(2)}(\theta) &= h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1} + h_{\kappa}^{(2)}\theta^{\kappa} \\
&= h_0^{(2)} + h_1^{(2)}\theta + \cdots + (h_{\kappa-1,0}^{(2)} + h_{\kappa-1,1}^{(2)}2^{\nu})\theta^{\kappa-1} + h_{\kappa}^{(2)}\theta^{\kappa} \\
&= h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1,0}^{(2)}\theta^{\kappa-1} + h_{\kappa-1,1}^{(2)}2^{\eta(\kappa-1)+\nu} + h_{\kappa}^{(2)}\theta^{\kappa} \\
&= h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1,0}^{(2)}\theta^{\kappa-1} + h_{\kappa-1,1}^{(2)}2^{m} + h_{\kappa}^{(2)}\theta^{\kappa} \\
&\equiv h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1,0}^{(2)}\theta^{\kappa-1} + h_{\kappa-1,1}^{(2)}\delta + h_{\kappa}^{(2)}c_p \mod p \\
&\qquad\qquad \text{[using (4.4) and (4.23)]} \\
&= h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1,0}^{(2)}\theta^{\kappa-1} + (h_{\kappa-1,1}^{(2)} + h_{\kappa}^{(2)}2^{\eta-\nu})\delta \text{ [using (4.23)]} \\
&= (r\delta + h_0^{(2)}) + h_1^{(2)}\theta + \cdots + h_{\kappa-1,0}^{(2)}\theta^{\kappa-1} \text{ [from Step 13]} \\
&= u + h_1^{(2)}\theta + \cdots + h_{\kappa-1,0}^{(2)}\theta^{\kappa-1} \text{ [from Step 14]} \\
&\equiv \underbrace{h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa}^{(3)}\theta^{\kappa}}_{\text{through Steps 14-21}} \mod p = h^{(3)}(\theta).
\end{aligned} \tag{4.35}$$

The analysis of the rest of the algorithm, i.e., Steps 14-30 is divided into two cases depending on whether $u < 2^{\eta}$ or $u \geq 2^{\eta}$.

Case 1. $u < 2^{\eta}$. In this case $q = \lfloor u/2^{\eta} \rfloor = 0$, and so from Steps 15-18 we have $0 \leq h_j^{(3)} = h_j^{(2)} < 2^{\eta}$ for $j = 1, 2, \ldots, \kappa - 2$. Also, we have $0 \leq h_{\kappa-1}^{(3)} = h_{\kappa-1}^{(2)} < 2^{\nu}$, because in Step 13 we have already updated $h_{\kappa-1}^{(2)}$ by $h_{\kappa-1}^{(2)} \bmod 2^{\nu}$. By Step 14 we have $h_0^{(3)} = u \bmod 2^{\eta} < 2^{\eta}$, and Step 21 gives $h_{\kappa}^{(3)} = 0$. So, in this case $h^{(3)}(\theta)$ returned at Step 20 has a (κ, η, ν)-representation irrespective of whether $\nu < \eta$ or $\nu = \eta$. By (4.27), (4.29) and (4.35) we have $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$. This proves the statement of the theorem on partial reduction for **Case 1**.

Further, we have $s = 0$ by Step 22, and so by the remaining steps of the algorithm we have $0 \leq h_j^{(4)} = h_j^{(3)} < 2^{\eta}$ for $j = 0, 1, \ldots, \kappa - 2$ and $0 \leq h_{\kappa-1}^{(4)} = h_{\kappa-1}^{(3)} < 2^{\nu}$, i.e., $h^{(4)}(\theta)$ has a (κ, η, ν)-representation. It follows that $h^{(4)}(\theta) = h^{(3)}(\theta)$ and using (4.27), (4.29) and (4.35), we have $h^{(4)}(\theta) \equiv h^{(0)}(\theta) \bmod p$ which proves the statement of the theorem on full reduction for **Case 1**.

Case 2. $u \geq 2^\eta$. Step 14 defines q to be $q = \lfloor u/2^\eta \rfloor$. Since in this case $u \geq 2^\eta$, the bounds on q are given by

$$\begin{aligned} 1 \;\leq\; q \;\leq\; & \left\lfloor \frac{2^\eta - 1 + 2^{2\eta-1} - 2^{\eta-1}}{2^\eta} \right\rfloor \text{ [using (4.34)]} \\ \Rightarrow\; 1 \;\leq\; q \;\leq\; & \left\lfloor 1 - \frac{1}{2^\eta} + 2^{\eta-1} - \frac{1}{2} \right\rfloor \\ \Rightarrow\; 1 \;\leq\; q \;\leq\; & 2^{\eta-1} + \left\lfloor \frac{1}{2} - \frac{1}{2^\eta} \right\rfloor \\ \Rightarrow\; 1 \;\leq\; q \;\leq\; & 2^{\eta-1} \;<\; 2^\eta - 1. \end{aligned} \tag{4.36}$$

In Step 15 the algorithm computes $v = h_1^{(2)} + q$. There are two sub cases to consider depending on whether $v < 2^\eta$ or $v \geq 2^\eta$.

Subcase 2a. $v < 2^\eta$. Step 15 defines $\mathfrak{c} = \lfloor v/2^\eta \rfloor$ and so $\mathfrak{c} = 0$ at this step. This simplifies the analysis and the rest of the proof is similar to that of **Case 1**.

Subcase 2b. $v \geq 2^\eta$. This is the non-trivial case and it is required to argue that there are no overflows. In this case, using (4.30) and (4.36) we have

$$2^\eta \;\leq\; v \;=\; h_1^{(2)} + q \;<\; 2^\eta + 2^\eta - 1 \;=\; 2^{\eta+1} - 1. \tag{4.37}$$

So, $2^\eta \leq v \leq 2^{\eta+1} - 2 = 2^\eta + 2^\eta - 2$ implying $v \bmod 2^\eta \leq 2^\eta - 2 < 2^\eta - 1$. After Step 15 we have

$$h_1^{(3)} = v \bmod 2^\eta < 2^\eta - 1 \text{ and } 0 \leq \mathfrak{c} \leq 1.$$

Consider $h^{(3)}(\theta)$ as given in (4.35).

- By Step 14 we have $0 \leq h_0^{(3)} < 2^\eta$.
- Since $\mathfrak{c} \leq 1$, considering Step 17 for $i = 2, 3, \ldots, \kappa - 2$, shows $0 \leq h_j^{(3)} < 2^\eta$ for $j = 2, 3, \ldots, \kappa - 2$.
- Recall that $h_{\kappa-1,0}^{(2)} = h_{\kappa-1}^{(2)} \bmod 2^\nu \leq 2^\nu - 1$ and consider Step 17 for $i = \kappa - 1$. Since $\mathfrak{c} \leq 1$, we have $h_{\kappa-1}^{(3)} \leq 2^\nu$ and so $h_{\kappa-1}^{(3)}$ is a $(\nu+1)$-bit integer.

Consequently, if $\nu < \eta$ then $h^{(3)}(\theta)$ has a $(\kappa, \eta, \nu+1)$-representation and by (4.27), (4.29) and (4.35) we have $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$. This proves the statement of the theorem on partial reduction for **Subcase 2b**.

On the other hand, if $\nu = \eta$ then $h_{\kappa-1}^{(3)}$ will be an $(\eta+1)$-bit string (equivalently, $h^{(3)}(\theta)$ will be an $(m+1)$-bit integer) and further reduction is required to ensure that the number of limbs in the final result is κ. So, for $\nu = \eta$ partial reduction is not useful. The general analysis for obtaining the final reduction irrespective of whether $\nu < \eta$ or $\nu = \eta$ is given below.

In Steps 22-23 the algorithm computes $s = 2^{\eta-\nu}h_\kappa^{(3)} + \lfloor h_{\kappa-1}^{(3)}/2^\nu \rfloor$ and $z = h_0^{(3)} + \delta s$ respectively. Define $h_{\kappa-1}^{(3)} = h_{\kappa-1,0}^{(3)} + h_{\kappa-1,1}^{(3)}2^\nu$ where $h_{\kappa-1,0}^{(3)} = h_{\kappa-1}^{(3)} \bmod 2^\nu$ and $h_{\kappa-1,1}^{(3)} = \lfloor h_{\kappa-1}^{(3)}/2^\nu \rfloor$. Then

$$\begin{aligned}
h^{(3)}(\theta) &= h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1}^{(3)}\theta^{\kappa-1} + h_\kappa^{(3)}\theta^\kappa \\
&= h_0^{(3)} + h_1^{(3)}\theta + \cdots + (h_{\kappa-1,0}^{(3)} + h_{\kappa-1,1}^{(3)}2^\nu)\theta^{\kappa-1} + h_\kappa^{(3)}\theta^\kappa \\
&= h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1,0}^{(3)}\theta^{\kappa-1} + h_{\kappa-1,1}^{(3)}2^{\eta(\kappa-1)+\nu} + h_\kappa^{(3)}\theta^\kappa \\
&= h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1,0}^{(3)}\theta^{\kappa-1} + h_{\kappa-1,1}^{(3)}2^m + h_\kappa^{(3)}\theta^\kappa \\
&\equiv h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1,0}^{(3)}\theta^{\kappa-1} + h_{\kappa-1,1}^{(3)}\delta + h_\kappa^{(3)}c_p \mod p \\
&\qquad\qquad \text{[using (4.4) and (4.23)]} \\
&= h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1,0}^{(3)}\theta^{\kappa-1} + (h_{\kappa-1,1}^{(3)} + h_\kappa^{(3)}2^{\eta-\nu})\delta \text{ [using (4.25)]} \\
&= (s\delta + h_0^{(3)}) + h_1^{(3)}\theta + \cdots + h_{\kappa-1,0}^{(3)}\theta^{\kappa-1} \text{ [from Step 22]} \\
&= z + h_1^{(3)}\theta + \cdots + h_{\kappa-1,0}^{(3)}\theta^{\kappa-1} \text{ [from Step 23].} \qquad (4.38)
\end{aligned}$$

Claim 4.4. *The value of s computed in Step 22 is at most 1.*

Proof. The value of s is $2^{\eta-\nu}h_\kappa^{(3)} + \lfloor h_{\kappa-1}^{(3)}/2^\nu \rfloor$. In Step 13 $h_{\kappa-1}^{(2)}$ is set to $h_{\kappa-1}^{(2)} \bmod 2^\nu$ and so after this step we have $0 \le h_{\kappa-1}^{(2)} < 2^\nu$. Consider Steps 19 and 21. We have $0 \le h_{\kappa-1}^{(2)} < 2^\nu$ and so the value of t at Step 21 is at most 2^ν.

- The value of $h_{\kappa-1}^{(3)}$ is set to be equal to $t \bmod 2^\eta$. So, if $\nu < \eta$ then $0 \le h_{\kappa-1}^{(3)} \le 2^\nu$, while if $\nu = \eta$ then $0 \le h_{\kappa-1}^{(3)} < 2^\eta$.
- The updated value of $\mathfrak{c}$ is $\lfloor t/2^\eta \rfloor$ and this can be equal to 1 only if $\nu = \eta$. This value of $\mathfrak{c}$ is assigned to $h_\kappa^{(3)}$ in Step 21. So, $h_\kappa^{(3)} = 1$ only if $\nu = \eta$.

If $\nu < \eta$ then $h_\kappa^{(3)} = 0$ and $\lfloor h_{\kappa-1}^{(3)}/2^\nu \rfloor \le 1$ implying that $s \le 1$. On the other hand, if $\nu = \eta$ then $2^{\eta-\nu}h_\kappa^{(3)} = h_\kappa^{(3)} \le 1$ and $\lfloor h_{\kappa-1}^{(3)}/2^\nu \rfloor = \lfloor h_{\kappa-1}^{(3)}/2^\eta \rfloor = 0$ again implying that $s \le 1$. □

If $s = 0$ then $z = h_0^{(3)}$ implying $h_0^{(4)} = h_0^{(3)}$ and $\mathfrak{c}$ at Step 27 is 0. So, $h_0^{(4)} = h_0^{(3)}$ and $h_1^{(4)} = h_1^{(3)}$ hold for both branches of the 'if' statement in Step 24. From Step 30 it follows that $h^{(4)}(\theta) = h^{(3)}(\theta)$.

If $s = 1$ then $z = h_0^{(3)} + \delta$. The termination arguments for the two branches of the 'if' statement at Step 24 are different.

First suppose that the boolean condition of the 'if' statement evaluates to true. We apply Lemma 4.1 with $x = h_0^{(2)}$, $y_1 = \delta r$ and $y_2 = \delta$. From (4.32) we have $0 \le \delta r < 2^{\eta-1}$ which also implies $0 < \delta < 2^{\eta-1}$. In Step 14 u is computed as $u = h_0^{(2)} + \delta r = x + y_1$ and $h_0^{(3)} = u \bmod 2^\eta = x + y_1 \bmod 2^\eta$. In Step 25 $h_4^{(0)} = z = h_0^{(3)} + \delta = y_2 + (x + y_1 \bmod 2^\eta)$. In **Case 2** $u \ge 2^\eta$, i.e., $x + y_1 \ge 2^\eta$. Then from Lemma 4.1 we have $h_4^{(0)} = y_2 + (x + y_1 \bmod 2^\eta) < 2^\eta$. So, the procedure terminates.

Now consider the case that the boolean condition of the 'if' statement evaluates to false. By Step 22 we have $0 \leq h^{(3)}_{\kappa-1} < 2^{\nu}$ and $0 \leq h^{(4)}_{0} < 2^{\eta}$ by Step 27. The value of $\mathfrak{c}$ in Step 27 can be at most 1, and since the bound of $h^{(3)}_{1}$ is $0 \leq h^{(3)}_{1} < 2^{\eta} - 1$, hence after Step 27 we have $0 \leq h^{(4)}_{1} < 2^{\eta}$.

So, after both branches of the 'if' statement in Step 24, the limb bounds of $h^{(4)}(\theta)$ are $0 \leq h^{(4)}_{j} < 2^{\eta}$ for $j = 0, 1, \ldots, \kappa - 2$, and $0 \leq h^{(4)}_{\kappa-1} < 2^{\nu}$. From (4.38) we can write

$$h^{(3)}(\theta) \equiv \underbrace{h^{(4)}_{0} + h^{(4)}_{1}\theta + \cdots + h^{(4)}_{\kappa-1}\theta^{\kappa}}_{\text{through Steps 22-30}} \mod p = h^{(4)}(\theta). \tag{4.39}$$

Combining (4.27), (4.29), (4.35) and (4.39) we have $h^{(4)}(\theta) \equiv h^{(0)}(\theta) \bmod p$, which proves the statement of the theorem on full reduction for **Subcase 2b**. □

4.5.3 Usefulness of Partial Reduction

The statement of Theorem 4.2 identifies two cases. If $\nu < \eta$ then the input to reduceSLPMP is considered to be a $(2m+2)$-bit integer, whereas if $\nu = \eta$ then the input to reduceSLPMP is considered to be a $2m$-bit integer. This is the consequence of whether partial reduction is used or not. In the case of $\nu < \eta$ a partial reduction strategy is used whereas for $\nu = \eta$ such a strategy is not used. For the partial reduction strategy, the output $h^{(3)}(\theta)$ returned by reduceSLPMP is an $(m+1)$-bit integer. So, if partial reduction strategy is used throughout then the inputs to mulSLDCC and sqrSLDCC will also be $(m+1)$-bit integers and so their outputs will be $(2m+2)$-bit integers. Subsequent applications of reduceSLPMP will have to handle $(2m+2)$-bit integers. This is the reason why the statement of Theorem 4.2 specifies the input to reduceSLPMP to be a $(2m+2)$-bit integer for the case $\nu < \eta$. On the other hand, for $\nu = \eta$ partial reduction is not used and so the output $h^{(4)}(\theta)$ of reduceSLPMP is an m-bit integer and consequently, the outputs of mulSLDCC and sqrSLDCC will be $2m$-bit integers.

Partial reduction is useful since it avoids the computation required to reduce $h^{(3)}(\theta)$ to $h^{(4)}(\theta)$. All intermediate computations are performed using partially reduced results and the full reduction is invoked only once at the end. This strategy leads to substantial savings in the number of operations and hence on the consequent speed of computation.

There does not seem to be an efficient way in which the partial reduction strategy can be made to work for Mersenne primes. A possible partially reduced result in reduceSLMP would be $h^{(2)}(\theta)$. It can be shown that when the input to reduceSLMP is the product of two m-bit integers each having (κ, η, ν)-representation, then $h^{(2)}(\theta)$ has a $(\kappa, \eta, \nu+1)$-representation, i.e., it is an $(m+1)$-bit integer. So, mulSLDCC and sqrSLDCC will produce as output $(2m+2)$-bit integers. Feeding such an integer as the input of reduceSLMP results in $h^{(2)}(\theta)$ having a $(\kappa, \eta, \nu+3)$-representation. In other words, the size of the last limb grows. This can be brought down, but doing this requires additional computation and results in the partial reduction being less efficient than the full reduction that we have described. In the case of pseudo-Mersenne primes the partial result returned by reduceSLPMP avoids such growth of the last limb.

Remark 4.6. *Step 7 of* reduceSLPMP *performs a* mulSCC *call. As described in Section 4.4, this call can be implemented using a single carry chain by using the* `mulx`, `add` *and* `adc` *instructions. In* reduceSLPMP *Steps 8-11 add the output of* mulSCC *to the initial κ limbs of the input $h^{(0)}$. It*

is possible to consider a strategy whereby the multiplications and the additions within mulSCC *are done simultaneously with the additions in Steps 8-11. It is possible to organize the code such that two independent carry chains arise so that one of the addition chains is implemented using* `adcx` *and the other using* `adox`*. We have implemented this strategy, but the gain in speed is not significant and so we do not describe the details.*

4.5.4 A Variant of reduceSLPMP

Bernstein et al [BDL+12] have used an algorithm for partial reduction using a 4-limb representation of $2^{255} - 19$. Function reduceSLPMPa in Algorithm 4.6 provides a generalization of this algorithm which works for a large class of pseudo-Mersenne primes.

Algorithm 4.6 Partial reduction for saturated limb representation. Performs reduction modulo p, where $p = 2^m - \delta$ is a pseudo-Mersenne prime; $c_p = 2^{\eta-\nu}\delta$ and $\theta = 2^\eta$.

1: **function** reduceSLPMPa($h_0^{(0)}(\theta)$)
2: **input**: $h^{(0)}(\theta)$.
3: **output**: $h^{(4)}(\theta)$.
4: **for** $i \leftarrow 0$ to $\kappa - 1$ **do**
5: $h_i^{(1)} \leftarrow h_i^{(0)}$
6: **end for**
7: $h_\kappa^{(1)} + h_{\kappa+1}^{(1)}\theta + \cdots + h_{2\kappa}^{(1)}\theta^\kappa \leftarrow$ mulSCC$(h_\kappa^{(0)} + h_{\kappa+1}^{(0)}\theta + \cdots + h_{2\kappa-1}^{(0)}\theta^{\kappa-1}, c_p)$
8: $t \leftarrow h_0^{(1)} + h_\kappa^{(1)}$; $h_0^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
9: **for** $i \leftarrow 1$ to $\kappa - 1$ **do**
10: $t \leftarrow h_i^{(1)} + h_{\kappa+i}^{(1)} + \mathfrak{c}$; $h_i^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
11: **end for**
12: $h_\kappa^{(2)} \leftarrow h_{2\kappa}^{(1)} + \mathfrak{c}$
13: $r \leftarrow h_\kappa^{(2)} \cdot c_p$
14: $u \leftarrow h_0^{(2)} + r$; $h_0^{(3)} \leftarrow u \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor u/2^\eta \rfloor$
15: **for** $i \leftarrow 1$ to $\kappa - 1$ **do**
16: $t \leftarrow h_i^{(2)} + \mathfrak{c}$; $h_i^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
17: **end for**
18: $h_\kappa^{(3)} \leftarrow \mathfrak{c}$
19: $s \leftarrow h_\kappa^{(3)} \cdot c_p$
20: $z = h_0^{(3)} + s$;
21: **if** $\max(2^{\eta-\nu+\alpha}, 2^{2\alpha+\nu'}) \leq 2^{\eta-1}$ **then**
22: $h_0^{(4)} \leftarrow z$; $h_1^{(4)} \leftarrow h_1^{(3)}$
23: **else**
24: $h_0^{(4)} \leftarrow z \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor z/2^\eta \rfloor$; $h_1^{(4)} = h_1^{(3)} + \mathfrak{c}$
25: **end if**
26: **for** $i \leftarrow 2$ to $\kappa - 1$ **do**
27: $h_i^{(4)} \leftarrow h_i^{(3)}$
28: **end for**
29: PARTIAL REDUCTION: **return** $h^{(4)}(\theta) = h_0^{(4)} + h_1^{(4)}\theta + \cdots + h_{\kappa-1}^{(4)}\theta^{\kappa-1}$
30: **end function**.

Similar to reduceSLPMP, the boolean condition in Step 21 of reduceSLPMPa does not depend on the input $h^{(0)}(\theta)$ and is determined entirely by η, ν and α. So, either the 'then' part of the 'if' statement will be required or, the 'else' part of the 'if' statement will be required. Among the primes considered in Table 4.3, the 'else' part is required only for the prime $2^{256} - 2^{32} - 977$.

The following result states the correctness of reduceSLPMPa.

Theorem 4.3. *Let $p = 2^m - \delta$ be a prime and let $\kappa \geq 2$, η and ν be such that m-bit integers have a (κ, η, ν)-representation. Let α be such that $2^{\alpha-1} \leq \delta < 2^{\alpha}$. Suppose that the input $h^{(0)}(\theta)$ to* reduceSLPMPa *is the output of either* mulSLDCC$(f(\theta), g(\theta))$ *or* sqrSLDCC$(f(\theta))$ *where $f(\theta)$ and $g(\theta)$ are $\kappa\eta$-bit integers having (κ, η, η)-representations. Then the output $h^{(4)}(\theta)$ of* reduceSLPMPa *has a (κ, η, η)-representation and $h^{(4)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

Proof. The proof is similar to the proof of Theorem 4.2. The inputs to reduceSLPMP and reduceSLPMPa are of different sizes. On the other hand, Steps 4-12 of reduceSLPMPa is exactly the same as that of reduceSLPMP. So, the bounds on the limbs of $h^{(2)}(\theta)$ can be derived in a manner similar to the bounds obtained in (4.30) and are as follows: $0 \leq h_j^{(2)} < 2^{\eta}$ for $j = 0, 1, \ldots, \kappa - 1$, and $0 \leq h_{\kappa}^{(2)} \leq \max(1, 2^{\nu+\alpha-\eta})$. Since r is computed as $r = h_{\kappa}^{(2)} \cdot c_p = h_{\kappa}^{(2)} \cdot 2^{\eta-\nu}\delta < 2^{\eta-\nu+\alpha}$, we have $0 \leq r \leq \max(2^{\eta-\nu+\alpha}, 2^{2\alpha})$. The upper bound on r gives rise to the boolean condition in Step 21. The rest of the argument proceeds along the same lines as that of Theorem 4.2 and is in fact a bit simpler. The 'if' statement in Step 21 determines whether the last addition takes place at limb number 0 (which is the 'then' part), or, whether it takes place at limb number 1 (which is the 'else' part). The 'else' part is required only if the addition to limb number 0 can produce a carry. This part of the argument is similar to the argument for termination corresponding to $s = 1$ in the proof of Theorem 4.2. □

4.5.5 Comparison of reduceSLPMP and reduceSLPMPa

We note the following points.

1. Full reduction can be obtained using reduceSLPMP, but reduceSLPMPa always performs partial reduction. So, if reduceSLPMPa is used, then the last reduction is to be done by reduceSLPMP, or, the final output of reduceSLPMPa is to be further reduced using some other method. On the other hand, if $\nu < \eta$ and reduceSLPMP is used then partial reduction will be done for all but the last invocation, and the last invocation will perform full reduction. No other code is required to ensure full reduction.

2. The computation of r in reduceSLPMP is slightly more expensive than the computation of r in reduceSLPMPa. Using partial reduction for reduceSLPMP avoids generating $h^{(4)}(\theta)$ from $h^{(3)}(\theta)$ saving a few instructions. Compared to reduceSLPMPa, saving these instructions more or less balances the extra cost of generating r.

3. We have implemented both reduceSLPMP and reduceSLPMPa as part of the various inversion algorithms. It has been found that the assembly implementations using reduceSLPMP performs better than the assembly implementations using reduceSLPMPa in the Skylake architecture.

Remark 4.7. *The outputs of* mulSLDCC *and* sqrSLDCC *can also be computed without using double carry chains. Let us call such algorithms as* mulSL *and* sqrSL *respectively. Theorems 4.1, 4.2 and 4.3 will also hold if the algorithms* mulSL *and* sqrSL *are used instead of* mulSLDCC *and* sqrSLDCC *respectively.*

4.6 Saturated Limb Computation Without Double Carry Chains

In Section 4.4 we have described how the saturated limb representation can be exploited in combination with two independent carry chains to obtain fast squaring and multiplication algorithms. Implementation of these algorithms require the use of the instructions `mulx`, `adcx` and `adox`. For processors which do not provide these instructions, the algorithms in Section 4.4 cannot be implemented. In this section we describe algorithms for saturated limb representation which do not use double carry chains and can be implemented on previous generation processors.

As before, let $p = 2^m - \delta$ where m-bit integers have (κ, η, ν)-representation. Since we consider the saturated limb representation, we have $\eta = 64$. As before, $\theta = 2^\eta$ and $c_p = 2^{\eta-\nu}\delta$. Let $f(\theta)$ and $g(\theta)$ be two elements of $\mathbb{F}_p$ written as

$$\begin{aligned} f(\theta) &= f_0 + f_1\theta + \cdots + f_{\kappa-1}\theta^{\kappa-1}, \\ g(\theta) &= g_0 + g_1\theta + \cdots + g_{\kappa-1}\theta^{\kappa-1}, \end{aligned}$$

where $0 \le f_i,\ g_i < 2^\eta,\ i = 0, 1, \ldots, \kappa-2$, and $0 \le f_{\kappa-1},\ g_{\kappa-1} < 2^\nu$. The schoolbook product of $f(\theta)$ and $g(\theta)$ modulo p can be written as $h(\theta) = h_0 + h_1\theta + \cdots + h_{\kappa-1}\theta^{\kappa-1}$ where

$$\begin{aligned} h_0 &= f_0g_0 + c_p(f_1g_{\kappa-1} + f_2g_{\kappa-2} + \cdots + f_{\kappa-2}g_2 + f_{\kappa-1}g_1), \\ h_1 &= f_0g_1 + f_1g_0 + c_p(f_2g_{\kappa-1} + \cdots + f_{\kappa-2}g_3 + f_{\kappa-1}g_2), \\ &\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ h_{\kappa-2} &= f_0g_{\kappa-2} + f_1g_{\kappa-3} + f_2g_{\kappa-4} + \cdots + f_{\kappa-2}g_0 + c_pf_{\kappa-1}g_{\kappa-1}, \\ h_{\kappa-1} &= f_0g_{\kappa-1} + f_1g_{\kappa-2} + f_2g_{\kappa-3} + \cdots + f_{\kappa-2}g_1 + f_{\kappa-1}g_0. \end{aligned} \tag{4.40}$$

Since we are working with $\eta = 64$, the coefficients h_i are not guaranteed to fit within 128 bits. We show how to tackle this problem. Define

$$f_i \cdot g_j = u_{i,j} + v_{i,j}2^\eta = u_{i,j} + v_{i,j}\theta \text{ for } i, j = 0, 1, \ldots, \kappa-1. \tag{4.41}$$

Using $\theta^\kappa = 2^{\kappa\eta} \equiv 2^{\eta-\nu}\delta \bmod p = c_p$ and (4.41) in (4.40), we have $h(\theta) \equiv z(\theta) \bmod p$ where $z(\theta) = z_0 + z_1\theta + \cdots + z_{\kappa-1}\theta^{\kappa-1}$ and

$$\begin{aligned} z_0 &= u_{0,0} + c_p(u_{1,\kappa-1} + u_{2,\kappa-2} + \cdots + u_{\kappa-2,2} + u_{\kappa-1,1} + \\ &\quad v_{0,\kappa-1} + v_{1,\kappa-2} + v_{2,\kappa-3} + \cdots + v_{\kappa-2,1} + v_{\kappa-1,0}), \\ z_1 &= u_{0,1} + u_{1,0} + v_{0,0} + c_p(u_{2,\kappa-1} + \cdots + u_{\kappa-2,3} + u_{\kappa-1,2} + \\ &\quad v_{1,\kappa-1} + v_{2,\kappa-2} + \cdots + v_{\kappa-2,2} + v_{\kappa-1,1}), \\ &\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ z_{\kappa-2} &= u_{0,\kappa-2} + u_{1,\kappa-3} + u_{2,\kappa-4} + \cdots + u_{\kappa-2,0} + v_{0,\kappa-3} + \\ &\quad v_{1,\kappa-4} + \cdots + v_{\kappa-3,0} + c_p(u_{\kappa-1,\kappa-1} + v_{\kappa-2,\kappa-1} + v_{\kappa-1,\kappa-2}), \\ z_{\kappa-1} &= u_{0,\kappa-1} + u_{1,\kappa-2} + u_{2,\kappa-3} + \cdots + u_{\kappa-2,1} + u_{\kappa-1,0} + \\ &\quad v_{0,\kappa-2} + v_{1,\kappa-3} + v_{2,\kappa-4} + \cdots + v_{\kappa-2,0} + c_pv_{\kappa-1,\kappa-1}. \end{aligned} \tag{4.42}$$

For all the primes in Table 4.4, it can be ensured that $0 \leq z_0, z_1, \ldots, z_{\kappa-1} < 2^{127}$. Substituting $g = f$, we get similar equations for squaring. Denote the resulting multiplication and squaring algorithms by mulSLa and sqrSLa respectively.

Next we describe how to reduce $z(\theta)$. Function reduceSL in Algorithm 4.7 performs the required computation. The following results states the correctness of reduceSL. The

Algorithm 4.7 Generic reduction algorithm using saturated limb representation for the primes in Table 4.4. It performs reduction modulo $p = 2^m - \delta$ and m-bit integers have a (κ, η, ν)-representation with $\eta = 64$; $\theta = 2^\eta$.

function reduceSL($h^{(0)}(\theta)$)
input: $h^{(0)}(\theta)$.
output: $h^{(2)}(\theta)$ or $h^{(3)}(\theta)$.
    $h_0^{(1)} \leftarrow h_0^{(0)} \mod 2^\eta$; $r_0 \leftarrow \lfloor h_0^{(0)}/2^\eta \rfloor$
    for $i \leftarrow 1$ to $\kappa - 2$ **do**
        $t_i \leftarrow h_i^{(0)} + r_{i-1}$; $h_i^{(1)} \leftarrow t_i \mod 2^\eta$; $r_i \leftarrow \lfloor t_i/2^\eta \rfloor$
    end for
    $t_{\kappa-1} \leftarrow h_{\kappa-1}^{(0)} + r_{\kappa-2}$; $h_{\kappa-1}^{(1)} \leftarrow t_{\kappa-1} \mod 2^\nu$; $r_{\kappa-1} \leftarrow \lfloor t_{\kappa-1}/2^\nu \rfloor$
    $t \leftarrow h_0^{(1)} + \delta r_{\kappa-1}$; $h_0^{(2)} \leftarrow t \mod 2^\eta$; $\mathfrak{c}_0 \leftarrow \lfloor t/2^\eta \rfloor$
    for $i \leftarrow 1$ to $\kappa - 2$ **do**
        $t \leftarrow h_i^{(1)} + \mathfrak{c}_{i-1}$; $h_i^{(2)} \leftarrow t \mod 2^\eta$; $\mathfrak{c}_i \leftarrow \lfloor t/2^\eta \rfloor$
    end for
    $h_{\kappa-1}^{(2)} \leftarrow h_{\kappa-1}^{(1)} + \mathfrak{c}_{\kappa-2}$
    PARTIAL REDUCTION: **return** $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}$
    $h_{\kappa-1}^{(3)} \leftarrow h_{\kappa-1}^{(2)} \mod 2^\nu$; $\mathfrak{c}_{\kappa-1} \leftarrow \lfloor h_{\kappa-1}^{(2)}/2^\nu \rfloor$
    $t \leftarrow h_0^{(2)} + \delta \mathfrak{c}_{\kappa-1}$; $h_0^{(3)} \leftarrow t \mod 2^\eta$; $\mathfrak{c}_0 \leftarrow \lfloor t/2^\eta \rfloor$
    $h_1^{(3)} \leftarrow h_1^{(2)} + \mathfrak{c}_0$
    for $i \leftarrow 2$ to $\kappa - 2$ **do**
        $h_i^{(3)} \leftarrow h_i^{(2)}$
    end for
    FULL REDUCTION: **return** $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1}^{(3)}\theta^{\kappa-1}$
end function.

proof is similar to the proofs of the previous results and hence we skip the proof.

Theorem 4.4. *Let $p = 2^m - \delta$ be a prime in Table 4.3 and m be such that m-bit integers have (κ, η, ν)-representation where $\eta = 64$. Suppose the input $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}$ to* reduceSL *is such that $0 \leq h_i^{(0)} < 2^{128}$ for $i = 0, 1, \ldots, \kappa - 1$.*

1. *For partial reduction, the output $h^{(2)}(\theta)$ of* reduceSL *has a $(\kappa, \eta, \nu + 1)$-representation and $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*
2. *For full reduction, the output $h^{(3)}(\theta)$ of* reduceSL *has a (κ, η, ν)-representation and $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

Remark 4.8. *The squaring algorithm has some disadvantages using the above strategy. For the primes satisfying $\nu < \eta - 1$, the doubling involved in the squaring operation can efficiently*

compute the terms $2f_{\kappa-1} \cdot f_j$ for $j = 0, 1, \ldots, \kappa - 1$, by first computing $2f_{\kappa-1}$, through a shift and then multiplying by f_j. For reduction, we can opt for partial reduction, keeping an extra bit in the last limb, but then we cannot take the advantage in the doubling operation. For the primes satisfying $\nu = \eta$, we do not get any advantage with the doubling operation in the squaring algorithm and also full reduction is required.

4.7 Multiplication Using Unsaturated Limb Representation

The unsaturated limb representation has been very effectively used in the various implementations of Curve25519 [Ber06b, BS12, BDL+12, Cho15].

In the case of saturated limb representation, the tasks of integer multiplication/squaring and reduction are completely separate, i.e., the integer multiplication step simply multiplies two integers while the integer squaring step simply squares an integer without any reference to the prime which will be used to perform the reduction step. In the case of unsaturated limb representation, the multiplication/squaring step is not simply an integer multiplication/squaring. It uses the underlying prime to return an intermediate reduced result which is then provided as input to the reduction algorithm. Two strategies are described below. The first strategy is a generalization of a strategy used for the prime $2^{255} - 19$ in the implementation of Curve25519 [BDL+12] to arbitrary pseudo-Mersenne primes. For some primes however, this strategy leads to overflow in the intermediate result. To handle such cases, we describe a modified strategy which works for a larger class of primes. To the best of our knowledge, this modified strategy has not appeared earlier in the literature either in its general form or for any particular prime.

As in Section 4.2, let $p = 2^m - \delta$, $\theta = 2^\eta$, and $c_p = 2^{\eta-\nu}\delta$. Since we are working with the unsaturated limb representation, $\eta < 64$. Let $f(\theta)$ and $g(\theta)$ be two elements of $\mathbb{F}_p$ written as

$$\begin{aligned} f(\theta) &= f_0 + f_1\theta + \cdots + f_{\kappa-1}\theta^{\kappa-1}, \\ g(\theta) &= g_0 + g_1\theta + \cdots + g_{\kappa-1}\theta^{\kappa-1}, \end{aligned}$$

where $0 \leq f_i, g_i < 2^\eta$ for $i = 0, 1, \ldots, \kappa - 2$, and $0 \leq f_{\kappa-1},\ g_{\kappa-1} < 2^\nu$. The product of $f(\theta)$ and $g(\theta)$ modulo p can be written as the polynomial $h(\theta) = h_0 + h_1\theta + \cdots + h_{\kappa-1}\theta^{\kappa-1}$ where the coefficients h_i are given by (4.40). Substituting $g = f$, we get similar equations for squaring and during the squaring computation, each cross-product term is computed only once. We have

$$\mathfrak{h}_{\max} = \max(h_0, h_1, \ldots, h_{\kappa-1}). \tag{4.43}$$

If $\mathfrak{h}_{\max} < 2^{128}$, then each of the coefficients h_i, $i = 0, 1, \ldots, \kappa - 1$ fit in two 64-bit words. In such cases, the above strategy for multiplication/squaring is feasible. We denote the resulting algorithm for multiplication (resp. squaring) as mulUSL (resp. sqrUSL). We note that for some primes, $\mathfrak{h}_{\max}$ is significantly below 2^{128} and this plays a role in the efficient implementation of the subsequent reduction algorithm.

4.7.1 Modified Multiplication Strategy

In the case where $\mathfrak{h}_{\max} \geq 2^{128}$, the coefficients in (4.40) do not fit within two 64-bit words. For example, such a situation arises for the prime $2^{256} - 2^{32} - 977$. In this case, $\delta =$

$2^{32} + 977$, $\kappa = 5$, $\eta = 52$ and $\nu = 48$ so that $c_p = 16(2^{32} + 977)$ is a 37-bit integer. With these values, it can be verified that $\mathfrak{h}_{\max} \geq 2^{128}$. To handle situations where $\mathfrak{h}_{\max} \geq 2^{128}$, we describe a simple modification of the previous strategy. Define

$$\begin{aligned} u_0 &= f_1 g_{\kappa-1} + f_2 g_{\kappa-2} + \cdots + f_{\kappa-2} g_2 + f_{\kappa-1} g_1, \\ u_1 &= f_2 g_{\kappa-1} + \cdots + f_{\kappa-2} g_3 + f_{\kappa-1} g_2, \\ &\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ u_{\kappa-2} &= f_{\kappa-1} g_{\kappa-1}, \end{aligned} \tag{4.44}$$

where $\max(u_0, u_1, \ldots, u_{\kappa-2}) = u_0 \leq \mathfrak{u}_{\max}$ with $\mathfrak{u}_{\max} = (2^\eta - 1)^2 (\kappa - 1)$. For $i = 0, 1, \ldots, \kappa - 2$, let $u_{i,0} = u_i \bmod 2^\eta$, $u_{j,1} = \lfloor u_j / 2^\eta \rfloor$, so that

$$u_j = u_{j,0} + u_{j,1} 2^\eta = u_{j,0} + u_{j,1}\theta. \tag{4.45}$$

Then for $f(\theta) \cdot g(\theta) = h(\theta) = h_0 + h_1\theta + \cdots + h_{\kappa-1}\theta^{\kappa-1}$ such that the coefficients $h_0, h_1 \ldots, h_{\kappa-1}$ are given by (4.40), we have $h(\theta) = h'(\theta) = h'_0 + h'_1\theta + \cdots + h'_{\kappa-1}\theta^{\kappa-1}$ where

$$\begin{aligned} h'_0 &= f_0 g_0 + c_p u_{0,0}, \\ h'_1 &= f_0 g_1 + f_1 g_0 + c_p (u_{1,0} + u_{0,1}), \\ &\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ h'_{\kappa-2} &= f_0 g_{\kappa-2} + f_1 g_{\kappa-3} + \cdots + f_{\kappa-2} g_0 + c_p (u_{\kappa-2,0} + u_{\kappa-1,1}), \\ h'_{\kappa-1} &= f_0 g_{\kappa-1} + f_1 g_{\kappa-2} + \cdots + f_{\kappa-1} g_0 + c_p u_{\kappa-2,1}. \end{aligned} \tag{4.46}$$

Let

$$\mathfrak{h}'_{\max} = \max(h'_0, h'_1, \ldots, h'_{\kappa-1}). \tag{4.47}$$

If $\mathfrak{u}_{\max} < 2^{128}$ and $\mathfrak{h}'_{\max} < 2^{128}$, then each of the coefficients u_i, $i = 0, 1, \ldots, \kappa - 2$ and also each of the coefficients h_j, $j = 0, 1, \ldots, \kappa - 1$ fit in two 64-bit words. So, even if some coefficient of $h(\theta)$ is greater than or equal to 2^{128}, it is still feasible to compute $h'(\theta)$ using 64-bit arithmetic without any overflow. We denote the resulting multiplication and squaring algorithms by mulUSLa and sqrUSLa respectively.

Remark 4.9. *The rationale for obtaining $h'_0, h'_1, \ldots, h'_{\kappa-1}$ is that $h_0, h_1, \ldots, h_{\kappa-1}$ are not 128-bit quantities. For certain primes, it may happen that there is an $i \in \{0, 1, \ldots, \kappa - 1\}$ such that $h_0, h_1, \ldots, h_i$ are greater than $2^{128} - 1$ while $h_{i+1}, h_{i+2}, \ldots, h_{\kappa-1}$ are each at most $2^{128} - 1$. In such cases, it would be sufficient to use $h'_0, h'_1 \ldots, h'_i, h_{i+1}, \ldots, h_{\kappa-1}$.*

In [KS20], a different strategy was used to tackle the situation when $\mathfrak{h}_{\max} \geq 2^{128}$. This strategy consists of 'expanding' $u_0, u_1, \ldots, u_{\kappa-2}$ to κ, η-bit quantities $\mathfrak{u}_0, \mathfrak{u}_1, \ldots, \mathfrak{u}_{\kappa-2}$ and then adding $c_p u_0$ to $f_0 g_0$; $c_p u_1$ to $f_0 g_1 + f_1 g_0$ and so on. In the present case, this strategy turns out to be less efficient than the strategy used to obtain $h'_0, h'_1, \ldots, h'_{\kappa-1}$.

4.7.2 Dovetailing with Reduction Algorithms

The outputs of the multiplication/squaring algorithms are fed as inputs into the reduction algorithms and the outputs of the reduction algorithms are fed as inputs to the multiplication/squaring algorithms. In the (κ, η, ν)-representation of an m-bit integer,

each of the first $\kappa - 1$ limbs is η bits long and the last limb is ν bits long. So, one may consider the goal of the reduction algorithms to ensure that the output indeed has a (κ, η, ν)-representation. It is however, more efficient to obtain a partial reduction, where some of the coefficients of the output of the reduction algorithms may have one extra bit. Such a strategy is feasible, if the multiplication/squaring/addition algorithms applied to such inputs do not lead to any overflow. Based on such criterion, we describe three reduction algorithms.

Let $f(\theta) = f_0 + f_1\theta + \cdots + f_{\kappa-1}\theta^{\kappa-1}$ and $g(\theta) = g_0 + g_1\theta + \cdots + g_{\kappa-1}\theta^{\kappa-1}$.

General reduction algorithm. Let $\mathfrak{f}(\theta) = \mathfrak{f}_0 + \mathfrak{f}_1\theta + \cdots + \mathfrak{f}_{\kappa-1}\theta^{\kappa-1}$. Define a predicate genCond($\mathfrak{f}$) to be true if and only if

$$\begin{array}{rcccl} 0 & \leq & \mathfrak{f}_0, \mathfrak{f}_2, \ldots, \mathfrak{f}_{\kappa-2} & < & 2^{\eta}; \\ 0 & \leq & \mathfrak{f}_1 & < & 2^{\eta+1}; \\ 0 & \leq & \mathfrak{f}_{\kappa-1} & < & 2^{\nu}. \end{array} \tag{4.48}$$

Consider the following conditions.

1. The inputs $f(\theta)$ and $g(\theta)$ to mulUSL/sqrUSL or mulUSLa/sqrUSLa satisfy genCond(f) and genCond(g).
2. Let the output of mulUSL/sqrUSL or mulUSLa/sqrUSLa on such $f(\theta)$ and $g(\theta)$ be $h(\theta)$ or $h'(\theta)$ respectively. Suppose that $\mathfrak{h}_{\max} < 2^{127}$ or $\mathfrak{h}'_{\max} < 2^{127}$ as the case may be.
3. $\kappa \geq 3$.

If the above conditions hold, then we describe the reduction algorithm reduceUSL which takes as input either $h(\theta)$ or $h'(\theta)$ as the case may be and produces an output for which (4.48) holds.

Reduction algorithm for primes of Type A. Let $\mathfrak{f}(\theta) = \mathfrak{f}_0 + \mathfrak{f}_1\theta + \cdots + \mathfrak{f}_{\kappa-1}\theta^{\kappa-1}$. Define a predicate condA($\mathfrak{f}$) to be true if and only if

$$\begin{array}{rcccl} 0 & \leq & \mathfrak{f}_1, \mathfrak{f}_2, \ldots, \mathfrak{f}_{\kappa-2} & < & 2^{\eta}; \\ 0 & \leq & \mathfrak{f}_0 & < & 2^{\eta+1}; \\ 0 & \leq & \mathfrak{f}_{\kappa-1} & < & 2^{\nu}. \end{array} \tag{4.49}$$

Consider the following conditions.

1. The inputs $f(\theta)$ and $g(\theta)$ to mulUSL/sqrUSL or mulUSLa/sqrUSLa satisfy condA(f) and condA(g).
2. Let the output of mulUSL/sqrUSL or mulUSLa/sqrUSLa on such $f(\theta)$ and $g(\theta)$ be $h(\theta)$ or $h'(\theta)$ respectively. Suppose that $\mathfrak{h}_{\max} < 2^{\ell}$ or $\mathfrak{h}'_{\max} < 2^{\ell}$ as the case may be and $\ell < 63 + \nu$.
3. $\kappa \geq 3$.

If the above three conditions hold, then we describe the reduction algorithm reduceUSLA which takes as input either $h(\theta)$ or $h'(\theta)$ as the case may be and produces an output for which (4.49) holds. *The primes for which* reduceUSLA *applies have been identified as Type A in Table 4.4.*

Reduction algorithm for primes of Type B. Let $\mathfrak{f}(\theta) = \mathfrak{f}_0 + \mathfrak{f}_1\theta + \cdots + \mathfrak{f}_{\kappa-1}\theta^{\kappa-1}$. Define a predicate condB($\mathfrak{f}$) to be true if and only if

$$\begin{array}{rcccl} 0 & \leq & \mathfrak{f}_0, \mathfrak{f}_1, \ldots, \mathfrak{f}_{\kappa-2} & < & 2^{\eta+1}; \\ 0 & \leq & \mathfrak{f}_{\kappa-1} & < & 2^{\nu+1}. \end{array} \tag{4.50}$$

Consider the following conditions.

1. The inputs $f(\theta)$ and $g(\theta)$ to mulUSL/sqrUSL or mulUSLa/sqrUSLa satisfy condA(f) and condA(g).
2. Let the output of mulUSL/sqrUSL or mulUSLa/sqrUSLa on such $f(\theta)$ and $g(\theta)$ be $h(\theta)$ or $h'(\theta)$ respectively. Suppose that $\mathfrak{h}_{\max} < 2^{\ell}$ or $\mathfrak{h}'_{\max} < 2^{\ell}$ as the case may be and $64 + \nu < \ell \leq 128$.
3. $\kappa \geq 3$.

If the above three conditions hold, then we describe the reduction algorithm reduceUSLB which takes as input either $h(\theta)$ or $h'(\theta)$ as the case may be and produces an output for which (4.50) holds. *The primes for which* reduceUSLB *applies have been identified as Type B in Table 4.4.* There are two such primes. Note that the condition $\ell < 63 + \nu$ used to identify Type A primes and the condition $64 + \nu < \ell \leq 128$ used to identify Type B primes are non-exhaustive. They do not cover the values of $\ell = 63 + \eta$ and $\ell = 64 + \eta$. None of the primes in Table 4.3 correspond to such values of ℓ and so this is not an issue.

For the three primes identified as Type G in Table 4.4, the conditions required to apply either reduceUSLA or reduceUSLB do not hold. It is possible to consider further conditions to develop an algorithm for the Type G primes and we have indeed implemented such an algorithm. This algorithm however, turns out to be slower than the generic reduceUSL and so we do not describe the details of it.

prime	$2^{127}-1$	$2^{221}-3$	$2^{222}-117$	$2^{251}-9$	$2^{255}-19$	$2^{256}-2^{32}-977$	$2^{266}-3$
type	A	A	A	A	A	A	A
prime	$2^{382}-105$	$2^{383}-187$	$2^{414}-17$	$2^{511}-187$	$2^{512}-569$	$2^{521}-1$	$2^{607}-1$
type	B	B	A	G	G	A	G

Table 4.4: Classification of primes for application of reduceUSL, reduceUSLA or reduceUSLB.

4.8 Reduction Using Unsaturated Limb Representation

In this section, we describe several reduction algorithms which work with the unsaturated limb representation. The first of this is Function reduceUSL and is shown in Algorithm 4.8.

Theorem 4.5 below states the correctness of reduceUSL. The correctness is based on two assumptions both of which are valid for all the primes considered in this work.

Theorem 4.5. *Let $p = 2^m - \delta$ and m be such that m-bit integers have (κ, η, ν)-representation with $\kappa \geq 3$ and $\delta < 2^{2\eta+\nu-129}$. Suppose the input $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}$ to* reduceUSL *is such that $0 \leq h_i^{(0)} < 2^{128} - 2^{128-\eta}$ for $i = 0, 1, \ldots, \kappa - 1$.*

Algorithm 4.8 Reduction for unsaturated limb representation. Performs reduction modulo $p = 2^m - \delta$; m-bit integers have a (κ, η, ν)-representation with $\eta < 64$; $\theta = 2^\eta$.

1: **function** reduceUSL($h^{(0)}(\theta)$)
2: **input**: $h^{(0)}(\theta)$.
3: **output**: $h^{(1)}(\theta)$ or $h^{(2)}(\theta)$.
4: $u \leftarrow h_0^{(0)} \bmod 2^\eta$; $r_0 \leftarrow \lfloor h_0^{(0)}/2^\eta \rfloor$
5: $t_1 \leftarrow h_1^{(0)} + r_0$; $v \leftarrow t_1 \bmod 2^\eta$; $r_1 \leftarrow \lfloor t_1/2^\eta \rfloor$
6: **for** $i \leftarrow 2$ to $\kappa - 2$ **do**
7: $t_i \leftarrow h_i^{(0)} + r_{i-1}$; $h_i^{(1)} \leftarrow t_i \bmod 2^\eta$; $r_i \leftarrow \lfloor t_i/2^\eta \rfloor$
8: **end for**
9: $t_{\kappa-1} \leftarrow h_{\kappa-1}^{(0)} + r_{\kappa-2}$; $h_{\kappa-1}^{(1)} \leftarrow t_{\kappa-1} \bmod 2^\nu$; $r_{\kappa-1} \leftarrow \lfloor t_{\kappa-1}/2^\nu \rfloor$
10: $t \leftarrow u + \delta r_{\kappa-1}$; $h_0^{(1)} \leftarrow t \bmod 2^\eta$; $r_0 \leftarrow \lfloor t/2^\eta \rfloor$
11: $h_1^{(1)} \leftarrow v + r_0$
12: PARTIAL REDUCTION: **return** $h^{(1)}(\theta) = h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1}$
13: $w \leftarrow h_1^{(1)} \bmod 2^\eta$; $\mathfrak{c}_1 \leftarrow \lfloor h_1^{(1)}/2^\eta \rfloor$
14: **for** $i \leftarrow 2$ to $\kappa - 2$ **do**
15: $t \leftarrow h_i^{(1)} + \mathfrak{c}_{i-1}$; $h_i^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c}_i \leftarrow \lfloor t/2^\eta \rfloor$
16: **end for**
17: $t \leftarrow h_{\kappa-1}^{(1)} + \mathfrak{c}_{\kappa-2}$; $h_{\kappa-1}^{(2)} \leftarrow t \bmod 2^\nu$; $\mathfrak{c}_{\kappa-1} \leftarrow \lfloor t/2^\nu \rfloor$
18: $t \leftarrow h_0^{(1)} + \delta\mathfrak{c}_{\kappa-1}$; $h_0^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c}_0 \leftarrow \lfloor t/2^\eta \rfloor$
19: $t \leftarrow w + \mathfrak{c}_0$; $h_1^{(2)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t/2^\eta \rfloor$
20: $h_2^{(2)} \leftarrow h_2^{(2)} + \mathfrak{c}$
21: FULL REDUCTION: **return** $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}$
22: **end function**.

1. *For partial reduction, the output of* reduceUSL *is* $h^{(1)}(\theta) = h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1}$, *where* $0 \le h_1^{(1)} < 2^{\eta+1}$, $0 \le h_0^{(1)}, h_2^{(1)}, \ldots, h_{\kappa-2}^{(1)} < 2^\eta$ *and* $0 \le h_{\kappa-1}^{(1)} < 2^\nu$ *satisfying* $h^{(1)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

2. *For full reduction, the output* $h^{(2)}(\theta)$ *of* reduceUSL *has a* (κ, η, ν)*-representation and* $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

Proof. Since $0 \le h_0^{(0)} < 2^{128} - 2^{128-\eta}$, after Step 4 the bounds on u and r_0 are $0 \le u < 2^\eta$ and $0 \le r_0 < 2^{128-\eta}$ respectively. In Step 5 t is set to $h_1^{(0)} + r_0$ implying $0 \le t < 2^{128}$. Consequently we have $0 \le v < 2^\eta$ and $0 \le r_1 < 2^{128-\eta}$ respectively. After Steps 4-5

$h^{(0)}(\theta)$ can be written as

$$\begin{aligned} h^{(0)}(\theta) &= h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= (u + r_0\theta) + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + (h_1^{(0)} + r_0)\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + t_1\theta + h_2^{(0)}\theta^2 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + (v + r_1\theta)\theta + h_2^{(0)}\theta^2 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + v\theta + (h_2^{(0)} + r_1)\theta^2 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}. \end{aligned} \tag{4.51}$$

The coefficients $h_2^{(1)}, h_3^{(1)} \ldots, h_{\kappa-1}^{(1)}$ are computed in Steps 6-9 as follows.

$$\begin{aligned} h_j^{(1)} &= (h_j^{(0)} + r_{j-1}) \bmod 2^\eta \\ r_j &= \lfloor (h_j^{(0)} + r_{j-1})/2^\eta \rfloor, \qquad j = 2, 3, \ldots, \kappa-2, \\ h_{\kappa-1}^{(1)} &= (h_{\kappa-1}^{(0)} + r_{\kappa-2}) \bmod 2^\nu, \\ r_{\kappa-1} &= \lfloor (h_{\kappa-1}^{(0)} + r_{\kappa-2})/2^\nu \rfloor \end{aligned} \tag{4.52}$$

where $0 \le r_j < 2^{128-\eta}$ and $0 \le r_{\kappa-1} < 2^{128-\nu}$. Continuing from (4.51) and using (4.52), the effect of Steps 6-9 can be written in the following manner.

$$\begin{aligned} h^{(0)}(\theta) &= u + v\theta + (h_2^{(0)} + r_1)\theta^2 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + v\theta + t_1\theta^2 + h_3^{(0)}\theta^3 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + v\theta + (h_2^{(1)} + r_2\theta)\theta^2 + h_3^{(0)}\theta^3 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= u + v\theta + h_2^{(1)}\theta^2 + (h_3^{(0)} + r_2)\theta^3 + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ &= u + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-2}^{(1)}\theta^{\kappa-2} + (h_{\kappa-1}^{(0)} + r_{\kappa-2})\theta^{\kappa-1} \\ &= u + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-2}^{(1)}\theta^{\kappa-2} + t_{\kappa-1}\theta^{\kappa-1} \\ &= u + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-2}^{(1)}\theta^{\kappa-2} + (h_{\kappa-1}^{(1)} + r_{\kappa-1}2^\nu)\theta^{\kappa-1} \\ &= u + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} + r_{\kappa-1}2^{(\kappa-1)\eta+\nu} \text{ [since } \theta = 2^\eta\text{]} \\ &\equiv u + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} + r_{\kappa-1}\delta \mod p \text{ [using (4.4)]} \end{aligned} \tag{4.53}$$

where $0 \le u, v, h_2^{(1)}, h_3^{(1)}, \ldots, h_{\kappa-2}^{(1)} < 2^\eta$ and $0 \le h_{\kappa-1}^{(1)} < 2^\nu$. The bounds on $\delta r_{\kappa-1}$ are $0 \le \delta r_{\kappa-1} < 2^{2\eta+\nu-129} \cdot 2^{128-\nu} = 2^{2\eta-1}$. In Step 10 t is assigned to $u + \delta r_{\kappa-1}$ and so $0 \le t < 2^{2\eta}$. By the remaining two instructions of Step 10 we get $0 \le h_0^{(1)} < 2^\eta$ and $0 \le r_0 < 2^\eta$. By Step 11 we have $0 \le h_1^{(1)} < 2^{\eta+1}$. Hence, from (4.53) through Steps 10

and 11 we obtain

$$\begin{aligned} h^{(0)}(\theta) &\equiv (u + \delta r_{\kappa-1}) + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} \mod p \\ &= t + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} \\ &= (t \bmod 2^\eta + \lfloor t/2^\eta \rfloor\theta) + v\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} \text{ [since } \theta = 2^\eta] \\ &= h_0^{(1)} + (v + r_0)\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} \\ &= h_0^{(1)} + h_1^{(1)}\theta + h_2^{(1)}\theta^2 + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} = h^{(1)}(\theta), \end{aligned} \tag{4.54}$$

where $0 \leq h_1^{(1)} < 2^{\eta+1}$, $0 \leq h_0^{(1)}, h_2^{(1)}, \ldots, h_{\kappa-2}^{(1)} < 2^\eta$, and $h_{\kappa-1}^{(1)} < 2^\nu$. From (4.54) we have $h^{(1)}(\theta) \equiv h^{(0)}(\theta) \mod p$ which proves the statement on partial reduction.

To ensure full reduction another pass over the limbs is required. This is performed in Steps 13-20. First assume that $h_2^{(2)}$ computed in Step 20 satisfies $0 \leq h_2^{(2)} < 2^\eta$. Then, it is routine to argue in a manner similar to above that the final computed $h^{(2)}(\theta)$ is such that $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \mod p$ and $0 \leq h_i^{(2)} < 2^\eta$ for $i = 0, 1, \ldots, \kappa - 2$ and $0 \leq h_{\kappa-1}^{(2)} < 2^\nu$.

We now show that $h_2^{(2)}$ computed in Step 20 satisfies $0 \leq h_2^{(2)} < 2^\eta$. Since $h_1^{(1)} < 2^{\eta+1}$, this implies $\mathfrak{c}_1 \leq 1$. In the first iteration of the loop in Steps 14-17 $\mathfrak{c}_1$ is added to $h_2^{(1)}$ to obtain t. From this t $h_2^{(2)}$ is obtained as $t \bmod 2^\eta$ and $\mathfrak{c}_2$ is obtained as $\mathfrak{c} = \lfloor t/2^\eta \rfloor$. So, $\mathfrak{c}_2 \leq 1$. Since $h_2^{(1)} < 2^\eta$, $\mathfrak{c}_2 = 1$ if and only if $h_2^{(1)} = 2^\eta - 1$ and in this case $h_2^{(2)} = 0$. The following observations can be noted.

1. $\mathfrak{c}_i, \mathfrak{c} \leq 1$ for $i = 0, 1, \ldots, \kappa - 1$.
2. If $\mathfrak{c}_2 = 0$, then $\mathfrak{c}_i = 0$ for $i = 3, 4, \ldots, \kappa - 1$ and $\mathfrak{c}_0 = \mathfrak{c} = 0$.

So, if $\mathfrak{c}_2 = 0$, then $\mathfrak{c} = 0$ and so the value of $h_2^{(2)}$ computed in Step 20 is equal to the value of $h_2^{(2)}$ computed in Step 15. Since the value of $h_2^{(2)}$ computed in this step satisfies $0 \leq h_2^{(2)} < 2^\eta$ so does the value of $h_2^{(2)}$ computed in Step 20. On the other hand, if $\mathfrak{c}_2 = 1$ then the value of $h_2^{(2)}$ computed in Step 15 is 0 and so, the value of $h_2^{(2)}$ computed in Step 20 is equal to $\mathfrak{c} \leq 1$. So, in both cases the bounds $0 \leq h_2^{(2)} < 2^\eta$ hold. □

4.8.1 An Important Implementation Issue

In Steps 4, 5, 7, 9 and 10, the operations $w \bmod 2^\tau$ and $\lfloor w/2^\tau \rfloor$ are performed on a 128-bit quantity w where τ is either η or ν. The operation $\lfloor w/2^\tau \rfloor$ heavily influences the overall performance of the algorithm. We describe the implementation of the operations $w \bmod 2^\tau$ and $\lfloor w/2^\tau \rfloor$ in more details. The 128-bit quantity w is stored in two 64-bit words w_0 and w_1 such that $w = w_0 + w_1 2^{64}$. There are two cases to consider.

Case 1. $0 \leq w < 2^{128-\tau}$. In this case $0 \leq w_1 < 2^{64-\tau}$, i.e., w_1 is at most a $(64 - \tau)$-bit word. So, it is possible to left shift w_1 by τ bits and at the same time move in the τ most significant bits of w_0 into the τ least significant bits of w_1. The assembly level Intel instruction for doing this is `shld` and the two operations $w \bmod 2^\tau$ and $\lfloor w/2^\tau \rfloor$ are executed as follows.

```
shld 64 − τ, w0, w1
and  2^τ − 1, w0.
```

After executing these two steps, w_1 stores $\lfloor w/2^\tau \rfloor$ and w_0 stores $w \bmod 2^\tau$.

Case 2. $w \geq 2^{128-\tau}$. In this case, $0 \leq w_1 < 2^{64-\tau}$ and the length of w_1 in bits is more than $64 - \tau$ bits. So, left shifting w_1 by τ bits will result in loss of information and the strategy of Case 1 does not work. Further, the result of $\lfloor w/2^\tau \rfloor$ is more than 64 bits in length and requires two 64-bit words to be stored. The strategy in this case is the following. First copy w_0 to another 64-bit location x_0. Right shift τ bits of w_0 while moving in τ least significant bits of w_1 into the most significant bits of w_0. (The Intel instruction for doing this is `shrd`.) Then, right shift w_1 by τ bits. The two operations $w \bmod 2^\tau$ and $\lfloor w/2^\tau \rfloor$ are executed as follows.

```
mov  w0, x0
and  2^τ − 1, x0
shrd τ, w1, w0
shr  τ, w1.
```

After executing the above steps, x_0 stores $w \bmod 2^\tau$; w_0 stores the 64 least significant bits of $\lfloor w/2^\tau \rfloor$ and the $(64 - \tau)$ least significant bits of w_1 stores the $(64 - \tau)$ most significant bits of $\lfloor w/2^\tau \rfloor$.

Clearly Case 2 is more time consuming than Case 1. The applicability of Case 1 and Case 2 to the primes that we have considered are as follows.

1. For primes identified as type A in Table 4.4, Case 1 can be applied, except for the prime $2^{222} - 117$ where Case 2 needs to be applied only for Step 4 of reduceUSL.

2. For primes identified as type B in Table 4.4, Case 2 needs to be applied.

4.8.2 A Computational Bottleneck

The various computations $\lfloor \cdot/2^\tau \rfloor$ in reduceUSL are strictly sequential. Correspondingly, the operations `shld` or `shrd` as the case may be, are not independent and have to be executed in sequence. These are relatively high-latency operations and so the strict sequential execution of these operations have a negative impact on the overall performance of the algorithm.

We next describe two other reduction algorithms. The main motivation of these algorithms is to try and ensure that the operations `shld` or `shrd` are independent. Achieving such independence comes at the cost of increasing the total number of operations. Even then, for certain primes, the independence of these operations result in an overall faster algorithm.

Remark 4.10. *Steps 13-19 also use the operation $\lfloor \cdot/2^\tau \rfloor$, but these are on 64-bit quantities and can be efficiently implemented using the* `shr` *instruction.*

Algorithm 4.9 Improved reduction algorithm for primes identified as type A in Table 4.4 using unsaturated limb representation. Performs reduction modulo $p = 2^m - \delta$ and m-bit integers have a (κ, η, ν)-representation with $\eta < 64$; $\theta = 2^\eta$.

function reduceUSLA($h^{(0)}(\theta)$)
input: $h^{(0)}(\theta)$.
output: $h^{(2)}(\theta)$ or $h^{(3)}(\theta)$.
  $r \leftarrow h_0^{(0)} \bmod 2^\eta$
  for $i \leftarrow 1$ to $\kappa - 2$ **do**
    $h_i^{(1)} \leftarrow h_i^{(0)} \bmod 2^\eta + \lfloor h_{i-1}^{(0)} / 2^\eta \rfloor$
  end for
  $h_{\kappa-1}^{(1)} \leftarrow h_{\kappa-1}^{(0)} \bmod 2^\nu + \lfloor h_{\kappa-2}^{(0)} / 2^\eta \rfloor$
  $s \leftarrow \lfloor h_{\kappa-1}^{(0)} / 2^\nu \rfloor$; $h_0^{(1)} \leftarrow r + \delta s$
  $u \leftarrow h_0^{(1)} \bmod 2^\eta$; $r_0 \leftarrow \lfloor h_0^{(1)} / 2^\eta \rfloor$
  for $i \leftarrow 1$ to $\kappa - 2$ **do**
    $t_i \leftarrow h_i^{(1)} + r_{i-1}$; $h_i^{(2)} \leftarrow t_i \bmod 2^\eta$; $r_i \leftarrow \lfloor t_i / 2^\eta \rfloor$
  end for
  $t_{\kappa-1} \leftarrow h_{\kappa-1}^{(1)} + r_{\kappa-2}$; $h_{\kappa-1}^{(2)} \leftarrow t_{\kappa-1} \bmod 2^\nu$; $r_{\kappa-1} \leftarrow \lfloor t_{\kappa-1} / 2^\nu \rfloor$
  $h_0^{(2)} \leftarrow u + \delta r_{\kappa-1}$
  PARTIAL REDUCTION: **return** $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}$
  $v \leftarrow h_0^{(2)} \bmod 2^\eta$; $\mathfrak{c}_0 \leftarrow \lfloor h_0^{(2)} / 2^\eta \rfloor$
  for $i \leftarrow 1$ to $\kappa - 2$ **do**
    $t \leftarrow h_i^{(2)} + \mathfrak{c}_{i-1}$; $h_i^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c}_i \leftarrow \lfloor t / 2^\eta \rfloor$
  end for
  $t \leftarrow h_{\kappa-1}^{(2)} + \mathfrak{c}_{\kappa-2}$; $h_{\kappa-1}^{(3)} \leftarrow t \bmod 2^\nu$; $\mathfrak{c}_{\kappa-1} \leftarrow \lfloor t / 2^\nu \rfloor$
  $t \leftarrow v + \delta \mathfrak{c}_{\kappa-1}$; $h_0^{(3)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t / 2^\eta \rfloor$
  $h_1^{(3)} \leftarrow h_1^{(3)} + \mathfrak{c}$
  FULL REDUCTION: **return** $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_{\kappa-1}^{(3)}\theta^{\kappa-1}$
end function.

4.8.3 Improved Reduction for Type A Primes

Function reduceUSLA in Algorithm 4.9 describes a reduction algorithm which improves upon reduceUSL for primes identified as type A in Table 4.4.

The following result states the correctness of reduceUSLA.

Theorem 4.6. *Let $p = 2^m - \delta$ be a type A prime as identified in Table 4.4; m be such that m-bit integers have (κ, η, ν)-representation and $\delta < 2^{2\eta+\nu-130}$. Suppose the input $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}$ to* reduceUSLA *is such that $0 \le h_i^{(0)} < 2^\ell$ for $i = 0, 1, \ldots, \kappa - 1$ where $\ell < 63 + \nu$.*

1. *For partial reduction, the output of* reduceUSLA *is $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}$, where $0 \le h_0^{(2)} < 2^{\eta+1}$, $0 \le h_1^{(2)}, h_2^{(2)}, \ldots, h_{\kappa-2}^{(2)} < 2^\eta$ and $0 \le h_{\kappa-1}^{(2)} < 2^\nu$ satisfying $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

2. *For full reduction, the output* $h^{(3)}(\theta)$ *of* reduceUSLA *has a* (κ, η, ν)*-representation and* $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

Proof. Note that for all the primes identified as type A in Table 4.4, we have $\eta < 62$. Steps 4-9 convert the $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$ ensuring $h^{(1)}(\theta) \equiv h^{(0)}(\theta) \bmod p$ and Steps 10-15 convert $h^{(1)}(\theta)$ to $h^{(2)}(\theta)$ ensuring $h^{(2)}(\theta) \equiv h^{(1)}(\theta) \bmod p$. Define

$$\begin{aligned} h_j^{(0)} &= h_{j,0}^{(0)} + h_{j,1}^{(0)} 2^{\eta} && \text{where } h_{j,0}^{(0)} = h_j^{(0)} \bmod 2^{\eta},\ h_{j,1}^{(0)} = \lfloor h_j^{(0)}/2^{\eta} \rfloor, \\ & && \text{for } j = 0, 1, \ldots, \kappa-2; \text{ and} \\ h_{\kappa-1}^{(0)} &= h_{\kappa-1,0}^{(0)} + h_{\kappa-1,1}^{(0)} 2^{\nu} && \text{where } h_{\kappa-1,0}^{(0)} = h_{\kappa-1}^{(0)} \bmod 2^{\nu},\ h_{\kappa-1,1}^{(0)} = \lfloor h_{\kappa-1}^{(0)}/2^{\nu} \rfloor. \end{aligned}$$

Clearly, $0 \le h_{j,0}^{(0)} < 2^{\eta} < 2^{62}$ and $0 \le h_{j,1}^{(0)} < 2^{\ell-\eta}$ for $j = 0, 1 \ldots, \kappa-2$; $0 \le h_{\kappa-1,0}^{(0)} < 2^{\nu}$ and $0 \le h_{\kappa-1,1}^{(0)} < 2^{\ell-\nu}$. Using $\ell < 63 + \nu$ and $\eta \ge \nu$, we have $0 \le h_{j,1}^{(0)} < 2^{62}$ for $j = 0, 1, \ldots, \kappa-1$.

In Step 4, r is assigned the value $h_{0,0}^{(0)}$; for $i = 1, 2, \ldots, \kappa-2$, the i-th iteration of the loop in Steps 5-7 assigns the value $(h_{i,0}^{(0)} + h_{i-1,1}^{(0)})$ to $h_i^{(1)}$; Step 8 assigns the value $(h_{\kappa-1,0}^{(0)} + h_{\kappa-2,1}^{(0)})$ to $h_{\kappa-1}^{(1)}$; Step 9 assigns the value $h_{\kappa-1,1}^{(0)}$ to s. So, $0 \le r < 2^{\eta}$ and $0 \le s < 2^{\ell-\nu}$. Note that $2^{(\kappa-1)\eta+\nu} = 2^m \equiv \delta \bmod p$ and so $0 \le \delta s < 2^{2\eta+\ell-130} < 2^{2\eta-2}$, since $\delta < 2^{2\eta+\nu-130}$ and $\ell < 63 + \nu < 128$. Step 9 assigns the value $r + \delta s$ to $h_0^{(1)}$. The bounds on $h_i^{(1)}$ are

$$0 \le h_0^{(1)} < 2^{2\eta-1} \quad \text{and} \quad 0 \le h_i^{(1)} < 2^{63} \text{ for } i = 1, 2, \ldots, \kappa-1. \tag{4.55}$$

Using $\theta = 2^{\eta}$, we can write $h^{(0)}(\theta)$ as

$$\begin{aligned} h^{(0)}(\theta) &= h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &= (h_{0,0}^{(0)} + h_{0,1}^{(0)}\theta) + (h_{1,0}^{(0)} + h_{1,1}^{(0)}\theta)\theta + \cdots + (h_{\kappa-1,0}^{(0)} + h_{\kappa-1,1}^{(0)}2^{\nu})\theta^{\kappa-1} \\ &= h_{0,0}^{(0)} + (h_{0,1}^{(0)} + h_{1,0}^{(0)})\theta + \cdots + (h_{\kappa-2,1}^{(0)} + h_{\kappa-1,0}^{(0)})\theta^{\kappa-1} + h_{\kappa-1,1}^{(0)}2^{(\kappa-1)\eta+\nu} \\ &\equiv (r + \delta s) + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} \bmod p \text{ [using (4.4)]} \\ &= h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} = h^{(1)}(\theta). \end{aligned} \tag{4.56}$$

The argument that Steps 10-14 computes $h^{(2)}(\theta)$ such that $h^{(2)}(\theta) \equiv h^{(1)}(\theta) \bmod p$ and the limbs of $h^{(2)}(\theta)$ satisfy the stated bounds for partial reduction is similar to the proof of Theorem 4.5. The points to be noted are the following.

1. Since $0 \le h_0^{(1)} < 2^{2\eta-1}$, the value of u and r_0 computed in Step 10 satisfies $0 \le u < 2^{\eta}$ and $0 \le r_0 < 2^{\eta-1}$ respectively.

2. Since $0 \le h_i^{(1)} < 2^{63}$ for $i = 1, 2, \ldots, \kappa-2$, and $\eta < 64$, in Step 12 we have $0 \le t_i < 2^{64}$, $0 \le h_i^{(2)} < 2^{\eta}$ and $0 \le r_i < 2^{64-\eta}$ for $i = 1, 2, \ldots, \kappa-2$.

3. Since $0 \le h_{\kappa-1}^{(1)} < 2^{63}$, in Step 14 we have $0 \le t_{\kappa-1} < 2^{64}$, $0 \le h_{\kappa-1}^{(2)} < 2^{\nu}$ and $0 \le r_{\kappa-1} < 2^{64-\nu}$.

4. Since $0 \leq \delta < 2^{2\eta+\nu-130}$ and $0 \leq r^{(1)}_{\kappa-1} < 2^{64-\nu}$, in Step 15 we have $0 \leq \delta r_{\kappa-1} < 2^{2\eta-66} < 2^{\eta}$ for all the primes identified as type A in Table 4.4. This, along with $0 \leq u < 2^{\eta}$ implies $0 \leq h^{(2)}_0 < 2^{\eta+1}$ in Step 15.

The effect of Steps 10-15 on $h^{(1)}_1(\theta)$ can be written as

$$\begin{aligned}
h^{(1)}(\theta) &= h^{(1)}_0 + h^{(1)}_1\theta + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \\
&= (u + r_0\theta) + h^{(1)}_1\theta + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \\
&= u + (h^{(1)}_1 + r_0)\theta + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \\
&= u + t_1\theta + h^{(1)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \\
&= u + (h^{(2)}_1 + r_1\theta)\theta + h^{(1)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \\
&= u + h^{(2)}_1\theta + (h^{(1)}_2 + r_1)\theta^2 + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \\
\cdots \quad & \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\
&= u + h^{(2)}_1\theta + h^{(2)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-2}\theta^{\kappa-2} + (h^{(0)}_{\kappa-1} + r_{\kappa-2})\theta^{\kappa-1} \\
&= u + h^{(2)}_1\theta + h^{(2)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-2}\theta^{\kappa-2} + t_{\kappa-1}\theta^{\kappa-1} \\
&= u + h^{(2)}_1\theta + h^{(2)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-2}\theta^{\kappa-2} + (h^{(1)}_{\kappa-1} + r_{\kappa-1}2^{\nu})\theta^{\kappa-1} \\
&= u + h^{(2)}_1\theta + h^{(2)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} + r_{\kappa-1}2^{(\kappa-1)\eta+\nu} \text{ [since } \theta = 2^{\eta}\text{]} \\
&\equiv (u + r_{\kappa-1}\delta) + h^{(2)}_1\theta + h^{(2)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} \bmod p \text{ [using (4.4)]} \\
&= h^{(2)}_0 + h^{(2)}_1\theta + h^{(2)}_2\theta^2 + \cdots + h^{(1)}_{\kappa-1}\theta^{\kappa-1} = h^{(2)}(\theta). \qquad (4.57)
\end{aligned}$$

Using the points 1-4 mentioned before, we have the desired bounds on the limbs of $h^{(2)}(\theta)$ as $0 \leq h^{(2)}_0 < 2^{\eta+1}$, $0 \leq h^{(2)}_1, h^{(2)}_2, \ldots, h^{(2)}_{\kappa-2} < 2^{\eta}$ and $0 \leq h^{(2)}_{\kappa-1} < 2^{\nu}$. Combining (4.56) and (4.57) we have $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$ which proves the statement of partial reduction.

The statement on full reduction can be proved in a manner which is very similar to that of Theorem 4.5. □

Function reduceUSLA makes two passes over the limbs compared to reduceUSL which makes a single pass over the limbs. So, the total number of operations required by reduceUSLA is more than that of reduceUSL. Even then, for primes of type A, it turns out that reduceUSLA is faster than reduceUSL. The reason is explained below.

4.8.4 Independent Double Word Shifts

The computations $\lfloor \cdot / 2^{\eta} \rfloor$ in Steps 6 and 8 are on ℓ-bit quantities with $\ell < 63 + \eta$. This computation falls under Case 1 discussed after the proof of Theorem 4.5 and can be completed using a single `shld` instruction. The important difference between reduceUSL and reduceUSLA is that in the later case, the `shld` instructions are *independent*. So, these can be appropriately pipelined and may also be simultaneously scheduled on separate ALUs. It is this feature that leads to the speed up of reduceUSLA over reduceUSL. For example, on the Intel Skylake processor, for $p = 2^{255} - 19$, reduceUSLA takes 25 cycles whereas reduceUSL takes 37 cycles.

Function reduceUSLA has another set of shift operations in Step 14. These operations are on 64-bit words and hence can be computed using the `shr` instruction. This is true for all the primes except for $2^{222} - 117$, for which the first two limbs of $h^{(1)}(\theta)$ have more than 64 bits and hence the `shld` instruction has to be applied to extract the required leading bits of these limbs. The latency of `shr` instruction is smaller than the latency of `shld` instruction. The independence of the `shld` instructions in reduceUSLA more than compensates for the extra `shr` operations.

It is possible to avoid `shld` instruction and instead implement the desired functionality with the four instructions `shl`, `mov`, `shr` and `or`. We have implemented this strategy to try and speed up reduceUSL, but the resulting speed is still slower than that of reduceUSLA.

4.8.5 Improved Reduction for Type B Primes

There are two primes identified as type B in Table 4.4. If reduceUSLA is applied to these two primes, then the sizes of all the coefficients of $h^{(1)}(\theta)$ will be more than 64 bits. So, the subsequent steps of reduceUSLA will require application of `shld` instead of `shr`. Further, these `shld` instructions would not be independent. To avoid this situation, it is possible to make an extra pass over the limbs as in Steps 4-9 of reduceUSLA. This results in Function reduceUSLB which is given in Algorithm 4.10. Each limb of the partially reduced output of reduceUSLB has an extra bit. As mentioned in Section 4.7.2, only those primes are identified as type B for which this does not lead to an overflow in the multiplication and squaring algorithms.

The following result states the correctness of reduceUSLB.

Theorem 4.7. *Let $p = 2^m - \delta$ be a type B prime as identified in Table 4.4; m be such that m-bit integers have (κ, η, ν)-representation; and $\delta < 2^{2\eta+\nu-130}$. Suppose the input $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1}$ to* reduceUSLA *is such that $0 \le h_i^{(0)} < 2^{128}$ for $i = 0, 1, \ldots, \kappa - 1$.*

1. *For partial reduction, the output of* reduceUSLB *is $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}$, where $0 \le h_0^{(2)}, h_1^{(2)}, \ldots, h_{\kappa-2}^{(2)} < 2^{\eta+1}$ and $0 \le h_{\kappa-1}^{(2)} < 2^{\nu+1}$ satisfying $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

2. *For full reduction, the output $h^{(4)}(\theta)$ of* reduceUSLB *has a (κ, η, ν)-representation and $h^{(4)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

Proof. The first iteration of the loop in Steps 4-11 converts $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$. The correctness of this conversion can be argued in a manner similar to the first part of the proof of Theorem 4.6. In particular, we obtain

$$\begin{aligned} h^{(0)}(\theta) &= h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{\kappa-1}^{(0)}\theta^{\kappa-1} \\ &\equiv \underbrace{h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1}}_{\text{through first iteration of Steps 4-11}} \bmod p = h^{(1)}(\theta). \end{aligned} \tag{4.58}$$

Proceeding in a manner similar to the first part of the proof of Theorem 4.6, it can be shown that $0 \le h_0^{(1)} < 2^{2\eta-1}$ and $0 \le h_1^{(1)}, h_2^{(1)}, \ldots, h_{\kappa-1}^{(1)} < 2^{129-\eta}$.

Algorithm 4.10 Improved reduction algorithm for primes identified as type B in Table 4.4 using unsaturated limb representation. Performs reduction modulo $p = 2^m - \delta$ and m-bit integers have a (κ, η, ν)-representation with $\eta < 64$; $\theta = 2^\eta$.

1: **function** reduceUSLB($h^{(0)}(\theta)$)
2: **input**: $h^{(0)}(\theta)$.
3: **output**: $h^{(2)}(\theta)$ or $h^{(4)}(\theta)$.
4: **for** $\lambda \leftarrow 0$ to 1 **do**
5: $r \leftarrow h_0^{(\lambda)} \bmod 2^\eta$
6: **for** $i \leftarrow 1$ to $\kappa - 2$ **do**
7: $h_i^{(\lambda+1)} \leftarrow h_i^{(\lambda)} \bmod 2^\eta + \lfloor h_{i-1}^{(\lambda)} / 2^\eta \rfloor$
8: **end for**
9: $h_{\kappa-1}^{(\lambda+1)} \leftarrow h_{\kappa-1}^{(\lambda)} \bmod 2^\nu + \lfloor h_{\kappa-2}^{(\lambda)} / 2^\eta \rfloor$
10: $s \leftarrow \lfloor h_{\kappa-1}^{(\lambda)} / 2^\nu \rfloor$; $h_0^{(\lambda+1)} \leftarrow r + \delta s$
11: **end for**
12: PARTIAL REDUCTION: **return** $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}$
13: **for** $\lambda \leftarrow 2$ to 3 **do**
14: $h_0^{(\lambda+1)} \leftarrow h_0^{(\lambda)} \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor h_0^{(\lambda)} / 2^\eta \rfloor$
15: **for** $i \leftarrow 1$ to $\kappa - 2$ **do**
16: $t \leftarrow h_i^{(\lambda)} + \mathfrak{c}$; $h_i^{(\lambda+1)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t / 2^\eta \rfloor$
17: **end for**
18: $t \leftarrow h_{\kappa-1}^{(\lambda)} + \mathfrak{c}$; $h_{\kappa-1}^{(\lambda+1)} \leftarrow t \bmod 2^\nu$; $\mathfrak{c} \leftarrow \lfloor t / 2^\nu \rfloor$
19: $h_0^{(\lambda+1)} \leftarrow h_0^{(\lambda+1)} + \delta\mathfrak{c}$
20: **end for**
21: $t \leftarrow h_0^{(4)}$; $h_0^{(4)} \leftarrow t \bmod 2^\eta$; $\mathfrak{c} \leftarrow \lfloor t / 2^\eta \rfloor$
22: $h_1^{(4)} \leftarrow h_1^{(4)} + \mathfrak{c}$
23: FULL REDUCTION: **return** $h^{(4)}(\theta) = h_0^{(4)} + h_1^{(4)}\theta + \cdots + h_{\kappa-1}^{(4)}\theta^{\kappa-1}$
24: **end function**.

The second iteration of the loop in Steps 4-11 converts $h^{(1)}(\theta)$ to $h^{(2)}(\theta)$. The correctness of this argument is also similar to the first part of the proof of Theorem 4.6. The only thing required is to argue that the coefficients of $h^{(2)}(\theta)$ satisfy the stated bounds.

Step 5 of the second iteration provides r satisfying $0 \leq r < 2^\eta$. We have $\lfloor h_{i-1}^{(1)} / 2^\eta \rfloor < 2^{129-2\eta}$, $h_i^{(1)} \bmod 2^\eta < 2^\eta$ for $i = 1, 2, \ldots, \kappa - 2$, and $h_{\kappa-1}^{(1)} \bmod 2^\nu < 2^\nu < 2^\eta$. The following three observations hold for both the primes identified as type B in Table 4.4. Their consequences are also mentioned.

1. $129 - 2\eta = 129 - 2 \cdot 55 = 129 - 110 = 19 < \nu < \eta$. Consequently, after Steps 6-9 of the second iteration are $0 \leq h_1^{(2)}, h_2^{(2)}, \ldots, h_{\kappa-2}^{(2)} < 2^{\eta+1}$ and $0 \leq h_{\kappa-1}^{(2)} < 2^{\nu+1}$.

2. $\delta < 2^8$. Consequently, after Step 10, $\delta s < 2^8 \cdot 2^{129-2\nu} = 2^{137-2\nu}$.

3. $137 - 2\nu \leq 137 - 2 \cdot 52 = 137 - 104 = 33 < \eta$. Consequently, after Step 10, $0 \leq h_0^{(2)} < 2^{\eta+1}$.

So, we have

$$\begin{aligned} h^{(1)}(\theta) &= h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_{\kappa-1}^{(1)}\theta^{\kappa-1} \\ &\equiv \underbrace{h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_{\kappa-1}^{(2)}\theta^{\kappa-1}}_{\text{through second iteration of Steps 4-11}} \mod p = h^{(2)}(\theta), \end{aligned} \tag{4.59}$$

where $0 \leq h_0^{(2)}, h_1^{(2)}, \ldots, h_{\kappa-2}^{(2)} < 2^{\eta+1}$ and $0 \leq h_{\kappa-1}^{(2)} < 2^{\nu+1}$. Combining (4.58) and (4.59) we have $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$, which proves the statement on partial reduction.

The statement on full reduction can be proved routinely. The only point to be noted is that a single pass over the limbs is not sufficient to ensure termination and instead two passes are required. □

4.9 Implementations and Timings

Timings on Haswell and Skylake are shown in Tables 4.5, 4.6 and 4.7 respectively.

field	implementation type - maa			
	previous	this work	algorithm	sup
$\mathbb{F}_{2^{127}-1}$	(32, 32, 2797) [BCLS14]	(32, 30, 2503)	farith-SLMP	(0, 10.3, 10.5)
$\mathbb{F}_{2^{221}-3}$	-	(54, 45, 8082) (57, 56, 9957)	farith-USLA farith-USL	-
$\mathbb{F}_{2^{222}-117}$	-	(58, 46, 8385) (64, 55, 10798)	farith-USLA farith-USL	-
$\mathbb{F}_{2^{251}-9}$	(72, 55, 12202) [KS20]	(52, 46, 11245) (78, 55, 11803)	farith-SLa farith-USLA	(27.8, 16.35, 7.8)
$\mathbb{F}_{2^{255}-19}$	(72, 51, 12359) [BDL+12, 5-limb] (77, 64, 15880) [BDL+12, 4-limb]	(71, 50, 11854) (62, 54, 12393)	farith-USLA farith-SLa	(1.4, 2.0, 4.1)
$\mathbb{F}_{2^{256}-2^{32}-977}$	(86, 62, 20209) [SEC]	(55, 51, 12809) (70, 63, 17202)	farith-SLa farith-USL	(36.0, 17.7, 36.6)
$\mathbb{F}_{2^{266}-3}$	(72, 52, 12705) [KS20]	(71, 51, 12413) (85, 50, 14892)	farith-USLA farith-USL	(1.4, 2.0, 2.3)
$\mathbb{F}_{2^{382}-105}$	-	(119, 100, 33437) (127, 102, 39722)	farith-USLB farith-USL	-
$\mathbb{F}_{2^{383}-187}$	-	(119, 101, 33699) (127, 102, 39825)	farith-USLB farith-USL	-
$\mathbb{F}_{2^{414}-17}$	-	(161, 117, 43218) (130, 109, 44239)	farith-USLA farith-USLa	-
$\mathbb{F}_{2^{511}-187}$	-	(199, 144, 72804)	farith-USL	-
$\mathbb{F}_{2^{512}-569}$	-	(199, 144, 73771)	farith-USL	-
$\mathbb{F}_{2^{521}-1}$	(210, 166, 76298) [GS15]	(177, 129, 62244) (178, 132, 71546)	farith-USLA farith-USL	(15.7, 22.3, 18.4)
$\mathbb{F}_{2^{607}-1}$	-	(230, 156, 94149)	farith-USL	-

Table 4.5: Comparison of timings of various field arithmetic algorithms on Haswell. The work [BDL+12] was targeted for the Intel's Nehalem/Westmere CPUs.

For the maa implementations, all applicable algorithms have been implemented and timings recorded. To simplify the presentation, we provide timings for farith-USL. Here, the notation farith-USL denotes the algorithm triplet which computes a field-multiplication,

field-squaring and field-inverse using mulUSL, sqrUSL and reduceUSL. Similar interpretation applies for the other notations used in the tables. If the algorithm invUSL of farith-USL is not the fastest, then the timing of the triplet is provided which has the fastest time for inversion. The speed-up percentage corresponds to the faster of the two timings.

For the prime $2^{127}-1$, the maa type implementation is done using farith-SLMP. This is basically an optimized version of the implementation by Bernstein et al [BCLS14].

Based on Tables 4.5 to 4.7, we make the following observations.

1. Among the primes considered in this work, only the prime $2^{255}-19$ has a previous maax type implementation. For the other primes, we provide the first maax type implementations.

2. For each prime, where a previous maa implementation is available, we report a faster maa implementation. On both processors, the speed-up percentage varies from 2% to about 36%.

3. In comparison to previous work, we provide the fastest implementations for all the primes that are considered in this work. The speed-up is significant for some important primes. For the latest processors, the maax type implementations are faster than maa type implementations. Apart from the prime $2^{255}-19$, for all the other primes we provide the first maax type implementations.

field	implementation type - maa			
	previous	this work	algorithm	sup
$\mathbb{F}_{2^{127}-1}$	(27, 26, 2505) [BCLS14]	(26, 24, 2263)	farith-SLMP	(3.7, 7.7, 9.7)
$\mathbb{F}_{2^{221}-3}$	-	(58, 41, 7949) (60, 43, 8936)	farith-USLA farith-USL	-
$\mathbb{F}_{2^{222}-117}$	-	(55, 42, 8033) (60, 44, 10067)	farith-USLA farith-USL	-
$\mathbb{F}_{2^{251}-9}$	(66, 65, 13632) [KS20]	(50, 46, 11783) (65, 52, 12415)	farith-SLa farith-USLA	(24.2, 29.2, 13.6)
$\mathbb{F}_{2^{255}-19}$	(67, 48, 13223) [BDL+12, 5-limb] (67, 58, 13901) [BDL+12, 4-limb]	(65, 47, 12671) (57, 52, 12906)	farith-USLA farith-SLa	(3.0, 2.1, 4.2)
$\mathbb{F}_{2^{256}-2^{32}-977}$	(74, 54, 18391) [SEC]	(52, 49, 13242) (74, 63, 15565)	farith-SLa farith-USLa	(29.7, 9.3, 28.0)
$\mathbb{F}_{2^{266}-3}$	(66, 50, 14472) [KS20]	(65, 48, 13350) (71, 48, 14651)	farith-USLA farith-USL	(1.51, 4.0, 7.8)
$\mathbb{F}_{2^{382}-105}$	-	(107, 92, 30419) (115, 93, 35465)	farith-USLB farith-USL	-
$\mathbb{F}_{2^{383}-187}$	-	(107, 92, 30680) (115, 93, 35552)	farith-USLB farith-USL	-
$\mathbb{F}_{2^{414}-17}$	-	(127, 98, 38096) (126, 97, 39371)	farith-USLA farith-USLa	-
$\mathbb{F}_{2^{511}-187}$	-	(179, 131, 66039)	farith-USL	-
$\mathbb{F}_{2^{512}-569}$	-	(179, 131, 66808)	farith-USL	-
$\mathbb{F}_{2^{521}-1}$	(184, 142, 64924) [GS15]	(150, 116, 54790) (162, 121, 63938)	farith-USLA farith-USL	(18.5, 18.3, 15.6)
$\mathbb{F}_{2^{607}-1}$	-	(202, 137, 83587)	farith-USL	-

Table 4.6: Comparison of maa-timings of various field arithmetic algorithms on Skylake. The work [BDL+12] was targeted for the Intel's Nehalem/Westmere CPUs.

field	implementation type - maax			
	previous	this work	reduction	sup
$\mathbb{F}_{2^{127}-1}$	-	(26, 24, 2154)	farithx-SLMP	-
$\mathbb{F}_{2^{221}-3}$	-	(62, 43, 7728)	farithx-SLPMP	-
$\mathbb{F}_{2^{222}-117}$	-	(64, 40, 7967)	farithx-SLPMP	-
$\mathbb{F}_{2^{251}-9}$	-	(54, 47, 8784)	farithx-SLPMP	-
$\mathbb{F}_{2^{255}-19}$	(62, 49, 12170) [OLH+17]	(54, 42, 9301)	farithx-SLPMP	(12.9, 14.3, 23.6)
$\mathbb{F}_{2^{256}-2^{32}-977}$	-	(65, 53, 11501)	farithx-SLPMP	-
$\mathbb{F}_{2^{266}-3}$	-	(65, 53, 12938)	farithx-SLPMP	-
$\mathbb{F}_{2^{382}-105}$	-	(81, 69, 24549)	farithx-SLPMP	-
$\mathbb{F}_{2^{383}-187}$	-	(81, 69, 24628)	farithx-SLPMP	-
$\mathbb{F}_{2^{414}-17}$	-	(97, 80, 30972)	farithx-SLPMP	-
$\mathbb{F}_{2^{511}-187}$	-	(118, 101, 47062)	farithx-SLPMP	-
$\mathbb{F}_{2^{512}-569}$	-	(125, 106, 49713)	farithx-SLPMP	-
$\mathbb{F}_{2^{521}-1}$	-	(128, 108, 53828)	farithx-SLMP	-
$\mathbb{F}_{2^{607}-1}$	-	(159, 129, 74442)	farithx-SLMP	-

Table 4.7: Comparison of maax-timings of various field arithmetic algorithms on Skylake.

Timings for field multiplications and squarings are given in Tables 4.5 to 4.7. A field multiplication/squaring consists of an integer multiplication/squaring and a reduction step. The reduction step requires a significant amount of the total time. To provide an idea of the time required by the reduction step, we have obtained timings for integer multiplication/squaring operations in the maax setting. These timings are shown in Table 4.8. Comparing the timings for integer multiplication/squaring in Table 4.8 to the timings for field multiplication/squarings in Table 4.7 provides an idea of the time required for the reduction steps.

field(s)	(mult, sqr)
$\mathbb{F}_{2^{127}-1}$	(14, 13)
$\mathbb{F}_{2^{221}-3}$, $\mathbb{F}_{2^{222}-117}$	(27, 23)
$\mathbb{F}_{2^{251}-9}$, $\mathbb{F}_{2^{255}-19}$, $\mathbb{F}_{2^{256}-2^{32}-977}$	(30, 25)
$\mathbb{F}_{2^{266}-3}$	(42, 33)
$\mathbb{F}_{2^{382}-105}$, $\mathbb{F}_{2^{383}-187}$	(59, 50)
$\mathbb{F}_{2^{414}-17}$	(73, 65)
$\mathbb{F}_{2^{511}-187}$, $\mathbb{F}_{2^{512}-569}$	(90, 82)
$\mathbb{F}_{2^{521}-1}$	(109, 96)
$\mathbb{F}_{2^{607}-1}$	(143, 113)

Table 4.8: Timings for integer multiplication and squaring in the maax setting on Skylake.

4.10 Conclusion

In this chapter, we have considered efficient algorithms for multiplication and squaring over (pseudo-)Mersenne prime order fields. Our contributions have been two fold. On the theoretical side, we provide various algorithms for multiplication/squaring and reduction. The correctness of the reduction algorithms have been rigorously proven. On the practical side, we provide efficient assembly implementation of the various algo-

rithms for modern Intel processors. For well known primes our implementations are faster than the previous works. We have made all our source codes publicly available.

CHAPTER 5

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

5.1 Introduction

Let $p = 2^{448} - 2^{224} - 1$. The implementation of elliptic curve operations require arithmetic in the underlying field $\mathbb{F}_p$. Specifically, addition, subtraction, multiplication and squaring are required. Additionally, to implement the Montgomery ladder for Curve448, it is required to implement multiplication by a small constant. 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 $\mathbb{F}_p$. All the field operations require reduction modulo p while dealing with saturated limb representation of field elements. 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.

We deal with both saturated and unsaturated limb representations of elements of $\mathbb{F}_p$. The field elements are represented using 7 limbs for saturated limb representation and 8 limbs for unsaturated limb representation. Our focus is on the reduction algorithms which are required to implement field arithmetic operations in $\mathbb{F}_p$. We 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.

5.1.1 Efficient 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 [LH16]. 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 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. We have made our software publicly available at the link

`https://github.com/kn-cs/x448`.

5.1.2 Related Work

Efficient implementation of elliptic curve cryptography requires efficient implementation of arithmetic in the underlying finite field. Good introductions to implementation of field arithmetic can be found in [CFA+05, MvOV96]. 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 [Ber06b]. 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 [Ber06b, Cho15, DHH+15, FL15]. Algorithms for implementing field arithmetic over Mersenne and pseudo-Mersenne primes have been discussed in Chapter 4. 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 [Mon85] is generally found to be helpful for efficient implementations [BM17]. An implementation of NIST P-256 based using Montgomery arithmetic has been reported in [GK15]. The NIST P-256 prime is a Montgomery-friendly prime and so reduction is faster than usual in Montgomery arithmetic.

5.2 Arithmetic in $\mathbb{F}_p$ using Saturated Limb Representation

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

$$f(\theta) \quad = \quad f_0 + f_1\theta + \cdots + f_d\theta^d \tag{5.1}$$

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 [OGGF12, OGG13]. Explicit descriptions of the squaring and multiplication algorithms in the general case have been provided in Chapter 4.

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 [Mon87] 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.

5.3 Reduction in $\mathbb{F}_p$ using Saturated Limb Representation

In Section 5.3.1 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 [1]. This is pointed out in Section 5.3.2. The case of subtraction in $\mathbb{F}_p$ is described in Section 5.3.3.

5.3.1 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 reducep448_7L of Algorithm 5.1. All the operations in reducep448_7L can be performed using 64-bit arithmetic instructions available in modern

[1] Knowing that the eighth limb is single bit in the case of addition doesn't provide any advantage in reduction.

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 reducep448_7L 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 reducep448_7L 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 5.1. 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 5.1. *Suppose the input* $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^{13}$ *to* reducep448_7L *is such that* $0 \leq h_i^{(0)} < 2^{64}$ *for* $i = 0, 1, \ldots, 13$. *Then the output* $h^{(3)}(\theta)$ *of* reducep448_7L *is such that* $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_6^{(3)}(\theta)$ *with* $0 \leq h_j^{(3)} < 2^{64}$ *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

$$2^{448} \;=\; \theta^7 \;\equiv\; 2^{224} + 1 \;=\; 2^{\eta/2}\theta^3 + 1 \mod p. \tag{5.2}$$

Reduction from $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$. The input $h^{(0)}(\theta)$ to reducep448_7L can be written as

$$\begin{aligned}
h^{(0)}(\theta) &= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)}\theta^7 + h_8^{(0)}\theta^8 + \cdots + h_{13}^{(0)}\theta^{13}),\\
&= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)\theta^7,\\
&\equiv (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)2^{\eta/2}\theta^3 +\\
&\quad (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6) \bmod p \qquad \text{[using (5.2)]},\\
&= (h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_6^{(0)}\theta^6) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6) +\\
&\quad (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)\theta^3 2^{\eta/2}.
\end{aligned} \tag{5.3}$$

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 $(r_0^{(0)} + r_1^{(0)}\theta +$

Algorithm 5.1 Reduction from 14-limb to 7-limb in $\mathbb{F}_p$. In the algorithm, $\eta = 64$.

function reducep448_7L($h^{(0)}(\theta)$)
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 $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.

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

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

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

$\cdots + r_7^{(0)}\theta^7)$. Hence, from (5.3) we write

$$h^{(0)}(\theta) \equiv \underbrace{(r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7)}_{\text{through Steps 4-8}} + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{13}^{(0)}\theta^6)\theta^3 2^{\eta/2} \mod p, \tag{5.4}$$

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

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

$$h_j^{(0)} = h_{j,0}^{(0)} + h_{j,1}^{(0)} 2^{\eta/2}, \text{ where } h_{j,0}^{(0)} = h_j^{(0)} \bmod 2^{\eta/2}, \text{ and } h_{j,1}^{(0)} = \lfloor h_j^{(0)} / 2^{\eta/2} \rfloor. \tag{5.5}$$

Using (5.5) for $j = 10$ we can write (5.4) as

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

which can be further written as

$$\begin{aligned} h^{(0)}(\theta) &\equiv (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2} + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)} 2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2})\theta^7 \mod p, \\ &\equiv (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2} + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)} 2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2})(\theta^3 2^{\eta/2} + 1) \mod p \quad \text{[using (5.2)]}, \\ &= (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (h_{10,1}^{(0)} + h_{11}^{(0)} 2^{\eta/2} + \cdots + h_{13}^{(0)}\theta^2 2^{\eta/2})\theta^3 2^{\eta/2} + \\ &\quad (h_{10,1}^{(0)} + h_{11}^{(0)} 2^{\eta/2} + \cdots + h_{13}^{(0)}\theta^2 2^{\eta/2}) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2}, \\ &= (r_0^{(0)} + r_1^{(0)}\theta + \cdots + r_7^{(0)}\theta^7) + (2^{\eta/2} h_{10,1}^{(0)}\theta^3 + h_{11}^{(0)}\theta^4 + \cdots + h_{13}^{(0)}\theta^6) + (h_{10,1}^{(0)} + \\ &\quad h_{11}^{(0)} 2^{\eta/2} + \cdots + h_{13}^{(0)}\theta^2 2^{\eta/2}) + (h_7^{(0)} + h_8^{(0)}\theta + \cdots + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2}. \end{aligned} \tag{5.6}$$

Steps 9-14 add the two 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 (5.6) we write

$$h^{(0)}(\theta) \equiv \underbrace{(s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7)}_{\text{through Steps 9-14}} + (h_{10,1}^{(0)} + h_{11}^{(0)} 2^{\eta/2} + h_{12}^{(0)}\theta 2^{\eta/2} + h_{13}^{(0)}\theta^2 2^{\eta/2}) + (h_7^{(0)} + h_8^{(0)}\theta + h_9^{(0)}\theta^2 + h_{10,0}^{(0)}\theta^3)\theta^3 2^{\eta/2} \mod p, \tag{5.7}$$

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 (5.5) we can further write (5.7) as

$$\begin{aligned} h^{(0)}(\theta) &\equiv (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7) + h_{10,1}^{(0)} + (h_{11,0}^{(0)} + h_{11,1}^{(0)}2^{\eta/2})2^{\eta/2} + \\ &\quad (h_{12,0}^{(0)} + h_{12,1}^{(0)}2^{\eta/2})\theta 2^{\eta/2} + (h_{13,0}^{(0)} + h_{13,1}^{(0)}2^{\eta/2})\theta^2 2^{\eta/2} + (h_{7,0}^{(0)} + h_{7,1}^{(0)}2^{\eta/2})\theta^3 2^{\eta/2} + \\ &\quad (h_{9,0}^{(0)} + h_{9,1}^{(0)}2^{\eta/2})\theta^5 2^{\eta/2} + (h_{8,0}^{(0)} + h_{8,1}^{(0)}2^{\eta/2})\theta^4 2^{\eta/2} + h_{10,0}^{(0)}\theta^6 2^{\eta/2} \mod p, \\ &= (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7) + (h_{10,1}^{(0)} + h_{11,0}^{(0)}2^{\eta/2}) + (h_{11,1}^{(0)} + h_{12,0}^{(0)}2^{\eta/2})\theta + \\ &\quad (h_{12,1}^{(0)} + h_{13,0}^{(0)}2^{\eta/2})\theta^2 + (h_{13,1}^{(0)} + h_{7,0}^{(0)}2^{\eta/2})\theta^3 + (h_{7,1}^{(0)} + (h_{8,0}^{(0)}2^{\eta/2})\theta^4 + \\ &\quad (h_{8,1}^{(0)} + h_{9,0}^{(0)}2^{\eta/2})\theta^5 + (h_{9,1}^{(0)} + h_{10,0}^{(0)}2^{\eta/2})\theta^6, \\ &= (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^7) + \underbrace{(t_0^{(0)} + t_1^{(0)}\theta + \cdots + t_6^{(0)}\theta^6)}_{\text{through Steps 15-21}}. \end{aligned} \tag{5.8}$$

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 (5.8) we can write

$$\begin{aligned} h^{(0)}(\theta) &\equiv (s_0^{(0)} + s_1^{(0)}\theta + \cdots + s_7^{(0)}\theta^6) + (t_0^{(0)} + t_1^{(0)}\theta + \cdots + t_6^{(0)}\theta^6) \mod p, \\ &= \underbrace{(h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_7^{(1)}\theta^7)}_{\text{through Steps 22-26}} = h^{(1)}(\theta), \end{aligned} \tag{5.9}$$

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 $h^{(2)}(\theta)$. The polynomial $h^{(1)}(\theta)$ can further be written as

$$\begin{aligned} h^{(1)}(\theta) &\equiv h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_6^{(1)}\theta^6 + h_7^{(1)}(2^{\eta/2}\theta^3 + 1) \mod p \text{ [using (5.2)]}, \\ &= (h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_6^{(1)}\theta^6) + (h_7^{(1)} + 2^{\eta/2}h_7^{(1)}\theta^3). \end{aligned} \tag{5.10}$$

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 (5.10) we write

$$\begin{aligned} h^{(1)}(\theta) &\equiv (h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_6^{(1)}\theta^6) + (h_7^{(1)} + 2^{\eta/2}h_7^{(1)}\theta^3) \mod p, \\ &= \underbrace{(h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_7^{(2)}\theta^7)}_{\text{through Steps 27-33}} = h^{(2)}(\theta), \end{aligned} \tag{5.11}$$

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 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 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 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 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

$$h_3^{(2)} < 2^{3\eta/4},\ h_4^{(2)} = h_5^{(2)} = h_6^{(2)} = 0. \tag{5.12}$$

have to hold.

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

$$\begin{aligned} h^{(2)}(\theta) &\equiv h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6 + h_7^{(2)}(2^{\eta/2}\theta^3 + 1) \mod p \text{ [using (5.2)]}, \\ &= (h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6) + (h_7^{(2)} + 2^{\eta/2}h_7^{(2)}\theta^3). \end{aligned} \tag{5.13}$$

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 (5.12) 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 (5.12) 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 $0 \le h_j^{(3)} < 2^\eta, j = 0, 1, 2, 3$, and $h_4^{(3)} = h_5^{(3)} = h_6^{(3)} = 0$. Hence, in any case from (5.13) it follows that

$$\begin{aligned} h^{(2)}(\theta) &\equiv (h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6) + (h_7^{(2)} + 2^{\eta/2}h_7^{(2)}\theta^3) \mod p, \\ &= \underbrace{(h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_6^{(3)}\theta^6)}_{\text{through Steps 34-38}} = h^{(3)}(\theta), \end{aligned} \tag{5.14}$$

where $0 \le h_0^{(3)}, h_1^{(3)}, \cdots, h_6^{(3)} < 2^\eta$. Also, by combining (5.9), (5.11) and (5.14) we have $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$, which proves the theorem. □

Remark 5.1. *Note that for a non-negative integer x and a positive integer μ, the operation $x \bmod 2^\mu$ extracts the μ least significant bits of x, while the operation $\lfloor x/2^\mu \rfloor$ returns an integer obtained by dropping the μ least significant 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 5.1 concatenate the 32 least significant 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 5.4.*

5.3.2 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 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 reducep448_7L which reduces $h^{(1)}(\theta)$ to $h^{(3)}(\theta)$. The correctness of the reduction is guaranteed by the part of the proof of Theorem 5.1 which argues the correctness of the reduction from $h^{(1)}(\theta)$ to $h^{(2)}(\theta)$ and from $h^{(2)}(\theta)$ to $h^{(3)}(\theta)$.

5.3.3 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 subp448_7L of Algorithm 5.2 performs this computation. The description of subp448_7L 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.

$$z = \begin{cases} x - (y + \mathfrak{b}_0) & \text{if } x \geq y + \mathfrak{b}_0, \\ 2^{64} + x - (y + \mathfrak{b}_0) & \text{if } x < y + \mathfrak{b}_0; \end{cases} \tag{5.15}$$

$$\mathfrak{b}_1 = \begin{cases} 0 & \text{if } x \geq y + \mathfrak{b}_0, \\ 1 & \text{if } x < y + \mathfrak{b}_0. \end{cases} \tag{5.16}$$

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 $\mathsf{sub}(x, y, \mathfrak{b}_0)$.

The correctness of subp448_7L is stated in the following theorem.

Theorem 5.2. *The output* $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6$ *of* subp448_7L *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 subp448_7L 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 $h^{(0)}(\theta) = 2^{448} + f(\theta) - g(\theta)$.

Algorithm 5.2 Subtraction in $\mathbb{F}_p$.

1: **function** subp448_7L($(f(\theta), g(\theta))$)
2: **input:** 7-limb quantities $f(\theta)$ and $g(\theta)$ such that $0 \leq f_i, g_j < 2^{64}$ for $i, j = 0, 1, \ldots, 6$.
3: **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 $h^{(2)}(\theta) \equiv (f(\theta) - g(\theta)) \bmod p$.

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

8: $\mathfrak{d} \leftarrow \mathfrak{b}$; $\mathfrak{d}' \leftarrow \mathfrak{d} \ll 32$
9: $\mathfrak{b} \leftarrow 0$
10: $(h_0^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_0^{(0)}, \mathfrak{d}, \mathfrak{b})$
11: $(h_1^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_1^{(0)}, 0, \mathfrak{b})$
12: $(h_2^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_2^{(0)}, 0, \mathfrak{b})$
13: $(h_3^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_3^{(0)}, \mathfrak{d}', \mathfrak{b})$
14: $(h_4^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_4^{(0)}, 0, \mathfrak{b})$
15: $(h_5^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_5^{(0)}, 0, \mathfrak{b})$
16: $(h_6^{(1)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_6^{(0)}, 0, \mathfrak{b})$

17: $\mathfrak{d} \leftarrow \mathfrak{b}$; $\mathfrak{d}' \leftarrow \mathfrak{d} \ll 32$
18: $\mathfrak{b} \leftarrow 0$
19: $(h_0^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_0^{(1)}, \mathfrak{d}, \mathfrak{b})$
20: $(h_1^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_1^{(1)}, 0, \mathfrak{b})$
21: $(h_2^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_2^{(1)}, 0, \mathfrak{b})$
22: $(h_3^{(2)}, \mathfrak{b}) \leftarrow \mathsf{sub}(h_3^{(1)}, \mathfrak{d}', \mathfrak{b})$
23: $h_4^{(2)} \leftarrow h_4^{(1)}$; $h_5^{(2)} \leftarrow h_5^{(1)}$; $h_6^{(2)} \leftarrow h_6^{(1)}$

24: **return** $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_6^{(2)}\theta^6$

25: **end function**.

(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[2]If the value of $\mathfrak{d}'$ in Step 17 is 0, then it follows that the value of $\mathfrak{b}$ in the output of

[2]An earlier version of this argument had an error. Thanks to Timothy Shelton [She21] for correcting it.

Step 22 is also 0. So, suppose that the value of $\mathfrak{d}'$ in Step 17 is 2^{32}. In this case, a sufficient condition for the value of $\mathfrak{b}$ produced by the sub call in Step 22 to be 0 is $h_3^{(1)} \geq 2^{32}+1$. The value of $\mathfrak{d}'$ in Step 17 is 2^{32}, 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. Denote by $\mathfrak{b}_0$ (resp. $\mathfrak{b}_1$) the value of $\mathfrak{b}$ in the input (resp. output) of Step 13. By the previous argument, we have $\mathfrak{b}_1 = 1$. From the definition of sub, $\mathfrak{b}_1 = 1$ occurs if and only if $h_3^{(0)} < \mathfrak{d}' + \mathfrak{b}_0$ and in this case $h_3^{(1)} = 2^{64} + h_3^{(0)} - (\mathfrak{d}' + \mathfrak{b}_0)$. Since $\mathfrak{d}' \leq 2^{32}$ and $\mathfrak{b}_0 \leq 1$, it follows that $h_3^{(1)} \geq 2^{32}+1$ as required. □

5.4 Comparison to the Reduction of [OLH+17]

Page 17 of [OLH+17], 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 is given below.

$$z \leftarrow (z \bmod 2^{672}) + (2^{448} + 2^{224})\lfloor z/2^{672} \rfloor, \tag{5.17}$$
$$z \leftarrow (z \bmod 2^{448}) + (2^{224} + 1)\lfloor z/2^{448} \rfloor, \tag{5.18}$$
$$z \leftarrow (z \bmod 2^{448}) + (2^{224} + 1)\lfloor z/2^{448} \rfloor. \tag{5.19}$$

The first two steps (5.17) and (5.18) 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 (5.17) reduces $h^{(0)}(\theta)$ to an 11-limb polynomial, say $f(\theta) = f_0 + f_1\theta + \cdots + f_{10}\theta^{10}$ and Step (5.18) reduces $f(\theta)$ to $h^{(1)}(\theta)$. Step (5.19) 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 [OLH+17] is incomplete. An example of the incompleteness in reduction has been discussed in the Appendix of [NS19].

5.4.1 Reduction Algorithms Used in the Code Accompanying [OLH+17]

We have studied the latest version[3] of the implementation corresponding to [OLH+17]. 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 5.1 proceeds only up to the fourth limb. Theorem 5.1 shows that this is sufficient. The present version of the code corresponding to [OLH+17] 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.

5.4.2 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 (5.2). We can also view $h^{(0)}(\theta)$ as an equivalent polynomial $\mathfrak{h}^{(0)}$ in base ϕ defined as $\mathfrak{h}^{(0)}(\phi) = a + b\phi + c\phi^2 + d\phi^3$,

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

where $a, b, c, d < \phi$. Under such a consideration, the reduction from $h^{(0)}(\theta)$ to $h^{(1)}(\theta)$ of Algorithm 5.1 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 5.1 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 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 [OLH+17] 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 require extra operations to back up d for computing $(d + c\phi)$ later on. As a result, the number of load/stores will increase.

5.4.3 Inline Assembly Code of Reduction from [OLH+17]

We produce below the inline assembly code of reduction after integer multiplication/squaring from the implementation of [OLH+17].

```
1  void red_EltFp448_1w_x64(uint64_t *c, uint64_t *a) {
2
3    __asm__ __volatile__(
4
5        /**
6         * (   ,2C13,2C12,2C11,2C10|C10,C9,C8, C7) + (C6,...,C0)
7         * (r14, r13, r12, r11,     r10,r9,r8,r15)
8         */
```

```
9         "movq 80(%1),%%rax; movq %%rax,%%r10;"
10        "movq $0xffffffff00000000, %%r8;"
11        "andq %%r8,%%r10;"
12
13        "movq $0,%%r14;"
14        "movq 104(%1),%%r13; shldq $1,%%r13,%%r14;"
15        "movq  96(%1),%%r12; shldq $1,%%r12,%%r13;"
16        "movq  88(%1),%%r11; shldq $1,%%r11,%%r12;"
17        "movq  72(%1), %%r9; shldq $1,%%r10,%%r11;"
18        "movq  64(%1), %%r8; shlq  $1,%%r10;"
19        "movq $0xffffffff,%%r15; andq %%r15,%%rax; orq %%rax,%%r10;"
20        "movq  56(%1),%%r15;"
21
22        "addq  0(%1),%%r15; movq %%r15, 0(%0); movq  56(%1),%%r15;"
23        "adcq  8(%1), %%r8; movq  %%r8, 8(%0); movq  64(%1), %%r8;"
24        "adcq 16(%1), %%r9; movq  %%r9,16(%0); movq  72(%1), %%r9;"
25        "adcq 24(%1),%%r10; movq %%r10,24(%0); movq  80(%1),%%r10;"
26        "adcq 32(%1),%%r11; movq %%r11,32(%0); movq  88(%1),%%r11;"
27        "adcq 40(%1),%%r12; movq %%r12,40(%0); movq  96(%1),%%r12;"
28        "adcq 48(%1),%%r13; movq %%r13,48(%0); movq 104(%1),%%r13;"
29        "adcq     $0,%%r14;"
30
31        /**
32         * (c10c9,c9c8,c8c7,c7c13,c13c12,c12c11,c11c10) + (c6,...,c0)
33         * (   r9,  r8, r15,  r13,   r12,   r11,   r10)
34         */
35        "movq %%r10, %%rax;"
36        "shrdq $32,%%r11,%%r10;"
37        "shrdq $32,%%r12,%%r11;"
38        "shrdq $32,%%r13,%%r12;"
39        "shrdq $32,%%r15,%%r13;"
40        "shrdq $32, %%r8,%%r15;"
41        "shrdq $32, %%r9, %%r8;"
42        "shrdq $32,%%rax, %%r9;"
43
44        "addq  0(%0),%%r10;"
45        "adcq  8(%0),%%r11;"
46        "adcq 16(%0),%%r12;"
47        "adcq 24(%0),%%r13;"
48        "adcq 32(%0),%%r15;"
49        "adcq 40(%0), %%r8;"
50        "adcq 48(%0), %%r9;"
51        "adcq     $0,%%r14;"
52
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",
78            "%r13", "%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 32 least significant 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 5.3.2. However, the operations in Steps 72, 73 and 74 in the code are redundant according to Theorem 5.1.

5.4.4 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 5.1. Please note that here a 64-bit register `r` is accessed using the notation `%r` instead of the notation `%%r` used while writing code 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 5.1 using the available 15 registers in assembly. By comparing the two implementations of reduction it is easy to see that the number of instructions in our assembly is much smaller than the code of [OLH+17].

```
1  xorq    %rdi,%rdi
2  addq    %r9, %rax
3  adcq    %r10, %rbx
4  adcq    %r11, %rcx
5  adcq    %r12, %rdx
6  adcq    %r13, %rbp
7  adcq    %r14, %rsi
8  adcq    %r15, %r8
9  adcq    $0, %rdi
10
11 movq    %rax, 536(%rsp)
12
13 movq    %r12, %rax
14 andq    mask32h, %rax
15
16 addq    %rax, %rdx
17 adcq    %r13, %rbp
18 adcq    %r14, %rsi
19 adcq    %r15, %r8
20 adcq    $0, %rdi
21
22 movq    %r12,  %rax
23 shrd    $32, %r13, %r12
24 shrd    $32, %r14, %r13
25 shrd    $32, %r15, %r14
26 shrd    $32, %r9,  %r15
27 shrd    $32, %r10, %r9
28 shrd    $32, %r11, %r10
29 shrd    $32, %rax, %r11
30 addq    536(%rsp), %r12
31 adcq    %r13, %rbx
32 adcq    %r14, %rcx
33 adcq    %r15, %rdx
34 adcq    %r9,  %rbp
35 adcq    %r10, %rsi
36 adcq    %r11, %r8
37 adcq    $0, %rdi
38
39 movq    %rdi, %r13
40 shlq    $32, %r13
41
42 xorq    %r14, %r14
43 addq    %rdi, %r12
44 adcq    $0, %rbx
45 adcq    $0, %rcx
46 adcq    %r13, %rdx
47 adcq    $0, %rbp
48 adcq    $0, %rsi
49 adcq    $0, %r8
50 adcq    $0, %r14
51
52 movq    %r14, %r13
53 shlq    $32, %r13
54
55 addq    %r14, %r12
56 adcq    $0, %rbx
57 adcq    $0, %rcx
58 adcq    %r13, %rdx
```

5.5 Arithmetic in $\mathbb{F}_p$ using Unsaturated Limb Representation

In this case the algorithms for the non-linear operations, i.e, field multiplication and field squaring are of major interest and we mainly focus on them.

5.5.1 Multiplication and Squaring in $\mathbb{F}_p$

A field element is represented as the polynomial in base $\theta = 2^{56}$ as the 8-limb polynomial $f(\theta) = \sum_{i=0}^{7} f_i\theta^i$, where $0 \leq f_i < \theta, i = 0, 1, \ldots, 7$. The product of two elements $f(\theta)$ and $g(\theta)$ is given by the polynomial $h(\theta) = \sum_{i=0}^{7} h_i\theta^i$, where

$$\begin{aligned}
h_0 &= f_0g_0 + f_1g_7 + f_2g_6 + f_3g_5 + f_4g_4 + f_5g_3 + f_6g_2 + f_7g_1 + f_5g_7 + f_6g_6 + f_7g_5,\\
h_1 &= f_0g_1 + f_1g_0 + f_2g_7 + f_3g_8 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2 + f_6g_7 + f_7g_6,\\
h_2 &= f_0g_2 + f_1g_1 + f_2g_0 + f_3g_7 + f_4g_6 + f_5g_5 + f_6g_4 + f_7g_3 + f_7g_7,\\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + f_4g_7 + f_5g_6 + f_6g_5 + f_7g_4,\\
h_4 &= f_0g_4 + f_1g_3 + f_2g_2 + f_3g_1 + f_4g_0 + f_1g_7 + f_2g_6 + f_3g_5 + f_4g_4 + f_5g_3 + f_6g_2 +\\
&\quad f_7g_1 + 2f_5g_7 + 2f_6g_6 + 2f_7g_5
\end{aligned}$$

$$\begin{aligned}
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0 + f_2g_7 + f_3g_6 + f_4g_5 + f_5g_4 + f_6g_3 +\\
&\quad f_7g_2 + 2f_6g_7 + 2f_7g_6\\
h_6 &= f_0g_6 + f_1g_5 + f_2g_4 + f_3g_3 + f_4g_2 + f_5g_1 + f_6g_0 + f_3g_7 + f_4g_6 + f_5g_5 + f_6g_4 +\\
&\quad f_7g_3 + 2f_7g_7\\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 + f_4g_7 + f_5g_6 + f_6g_5 +\\
&\quad f_7g_4,
\end{aligned}$$

and $0 \leq h_i < 2^{128}$. The equations are found by multiplying the polynomials $f(\theta)$ and $g(\theta)$ and applying an immediate reduction using the congruence $\theta^8 \equiv \theta^4 + 1 \bmod p$. If we set $g_i \leftarrow f_i$ for $i = 0, 1, \ldots, 7$ in the above equations then we get the corresponding equations for $h(\theta) = f^2(\theta)$.

The polynomial $h(\theta)$ is reduced using function reduce448_8L given in Algorithm 5.3. For the reduction algorithm, the input is considered to be a polynomial $h^{(0)}(\theta)$, and the output is $h^{(2)}(\theta)$ or $h^{(3)}(\theta)$, such that

$$h^{(0)}(\theta) \equiv h^{(1)}(\theta) \equiv h^{(2)}(\theta) \equiv h^{(3)}(\theta) \bmod p.$$

The following result states the correctness of reducep448_8L. The proof of correctness shows that $h^{(i)}(\theta) \equiv h^{(i-1)}(\theta) \bmod p$ and also provide precise bounds on the coefficients of $h^{(i)}(\theta)$.

Theorem 5.3. *Let the elements in $\mathbb{F}_p$ have 8-limb representation in base $\theta = 2^{56}$. Suppose the input $h^{(0)}(\theta) = h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_7^{(0)}\theta^7$ to* reducep448_8L *is such that $0 \leq h_i^{(0)} < 2^{128}$ for $i = 0, 1, \ldots, 7$.*

1. *For partial reduction, the output of* reducep448_8L *is $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_7^{(2)}\theta^7$, where $0 \leq h_0^{(2)}, h_4^{(2)} < 2^{57}$, $0 \leq h_1^{(2)}, h_2^{(2)}, h_3^{(2)}, h_5^{(2)}, h_6^{(2)}, h_7^{(2)} < 2^{56}$ satisfying $h^{(2)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

2. *For full reduction, the output of* reducep448_8L *is $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_7^{(3)}\theta^7$, where $0 \leq h_0^{(3)}, h_1^{(3)}, \cdots, h_7^{(3)} < 2^{56}$ satisfying $h^{(3)}(\theta) \equiv h^{(0)}(\theta) \bmod p$.*

Algorithm 5.3 Reduction in $\mathbb{F}_p$.

1: **function** reducep448_8L($h^{(0)}(\theta)$)
2: **input:** $h^{(0)}(\theta)$.
3: **output:** $h^{(2)}(\theta)$ or $h^{(3)}(\theta)$.
4: $h_0^{(1)} \leftarrow h_0^{(0)} \bmod 2^{56}$
5: **for** $i \leftarrow 1$ to 7 **do**
6: $h_i^{(1)} \leftarrow h_i^{(0)} \bmod 2^{56} + \lfloor h_{i-1}^{(0)}/2^{56} \rfloor$
7: **end for**
8: $h_0^{(1)} \leftarrow h_0^{(1)} + \lfloor h_7^{(0)}/2^{56} \rfloor$; $h_4^{(1)} \leftarrow h_4^{(1)} + \lfloor h_7^{(0)}/2^{56} \rfloor$
9: $h_0^{(2)} \leftarrow h_0^{(1)} \bmod 2^{56}$
10: **for** $i \leftarrow 1$ to 7 **do**
11: $t \leftarrow h_i^{(1)} + \lfloor h_{i-1}^{(1)}/2^{56} \rfloor$; $h_i^{(2)} \leftarrow t \bmod 2^{56}$
12: **end for**
13: $h_0^{(2)} \leftarrow h_0^{(2)} + \lfloor h_7^{(1)}/2^{56} \rfloor$; $h_4^{(2)} \leftarrow h_4^{(2)} + \lfloor h_7^{(1)}/2^{56} \rfloor$
14: PARTIAL REDUCTION: **return** $h^{(2)}(\theta) = h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_7^{(2)}\theta^7$
15: $h_0^{(3)} \leftarrow h_0^{(2)} \bmod 2^{56}$
16: **for** $i \leftarrow 1$ to 7 **do**
17: $t \leftarrow h_i^{(2)} + \lfloor h_{i-1}^{(2)}/2^{56} \rfloor$; $h_i^{(3)} \leftarrow t \bmod 2^{56}$
18: **end for**
19: $t \leftarrow h_0^{(3)} + \lfloor h_7^{(2)}/2^{56} \rfloor$; $h_0^{(3)} \leftarrow t \bmod 2^{56}$; $h_1^{(3)} \leftarrow h_1^{(3)} + \lfloor t/2^{56} \rfloor$
20: $t \leftarrow h_4^{(3)} + \lfloor h_7^{(2)}/2^{56} \rfloor$; $h_4^{(3)} \leftarrow t \bmod 2^{56}$; $h_5^{(3)} \leftarrow h_5^{(3)} + \lfloor t/2^{56} \rfloor$
21: FULL REDUCTION: **return** $h^{(3)}(\theta) = h_0^{(3)} + h_1^{(3)}\theta + \cdots + h_7^{(3)}\theta^7$
22: **end function**.

Proof. Define

$$h_j^{(0)} = h_{j,0}^{(0)} + h_{j,1}^{(0)} 2^{56} \text{ where } h_{j,0}^{(0)} = h_j^{(0)} \bmod 2^{56},\ h_{j,1}^{(0)} = \lfloor h_j^{(0)}/2^{56} \rfloor,\ j = 0, 1, \ldots, 7. \quad (5.20)$$

As $\eta = 56$, we have the bounds $0 \leq h_{j,0}^{(0)} < 2^{56}$ and $0 \leq h_{j,1}^{(0)} < 2^{128-56} = 2^{72}$ for $j = 0, 1 \ldots, 7$. We can write $h^{(0)}(\theta)$ as

$$\begin{aligned}
h^{(0)}(\theta) &= h_0^{(0)} + h_1^{(0)}\theta + \cdots + h_7^{(0)}\theta^7 \\
&= (h_{0,0}^{(0)} + h_{0,1}^{(0)}\theta) + (h_{1,0}^{(0)} + h_{1,1}^{(0)}\theta)\theta + \cdots + (h_{7,0}^{(0)} + h_{7,1}^{(0)}\theta)\theta^7 \\
&= h_{0,0}^{(0)} + (h_{0,1}^{(0)} + h_{1,0}^{(0)})\theta + \cdots + (h_{6,1}^{(0)} + h_{7,0}^{(0)})\theta^7 + h_{7,1}^{(0)}\theta^8 \\
&\equiv h_{0,0}^{(0)} + (h_{0,1}^{(0)} + h_{1,0}^{(0)})\theta + \cdots + (h_{6,1}^{(0)} + h_{7,0}^{(0)})\theta^7 + h_{7,1}^{(0)}(\theta^4 + 1) \bmod p \quad (5.21) \\
&\qquad\qquad [\text{using } \theta^8 \equiv \theta^4 + 1 \bmod p\ ] \\
&= (h_{0,0}^{(0)} + h_{7,1}^{(0)}) + (h_{0,1}^{(0)} + h_{1,0}^{(0)})\theta + \cdots + (h_{3,1}^{(0)} + h_{4,0}^{(0)} + h_{7,1}^{(0)})\theta^4 + \cdots + \\
&\quad (h_{6,1}^{(0)} + h_{7,0}^{(0)})\theta^7 \quad (5.22)
\end{aligned}$$

Steps 4-8 of reducep448_8L performs the additions in (5.21) and we have

$$h^{(0)}(\theta) \equiv h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_7^{(1)}\theta^7 \bmod p = h^{(1)}(\theta), \quad (5.23)$$

where $0 \leq h_0^{(1)}, h_1^{(1)}, h_2^{(1)}, h_3^{(1)}, h_5^{(1)}, h_6^{(1)}, h_7^{(1)} < 2^{73}$ and $0 \leq h_4^{(1)} < 2^{74}$. Define

$$h_j^{(1)} \quad = \quad h_{j,0}^{(1)} + h_{j,1}^{(1)} 2^{56} \text{ where } h_{j,0}^{(1)} = h_j^{(1)} \bmod 2^{56},\ h_{j,1}^{(1)} = \lfloor h_j^{(1)} / 2^{56} \rfloor,\ j = 0, 1, \ldots, 7. \tag{5.24}$$

from which we have the bounds $0 \leq h_{j,1}^{(1)} < 2^{73-56} = 2^{17}$ for $j = 0, 1, 2, 3, 5, 6, 7$ and $0 \leq h_{4,1}^{(1)} < 2^{74-56} = 2^{18}$. Using these bounds in Steps 9-13 which converts the polynomial $h^{(1)}(\theta)$ to $h^{(2)}(\theta)$, we have

$$h^{(1)}(\theta) \quad \equiv \quad h_0^{(2)} + h_1^{(2)}\theta + \cdots + h_7^{(2)}\theta^7 \bmod p = h^{(2)}(\theta), \tag{5.25}$$

where $0 \leq h_1^{(2)}, h_2^{(2)}, h_3^{(2)}, h_5^{(2)}, h_6^{(2)}, h_7^{(2)} < 2^{56}$ and $0 \leq h_0^{(2)}, h_4^{(2)} < 2^{57}$. Combining (5.23) and (5.25) we have $h^2(\theta) \equiv h^0(\theta) \bmod p$ and this completes the proof for partial reduction.

Now, if there is a significant one-bit carry from the first and/or fourth limb of $h^2(\theta)$, it gets absorbed in the second and/or fifth limb of $h^3(\theta)$ through Steps 15-20, otherwise the limbs of $h^2(\theta)$ and $h^3(\theta)$ are same. In both the cases the limbs of $h^3(\theta)$ satisfy $0 \leq h_i^{(3)} < 2^{56},\ i = 0, 1, \ldots, 7$. Also, we have $h^3(\theta) \equiv h^0(\theta) \bmod p$ and this completes the proof for full reduction. □

5.5.2 Multiplication With a Small Constant in $\mathbb{F}_p$

The small field constant is $c = 39082$, which is of 16 bits. The obvious algorithm to multiply a field element by c and then use the reduction algorithm `reducep448_8L`. But, the 64-bit assembly implementation of `reducep448_8L` uses the high-latency `shld` instructions, which can be avoided in the present context using a different method that performs better. The method is formalized in Algorithm 5.4, which performs limb-wise multiplication and reduction in an interleaved way. We have developed the algorithm using the idea applied to multiply with the small constant 121666 in the unsaturated limb implementation of scalar multiplication of Curve25519 [BDL+12].

In the assembly implementation of `mulcp448_8L` we don't have to do anything extra for the operations $\lfloor \cdot / 2^{64} \rfloor$; the result is found in the `%rdx` register after the multiplication operation which is performed using the `mul` instruction. The operations $\lfloor \cdot / 2^8 \rfloor$ can be efficiently implemented using the `shr` instruction. The limbs of the output polynomial $h^{(1)}(\theta)$ have one bit extra and this doesn't produce any overflow situation for any integer multiplication which is done further. A formal proof of the algorithm is straight-forward and we skip this.

5.5.3 Addition and Subtraction in $\mathbb{F}_p$

Within the Montgomery ladder two elements $f(\theta)$ and $g(\theta)$ are limb-wise added to perform the addition. To subtract an element $g(\theta)$ from $f(\theta)$ we compute $(f(\theta) + 2p(\theta) - g(\theta))$. To do this, we use the pre-computed 8-limb value of $2p(\theta)$. The added or subtracted 8-limb value is kept unreduced within the ladder computation as it does not produce any overflow situation for the integer multiplication algorithm given in Section 5.5.1.

Algorithm 5.4 Multiplication with small constant in $\mathbb{F}_p$.

function mulcp448_8L($h^{(0)}(\theta)$)
input: $h^{(0)}(\theta)$.
output: $h^{(1)}(\theta)$.
    $c' \leftarrow c \cdot 2^8$
    $t \leftarrow h_0^{(0)} \cdot c'$; $r \leftarrow t \bmod 2^{64}$
    $h_0^{(1)} \leftarrow \lfloor r/2^8 \rfloor$; $h_1^{(1)} \leftarrow \lfloor t/2^{64} \rfloor$
    for $i \leftarrow 1$ to 6 **do**
        $t \leftarrow h_i^{(0)} \cdot c'$; $r \leftarrow t \bmod 2^{64}$
        $h_i^{(1)} \leftarrow h_i^{(1)} + \lfloor r/2^8 \rfloor$; $h_{i+1}^{(1)} \leftarrow \lfloor t/2^{64} \rfloor$
    end for
    $t \leftarrow h_7^{(0)} \cdot c'$; $r \leftarrow t \bmod 2^{64}$
    $h_7^{(1)} \leftarrow h_7^{(1)} + \lfloor r/2^8 \rfloor$; $h_0^{(1)} \leftarrow h_0^{(1)} + \lfloor t/2^{64} \rfloor$; $h_4^{(1)} \leftarrow h_4^{(1)} + \lfloor t/2^{64} \rfloor$
    PARTIAL REDUCTION: **return** $h^{(1)}(\theta) = h_0^{(1)} + h_1^{(1)}\theta + \cdots + h_7^{(1)}\theta^7$
end function.

5.6 Implementations and Timings

We present two 7-limb (`maax`-type and `mxaa`-type) and one 8-limb (`maa`-type) 64-bit implementations for shared secret computation phase of the X448 function. We first discuss about the performance of our 7-limb implementations.

5.6.1 Performance of 7-limb Implementations

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 the function reducep448_7L while for the integer multiplication/squaring we have used Algorithms 4.2 and 4.3. Implementations of field addition, subtraction and multiplication by the curve constant are as described in Sections 5.3.2 and 5.3.3. 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 3.6 and the key generation was computed using the right-to-left Montgomery ladder given in Algorithm 5 of [OLH+17]. 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.

Timings in the form of CPU-cycles are provided in Table 5.1. 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

Operation	Haswell	Skylake	Reference	Implementation Type
Shared secret	732013	587389	[OLH+17]	`mxaa`, inline assembly
	-	530984	[OLH+17]	`maax`, inline assembly
	719217	461379	this work	`mxaa`, assembly
	-	434831	this work	`maax`, assembly
Key generation	423703	356113	[OLH+17]	`mxaa`, inline assembly
	-	315890	[OLH+17]	`maax`, inline assembly
	420453	278743	this work	`mxaa`, assembly
	-	261683	this work	`maax`, assembly

Table 5.1: CPU-cycle counts for shared secret computation and key generation on Curve448 using 7-limb representation. Computation of key generation has been done using Algorithm 5 of [OLH+17].

corresponding to the work [OLH+17][4]. This implementation produces code targeting the Haswell and Skylake architectures. 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 5.1.

- 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 these are improvements, they are not substantial as it has been achieved on Skylake.

While the reduction algorithms that we have described avoid certain redundant operations performed by the code corresponding to [OLH+17], 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.

The key generation has also been computed using the left-to-right Montgomery ladder of Algorithm 3.6. However, the timings of these implementations reported in Table 5.2 are slower in comparison to the timings reported in Table 5.1.

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

Operation	Haswell	Skylake	Reference	Implementation Type
Key generation	653035	427058	this work	`mxaa`, assembly
	-	396583	this work	`maax`, assembly

Table 5.2: CPU-cycle counts for shared secret computation and key generation on Curve448 using 7-limb representation. Computation of key generation has been done using Algorithm 3.6.

5.6.2 Performance of 8-limb Implementations

The performances of the 8-limb `maa`-type implementations are slower than the `maax`-type and `mxaa`-type implementations. Hence, we report the timings of the `maa`-type implementations separately in Table 5.3. Both the computations of shared secret and key generation have been done using Algorithm 3.6. We did not find any 8-limb implementation of the x448 function in literature.

Operation	Haswell	Skylake	Implementation	Implementation Type
Shared secret	721044	558740	this work	`maa`, assembly
Key generation	644288	504808	this work	`maa`, assembly

Table 5.3: CPU-cycle counts for shared secret computation and key generation on Curve448 using 8-limb representation. Computation for both has been done using Algorithm 3.6.

5.7 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 $2^{480} - 2^{240} - 1$.

CHAPTER 6

Efficient Field Arithmetic Using 4-way Vector Instructions

6.1 Introduction

Computation of cryptographic primitives over elliptic curves require arithmetic in the underlying field $\mathbb{F}_p$. The primary operation of interest is the field multiplication whose cost mainly depends on the number of involved word multiplications needed at the basic level of the operation.

Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008 and first supported by Intel with the Sandy Bridge processor shipping in Q1 2011 and later on by AMD with the Bulldozer processor shipping in Q3 2011. AVX provides new features, new instructions and a new coding scheme. AVX2 expands most integer commands to 256 bits and introduces fused multiply-accumulate (FMA) operations. They were first supported by Intel with the Haswell processor, which shipped in 2013. For details one may follow [AVX]. The AVX2 instruction set architecture enables 4-way vectorized computation with 64-bit words. While computing the field multiplication using 4-way vector instructions available in the modern Intel architectures, the 32-bit multiplier `vpmuludq` is used to multiply two words of at most 32-bits. But, in the vectorized setup there are no facilities to keep track of the generated carries. As a reason an unsaturated limb representation of elements in $\mathbb{F}_p$ is used to accumulate the summands involved in an integer multiplication. We will be using `AVX2` instructions to vectorize multiple independent $\mathbb{F}_p$ operations instead of implementing a single $\mathbb{F}_p$ operation.

We introduce a notation to denote the primes. For example, the prime $2^{251} - 9$ will be denoted as p251-9 and similar notation applies for the other primes of the for $2^m - \delta$. The prime $2^{448} - 2^{224} - 1$ will be denoted as p448-224-1. The different primes with which vectorized arithmetic have been done in this thesis are p251-9 ,p255-19, p444-17, p448-224-1, p506-45, p510-75 and p521-1.

6.2 Representation of Field Elements and Primes

6.2.1 Representation of Field Elements

Let $m = \lceil \log_2 p \rceil$. Elements of $\mathbb{F}_p$ can be represented as m-bit strings using κ words called limbs, where $m = \eta(\kappa - 1) + \nu$ such that $1 \leq \nu \leq \eta < 32$. So, the first $(\kappa - 1)$ limbs are η bits long, while the size of the last limb is ν which lies between 1 and η.

The representations of field elements over the various fields are summarized in Table 6.1. For the elements of $\mathbb{F}_{2^{255}-19}$ we consider two different representations, one using 9 words and the other using 10 words. The representation of a field element $A \in \mathbb{F}_{2^{255}-19}$ using 10 words is the standard representation introduced by Bernstein [Ber06b] given by $A = \sum_{i=0}^{9} a_i 2^{\lceil 25.5i \rceil}$ where $0 \leq a_0 \leq 2^{26} - 19$, $0 \leq a_2, a_4, a_6, a_8 < 2^{26}$ and $0 \leq a_1, a_3, a_5, a_7, a_9 < 2^{25}$.

6.2.2 Representation of the Primes

A *pseudo-Mersenne* prime p having κ limbs (except p255-19 with $\kappa = 10$) is represented as a polynomial $\mathfrak{P}$ in base $\theta = 2^{\eta}$ given by

$$\mathfrak{P} = \sum_{i=0}^{\kappa-1} \mathfrak{p}_i \theta^i, \text{ where } \mathfrak{p}_0 = 2^{\eta} - \delta,\ \mathfrak{p}_1, \mathfrak{p}_2, \ldots, \mathfrak{p}_{\kappa-2} = 2^{\eta} - 1,\ \mathfrak{p}_{\kappa-1} = 2^{\nu} - 1.$$

The prime p255-19 having $\kappa = 10$ limbs is represented as

$$\mathfrak{P} = \sum_{i=0}^{9} p_i 2^{\lceil 25.5i \rceil}, \text{ where } \mathfrak{p}_0 = 2^{26} - 19;\ \mathfrak{p}_2, \mathfrak{p}_4, \mathfrak{p}_6, \mathfrak{p}_8 = 2^{26} - 1;\ \mathfrak{p}_1, \mathfrak{p}_3, \mathfrak{p}_5, \mathfrak{p}_7, \mathfrak{p}_9 = 2^{25} - 1.$$

The prime p448-224-1 is represented as

$$\mathfrak{P} = \sum_{i=0}^{9} \mathfrak{p}_i \theta^i, \text{ where } \mathfrak{p}_0, \mathfrak{p}_1, \ldots, \mathfrak{p}_7, \mathfrak{p}_9, \ldots, \mathfrak{p}_{15} = 2^{28} - 1;\ \mathfrak{p}_8 = 2^{28} - 2.$$

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

Table 6.1: Representations of field elements for vectorized arithmetic.

6.3 Multiplication and Squaring in $\mathbb{F}_p$

6.3.1 Applying Schoolbook

The 9-limb multiplication for p251-9 is done using the schoolbook method. The equations for integer multiplication with immediate reduction are formed using the equations (4.40). Following this method we directly get a k-limb product polynomial $H = h(\theta)$ in a semi-reduced form whose limb-values are less than 2^{64}. For p255-19, the equations due to 10-limb integer multiplication that produces the limbs of the product polynomial H are according to the equations provided in [BS12, Cho15]. The polynomial H needs to be reduced further which should satisfy certain computation-friendly limb-bounds. The algorithm for this is known as a *reduction chain* and we discuss about this after the following section.

6.3.2 Applying Karatsuba

For p255-19 with $\kappa = 9$ and all other primes at the 128-bit and 256-bit levels we apply the Karatsuba multiplication algorithm for integer multiplication. The 9-limb multiplication of p255-19 is done using the (5+4)-Karatsuba strategy and for all other primes we make use of the $(\kappa/2, \kappa/2)$-Karatsuba strategy with one level of recursion in which the integer multiplication of the subproblems are done following the schoolbook method. Integer squaring is also done using the same Karatsuba strategies except for the 9-limb squaring of p255-19, for which, we apply the schoolbook method directly instead of dividing it. Let $\theta = 2^{\eta}$ and $\phi = \theta^{\kappa/2}$. Then a field element $A(\theta) = A \in \mathbb{F}_p$ can be represented as

$$\begin{aligned} A &= a_0 + a_1\theta + \cdots + a_{\kappa-1}\theta^{\kappa-1} \\ &= (a_0 + a_1\theta + \cdots + a_{\kappa/2-1}\theta^{\kappa/2-1}) + (a_{\kappa/2} + a_{\kappa/2+1}\theta + \cdots + a_{\kappa-1}\theta^{\kappa/2-1})\theta^{\kappa/2} \\ &= U + V\phi, \end{aligned} \tag{6.1}$$

where $U = a_0 + a_1\theta + \cdots + a_{\kappa/2-1}\theta^{\kappa/2-1}$ and $V = a_{\kappa/2} + a_{\kappa/2+1}\theta + \cdots + a_{\kappa-1}\theta^{\kappa/2-1}$. For a uniform mathematical treatment we will consider the field element as a 10-limb polynomial such that the last limb is 0 in case of p255-19 with $\kappa = 9$. Like A, let us consider another field element $B(\theta) = B \in \mathbb{F}_p$ such that $B = W + Z\phi$. Then the product of A and B can be written as

$$\begin{aligned} C &= AB \\ &= (U + V\phi)(W + Z\phi) \\ &= UW + (UZ + VW)\phi + VZ\phi^2, \end{aligned} \tag{6.2}$$

which provides the $(2\kappa - 1)$-limb product polynomial. For reduction, C needs to be transformed to an equivalent κ-limb polynomial which is achieved through two phases. In the first phase the $(2\kappa - 1)$-limb product polynomial C is converted to an equivalent k-limb polynomial $H = h(\theta)$ whose limbs-values are less than 2^{64}. In the second phase, H needs to be further reduced which satisfy certain well-defined computation-friendly limb-bounds. This is done using the *reduction chain* as mentioned for the previous case. We first discuss about the arithmetic to achieve the κ-limb polynomial H whose limbs values are less than 2^{64}. The mathematical treatment to do so for the primes p448-224-1 and p521-1 are addressed next as two specific cases.

Case of the prime $p448$**-224-1.** For this prime $\kappa = 16$, $\theta = 2^{28}$, $\phi = \theta^8$ and we have the identity $\phi^2 \equiv \phi + 1 \mod p$. Applying the identity on (6.2) as the first step of reduction we get

$$\begin{aligned} C &= UW + (UZ + VW)\phi + VZ\phi^2 \\ &\equiv (UW + VZ) + (UZ + VW + VZ)\phi \mod p \\ &= (UW + VZ) + ((U + V)(W + Z) - UW))\phi. \end{aligned} \tag{6.3}$$

We now compute the three products UW, VZ and $(U + V)(W + Z)$ with the schoolbook method using $3 \times 8 \times 8 = 192$ limb-multiplications and combine the results to find the product C. This gives us a saving of 64 limb-multiplications as compared to the schoolbook method when applied to the entire 16-limb polynomials A and B. We can find the similar equation for squaring as

$$C = A^2 \equiv (U^2 + V^2) + ((U + V)^2 - U^2))\phi \mod p. \tag{6.4}$$

The product UW is computed as the polynomial $R = UW = \sum_{j=0}^{14} r_j\theta^j$, where

$$r_j = \sum_{i=0}^{j} a_i b_{j-i}, \text{ for } j = 0, 1, \ldots, 7; \tag{6.5}$$

$$r_{j+7} = \sum_{i=j}^{7} a_i b_{7-i+j}, \text{ for } j = 1, 2, \ldots, 7. \tag{6.6}$$

Similarly, let the products VZ and $(U + V)(W + Z)$ be denoted by $S = \sum_{j=0}^{14} s_j\theta^j$ and $T = \sum_{j=0}^{14} t_j\theta^j$ respectively. Then we can write

$$\begin{aligned} C &\equiv (R + S) + (T - R)\phi \mod p \\ &= E + F\phi, \end{aligned} \tag{6.7}$$

where $E = \sum_{j=0}^{14} e_j\theta^j$ and $F = \sum_{j=0}^{14} f_j\theta^j$, such that $0 \leq e_j, f_j < 2^{64}$, $j = 0, 1, \ldots, 14$.

To perform the first phase of reduction on the product $C = E + F\phi$, we perform some carry-less additions with specific coefficients of the polynomial C to arrive to a certain polynomial on which the second phase of the reduction can be applied. These carry-less additions do not lead to any overflow conditions. We describe the method below.

$$\begin{aligned} C &\equiv E + F\phi \mod p \\ &= \sum_{j=0}^{14} e_j\theta^j + \sum_{j=0}^{14} f_j\theta^{j+8} \\ &= \sum_{j=0}^{7} e_j\theta^j + \sum_{j=8}^{14} (e_j + f_{j-8})\theta^j + \sum_{j=15}^{22} f_{j-8}\theta^j \\ &= \sum_{j=0}^{7} (r_j + s_j)\theta^j + \sum_{j=8}^{14} (r_j + s_j + t_{j-8} - r_{j-8})\theta^j + \sum_{j=15}^{22} (t_{j-8} - r_{j-8})\theta^j \\ &= \sum_{j=0}^{22} g_j\theta^j \text{ (say)}. \end{aligned} \tag{6.8}$$

From (6.8) we can further write

$$\begin{aligned} C &\equiv \sum_{j=0}^{6}(g_j + g_{j+16})\theta^j + g_7\theta^7 + \sum_{j=8}^{14}(g_j + g_{j+8})\theta^j + g_{15}\theta^{15} \mod p \\ &= \sum_{j=0}^{6}(r_j + s_j + t_{j+8} - r_{j+8})\theta^j + (r_7 + s_7)\theta^7 + \\ &\quad \sum_{j=8}^{14}(s_j + t_{j-8} + t_j - r_{j-8})\theta^j + (t_7 - r_7)\theta^{15} \\ &= \sum_{j=0}^{15} h_j\theta^j = h(\theta) = H \text{ (say)}, \end{aligned} \tag{6.9}$$

which gives the desired 16-limb polynomial for the prime $p448\text{-}224\text{-}1$.

Case of the prime $p521\text{-}1$**.** For this prime $\kappa = 18$, $\theta = 2^{29}$, $\phi = \theta^9$ and we have the identity $\phi^2 \equiv 2\phi \mod p$. Applying the identity on (6.2) as the first step of reduction we get

$$\begin{aligned} C &= AB \\ &\equiv (UW + 2VZ) + (UZ + VW)\phi \mod p \\ &= (UW + 2VZ) + ((U + V)(W + Z) - (UW + VZ))\phi. \end{aligned} \tag{6.10}$$

For squaring the equation becomes

$$C = A^2 \equiv (U^2 + 2V^2) + ((U + V)^2 - (U^2 + V^2))\phi \mod p. \tag{6.11}$$

We compute the three products UW, VZ and $(U + V)(W + Z)$ with the schoolbook method using $3 \times 9 \times 9 = 243$ limb-multiplications and combine the results to find the product C. This gives us a saving of 81 limb-multiplications as compared to the schoolbook method when applied to the entire 18-limb polynomials A and B. Similar to the previous case, the product UV is computed as the polynomial $r = r(\theta) = UW = \sum_{j=0}^{16} r_j\theta^j$. Let the products VZ and $(U + V)(W + Z)$ be denoted by $S = \sum_{j=0}^{16} s_j\theta^j$ and $T = \sum_{j=0}^{16} t_j\theta^j$ respectively. Then, we can write

$$\begin{aligned} C &\equiv (R + 2S) + (T - R - S)\phi \mod p \\ &= E + F\phi \mod p \text{ (say)} \\ &= \sum_{j=0}^{16} e_j\theta^j + \sum_{j=0}^{16} f_j\theta^{j+9} \\ &= \sum_{j=0}^{8} e_j\theta^j + \sum_{j=9}^{16}(e_j + f_{j-9})\theta^j + \sum_{j=17}^{25} f_{j-9}\theta^j \\ &= \sum_{j=0}^{8}(r_j + 2s_j)\theta^j + \sum_{j=9}^{16}(r_j + 2s_j + t_{j-9} - r_{j-9} - s_{j-9})\theta^j + \sum_{j=17}^{25}(t_{j-9} - r_{j-9} - s_{j-9})\theta^j \\ &= \sum_{j=0}^{25} g_j\theta^j, \text{ (say)}. \end{aligned} \tag{6.12}$$

From (6.12) we can further write

$$\begin{aligned} C &\equiv \sum_{j=0}^{7}(g_j + 2g_{j+18})\theta^j + \sum_{j=8}^{17} g_j\theta^j \mod p \\ &= \sum_{j=0}^{7}(r_j + 2s_j + 2t_{j+9} - 2r_{j+9} - 2s_{j+9})\theta^j + (r_8 + 2s_8)\theta^8 + \\ &\quad \sum_{j=9}^{16}(r_j + 2s_j + t_{j-9} - r_{j-9} - s_{j-9})\theta^j + (t_8 - r_8 - s_8)\theta^{17} \\ &= \sum_{j=0}^{17} h_j\theta^j = h(\theta) = H \text{ (say)}, \end{aligned} \tag{6.13}$$

which gives the desired 18-limb polynomial for the prime $p521$-1.

Cases of the remaining primes. For the remaining cases we will have the congruency relationship $\phi^2 \equiv 2^{\eta-\nu}\delta \mod p$. Since for these cases the difference $(\eta - \nu)$ and the value of δ is relatively large, it is not possible to apply a reduction step like (6.3) and (6.10) without loosing information. To manage the situation, first an expansion of the $(2\kappa - 1)$-limb product polynomial C is done which converts C to an equivalent 2κ-limb polynomial D such that the values of the last κ limbs of D are small enough to afford a limb-multiplication with the constant $\mathfrak{c}_\mathfrak{p} = 2^{\eta-\nu}\delta$ and no loss of information occurs. After this, the last κ-limbs of D are added consecutively to the first κ limbs D, which gives the reduced polynomial $H = h(\theta)$ obtained after the first phase of the reduction. The method is actually the multe algorithm due to [KS20] which is re-versioned in Algorithm 6.1. In the algorithm, it has to be assured that we don't have any overflows while performing the additions in Steps 8, 9, 12 and 13.

The multiplication algorithm which multiplies two field elements A and B and produces the product H will be called as $\mathsf{mul}(A, B)$. The squaring algorithm will be termed as $\mathsf{sqr}(A)$.

6.3.3 Reduction Chain

The limbs of the polynomial $H(\theta)$ found after integer multiplication/squaring needs to be reduced further by a method which is usually known as the *reduction chain*. The reduction chain can be simple or interleaved, applications of which comes out to be efficient for different primes.

Simple reduction chain. The simple reduction chain for the pseudo-Mersenne primes is essentially the method for partial reduction provided in Algorithm 4.8. This is more popularly denoted by

$$h_0 \to h_1 \to \cdots \to h_{\kappa-2} \to h_{\kappa-1} \to h_0 \to h_1.$$

A single carry step is denoted by $h_{j \bmod \kappa} \to h_{(j+1) \bmod \kappa}$ which perform the following operations.

Algorithm 6.1 Expansion of product polynomial.

1: **function** EXPAND(C, $\mathfrak{c}_\mathfrak{p}$)
2: **input:** C is a $(2\kappa-1)$-limb product polynomial having coefficients from $\mathbb{F}_p$ and $\mathfrak{c}_\mathfrak{p} = 2^{\eta-\nu}\delta$.
3: **output:** $H = h(\theta)$, a κ-limb equivalent polynomial having coefficients from $\mathbb{F}_p$.
4: **for** $i \leftarrow 0$ to $\kappa-1$ **do**
5: $d_i \leftarrow c_i$
6: **end for**
7: **for** $i \leftarrow \kappa$ to $2\kappa-3$ **do**
8: $c_{i+1} \leftarrow c_{i+1} + \lceil c_i/2^\eta \rceil$; $\mathbf{t} \leftarrow c_i \bmod 2^\eta$
9: $d_i \leftarrow \mathfrak{c}_\mathfrak{p} \cdot \mathbf{t}$; $h_{\kappa-i} \leftarrow d_{i-\kappa} + d_i$
10: **end for**
11: $\mathbf{r} \leftarrow \lceil c_{2\kappa-2}/2^\eta \rceil$; $\mathbf{s} \leftarrow c_{2\kappa-2} \bmod 2^\eta$
12: $d_{2\kappa-2} \leftarrow \mathfrak{c}_\mathfrak{p} \cdot \mathbf{r}$; $h_{\kappa-2} \leftarrow d_{\kappa-2} + d_{2\kappa-2}$
13: $d_{2\kappa-1} \leftarrow \mathfrak{c}_\mathfrak{p} \cdot \mathbf{s}$; $h_{\kappa-1} \leftarrow d_{\kappa-1} + d_{2\kappa-1}$
14: **return** H
15: **end function**.

- Logically right shift the 64-bit word in $h_{j \bmod \kappa}$ by η bits. For $j = \kappa-1$ the amount of shift is ν bits. Let this amount be $\mathfrak{c}$.
- Add $\mathfrak{c}$ to $h_{(j+1) \bmod \kappa}$.
- Mask out the most significant $(64-\eta)$ bits of $h_{j \bmod \kappa}$. For $j = \kappa-1$ the masking amount is $(64-\nu)$ bits.

The modified polynomial H obtained after the chain is applied, satisfies the limb-bounds $0 \leq h_0, h_2, h_3 \ldots, h_{\kappa-2} < 2^\eta$, $0 \leq h_1 < 2^{\eta+1}$ and $0 \leq h_{\kappa-1} < 2^\nu$. Note that the second limb is relaxed with an extra bit and so the reduction is partial. This doesn't usually create any overflow problems while doing the computations. A full reduction can be done on the polynomial H by applying the reduction chain one more time.

For the prime p448-224-1 the simple reduction chain is a little different and is denoted by

$$h_0 \rightarrow h_1 \rightarrow \cdots \rightarrow h_{15} \rightarrow (h_0, h_8) \rightarrow (h_1, h_9),$$

which performs a partial reduction on the coefficients of H, by keeping one bit extra in the limbs h_1 and h_9 of the reduced polynomial. Here the notation $(h_0, h_8) \rightarrow (h_1, h_9)$ means performing the reductions $h_0 \rightarrow h_1$ and $h_1 \rightarrow h_9$ sequentially.

Interleaved reduction chain. The interleaved reduction chain for the pseudo-Mersenne primes is the working of two reduction sub-chains which does not have any dependencies and is denoted by

$$h_0 \rightarrow h_1 \rightarrow \cdots \rightarrow h_{\kappa/2-1} \rightarrow h_{\kappa/2} \rightarrow h_{\kappa/2+1},$$
$$h_{\kappa/2} \rightarrow h_{\kappa/2+1} \rightarrow \cdots \rightarrow h_{\kappa-1} \rightarrow h_0 \rightarrow h_1.$$

In this case the limbs h_1 and $h_{\kappa/2+1}$ have one bit extra. For p251-9 and p255-19 with $\kappa = 9$, the interleaved chains are

$$h_0 \to h_1 \to \cdots \to h_3 \to h_4 \to h_5,$$
$$h_4 \to h_5 \to \cdots \to h_8 \to h_0 \to h_1.$$

For p448-224-1 the interleaved chains are

$$h_0 \to h_1 \to \cdots \to h_7 \to h_8 \to h_9,$$
$$h_8 \to h_9 \to \cdots \to h_{15} \to (h_0, h_8) \to (h_1, h_9).$$

The reduction algorithm which reduces H will be called as $\mathsf{reduce}_1(H)$ in general.

Application of simple and interleaved chains. The simple chains have been applied to the primes at 128-bit and 224-bit security level and found to be slightly advantageous. For the primes at 256-bit security level we apply the simple reduction chain as it performs better than the interleaved chains.

6.4 Multiplication by a Small Constant in $\mathbb{F}_p$

Let $A \in \mathbb{F}_p$ have a κ-limb representation $\langle a_0, a_1, \ldots, a_{\kappa-1}\rangle$. Let $\mathfrak{c}$ be a small element in $\mathbb{F}_p$, which can be represented using a single limb. Then multiplication of A by $\mathfrak{c}$ provides the κ-limb polynomial represented by the tuple $\langle h_0, h_1, \ldots, h_{\kappa-1}\rangle = \langle a_0 \cdot \mathfrak{c}, a_1 \cdot \mathfrak{c}, \ldots, a_{\kappa-1} \cdot \mathfrak{c}\rangle$. This needs to be reduced. Here also we apply the reductions chains discussed in the previous section. However, the applied chains are short by length one in which the last reduction step is avoided. This can be done because the values of the limbs obtained by the products $a_i \cdot \mathfrak{c}$ are much smaller than 2^{64} as the constant $\mathfrak{c}$ is small. The reduction algorithm will be called reduce_2 and the algorithm to multiply with a small constant will be termed as mulc.

6.5 Dense Packing of Field Elements

Let $A = \sum_{i=0}^{\kappa-1} a_i\theta^i$. Consider that every limb a_i is less than 2^{32} and is stored in a 64-bit word. Then it is possible to pack $a_{\lfloor\kappa/2\rfloor}$with a_0, $a_{\lfloor\kappa/2\rfloor+1}$with $a_1, \ldots, a_{2\lfloor\kappa/2\rfloor-1}$with $a_{\lfloor\kappa/2\rfloor-1}$, so that every pair can be represented using a 64-bit word without losing any information. If κ is odd then $a_{\kappa-1}$ can be left alone. We denote this operation as dense packing of limbs. In general limb v is densely packed with limb u to produce the packed limb $\underline{u}$ using a `left-shift` and an `or` operation through $\underline{u} \leftarrow u \mid (v \ll 32)$. The 32-bit values can be extracted by splitting a 64-bit limb $\underline{u}$ through the operations $v \leftarrow \underline{u} \gg 32$ and $u \leftarrow \underline{u}$ `and` 1^{32}. Using dense packing of limbs we can think that a κ-limb quantity is represented using $\lceil\kappa/2\rceil$ limbs. We define the dense packing operation as $\mathrm{N2D}(A)$ which returns $\underline{A} \leftarrow \sum_{i=0}^{\lceil\kappa/2\rceil-1} \underline{a_i}\theta^i$, where $\underline{a_{\lceil\kappa/2\rceil-1}} = a_{\kappa-1}$ if κ is odd. To convert back a densely packed element to a normally packed element we use the operation $\mathrm{D2N}(\underline{A})$, which returns $A = \sum_{i=0}^{\kappa-1} a_i\theta^i$.

6.6 Linear Operations in $\mathbb{F}_p$

Let $A = \sum_{i=0}^{\kappa-1} a_i\theta^i$, $B = \sum_{i=0}^{\kappa-1} b_i\theta^i$ be two elements in $\mathbb{F}_p$. Using the operation N2D on A and B we obtain the densely packed elements $\underline{A} \leftarrow \sum_{i=0}^{\lceil \kappa/2 \rceil -1} \underline{a_i}\theta^i$ and $\underline{B} \leftarrow \sum_{i=0}^{\lceil \kappa/2 \rceil -1} \underline{b_i}\theta^i$ respectively. The linear operations can be applied over both normal and densely packed field elements.

6.6.1 Addition

The addition $c_i \leftarrow a_i + b_i$ adds the two limbs a_i and b_i and stores the result in c_i for $i = 0, 1, \ldots, \kappa - 1$. This addition is simply over normally packed field elements.

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

6.6.2 Negation

Here we wish to compute $-A \bmod p$. Let n be the least integer such that all the coefficients of $(2^n\mathfrak{P} - A)$ are non-negative. The negation of the element A is then defined by $\mathsf{negate}(A) = 2^n\mathfrak{P} - A = C$ in unreduced form, while reducing C modulo p gives us the desired value in $\mathbb{F}_p$.

Let $C = \sum_{i=0}^{\kappa-1} c_i\theta^i$ so that $c_i = 2^n\mathfrak{p}_i - a_i \geq 0 \ \forall i$. The c_i's are computed using 2's complement subtraction. The result of a subtraction can be negative. By ensuring that the c_i's are non-negative, this situation is avoided. Let α be a quantity such that $\alpha = 32$ or $\alpha = 64$. Considering all values to be α-bit quantities, the computation of c_i is done as

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

The operation $(2^\alpha - 1) - a_i$ is equivalent to taking the bitwise complement of a_i, which is equivalent to $1^\alpha \oplus a_i$. When $\alpha = 64$, the operation in applied over normally packed field elements. When $\alpha = 32$, the operation can be done over the densely packed element $\underline{A} = \sum_{i=0}^{\lceil \kappa/2 \rceil -1} \underline{a_i}\theta^i$ in parallel similar to addition. Usually, it is sufficient to consider $n = 1$ for the computations. But, while applying over normally packed field element obtained after an application of unreduced-mulc, the value of n has to be determined accordingly depending on the bit-sizes of the input-limbs.

6.6.3 Subtraction

Subtraction is done by first negating the subtrahend B and then adding to the minuend A. This operation can also be done over $\underline{A}$ and $\underline{B}$ simultaneously similar to addition.

6.6.4 Reduction after Linear Operations

After the addition/subtraction operation the bit-sizes of the output limbs are at most two more than the bit-sizes of the input limbs which can be further reduced, if required. While computing with normally packed limbs reduce_2 is used.

But, while computing with densely packed elements we take the benefit of reducing the elements in parallel through the reduction chain

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

Here, the notation $(c_i, c_j) \rightarrow (c_k, c_\ell)$ means performing the reductions $c_i \rightarrow c_k$ and $c_j \rightarrow c_\ell$ simultaneously. Some carry chain steps are performed sequentially if required. For example p255-19, the reductions $c_3 \rightarrow c_4$, $c_7 \rightarrow c_8$, $c_8 \rightarrow c_0$ when $\kappa = 9$ and the reductions $c_4 \rightarrow c_5$, $c_9 \rightarrow c_0$ when $\kappa = 10$ can be done sequentially if required. Similarly, for p448-224-1 the reductions $c_7 \rightarrow c_8$, $c_{15} \rightarrow (c_0, c_8)$ can also be done sequentially, if required. We name this reduction operation applied over densely packed elements as reduce_3.

6.6.5 Hadamard Transformations

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

$$\begin{aligned} C &= \mathsf{reduce}_3(A + B), \text{ and} \\ D &= \mathsf{reduce}_3(A + \mathsf{negate}(B)). \end{aligned}$$

The transformation can also be applied to densely packed elements which is denoted as $\mathcal{H}(\underline{A}, \underline{B})$ which the pair $\langle \underline{C}, \underline{D}\rangle$ where

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

The Hadamard transform $\mathcal{H}_1(\underline{A}, \underline{B})$ outputs the pair $\langle \underline{D}, \underline{C}\rangle$, where $\underline{C}$, $\underline{D}$ are defined as above. The transform $\mathcal{H}_2(\underline{A}, \underline{B})$ outputs the pair $\langle \underline{C}, \underline{D}\rangle$ where

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

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

6.7 Vector Operations

SIMD instructions in modern processors allow parallelism where the same instruction can be applied to multiple data. To take advantage of SIMD instructions it is convenient to organize the data as vectors. The Intel instructions that we target apply to 256-bit registers which are considered to be 4 64-bit words (or, as 8 32-bit words). So, we consider vectors of length 4.

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

6.7.1 Vector Representation of Field Elements

Define $\mathbf{A} = \langle A_0, A_1, A_2, A_3 \rangle$ where $A_k = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i \in \mathbb{F}_p$. Hence, $\mathbf{A}$ is a 4-element vector. Each $a_{k,i}$ is stored in a 64-bit word, and conceptually one may think of $\mathbf{A}$ to be given by a $\kappa \times 4$ matrix of 64-bit words. If we consider $\underline{A_k}$, i.e., densely packed form of A_k, then we have $\underline{\mathbf{A}} = \langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle$ where $\underline{A_k} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{a_{k,i}}\theta^i$. Then we can conceptually think of $\underline{\mathbf{A}}$ as a $\lceil \kappa/2 \rceil \times 8$ matrix of 32-bit words. This visualization helps to 2-way parallelize the vector Hadamard transformations and other linear operations within the ladder. We will observe this explicitly in the final algorithm.

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

6.7.2 Dense Packing of Vector Elements

Let $\langle A_0, A_1, A_2, A_3 \rangle = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$, where $A_k = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i$. The vectorized normal to dense packing operation PACK-N2D($\langle A_0, A_1, A_2, A_3 \rangle$) returns the 4-tuple $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}_i}\theta^i$, where $\underline{A_k} = \text{N2D}(A_k)$, such that $\underline{A_k} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{a_{k,i}}\theta^i$.

Let $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}}\theta^i$, where $\underline{A_k} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{a_{k,i}}\theta^i$. The vectorized dense to normal operation PACK-D2N($\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle$) returns the 4-tuple $\langle A_0, A_1, A_2, A_3 \rangle = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$, where $A_k = \text{D2N}(\underline{A_k})$, such that $A_k = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i$. A similar packing strategy called squeeze/unsqueeze has been used earlier in [BCLS14, HEY20]. In Figure 6.1 and Figure 6.2 we provide diagrammatic representation of normally and densely packed vector elements for the prime $p = 2^{255} - 19$.

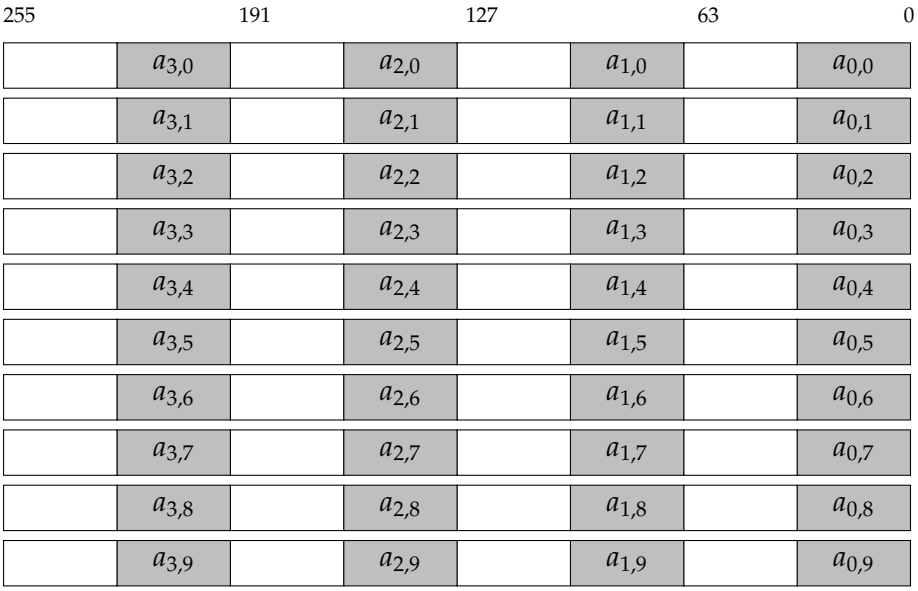

Figure 6.1: Normally packed vector field elements for the prime $p = 2^{255} - 19$ stored in 10 256-bit registers. The 32-bit wide white blocks are free.

255		191		127		63	0
$a_{3,5}$	$a_{3,0}$	$a_{2,5}$	$a_{2,0}$	$a_{1,5}$	$a_{1,0}$	$a_{0,5}$	$a_{0,0}$
$a_{3,6}$	$a_{3,1}$	$a_{2,6}$	$a_{2,1}$	$a_{1,6}$	$a_{1,1}$	$a_{0,6}$	$a_{0,1}$
$a_{3,7}$	$a_{3,2}$	$a_{2,7}$	$a_{2,2}$	$a_{1,7}$	$a_{1,2}$	$a_{0,7}$	$a_{0,2}$
$a_{3,8}$	$a_{3,3}$	$a_{2,8}$	$a_{2,3}$	$a_{1,8}$	$a_{1,3}$	$a_{0,8}$	$a_{0,3}$
$a_{3,9}$	$a_{3,4}$	$a_{2,9}$	$a_{2,4}$	$a_{1,9}$	$a_{1,4}$	$a_{0,9}$	$a_{0,4}$

Figure 6.2: Densely packed vector field elements for the prime $p = 2^{255} - 19$ stored in 5 256-bit registers. All 32-bit blocks are used.

6.7.3 Vector Reduction

There are three type of vector reduction operations will be used, namely REDUCE_1, REDUCE_2 and REDUCE_3 out of which REDUCE_3 will be used on densely packed limbs after the Hadamard transformations. We define them below.

- $\text{REDUCE}_1(\langle A_0, A_1, A_2, A_3\rangle)$: This is used in the vectorized field multiplication and squaring algorithms which returns $\langle \mathsf{reduce}_1(A_0), \mathsf{reduce}_1(A_1), \mathsf{reduce}_1(A_2), \mathsf{reduce}_1(A_3)\rangle$.
- $\text{REDUCE}_2(\langle A_0, A_1, A_2, A_3\rangle)$: This is used in the vectorized algorithm for multiplication by a field constant which returns $\langle \mathsf{reduce}_2(A_0), \mathsf{reduce}_2(A_1), \mathsf{reduce}_2(A_2), \mathsf{reduce}_2(A_3)\rangle$. The same reduction is also used after addition of two vector elements.
- $\text{REDUCE}_3(\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle)$: This is used in the vectorized algorithms for Hadamard transformations which returns $\langle \mathsf{reduce}_3(\underline{A_0}), \mathsf{reduce}_3(\underline{A_1}), \mathsf{reduce}_3(\underline{A_2}), \mathsf{reduce}_3(\underline{A_3})\rangle$. Details of reduce_3 will be defined later in the context of vectorized Hadamard transformation.

6.7.4 Vector Multiplication and Squaring

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

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

6.7.5 Vector Multiplication by a Field Constant

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

6.7.6 Vector Addition

The vectorized Montgomery ladder has a vector addition which is done over normally packed field elements. The operation is defined as $\text{ADD}(\langle A_0, A_1, A_2, A_3\rangle, \langle B_0, B_1, B_2, B_3\rangle)$ which returns $\text{REDUCE}_2(\langle C_0, C_1, C_2, C_3\rangle)$, where

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

Algorithms for vector Hadamard operations. For a normally packed 256-bit vector quantity $\mathbf{a} = \langle a_0, a_1, a_2, a_3\rangle$ we define $\mathsf{copy}_1(\mathbf{a}) = \langle a_0, a_0, a_2, a_2\rangle$ and $\mathsf{copy}_2(\mathbf{a}) = \langle a_1, a_1, a_3, a_3\rangle$. Similarly, for a densely packed 256-bit quantity $\underline{\mathbf{a}} = \langle \underline{a_0}, \underline{a_1}, \underline{a_2}, \underline{a_3}\rangle$ we define $\mathsf{copy}_3(\underline{\mathbf{a}}) = \langle \underline{a_0}, \underline{a_0}, \underline{a_2}, \underline{a_2}\rangle$ and $\mathsf{copy}_4(\underline{\mathbf{a}}) = \langle \underline{a_1}, \underline{a_1}, \underline{a_3}, \underline{a_3}\rangle$. The *copy* operations can be implemented using the assembly instruction `vpshufd`. The instruction `vpshufd` uses an additional parameter known as the shuffle mask, whose values for $\mathsf{copy}_1(\cdot)$ is 68 and for $\mathsf{copy}_2(\cdot)$ is 238. The vector Hadamard operation DENSE-H-H_1 and DENSE-H_2-H are described in Algorithm 6.2 and Algorithm 6.3 respectively. DENSE-H-H_1 implements the transformation $\mathcal{H}$-$\mathcal{H}_1$ and DENSE-H_2-H implements $\mathcal{H}_2$-$\mathcal{H}$. Due to the extra Step 5 in Algorithm 6.3, the function DENSE-H_2-H is slightly more costly than DENSE-H-H_1.

Algorithm 6.2 Vector Hadamard transformation.

1: **function** DENSE-H-H_1($\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle$)
2: **Input:** $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle = \sum_{i=0}^{\lceil \kappa/2\rceil - 1} \underline{\mathbf{a}_i}\theta^i$.
3: **Output:** $\underline{\mathbf{C}} = \sum_{i=0}^{\lceil \kappa/2\rceil - 1} \underline{\mathbf{c}_i}\theta^i$ representing $\langle \underline{A_0} + \underline{A_1}, \underline{A_0} - \underline{A_1}, \underline{A_2} - \underline{A_3}, \underline{A_2} + \underline{A_3}\rangle$, where each component is reduced modulo p_1 or p_2 depending on the chosen prime.
4: **for** $i \leftarrow 0$ to $\lceil \kappa/2\rceil - 1$ **do**
5: $\mathbf{s} \leftarrow \mathsf{copy}_3(\underline{\mathbf{a}_i})$
6: $\mathbf{t} \leftarrow \mathsf{copy}_4(\underline{\mathbf{a}_i})$
7: $\mathbf{t} \leftarrow \mathbf{t} \oplus \langle 0^{32}, 0^{32}, 1^{32}, 1^{32}, 1^{32}, 1^{32}, 0^{32}, 0^{32}\rangle$
8: $\mathbf{t} \leftarrow \mathbf{t} + \langle 0^{32}, 0^{32}, 2\mathfrak{p}_i + 1, 2\mathfrak{p}_{i+\lceil \kappa/2\rceil} + 1, 2\mathfrak{p}_i + 1, 2\mathfrak{p}_{i+\lceil \kappa/2\rceil} + 1, 0^{32}, 0^{32}\rangle$
9: $\underline{\mathbf{c}_i} \leftarrow \mathbf{s} + \mathbf{t}$
10: **end for**
11: **return** $\text{REDUCE}_3(\underline{\mathbf{C}})$
12: **end function**.

6.7.7 Vector Duplication

For the 256-bit quantity $\underline{\mathbf{a}} = \langle \underline{a_0}, \underline{a_1}, \underline{a_2}, \underline{a_3}\rangle$ let us define the operation $\mathsf{copy}_3(\underline{\mathbf{a}}) = \langle \underline{a_0}, \underline{a_1}, \underline{a_0}, \underline{a_1}\rangle$, which can be implemented using the assembly instruction `vpermq`. The instruction `vpermq` uses an additional parameter known as the shuffle mask, whose value for $\mathsf{copy}_3(\cdot)$ is 68. Let $\underline{\mathbf{A}} = \sum_{i=0}^{\lceil \kappa/2\rceil - 1} \underline{\mathbf{a}_i}\theta^i$. Define the operation DENSE-DUP($\underline{\mathbf{A}}$) to return $\sum_{i=0}^{\lceil \kappa/2\rceil - 1} \mathsf{copy}_3(\underline{\mathbf{a}_i})\theta^i$. If $\underline{\mathbf{A}}$ represents $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3}\rangle$, then DENSE-DUP($\underline{\mathbf{A}}$) $= \langle \underline{A_0}, \underline{A_1}, \underline{A_0}, \underline{A_1}\rangle$.

Algorithm 6.3 Vector Hadamard transformation.

function DENSE-H_2-H($\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle$)
Input: $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}_i} \theta^i$.
Output: $\underline{\mathbf{C}} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{c}_i} \theta^i$ representing $\langle \underline{A_1}, \underline{A_0} - \underline{A_1}, \underline{A_2} + \underline{A_3}, \underline{A_2} - \underline{A_3} \rangle$, where each component is reduced modulo p_1 or p_2 depending on the chosen prime.
    for $i \leftarrow 0$ to $\lceil \kappa/2 \rceil - 1$ **do**
        $\mathbf{s} \leftarrow \mathsf{copy}_3 \langle \underline{\mathbf{a}_i} \rangle$
        $\mathbf{s} \leftarrow \mathbf{s}$ and $\langle 0^{64}, 1^{64}, 1^{64}, 1^{64} \rangle$
        $\mathbf{t} \leftarrow \mathsf{copy}_4(\mathbf{a}_i)$
        $\mathbf{t} \leftarrow \mathbf{t} \oplus \langle 0^{32}, 0^{32}, 1^{32}, 1^{32}, 0^{32}, 0^{32}, 1^{32}, 1^{32} \rangle$
        $\mathbf{t} \leftarrow \mathbf{t} + \langle 0^{32}, 0^{32}, 2\mathfrak{p}_i + 1, 2\mathfrak{p}_{i+\lceil \kappa/2 \rceil} + 1, 0^{32}, 0^{32}, 2\mathfrak{p}_i + 1, 2\mathfrak{p}_{i+\lceil \kappa/2 \rceil} + 1 \rangle$
        $\underline{\mathbf{c}_i} \leftarrow \mathbf{s} + \mathbf{t}$
    end for
    return REDUCE$_3$($\underline{\mathbf{C}}$)
end function.

6.7.8 Vector Blending

For the 256-bit quantities $\mathbf{a} = \langle a_0, a_1, a_2, a_3 \rangle$ and $\mathbf{b} = \langle b_0, b_1, b_2, b_3 \rangle$ define the operation $\mathsf{mix}(\mathbf{a}, \mathbf{b}, \mathfrak{b}_0\mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3) = \langle c_0, c_1, c_2, c_3 \rangle$ such that

$$c_k \leftarrow \begin{cases} a_k & \text{if } \mathfrak{b}_k = 0, \\ b_k & \text{if } \mathfrak{b}_k = 1. \end{cases}$$

$\mathsf{mix}(\mathbf{a}, \mathbf{b}, \mathfrak{b}_0\mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3)$ can be implemented using the assembly instruction `vpblendd`. Let $\mathbf{A} = \sum_{i=0}^{\kappa-1} \mathbf{a}_i \theta^i$. Define the operation BLEND($\mathbf{A}, \mathbf{B}, \mathfrak{b}_0\mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3$) to return $\sum_{i=0}^{\kappa-1} \mathsf{mix}(\mathbf{a}_i, \mathbf{b}_i, \mathfrak{b}_0\mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3)\theta^i$. If $\mathbf{A}$ represents $\langle A_0, A_1, A_2, A_3 \rangle$, then BLEND($\mathbf{A}, \mathbf{B}, \mathfrak{b}_0\mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3$) $= \langle C_0, C_1, C_2, C_3 \rangle$ such that

$$C_k \leftarrow \begin{cases} A_k & \text{if } \mathfrak{b}_k = 0, \\ B_k & \text{if } \mathfrak{b}_k = 1. \end{cases}$$

The blending function BLEND can also be used over the densely packed operands $\underline{\mathbf{A}}, \underline{\mathbf{B}}$, and the working of the function does not change from the one defined above. We will call such a function as DENSE-BLEND.

6.7.9 Vector Swapping

Let $\underline{\mathbf{a}} = \langle \underline{a_0}, \underline{a_1}, \underline{a_2}, \underline{a_3} \rangle$ and $\mathfrak{b}$ be a bit. We define an operation $\mathsf{swap}(\underline{\mathbf{a}}, \mathfrak{b})$ as

$$\mathsf{swap}(\underline{\mathbf{a}}, \mathfrak{b}) \leftarrow \begin{cases} \langle \underline{a_0}, \underline{a_1}, \underline{a_2}, \underline{a_3} \rangle & \text{if } \mathfrak{b} = 0, \\ \langle \underline{a_2}, \underline{a_3}, \underline{a_0}, \underline{a_1} \rangle & \text{if } \mathfrak{b} = 1. \end{cases}$$

The operation $\mathsf{swap}(\underline{\mathbf{a}}, \mathfrak{b})$ is implemented using the assembly instruction `vpermd`. Let $\underline{\mathbf{A}} = \sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \underline{\mathbf{a}_i} \theta^i$. We define the operation DENSE-SWAP($\underline{\mathbf{A}}, \mathfrak{b}$) to return $\sum_{i=0}^{\lceil \kappa/2 \rceil - 1} \mathsf{swap}(\underline{\mathbf{a}_i}, \mathfrak{b})\theta^i$. If $\underline{\mathbf{A}}$ represents the vector $\langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle$, then

$$\text{DENSE-SWAP}(\underline{\mathbf{A}}, \mathfrak{b}) \leftarrow \begin{cases} \langle \underline{A_0}, \underline{A_1}, \underline{A_2}, \underline{A_3} \rangle & \text{if } \mathfrak{b} = 0, \\ \langle \underline{A_2}, \underline{A_3}, \underline{A_0}, \underline{A_1} \rangle & \text{if } \mathfrak{b} = 1. \end{cases}$$

The following summary classifies the different vector operations in terms of the type of packing of the operands.

- MUL, SQR, MULC, ADD, BLEND, PACK-N2D are applied to normally packed field elements.
- DENSE-SWAP, DENSE-H-H_1, DENSE-H_2-H, DENSE-DUP, DENSE-BLEND, PACK-D2N are applied to densely packed field elements.

6.8 Conclusion

In this chapter we have formalized the different algorithms of field operations which can be computed using 4-way vector instructions. These algorithms will be used in the 4-way vectorized implementations of the Montgomery ladder which we address next.

CHAPTER 7

Efficient 4-way Vectorizations of the Montgomery Ladder

This chapter is dedicated to the memory of Peter Lawrence Montgomery.

7.1 Introduction

Due to the practical importance of Curve25519 and also Curve448, the efficient implementations of X25519 and X448 are of major interest. The first efficient implementation of X25519 was provided by Bernstein himself in the paper which introduced the curve [Ber06b]. Since then, there has been a substantial amount of work on implementing X25519 on a variety of architectures [BS12, Cho15, CS09, DHH+15, FL15, FHLD19, FA17, HEY20, HS13, Moo15, OLH+17]. Several works have also provided efficient implementations of X448 [OLH+17, FHLD19].

Modern processor architectures provide support for single instruction multiple data (SIMD) operations. This allows performing the same operation on a vector of inputs. Vectorization leads to efficiency gains. Arguments in favor of vectorization have been put forward by Bernstein[1].

Scalar multiplication on Montgomery form curves is performed using the so-called Montgomery ladder algorithm. This is an iterative algorithm where each iteration or ladder-step performs a combined double and differential addition of curve points. The ladder-step is the primary target for vectorization. The idea behind such vectorization is to form groups of independent multiplications so that the SIMD instructions can be applied to the groups. To the best of our knowledge, the first work which considered grouping together four independent multiplications was by Costigan and Schwabe [CS09]. Subsequent work by Bernstein and Schwabe [BS12] and Chou [Cho15] considered grouping together two independent multiplications. A modification of the algorithm of Chou [Cho15], also grouping together two independent multiplications was proposed by Faz-Hernández and López [FL15]. Even though the algorithm grouped together two independent multiplications, in [FL15] it was implemented using the 4-way SIMD instructions. An improved implementation of the same algorithm has been reported in

[1] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/mmsH4k3j_1g/m/JfzP1EBuBQAJ, accessed on March 10, 2020.

[FHLD19]. Recently, the work [HEY20] proposed a vectorization strategy which groups together four independent multiplications and provided its implementation using 4-way SIMD instructions.

Modern processors provide support for 4-way SIMD instructions. To fully exploit this feature, it is required to form groups of four independent multiplications. As mentioned above, the only previous works to consider this are [CS09, HEY20]. For variable base scalar multiplication, the vectorization strategy of [HEY20] is faster than that of [CS09], while for fixed base scalar multiplication, the vectorization strategy of [CS09] is faster than that of [HEY20].

In this work, we present new 4-way vectorizations of the Montgomery ladder-step. The first algorithm that we propose consists of two general multiplication rounds (one round consisting of two squarings and two multiplications and the other round consisting of three multiplications), one squaring round (consisting of two squarings) and a round which performs a multiplication by a curve constant. The second algorithm has two groups of four multiplications, one multiplication by the curve constant and one multiplication by the x-coordinate of the base point. In the case where the base point is fixed and its x-coordinate is small, the second strategy is faster than the first strategy.

For variable base scalar multiplication, a comparison of our first algorithm with [HEY20] shows a trade-off. While [HEY20] does not require the round consisting of multiplication by a constant, it requires several extra non-multiplication operations. It has been observed through our concrete implementations that the advantage of avoiding the multiplication by constant is outweighed by the overhead of the additional non-multiplication operations. For fixed base scalar multiplication, our second algorithm is shown to be clearly faster than [CS09].

We provide efficient constant time assembly implementations of both our vectorized algorithms for X25519 and X448. For X25519, an Intel intrinsics based implementation has been reported in [HEY20]. We provide improved implementation of the vectorized algorithm of [HEY20] for X25519; the improvement comes in two parts – an assembly implementation and faster multiplication/squaring. For X448, we provide the first efficient assembly implementation of the vectorized algorithm of [HEY20]. We have made the source codes of all our implementations publicly available at the link

`https://github.com/kn-cs/vec-ladder`.

Timing results on the Skylake and Haswell processors have been obtained for all the implementations that we have made. For comparison, we have measured the performance of previous codes [FHLD19, HEY20, OLH+17] on the same computers where we measured our code. For variable base scalar multiplication, the new algorithm proposed here shows a major improvement in speed over [FHLD19, OLH+17] and a modest, but noticeable improvement in speed over [HEY20]. These results indicate that for practical implementations of shared secret generation phase of ECDH protocol over Curve25519 and Curve448, the new vectorized algorithm proposed in this work is preferable over previous works.

For fixed base scalar multiplication, the second vectorized algorithm that we present significantly improves upon the speed of variable base scalar multiplication. If implementation of the key generation phase of the ECDH protocol is to be done over Montgomery curves, then this is the algorithm of choice.

7.2 The 4-way Vectorization of [HEY20]

In [HEY20], the computation of $(X_2' : Z_2')$ was done in a manner different from that shown in (2.1). Retracing the computation of $(X_2' : Z_2')$ given in Figure 1 of [HEY20] shows that the following formula was used.

$$\left.\begin{array}{rcl}(X_2' : Z_2') & = & \big(2((X_2+Z_2)^2+(X_2-Z_2)^2)^2-(4X_2Z_2)^2 \\ & & \quad : 2((X_2+Z_2)^2+(X_2-Z_2)^2)4X_2Z_2+A(4X_2Z_2)^2\big)\,.\end{array}\right\} \tag{7.1}$$

Assuming that $4X_2Z_2$ is computed as mentioned above, the computations of $(X_3' : Z_3')$ using (2.1) and of $(X_2' : Z_2')$ using (7.1) require 4 multiplications, 6 squarings and one multiplication by the curve constant A. The total number of multiplications and squaring is one more than that required for computation of (2.1). For a sequential computation this would be inefficient, but for 4-way vectorization, the extra operation does not necessarily lead to a less efficient method. Note that $2((X_2+Z_2)^2+(X_2-Z_2)^2)^2-(4X_2Z_2)^2$ can also be computed more simply as $4(X_2+Z_2)^2(X_2-Z_2)^2$. We have investigated this possibility and it turns out that the resulting 4-way vectorization is somewhat less efficient than the 4-way vectorization obtained using (7.1).

7.3 New 4-way Vectorizations of the Montgomery Ladder

In this section, we present two new vectorization strategies for the Montgomery ladder algorithm.

Before describing the new algorithms, we introduce some notation. By **0** and **1**, we will denote the additive and the multiplicative identities of $\mathbb{F}_p$ respectively. The ladder algorithm uses the constant $(A+2)/4$. For practical curves like Curve25519 and Curve448, the value of this constant is small and the constant can be represented using a single 64-bit word. We denote the constant by $a24$.

The batching strategies for the Montgomery ladder-step proposed in this work are shown in Figures 7.1 and 7.2. It is not difficult to verify that the computations done in Figures 7.1 and 7.2 are essentially different ways of computing the formulas given in (2.1). So, the new algorithms provide different ways of computing the Montgomery ladder-step. The figures only show groupings of multiplications and other operations. To obtain vectorized algorithms, it is required to convert the algorithms using 4-way vector operations. For this, we need to introduce some top-level vector operations. Later we discuss how these vector operations can be realized with the 4-way SIMD instructions.

For $a, b \in \mathbb{F}_p$, define $\mathcal{H}(a,b) = (a+b, a-b)$, $\mathcal{H}_1(a,b) = (a-b, a+b)$, $\mathcal{H}_2(a,b) = (0+b, a-b)$. The following vector operations will be used to provide top-level descriptions of the different vectorization strategies. The vector $\langle A_0, A_1, A_2, A_3\rangle$ represents 4 field elements A_0, A_1, A_2, A_3, where each A_i is represented using κ limbs. Similar interpretation holds for the vectors $\langle B_0, B_1, B_2, B_3\rangle$ and $\langle C_0, C_1, C_2, C_3\rangle$. The vector $\langle c_0, c_1, c_2, c_3\rangle$ represents the 4 single limb quantities c_0, c_1, c_2 and c_3.

- $\mathcal{H}$-$\mathcal{H}(\langle A_0, A_1, A_2, A_3\rangle) = \langle A_0+A_1, A_0-A_1, A_2+A_3, A_2-A_3\rangle$.
- $\mathcal{H}$-$\mathcal{H}_1(\langle A_0, A_1, A_2, A_3\rangle) = \langle A_0+A_1, A_0-A_1, A_2-A_3, A_2+A_3\rangle$.
- $\mathcal{H}_2$-$\mathcal{H}(\langle A_0, A_1, A_2, A_3\rangle) = \langle A_1, A_0-A_1, A_2+A_3, A_2-A_3\rangle$.

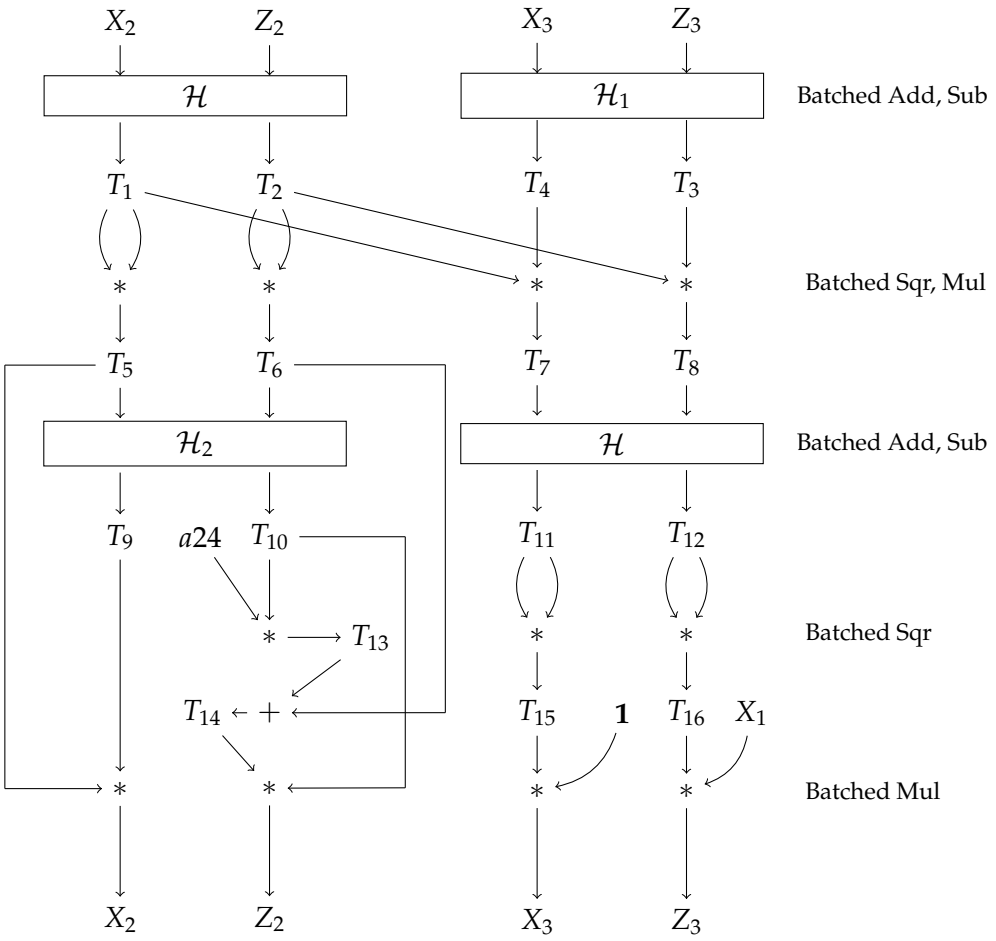

Figure 7.1: A batching strategy for computing the formulas in (2.1)

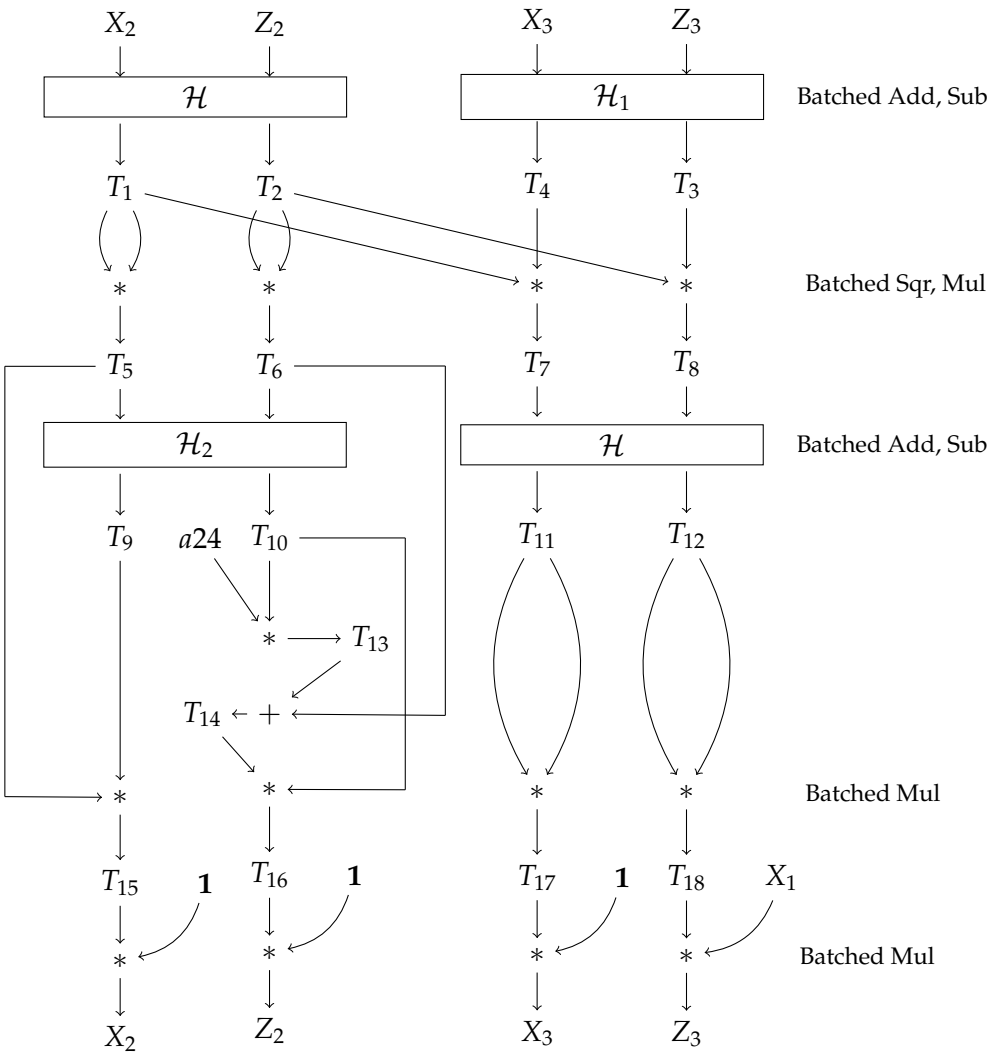

Figure 7.2: A batching strategy for computing the formulas in (2.1)

- $\mathcal{ADD}(\langle A_0, A_1, A_2, A_3\rangle, \langle B_0, B_1, B_2, B_3\rangle) = \langle A_0 + B_0, A_1 + B_1, A_2 + B_2, A_3 + B_3\rangle$.
- $\mathcal{SUB}(\langle A_0, A_1, A_2, A_3\rangle, \langle B_0, B_1, B_2, B_3\rangle) = \langle A_0 - B_0, A_1 - B_1, A_2 - B_2, A_3 - B_3\rangle$.
- $\mathcal{MUL}(\langle A_0, A_1, A_2, A_3\rangle, \langle B_0, B_1, B_2, B_3\rangle) = \langle A_0 \cdot B_0, A_1 \cdot B_1, A_2 \cdot B_2, A_3 \cdot B_3\rangle$.
- $\mathcal{SQR}(\langle A_0, A_1, A_2, A_3\rangle) = \langle A_0^2, A_1^2, A_2^2, A_3^2\rangle$.
- $\mathcal{MULC}(\langle A_0, A_1, A_2, A_3\rangle, \langle c_0, c_1, c_2, c_3\rangle) = \langle c_0 \cdot A_0, c_1 \cdot A_1, c_2 \cdot A_2, c_3 \cdot A_3\rangle$.
- $\mathcal{DUP}(\langle A_0, A_1, A_2, A_3\rangle) = \langle A_0, A_1, A_0, A_1\rangle$.
- $\mathcal{SHUFFLE}(\langle A_0, A_1, A_2, A_3\rangle) = \langle A_1, A_0, A_3, A_2\rangle$.
- $\mathcal{BLEND}(\langle A_0, A_1, A_2, A_3\rangle, \langle B_0, B_1, B_2, B_3\rangle, \mathfrak{b}_0\mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3) = \langle C_0, C_1, C_2, C_3\rangle)$, where $C_i = A_i$ if $\mathfrak{b}_i = 0$ and $C_i = B_i$ if $\mathfrak{b}_i = 1$.

Vectorized descriptions of Figures 7.1 and 7.2 are provided in Algorithms 7.1 and 7.2 respectively. For the purpose of comparison, in Algorithms 7.3 and 7.4 we provide the 4-way vectorization strategies obtained from the descriptions given in [CS09] and [HEY20] respectively. We note that Algorithms 7.1, 7.2 and 7.3 implement the formulas given by (2.1), whereas Algorithm 7.4 implements the formulas given by (2.1) as modified in (7.1).

Algorithm 7.1 4-way vectorization of Montgomery ladder-step corresponding to Figure 7.1.

function VECTORIZED-LADDER-STEP($\langle X_2, Z_2, X_3, Z_3\rangle, \langle \mathbf{0}, \mathbf{0}, \mathbf{1}, X_1\rangle$)
  $\langle T_1, T_2, T_4, T_3\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}_1(\langle X_2, Z_2, X_3, Z_3\rangle)$
  $\langle T_1, T_2, T_1, T_2\rangle \leftarrow \mathcal{DUP}(\langle T_1, T_2, T_4, T_3\rangle)$
  $\langle T_5, T_6, T_7, T_8\rangle \leftarrow \mathcal{MUL}(\langle T_1, T_2, T_4, T_3\rangle, \langle T_1, T_2, T_1, T_2\rangle)$
  $\langle T_9, T_{10}, T_{11}, T_{12}\rangle \leftarrow \mathcal{H}_2\text{-}\mathcal{H}(\langle T_5, T_6, T_7, T_8\rangle)$
  $\langle T_9, T_{10}, \mathbf{1}, X_1\rangle \leftarrow \mathcal{BLEND}(\langle \mathbf{0}, \mathbf{0}, \mathbf{1}, X_1\rangle, \langle T_9, T_{10}, T_{11}, T_{12}\rangle, 1100)$
  $\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0}\rangle \leftarrow \mathcal{MULC}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle, \langle 0^{64}, a24, 0^{64}, 0^{64}\rangle\rangle$
  $\langle T_5, T_{14}, T_7, T_8\rangle \leftarrow \mathcal{ADD}(\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0}\rangle, \langle T_5, T_6, T_7, T_8\rangle)$
  $\langle *, *, T_{15}, T_{16}\rangle \leftarrow \mathcal{SQR}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle)$
  $\langle T_5, T_{14}, T_{15}, T_{16}\rangle \leftarrow \mathcal{BLEND}(\langle T_5, T_{14}, T_7, T_8\rangle, \langle *, *, T_{15}, T_{16}\rangle, 0011)$
  $\langle X_2, Z_2, X_3, Z_3\rangle \leftarrow \mathcal{MUL}(\langle T_5, T_{14}, T_{15}, T_{16}\rangle, \langle T_9, T_{10}, \mathbf{1}, X_1\rangle)$
  return $\langle X_2, Z_2, X_3, Z_3\rangle$
end function.

The numbers of various vector operations required by the Algorithms 7.1, 7.2, 7.3 and 7.4 are shown in Table 7.1. In the table, the numbers corresponding to $\mathcal{HAD}$ are the counts of $\mathcal{H}$-$\mathcal{H}$, $\mathcal{H}$-$\mathcal{H}_1$, or $\mathcal{H}_2$-$\mathcal{H}$ operations.

While the vector multiplications are indeed the most time consuming operations, the other operations can also take a significant amount of time. Regarding these other operations, we note that Algorithm 7.4 requires the maximum number of such operations and Algorithms 7.1 and 7.2 require the least. Among the non-multiplication operations, the $\mathcal{HAD}$ operations require the maximum amount of time. We note that three $\mathcal{HAD}$

Algorithm 7.2 4-way vectorization of Montgomery ladder-step corresponding to Figure 7.2.

1: **function** VECTORIZED-LADDER-STEP($\langle X_2, Z_2, X_3, Z_3\rangle, \langle \mathbf{1}, \mathbf{1}, \mathbf{1}, X_1\rangle$)
2: $\langle T_1, T_2, T_4, T_3\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}_1(\langle X_2, Z_2, X_3, Z_3\rangle)$
3: $\langle T_1, T_2, T_1, T_2\rangle \leftarrow \mathcal{DUP}(\langle T_1, T_2, T_4, T_3\rangle)$
4: $\langle T_5, T_6, T_7, T_8\rangle \leftarrow \mathcal{MUL}(\langle T_1, T_2, T_4, T_3\rangle, \langle T_1, T_2, T_1, T_2\rangle)$
5: $\langle T_9, T_{10}, T_{11}, T_{12}\rangle \leftarrow \mathcal{H}_2\text{-}\mathcal{H}(\langle T_5, T_6, T_7, T_8\rangle)$
6: $\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0}\rangle \leftarrow \mathcal{MULLC}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle, \langle 0^{64}, a24, 0^{64}, 0^{64}\rangle\rangle$
7: $\langle T_5, T_{14}, T_7, T_8\rangle \leftarrow \mathcal{ADD}(\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0}\rangle, \langle T_5, T_6, T_7, T_8\rangle)$
8: $\langle T_5, T_{14}, T_{11}, T_{12}\rangle \leftarrow \mathcal{BLENDD}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle, \langle T_5, T_{14}, T_7, T_8\rangle, 1100)$
9: $\langle T_{15}, T_{16}, T_{17}, T_{18}\rangle \leftarrow \mathcal{MUL}(\langle T_5, T_{14}, T_{11}, T_{12}\rangle, \langle T_9, T_{10}, T_{11}, T_{12}\rangle)$
10: $\langle X_2, Z_2, X_3, Z_3\rangle \leftarrow \mathcal{MUL}(\langle T_{15}, T_{16}, T_{17}, T_{18}\rangle, \langle \mathbf{1}, \mathbf{1}, \mathbf{1}, X_1\rangle)$
11: **return** $\langle X_2, Z_2, X_3, Z_3\rangle$
12: **end function**.

Algorithm 7.3 4-way vectorization of Montgomery ladder-step obtained from [CS09].

1: **function** VECTORIZED-LADDER-STEP($\langle X_2, Z_2, X_3, Z_3\rangle, \langle \mathbf{1}, \mathbf{1}, \mathbf{1}, X_1\rangle$)
2: $\langle T_1, T_2, T_4, T_3\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}_1(\langle X_2, Z_2, X_3, Z_3\rangle)$
3: $\langle T_1, T_2, T_1, T_2\rangle \leftarrow \mathcal{DUP}(\langle T_1, T_2, T_4, T_3\rangle)$
4: $\langle T_5, T_6, T_7, T_8\rangle \leftarrow \mathcal{MUL}(\langle T_1, T_2, T_4, T_3\rangle, \langle T_1, T_2, T_1, T_2\rangle)$
5: $\langle T_9, T_{10}, \mathbf{0}, \mathbf{0}\rangle \leftarrow \mathcal{MULLC}(\langle T_5, T_6, T_7, T_8\rangle, \langle a24, a24 - 1, 0^{64}, 0^{64}\rangle)$
6: $\langle *, T_{11}, T_{13}, T_{14}\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}(\langle T_5, T_6, T_7, T_8\rangle)$
7: $\langle *, T_{12}, *, *\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}(\langle T_9, T_{10}, \mathbf{0}, \mathbf{0}\rangle)$
8: $\langle T_5, T_{11}, T_{13}, T_{14}\rangle \leftarrow \mathcal{BLENDD}(\langle T_5, T_6, T_7, T_8\rangle, \langle *, T_{11}, T_{13}, T_{14}\rangle, 0111)$
9: $\langle T_6, *, *, *\rangle \leftarrow \mathcal{SHUFFLE}(\langle T_5, T_6, T_7, T_8\rangle)$
10: $\langle T_6, T_{12}, *, *\rangle \leftarrow \mathcal{BLENDD}(\langle T_6, *, *, *\rangle, \langle *, T_{12}, *, *\rangle, 01dd)$
11: $\langle T_6, T_{12}, T_{13}, T_{14}\rangle \leftarrow \mathcal{BLENDD}(\langle T_6, T_{12}, *, *\rangle, \langle T_5, T_{11}, T_{13}, T_{14}\rangle, 0011)$
12: $\langle X_2, Z_2, X_3, T_{15}\rangle \leftarrow \mathcal{MUL}(\langle T_5, T_{11}, T_{13}, T_{14}\rangle, \langle T_6, T_{12}, T_{13}, T_{14}\rangle)$
13: $\langle X_2, Z_2, X_3, Z_3\rangle \leftarrow \mathcal{MUL}(\langle X_2, Z_2, X_3, T_{15}\rangle, \langle \mathbf{1}, \mathbf{1}, \mathbf{1}, X_1\rangle)$
14: **return** $\langle X_2, Z_2, X_3, Z_3\rangle$
15: **end function**.

operations are required by Algorithm 7.3 while two $\mathcal{HAD}$ operations are required by the other algorithms.

Step 13 of Algorithm 7.3 and Step 11 of Algorithm 7.2 perform the product of $\langle X_2, Z_2, X_3, T_{15}\rangle$ and $\langle \mathbf{1}, \mathbf{1}, \mathbf{1}, X_1\rangle$. In Table 7.1, this multiplication has been counted as a general field multiplication. On the other hand, if X_1 is a small constant, then this multiplication should be counted as multiplication by a small field constant. Based on this distinction, to compare between the algorithms based on the operation counts given in Table 7.1, we consider two situations.

Algorithm 7.4 4-way vectorization of Montgomery ladder-step obtained from Figure 1 in [HEY20].

function VECTORIZED-LADDER-STEP($\langle X_2, Z_2, X_3, Z_3\rangle, \langle \mathbf{0}, A, \mathbf{1}, X_1\rangle$)
  $\langle T_1, T_2, T_3, T_4\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}(\langle X_2, Z_2, X_3, Z_3\rangle)$
  $\langle T_1, T_2, T_2, T_1\rangle \leftarrow \mathcal{DUP}(\langle T_1, T_2, T_3, T_4\rangle)$
  $\langle T_5, T_6, T_7, T_8\rangle \leftarrow \mathcal{MUL}(\langle T_1, T_2, T_3, T_4\rangle, \langle T_1, T_2, T_2, T_1\rangle)$
  $\langle T_9, T_{10}, T_{11}, T_{12}\rangle \leftarrow \mathcal{H}\text{-}\mathcal{H}(\langle T_5, T_6, T_7, T_8\rangle)$
  $\langle T_{10}, T_9, T_{12}, T_{11}\rangle \leftarrow \mathcal{SHUFFLE}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle)$
  $\langle T_{10}, A, \mathbf{1}, X_1\rangle \leftarrow \mathcal{BLEND}(\langle \mathbf{0}, A, \mathbf{1}, X_1\rangle, \langle T_{10}, T_9, T_{12}, T_{11}\rangle, 1000)$
  $\langle T_{13}, T_{14}, T_{15}, T_{16}\rangle \leftarrow \mathcal{SQR}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle)$
  $\langle T_{14}, T_{13}, T_{16}, T_{15}\rangle \leftarrow \mathcal{SHUFFLE}(\langle T_{13}, T_{14}, T_{15}, T_{16}\rangle)$
  $\langle X_2, *, *, *\rangle \leftarrow \mathcal{SUB}(\langle T_{13}, T_{14}, T_{15}, T_{16}\rangle, \langle T_{14}, T_{13}, T_{15}, T_{16}\rangle)$
  $\langle T_9, T_{14}, T_{15}, T_{16}\rangle \leftarrow \mathcal{BLEND}(\langle T_9, T_{10}, T_{11}, T_{12}\rangle, \langle T_{13}, T_{14}, T_{15}, T_{16}\rangle, 0111)$
  $\langle T_{17}, T_{18}, X_3, Z_3\rangle \leftarrow \mathcal{MUL}(\langle T_{10}, A, \mathbf{1}, X_1\rangle, \langle T_9, T_{14}, T_{15}, T_{16}\rangle)$
  $\langle T_{19}, *, *, *\rangle \leftarrow \mathcal{ADD}(\langle T_{17}, T_{18}, X_3, Z_3\rangle, \langle T_{17}, T_{18}, X_3, Z_3\rangle)$
  $\langle *, T_{19}, *, *\rangle \leftarrow \mathcal{SHUFFLE}(\langle T_{19}, *, *, *\rangle)$
  $\langle *, Z_2, *, *\rangle \leftarrow \mathcal{ADD}(\langle T_{17}, T_{18}, X_3, Z_3\rangle, \langle *, T_{19}, *, *\rangle)$
  $\langle X_2, Z_2, *, *\rangle \leftarrow \mathcal{BLEND}(\langle X_2, *, *, *\rangle, \langle *, Z_2, *, *\rangle, 01\_\_)$
  $\langle X_2, Z_2, X_3, Z_3\rangle \leftarrow \mathcal{BLEND}(\langle X_2, Z_2, *, *\rangle, \langle T_{17}, T_{18}, X_3, Z_3\rangle, 0011)$
  return $\langle X_2, Z_2, X_3, Z_3\rangle$
end function.

Vector Operations	Algorithm 7.1	Algorithm 7.2	Algorithm 7.3	Algorithm 7.4
$\mathcal{MUL}$	2	3	3	2
$\mathcal{SQR}$	1	-	-	1
$\mathcal{MULC}$	1	1	1	-
$\mathcal{HAD}$	2	2	3	2
$\mathcal{ADD}$	1	1	-	2
$\mathcal{SUB}$	-	-	-	1
$\mathcal{DUP}$	1	1	1	1
$\mathcal{BLEND}$	2	1	3	4
$\mathcal{SHUFFLE}$	-	-	1	3

Table 7.1: Comparison of the vector operations required by different algorithms.

7.3.1 Variable Base Scalar Multiplication

In this case, the quantity X_1 is a general element of the field. Clearly, from Table 7.1 we see that both Algorithms 7.2 and 7.3 will be slower than either of the Algorithms 7.1 or 7.4. So, for variable base scalar multiplication, the comparison is really between Algorithms 7.1 and 7.4. Both require 2 $\mathcal{MUL}$+1 $\mathcal{SQR}$. The trade-off between the two algorithms is that Algorithm 7.1 requires 1 $\mathcal{MULC}$ whereas Algorithm 7.4 requires quite

a few extra non-multiplication operations. So, from the operation count itself it is not immediately clear which of the two algorithms will be faster. The implementation results that we report later show that in practice Algorithm 7.1 turns out to be faster on the Intel processors we have benchmarked.

7.3.2 Fixed Base Scalar Multiplication

In this case, the quantity X_1 is small. In Algorithms 7.2 and 7.3, the product of $\langle X_2, Z_2, X_3, T_{15}\rangle$ and $\langle \mathbf{1}, \mathbf{1}, \mathbf{1}, X_1\rangle$ is to be counted as $\mathcal{MULC}$ instead of $\mathcal{MUL}$. In this case, the number of vector multiplications performed by Algorithms 7.2 and 7.3 will be $2\,\mathcal{MUL} + 2\,\mathcal{MULC}$ operations. The resulting cost of Algorithms 7.2 and 7.3 will be lower than that of Algorithms 7.1 and 7.4. From Table 7.1, a comparison between Algorithms 7.2 and 7.3 shows that the number of non-multiplication steps required by Algorithm 7.2 is smaller than that of Algorithm 7.3. In particular, the number of $\mathcal{HAD}$ operations is two for Algorithm 7.2 while it is three for Algorithm 7.3. So, for fixed base scalar multiplication, Algorithm 7.2 will be faster than Algorithm 7.3 and also faster than Algorithms 7.1 and 7.4.

7.4 Vectorized Montgomery Ladder

Algorithm 7.5 describes the vectorized Montgomery ladder. For variable base scalar multiplication, Algorithm 7.6 describes a single step of the ladder. The x-coordinate X_1 of the point P is represented as a κ-limb quantity (Recall that $\kappa = 9$ or 10 for p_1 and $\kappa = 16$ for p_2). The variables X_2 and Z_3 are initialized with the κ-limb representation of $\mathbf{1}$. The variable Z_2 is initialized with the κ-limb representation of $\mathbf{0}$ and the variable X_3 is initialized with the κ-limb representation of X_1. So, the vector $\langle X_2, Z_2, X_3, Z_3\rangle$ is represented by a $\kappa \times 4$ matrix. We use the pre-calculated 4-tuple $\langle \mathbf{0}, \mathbf{0}, \mathbf{1}, X_1\rangle$ as a fixed value before the ladder-loop starts.

Algorithm 7.6 is an optimized version of Algorithm 7.1. The steps of Algorithm 7.6 can be easily related to the various steps of Algorithm 7.1. The operation DENSE-H-H_1 of Step 2 realizes the Hadamard operation $\mathcal{H}$-$\mathcal{H}_1$ and DENSE-H_2-H of Step 8 realizes $\mathcal{H}_2$-$\mathcal{H}$. The operation DENSE-DUP of Step 4 realizes the operation $\mathcal{DUP}$ and the operation DENSE-BLEND of Step 9 realizes the $\mathcal{BLEND}$ operation of Step 6. All these operations are performed on densely packed operands. The BLEND operation of Step 15 realizes the $\mathcal{BLEND}$ of Step 10 with normally packed operand. The operations $\mathcal{MUL}$, $\mathcal{SQR}$, $\mathcal{MULC}$ and $\mathcal{ADD}$ of Algorithm 7.1, which are performed on normally packed operands are realized respectively by Steps 6,16,14,12,13 of Algorithm 7.6.

Below we mention a few important points regarding the implementations of Algorithm 7.6 for Curve25519 and Curve448.

1. For Curve25519 with $\kappa = 10$, the outputs of the vector Hadamard transformations in Steps 2 and 8 of the VECTORIZED-LADDER-STEP can be kept unreduced. This is so because, a size increment by at most 2 bits in the limbs does not produce any overfull in the integer multiplication/squaring algorithm for p_1.

2. For Curve25519 with $\kappa = 9$, the outputs of the vector Hadamard transformations cannot be kept unreduced. In this case a size increment by at most 2 bits in the

Algorithm 7.5 Montgomery ladder with 4-way vectorization. In the algorithm $m = \lceil \lg p \rceil$.

1: **function** VECTORIZED-MONT-LADDER(X_1, n)
2: **input:** A point $P = (X_1 : \cdot : 1)$ on $E_{M,A,B}(\mathbb{F}_p)$ and an m-bit scalar n.
3: **output:** $\mathbf{x}_0(nP)$.
4: $X_2 = \mathbf{1}; Z_2 = \mathbf{0}; X_3 = X_1; Z_3 = \mathbf{1}$
5: prevbit $\leftarrow 0$
6: $\langle \underline{\mathbf{0}}, \underline{\mathbf{0}}, \underline{\mathbf{1}}, \underline{X_1} \rangle \leftarrow$ PACK-N2D$(\langle \mathbf{0}, \mathbf{0}, \mathbf{1}, X_1 \rangle)$
7: $\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle \leftarrow$ PACK-N2D$(\langle X_2, Z_2, X_3, Z_3 \rangle)$
8: **for** $i \leftarrow m - 1$ **down to** 0 **do**
9: bit $\leftarrow$ bit at index i of n
10: $\mathfrak{b} \leftarrow$ bit $\oplus$ prevbit
11: prevbit $\leftarrow$ bit
12: $\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle \leftarrow$ DENSE-SWAP$(\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle, \mathfrak{b})$
13: $\langle X_2, Z_2, X_3, Z_3 \rangle \leftarrow$ VECTORISED-LADDER-STEP$(\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle, \langle \underline{\mathbf{0}}, \underline{\mathbf{0}}, \underline{\mathbf{1}}, \underline{X_1} \rangle)$
14: $\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle \leftarrow$ PACK-N2D$(\langle X_2, Z_2, X_3, Z_3 \rangle)$
15: **end for**
16: $\langle X_2, Z_2, X_3, Z_3 \rangle \leftarrow$ PACK-D2N$(\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle)$
17: $\langle X_2, Z_2, X_3, Z_3 \rangle \leftarrow$ REDUCE$_2(\langle X_2, Z_2, X_3, Z_3 \rangle)$
18: **return** $X_2 \cdot Z_2^{p-2}$
19: **end function**.

limbs produces overfull in the integer multiplication/squaring algorithm. We apply the reduction chain $(c_0, c_4) \rightarrow (c_1, c_5) \rightarrow (c_2, c_6) \rightarrow (c_3, c_7)$ in parallel over densely packed field elements. The reductions $c_3 \rightarrow c_4$, $c_7 \rightarrow c_8$ and $c_8 \rightarrow c_0$ are applied sequentially.

3. For Curve448 the outputs of the vector Hadamard transformations cannot be kept unreduced since in this case also, a size increment by at most 2 bits in the limbs produces overfull in the integer multiplication/squaring algorithm for p_2. On the other hand, it is sufficient to use only the reduction steps covered by the parallel reduction chain $(c_0, c_8) \rightarrow (c_1, c_9) \rightarrow \cdots \rightarrow (c_7, c_{15})$. Such a reduction keeps at most 3 extra bits in the limbs at index 7 and 15 of the field element and this does not lead to any overfull for the multiplication/squaring algorithm applied further. The sequential reduction steps $c_7 \rightarrow c_8$, $c_{15} \rightarrow (c_0, c_8)$ can be skipped and this provides some time saving in the computation.

4. The output of MULC operation in Step 12 is kept unreduced.

5. The PACK-D2N operation can be implemented using the `vpsrlq` and `vpand` instructions. On the other hand, for the implementation of Algorithm 7.6 it is sufficient to use only the `vpsrlq` instruction, which helps to extract the lower $\lceil \kappa/2 \rceil$ limbs of the field elements from the densely packed limbs. It is not necessary to mask off the upper 32-bits of the densely packed limbs because the `vpmuludq` instruction is not dependent on the values stored in the upper 32-bits. This makes

Algorithm 7.6 Vectorized algorithm of Montgomery ladder-step corresponding to Algorithm 7.1.

1: **function** VECTORIZED-LADDER-STEP($\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle, \langle \underline{\mathbf{0}}, \underline{\mathbf{0}}, \underline{\mathbf{1}}, \underline{X_1} \rangle$)
2: $\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle \leftarrow$ DENSE-H-H$_1$($\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle$)
3: $\langle T_1, T_2, T_4, T_3 \rangle \leftarrow$ PACK-D2N($\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle$)
4: $\langle \underline{T_1}, \underline{T_2}, \underline{T_1}, \underline{T_2} \rangle \leftarrow$ DENSE-DUP($\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle$)
5: $\langle T_1, T_2, T_1, T_2 \rangle \leftarrow$ PACK-D2N($\langle \underline{T_1}, \underline{T_2}, \underline{T_1}, \underline{T_2} \rangle$)
6: $\langle T_5, T_6, T_7, T_8 \rangle \leftarrow$ MUL($\langle T_1, T_2, T_4, T_3 \rangle, \langle T_1, T_2, T_1, T_2 \rangle$)
7: $\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8} \rangle \leftarrow$ PACK-N2D($\langle T_5, T_6, T_7, T_8 \rangle$)
8: $\langle \underline{T_9}, \underline{T_{10}}, \underline{T_{11}}, \underline{T_{12}} \rangle \leftarrow$ DENSE-H$_2$-H($\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8} \rangle$)
9: $\langle \underline{T_9}, \underline{T_{10}}, \underline{\mathbf{1}}, \underline{X_1} \rangle \leftarrow$ DENSE-BLEND($\langle \underline{\mathbf{0}}, \underline{\mathbf{0}}, \underline{\mathbf{1}}, \underline{X_1} \rangle, \langle \underline{T_9}, \underline{T_{10}}, \underline{T_{11}}, \underline{T_{12}} \rangle, 1100$)
10: $\langle T_9, T_{10}, \mathbf{1}, X_1 \rangle \leftarrow$ PACK-D2N($\langle \underline{T_9}, \underline{T_{10}}, \underline{\mathbf{1}}, \underline{X_1} \rangle$)
11: $\langle T_9, T_{10}, T_{11}, T_{12} \rangle \leftarrow$ PACK-D2N($\langle \underline{T_9}, \underline{T_{10}}, \underline{T_{11}}, \underline{T_{12}} \rangle$)
12: $\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0} \rangle \leftarrow$ UNREDUCED-MULC($\langle T_9, T_{10}, T_{11}, T_{12} \rangle, \langle 0^{64}, a24, 0^{64}, 0^{64} \rangle$)
13: $\langle T_5, T_{14}, T_7, T_8 \rangle \leftarrow$ ADD($\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0} \rangle, \langle T_5, T_6, T_7, T_8 \rangle$)
14: $\langle *, *, T_{15}, T_{16} \rangle \leftarrow$ SQR($\langle T_9, T_{10}, T_{11}, T_{12} \rangle$)
15: $\langle T_5, T_{14}, T_{15}, T_{16} \rangle \leftarrow$ BLEND($\langle T_5, T_{14}, T_7, T_8 \rangle, \langle *, *, T_{15}, T_{16} \rangle, 0011$)
16: $\langle X_2, Z_2, X_3, Z_3 \rangle \leftarrow$ MUL($\langle T_5, T_{14}, T_{15}, T_{16} \rangle, \langle T_9, T_{10}, \mathbf{1}, X_1 \rangle$)
17: **return** $\langle X_2, Z_2, X_3, Z_3 \rangle$
18: **end function**.

the PACK-D2N operation less costly than PACK-N2D.

6. The DENSE-DUP operation in Step 4 is applied to the densely packed elements $\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle$ instead of $\langle T_1, T_2, T_4, T_3 \rangle$. This is done considering the latency of the `vpermq` instruction. Doing so, needs $\lceil \kappa/2 \rceil$ `vpermq` and $\lfloor \kappa/2 \rfloor$ `vpsrlq` instructions to produce the vector $\langle T_1, T_2, T_1, T_2 \rangle$. This is slightly advantageous compared to applying the DUP operation to $\langle T_1, T_2, T_4, T_3 \rangle$, which will need κ `vpermq` instructions.

7.4.1 Constant Time Conditional Swap

The conditional swap is performed over densely packed vector elements $\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle$. To perform the swapping in constant time we make use of the `vpermd` assembly instruction. First, a swapping index is created using the value of the present bit of the scalar and stored in a 256-bit `ymm` register. This index is then used by `vpermd` to swap the limbs of $\langle \underline{X_2}, \underline{Z_2} \rangle$ and $\langle \underline{X_3}, \underline{Z_3} \rangle$. The function DENSE-SWAP calls $\lceil \kappa/2 \rceil$ `vpermd` instructions to swap the field elements represented by the pairs $\langle \underline{X_2}, \underline{Z_2} \rangle$ and $\langle \underline{X_3}, \underline{Z_3} \rangle$.

7.4.2 Optimizing the Squaring in Ladder-step.

The instruction $\langle *, *, T_{15}, T_{16} \rangle \leftarrow$ SQR($\langle T_9, T_{10}, T_{11}, T_{12} \rangle$) in Step 14 of Algorithm 7.6 computes the four squarings $T_9^2, T_{10}^2, T_{11}^2$ and T_{12}^2 simultaneously. The squares T_9^2, T_{10}^2 are not needed by the algorithm and hence has been denoted as $*$ in the vector $\langle *, *, T_{15}, T_{16} \rangle$. Some crucial optimization can be done on this squaring operation. Considering the input $\langle T_9, T_{10}, T_{11}, T_{12} \rangle$ as a $\kappa \times 4$ matrix, of 64-bit integers, it can be considered that the first

two columns of the inputs are not required for computing the squares. This feature can be efficiently exploited while computing T_{11}^2 and T_{12}^2 via the last two columns of the input matrix while using the Karatsuba technique. The idea is to use the symmetry involved in the integer squarings of the subproblems while applying Karatsuba. We provide some details of the optimization technique that we have used for X448.

For convenience of notation, let us denote the vector $\langle T_9, T_{10}, T_{11}, T_{12}\rangle$ as $\mathbf{A} = \langle A_0, A_1, A_2, A_3\rangle$. So, we have $A_2 = T_{11}$ and $A_3 = T_{12}$. As defined before, considering $\mathbf{a}_i\theta^i = \langle a_{0,i}\theta^i, a_{1,i}\theta^i, a_{2,i}\theta^i, a_{3,i}\theta^i\rangle$, we can then write $\mathbf{A} = \sum_{i=0}^{15} \mathbf{a}_i\theta^i$. In the 16×4 matrix the values $a_{2,i}$ and $a_{3,i}$ are significant in the context and the values $a_{0,i}$ and $a_{1,i}$ can be ignored. The limbs $a_{2,8}, a_{2,9}, \ldots, a_{2,15}$ and $a_{3,8}, a_{3,9}, \ldots, a_{3,15}$ constitute the upper sub-problems of the field elements A_2 and A_3 respectively. We can copy these upper sub-problems to the first two columns of the matrix using 8 `vpermq` and 8 `vpblendd` instructions. Upon doing so, the entire limb information of the field elements A_2 and A_3 can be kept in a 8×4 matrix lying within $\mathbf{A} = \sum_{i=0}^{7} \mathbf{a}_i\theta^i$, where $\mathbf{a}_i\theta^i = \langle a_{2,8+i}\theta^i, a_{3,8+i}\theta^i, a_{2,i}\theta^i, a_{3,i}\theta^i\rangle$. Now, the integer squaring of the lower sub-problems and upper sub-problems of A_2 and A_3 can be done simultaneously using 36 `vpmuludq` instructions instead of 72. Along with this we also have a saving in reduced number of `vpaddq` instructions for accumulating the limb products.

We can also optimize the computation of the combined sub-problems using a similar technique. The combined sub-problems are denoted by the 8×4 matrix lying within $\mathbf{A} = \sum_{i=0}^{7} \mathbf{a}_i\theta^i$, where $\mathbf{a}_i\theta^i = \langle (a_{0,i} + a_{0,8+i})\theta^i, (a_{1,i} + a_{1,8+i})\theta^i, (a_{2,i} + a_{2,8+i})\theta^i, (a_{3,i} + a_{3,8+i})\theta^i\rangle$. As before, the values $(a_{2,i} + a_{2,8+i})$ and $(a_{3,i} + a_{3,8+i})$ are of interest and the values $(a_{0,i} + a_{0,8+i})$ and $(a_{1,i} + a_{1,8+i})$ can be ignored. In this situation we copy the values of the combined sub-problem in the order from bottom to top to the unused slots in the first two columns of the 8×4 matrix to get $\mathbf{A} = \sum_{i=0}^{7} \mathbf{a}_i\theta^i$, where $\mathbf{a}_i\theta^i = \langle (a_{2,7-i} + a_{2,15-i})\theta^i, (a_{3,7-i} + a_{3,15-i})\theta^i, (a_{2,i} + a_{2,8+i})\theta^i, (a_{3,i} + a_{3,8+i})\theta^i\rangle$. This is done using 8 `vpermq` and 8 `vpblendd` instructions. With such a setup, we can compute the integer squaring of the combined sub-problem using 20 `vpmuludq` instructions instead of 36. Here also we have an additional saving in reduced number of `vpaddq` instructions for accumulating the limb-products.

So, the integer squaring of A_2 and A_3 can be done using 56 `vpmuludq` instructions instead of 108. It is to be noted that the values of the accumulated limb-products has to brought back to the last two columns from the first two columns of the matrix for both the above cases to perform the linear operations needed for reduction. This is done by a total $(15 + 7) = 22$ `vpermq` instructions. So, the total number of `vpermq` instructions needed for achieving the speed-up due to the optimization is $(16 + 22) = 38$, whereas, the total number of `vpblendd` instructions needed is 16.

The above optimization technique can also be applied to the 9-limb implementation of X25519, but we did not find any benefit after applying it. The latency of the `vpermq` instruction plays a dominant role over here which neutralizes the benefit obtained due to the optimization. If the latency of `vpermq` gets minimized in future architectures, then applying the optimization strategy while computing the ladder-step for X25519 might produce some speed-up benefits.

7.4.3 Comparison of Algorithm 7.6 with the Vectorization of [HEY20]

The vectorization strategy given in Algorithm 7.4 has been derived from Figure 1 of [HEY20] and the corresponding implementation. This algorithm can be converted to

vectorized algorithm in the manner that Algorithm 7.1 has been converted to Algorithm 7.6. The trade-off between the two algorithms can be understood based on the following points.

Operation count: An operation level comparison between Algorithms 7.1 and 7.4 has been shown in Table 7.1. Both the algorithms require 2 $\mathcal{MUL}$ and 1 $\mathcal{SQR}$ operations. The trade-off in the operation counts is that Algorithm 7.4 does not require a $\mathcal{MULC}$ operation, but requires extra non-multiplication operations consisting of 1 $\mathcal{ADD}$, 1 $\mathcal{SUB}$, 2 $\mathcal{BLEND}$ and 3 $\mathcal{SHUFFLE}$ operations. The subtraction $\mathcal{SUB}$ is implemented by adding $2\mathfrak{p}$ to the minuend and then subtracting the subtrahend from the sum.

Conversions: Due to the extra non-multiplication operations in Algorithm 7.4, the number of conversions between normal and dense packings also increases.

Unreduced Hadamard: The outputs of the Hadamard operations of Algorithm 7.4 need to be reduced for the 9-limb implementation of X25519 and 16-limb implementation of X448. For the 10-limb implementation of X25519, the outputs of the Hadamard operations of Algorithm 7.4 can be kept unreduced. But, to afford this, the output of the ladder-step has to be reduced, as otherwise, the output of the first Hadamard operation of Algorithm 7.4 cannot be kept unreduced. For the 10-limb implementation of X25519, this reduction comes out to be extra in Algorithm 7.4, in comparison to Algorithm 7.1.

Optimizing the squaring step: As explained above, in Algorithm 7.1 there is the possibility of utilizing the free lanes in the squaring step to speed up the squaring operation. This has been seen to be advantageous for X448. There is no scope of applying such an optimization to a vectorized algorithm based on Algorithm 7.4. This is because the squaring step in Algorithm 7.4 simultaneously squares four elements and there are no free lanes.

The above discussion suggests that for Algorithm 7.4, the advantage of not having a single $\mathcal{MULC}$ operation is outweighed by the extra computations that need to be done. Timing results obtained from actual implementations support this observation.

7.4.4 Fixed Base Scalar Multiplication

Algorithm 7.7 shows the vectorized ladder-step for fixed base scalar multiplication. It is an optimized version of Algorithm 7.2. As before, the steps of Algorithm 7.7 can also be easily related to the various steps of Algorithm 7.2. It is to be noted that Step 16 in Algorithm 7.7 is MULC instead of MUL because X_1 is small. Hence the second parameter to MULC, which is $\langle 1, 1, 1, X_1 \rangle$ is a vector constant of 4 64-bit words.

7.4.5 Possible Optimization of the Multiplication Operations Using 512-bit `zmm` Registers

A similar kind of optimization discussed for squaring can be applied to the multiplications in Steps 6 and 16 of VECTORIZED-LADDER-STEP. For the fields where we apply the

Algorithm 7.7 Vectorized algorithm of Montgomery ladder-step corresponding to Algorithm 7.2.

function VECTORIZED-LADDER-STEP-FB($\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle, \langle 1, 1, 1, X_1 \rangle$)
  $\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle \leftarrow$ DENSE-H-H$_1$($\langle \underline{X_2}, \underline{Z_2}, \underline{X_3}, \underline{Z_3} \rangle$)
  $\langle T_1, T_2, T_4, T_3 \rangle \leftarrow$ PACK-D2N($\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle$)
  $\langle \underline{T_1}, \underline{T_2}, \underline{T_1}, \underline{T_2} \rangle \leftarrow$ DENSE-DUP($\langle \underline{T_1}, \underline{T_2}, \underline{T_4}, \underline{T_3} \rangle$)
  $\langle T_1, T_2, T_1, T_2 \rangle \leftarrow$ PACK-D2N($\langle \underline{T_1}, \underline{T_2}, \underline{T_1}, \underline{T_2} \rangle$)
  $\langle T_5, T_6, T_7, T_8 \rangle \leftarrow$ MUL($\langle T_1, T_2, T_4, T_3 \rangle, \langle T_1, T_2, T_1, T_2 \rangle$)
  $\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8} \rangle \leftarrow$ PACK-N2D($\langle T_5, T_6, T_7, T_8 \rangle$)
  $\langle \underline{T_9}, \underline{T_{10}}, \underline{T_{11}}, \underline{T_{12}} \rangle \leftarrow$ DENSE-H$_2$-H($\langle \underline{T_5}, \underline{T_6}, \underline{T_7}, \underline{T_8} \rangle$)
  $\langle T_9, T_{10}, T_{11}, T_{12} \rangle \leftarrow$ PACK-D2N($\langle \underline{T_9}, \underline{T_{10}}, \underline{T_{11}}, \underline{T_{12}} \rangle$)
  $\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0} \rangle \leftarrow$ UNREDUCED-MULC($\langle T_9, T_{10}, T_{11}, T_{12} \rangle, \langle 0^{64}, a24, 0^{64}, 0^{64} \rangle$)
  $\langle T_5, T_{14}, T_7, T_8 \rangle \leftarrow$ ADD($\langle \mathbf{0}, T_{13}, \mathbf{0}, \mathbf{0} \rangle, \langle T_5, T_6, T_7, T_8 \rangle$)
  $\langle T_5, T_{14}, T_{11}, T_{12} \rangle \leftarrow$ BLEND($\langle T_9, T_{10}, T_{11}, T_{12} \rangle, \langle T_5, T_{14}, T_7, T_8 \rangle, 1100$)
  $\langle T_{15}, T_{16}, T_{17}, T_{18} \rangle \leftarrow$ MUL($\langle T_5, T_{14}, T_{11}, T_{12} \rangle, \langle T_9, T_{10}, T_{11}, T_{12} \rangle$)
  $\langle X_2, Z_2, X_3, Z_3 \rangle \leftarrow$ MULC($\langle T_{15}, T_{16}, T_{17}, T_{18} \rangle, \langle 1, 1, 1, X_1 \rangle$)
  return $\langle X_2, Z_2, X_3, Z_3 \rangle$
end function.

Karatsuba technique for multiplication, the upper sub-problems can be copied to the upper half of the `zmm` registers using `vpermq` and `vpblendmd` instructions. Upon doing this, the integer multiplications for both the lower and upper sub-problems can be done simultaneously. Using the same technique, we can also avoid roughly 50% of the `vpmuludq` operations while computing the integer multiplication of the combined sub-problem. Also since there are a total of 32 registers of 512 bits, the present implementations can also be optimized for the load/store instructions to achieve higher speed.

7.5 Implementations and Timings

We have developed constant-time assembly implementations for the following targeting the modern Intel architectures.

Variable base scalar multiplication:

1. Implementations of Algorithm 7.6 have been made for both X25519 and X448.
2. Implementations of the vectorization strategy from [HEY20] given in Algorithm 7.4 have been made for both X25519 and X448.

For X25519, two implementations were done for both the above cases – one with 9-limb representation using (5+4)-Karatsuba for multiplication and schoolbook for squaring; the other one with 10-limb representation using schoolbook method. In [HEY20], a field element is represented as $A = a_0 + a_1 2^{85} + a_2 2^{170}$, such that each a_i is represented by 3 limbs of sizes 29, 28 and 28 bits. Multiplication and squaring have been done using a Karatsuba strategy based on a (3+3+3)-decomposition.

For 9-limb representation, we have found that the (5+4)-Karatsuba strategy described above turns out to be more efficient than the method described in [HEY20]. For Skylake, the CPU-cycles for field multiplication and squaring, using (5+4)-Karatsuba are 112 and 91, whereas, the CPU-cycles of field multiplication and squaring from [HEY20] are 119 and 96 respectively. So, we did not make any implementation using the representation suggested in [HEY20].

Fixed base scalar multiplication: Implementations of Algorithm 7.7 have been made for both X25519 and X448.

In [HEY20], a field element is represented as $A = a_0 + a_1 2^{85} + a_2 2^{170}$, such that each a_i is represented by 3 limbs of sizes 29, 28 and 28 bits. Multiplication and squaring have been done using a Karatsuba strategy based on a (3+3+3)-decomposition. For 9-limb representation, we have found that the (5+4)-Karatsuba strategy described above turns out to be more efficient than the method described in [HEY20]. For Skylake, the CPU-cycles for field multiplication and squaring, using (5+4)-Karatsuba are 112 and 91, whereas, the CPU-cycles of field multiplication and squaring from [HEY20] are 119 and 96 respectively.

For comparison, we also obtained the numbers of CPU-cycles required by the implementations corresponding to previous works [FHLD19, HEY20, OLH+17]. The work [FHLD19] uses `AVX2` instructions to implement a 2-way SIMD algorithm. The implementations corresponding to [OLH+17] do not use SIMD instructions; they use 64-bit arithmetic based on the instructions `mulx, adcx, adox` for Skylake, and the instructions `mulx, add, adc` for Haswell. The implementations in [HEY20] implement a 4-way SIMD algorithm using `AVX2` instructions. We have found the timings of the 9-limb and 10-limb implementations of [HEY20] as 104519 and 124077 CPU-cycles respectively on a Skylake i7-6500U machine, which has been reported as 98484 and 116654 respectively in [HEY20] on a Skylake i9-7900X machine. The difference in the timings is due to the difference in the CPU architectures of the two Skylake machines. Similarly, we note that the timings reported in [FHLD19] and [OLH+17] are lower than those given in Table 7.2 and these differences are also attributable to the differences in the actual processors.

The work [HEY20] mentions that the inversion code that is used in their implementations is the `maax`-type implementation for p255-19 from Chapter 4. This inversion code is for Skylake and does not run on Haswell. To obtain performance results for the code of [HEY20] on Haswell, we replaced the inversion code with the inversion code for Haswell which is the `maa`-type implementation for p255-19 from Chapter 4.

The numbers of CPU-cycles required by X25519 and X448 for the shared secret computation phase of the ECDH protocol are given in Table 7.2. The number given in the gray cells of the table are the best speeds for X25519 and X448.

The first point to note from Table 7.2 is that as expected, 4-way vectorization using `AVX2` provides faster speed than `maax` or 2-way vectorization using `AVX2`. So, the comparison is between the vectorization strategy in [HEY20] and the strategy proposed in the present work.

One may note the following points from Table 7.2.

- On Haswell, the best performance of X25519 is obtained using a 9-limb representation and (5+4)-Karatsuba for multiplication, schoolbook for squaring while on Skylake, the best performance is obtained using a 10-limb representation and schoolbook multiplication.

Operation	Haswell	Skylake	κ	Strategy	Reference	Implementation Type
X25519	143956	118231	4	64-bit seq, [OLH+17]	[OLH+17]	`mxaa,maax`, assembly
	146363	128202	10	2-way SIMD, [FHLD19]	[FHLD19]	`AVX2`, intrinsics
	140996	104519	9	4-way SIMD, [HEY20]	[HEY20]	`AVX2`, intrinsics
	174129	124077	10	4-way SIMD, [HEY20]	[HEY20]	`AVX2`, intrinsics
	121539	99898	9	4-way SIMD, [HEY20]	this work	`AVX2`, assembly
	126521	97590	10	4-way SIMD, [HEY20]	this work	`AVX2`, assembly
	120108	99194	9	4-way SIMD, Alg. 7.6	this work	`AVX2`, assembly
	123899	95437	10	4-way SIMD, Alg. 7.6	this work	`AVX2`, assembly
X448	720698	536362	7	64-bit seq, [OLH+17]	[OLH+17]	`mxaa,maax`, assembly
	518467	421211	16	2-way SIMD, [FHLD19]	[FHLD19]	`AVX2`, intrinsics
	462277	373006	16	4-way SIMD, [HEY20]	this work	`AVX2`, assembly
	441715	357095	16	4-way SIMD, Alg. 7.6	this work	`AVX2`, assembly

Table 7.2: CPU-cycle counts on Haswell and Skylake processors required by X25519 and X448 for variable base scalar multiplication.

- On both Haswell and Skylake, the 10-limb implementation of X25519 using Algorithm 7.6 is noticeably better than the implementation using the vectorization strategy in [HEY20]. This is mainly due to the extra reduction needed at the end of the ladder-step of [HEY20], which can be avoided in Algorithm 7.6.

- The implementation in [HEY20] is slower than our implementation of the vectorization in [HEY20]. One reason for this is a change from intrinsic to assembly. For the implementation using 9-limb representation, a reason for the speed improvement is our use of (5+4)-Karatsuba in comparison to the (3+3+3)-Karatsuba used in [HEY20].

- On both Haswell and Skylake, the 16-limb implementation of X448 using Algorithm 7.6 is noticeably better than the implementation using the vectorization strategy in [HEY20]. This is mainly due to the benefit earned by optimizing the squaring operation of Algorithm 7.6, which is not possible while using the vectorization strategy of [HEY20].

Overall, from the timing information provided in Table 7.2 we see that for both X25519 and X448, the vectorization given by Algorithm 7.6 provides better performance compared to all previous works on both Haswell and Skylake. Compared to [FHLD19, OLH+17], major speed improvements are obtained by Algorithm 7.6. On the other hand, the performance differences of Algorithm 7.6 to our optimized assembly implementation of the vectorization strategy of [HEY20] are modest but nonetheless noticeable. Given that both X25519 and X448 are part of TLS version 1.3 and are likely to extensively used, a noticeable speed improvement is of practical interest. So, for practical deployment, in comparison to previous algorithms, it is preferable to use Algorithm 7.6 for implementing variable base scalar multiplication on Curve25519 and Curve448.

The speed-ups achieved by our assembly implementations of variable base scalar multiplication using Algorithm 7.6 over the existing vectorized implementations of X255-19 and X448 are given below.

1. In Skylake, our 9-limb implementation of X25519 is 22.6% faster than the 10-limb implementation of [FHLD19], 20.1% faster than the 10-limb implementation of [HEY20] and 5.1% faster than the 9-limb implementation of [HEY20]. In Haswell the corresponding improvements are 18%, 31% and 14.8% respectively.

2. In Skylake, our 10-limb implementation of X25519 is 25.6% faster than the 10-limb implementation of [FHLD19], 8.9% faster than the 10-limb implementation of [HEY20] and 23% faster than the 9-limb implementation of [HEY20]. In Haswell the corresponding improvements are 15.3%, 28.8% and 12.1% respectively.

3. In Skylake, our 16-limb implementation of X448 is 15.2% faster than the 16-limb implementation of [FHLD19] and in Haswell the corresponding improvement is 14.8%.

Operation	Haswell	Skylake	κ	Strategy	Reference	Implementation Type
X25519	100127	86885	9	4-way SIMD, Alg. 7.7	this work	`AVX2`, assembly
	106190	84047	10	4-way SIMD, Alg. 7.7	this work	`AVX2`, assembly
X448	381417	317778	16	4-way SIMD, Alg. 7.7	this work	`AVX2`, assembly

Table 7.3: CPU-cycle counts on Haswell and Skylake processors required by X25519 and X448 for fixed base scalar multiplication.

For fixed base scalar multiplication, neither Algorithm 7.6 nor the vectorization strategy in [HEY20] can take advantage of the fact that X_1 is small. The previous 4-way vectorized algorithm from [CS09] can indeed take advantage of this point. As discussed in Section 7.3, Algorithm 7.2 is faster than the algorithm in [CS09]. So, we did not implement the vectorized algorithm from [CS09]. Algorithm 7.7 is the detailed vectorized algorithm corresponding to Algorithm 7.2. Timing results for Algorithm 7.7 are given in Table 7.3. It may be noted that compared to the best timings for variable base scalar multiplication given in Table 7.2, there is a speed improvement of about 12% to 15% for both X25519 and X448.

Reasons for improvement. The reported performance improvement can be attributed to three factors, namely, vectorization strategy, algorithm design and code optimization. For the improvement in vectorization strategy, we refer to Table 7.1 and the detailed discussion based on this table in Section 7.3. For certain operations, careful algorithm design has been helpful. Examples are the proper choices of the numbers of limbs to represent the elements of the field and the issue of availing free lanes to speed up squaring for $2^{448} - 2^{224} - 1$. Finally, we have developed hand-written assembly. Due to this we had complete control over the available 4-way SIMD registers and was able to appropriately schedule the instructions so as to minimize the load/store instructions as much as possible.

7.6 Conclusion

In this chapter we have described new and efficient vectorization strategies for implementing the Montgomery ladder for both variable base and fixed base scalar multiplications. Constant time assembly implementations have been made for Curve25519 and Curve448. For fixed base scalar multiplication, the new algorithm is shown to be clearly faster than previous work. Timing results on the Haswell and the Skylake processors show that for variable base scalar multiplication, the new vectorization strategy provides speed improvements over all previous implementations of the Montgomery ladder.

Part II

New Curves and Security/Efficiency Trade-off

CHAPTER 8

Security and Efficiency Trade-off of ECDH over Prime Order Fields

8.1 Introduction

One of the most extensively used modern cryptographic primitives is the Diffie-Hellman (DH) [DH76] key agreement protocol. Koblitz [Kob87] and Miller [Mil85] have independently shown that the DH protocol can be instantiated using cyclic groups arising from the theory of elliptic curves. Since their introduction, a large body of literature has developed around the theory and application of elliptic curves in cryptography. Presently, elliptic curve cryptography is widely used in practical systems. Several standards and proposals have been put forward by a number of influential organizations [Braa, Cur99, Res10, NUM]. Among the various models of elliptic curves, the Montgomery form [Mon87] provides the most efficient model for implementing DH key agreement over prime order fields. The famous and widely deployed curve known as Curve25519 [Ber06b] is a Montgomery form curve.

The Transport Layer Security (TLS) Protocol, Version 1.3 [TP18] has been proposed by the Internet Engineering Task Force. This includes RFC 7748 [LH16] which specifies two elliptic curves, namely Curve25519 and Curve448, for DH key agreement. As part of the Transport Layer Security (TLS) protocol, Version 1.3 [TP18], RFC 7748 [LH16] specifies Curve25519 provides security at the 128-bit security level and Curve448 provides security at the 224-bit security level. The document specifies Montgomery form curves and their birationally equivalent twisted Edwards form curves.

We use the notation for primes introduced in Chapter 6. For the different curves we use the following notation. A Montgomery curve $E_{M,A,1}$ will be denoted as $\mathrm{M}[\![A]\!]$, an Edwards curve $E_{E,1,d}$ will be denoted as $\mathrm{E}[\![d]\!]$, and a twisted Edwards curve $\mathrm{E}_{E,-1,d}$ will be denoted as $\widetilde{\mathrm{E}}[\![d]\!]$. If we wish to emphasize the underlying field $\mathbb{F}_p$, we will write $\mathrm{M}[\![p, A]\!]$, $\mathrm{E}[\![p, d]\!]$ and $\widetilde{\mathrm{E}}[\![p, d]\!]$ instead of $\mathrm{M}[\![A]\!]$, $\mathrm{E}[\![d]\!]$ and $\widetilde{\mathrm{E}}[\![d]\!]$ respectively. In terms of this naming convention, the parameters of the new curves and those in RFC 7748 are shown in Table 8.1.

We work at the 128-bit, 224-bit and 256-bit security levels. At 128-bit we work with the primes p251-9, p255-19, at 224-bit security level we work with p444-17, p448-224-1 and at the 256-bit security level we work with the primes p506-45, p506-75, p521-1.

8.2 Curves Proposed in RFC 7748

Over p255-19: The birationally equivalent pair $(\mathrm{M}[\![486662]\!], \widetilde{\mathrm{E}}[\![121665/121666]\!])$ has been proposed. The curve $\mathrm{M}[\![486662]\!]$ is the famous Curve25519 and was introduced in [Ber06b]. The curve $\widetilde{\mathrm{E}}[\![121665/121666]\!]$ is the famous Ed25519 curve and was introduced in [BDL+12].

Over p448-224-1: The curves $\mathrm{M}[\![156326]\!]$, $\mathrm{E}[\![39082/39081]\!]$ and $\mathrm{E}[\![39081]\!]$ have been proposed. The curve $\mathrm{M}[\![156326]\!]$ has been named Curve448 in [LH16]. The curve $\mathrm{E}[\![39081]\!]$ was proposed in [Ham15] where it was named Ed448-Goldilocks, and in [LH16] it has been called Edwards448. The isogenies between $\mathrm{M}[\![156326]\!]$ and $\mathrm{E}[\![39081]\!]$ and the birational equivalence between $\mathrm{M}[\![156326]\!]$ and $\mathrm{E}[\![39082/39081]\!]$ have been identified in [LH16].

Curve25519 and Ed25519 are targeted at the 128-bit security level while Curve448 and Edwards448 are targeted at the 224-bit security level.

8.3 Our Contributions

One of our contribution is to provide new and the presently most efficient 64-bit implementations of Diffie-Hellman shared secret computations using Curve25519. These implementations are targeted at Intel processors. Compared to the previous best implementations [OLH+17], on the Skylake processor, the new implementations provide about 17% speed-up compared to [OLH+17]; for the previous generation Haswell processor, the improvement is nominal. The major speed improvement in Skylake makes the new implementation attractive for practical applications.

The main contribution of the paper is to propose new curves at the 128-bit, 224-bit and 256-bit security levels. Similar to Curve25519 and Curve448, we consider prime order fields and pairs of birationally equivalent curves. In particular the following pairs of curves are introduced.

Over p251-9*:* $(\mathrm{M}[\![4698]\!], \mathrm{E}[\![1175/1174]\!])$.

Over p444-17*:* $(\mathrm{M}[\![4058]\!], \mathrm{E}[\![1015/1014]\!])$.

Over p506-45*:* $(\mathrm{M}[\![996558]\!], \mathrm{E}[\![249140/249139]\!])$.

Over p510-75*:* $(\mathrm{M}[\![952902]\!], \mathrm{E}[\![-238225/238226]\!])$.

Over p521-1*:* $(\mathrm{M}[\![1504058]\!], \mathrm{E}[\![376015/376014]\!])$.

The prime p251-9 was considered in [BHKL13] in which the curve $u^2 + v^2 = 1 - 1174u^2v^2$ was introduced and named Curve1174. The Montgomery curve $(4/1175)y^2 = x^3 + (4/1175 - 2)x^2 + x$ with base point $(4, \cdot)$ was considered as birationally equivalent to Curve1174; the corresponding base point on Curve1174 is $(\cdot, 3/5)$. Using the isogenies given in [CN15], it can be shown that $\mathrm{M}[\![p251\text{-}9, 4698]\!]$ is 4-isogenous to Curve1174 which was introduced in [BHKL13].

To the best of our knowledge, neither $\mathrm{M}[\![p251\text{-}9, 4698]\!]$ nor $\mathrm{E}[\![p251\text{-}9, 1175/1174]\!]$ has been earlier considered in the literature. The prime p444-17 has been mentioned

Security Level	Prime	Mont	(h, h_T)	(k, k_T)	Security	Mont Base Pt	Ed	Ed Base Pt
128-bit	p251-9	M$[\![4698]\!]$	$(4,4)$	$(\ell-1, \frac{\ell_T-1}{2})$	124.5	$(3,\cdot)$	E$[\![\frac{1175}{1174}]\!]$	$(\cdot,2)$
	p255-19	M$[\![486662]\!]$	$(8,4)$	$(\frac{\ell-1}{6}, \frac{\ell_T-1}{2})$	126	$(9,\cdot)$	$\widetilde{\text{E}}[\![\frac{121665}{121666}]\!]$	$(\cdot,4/5)$
224-bit	p444-17	M$[\![4058]\!]$	$(4,4)$	$(\frac{\ell-1}{3}, \ell_T-1)$	221	$(3,\cdot)$	E$[\![\frac{1015}{1014}]\!]$	$(\cdot,2)$
	p448-224-1	M$[\![156326]\!]$	$(4,4)$	$(\frac{\ell-1}{2}, \frac{\ell_T-1}{4})$	223	$(5,\cdot)$	E$[\![\frac{39082}{39081}]\!]$	$(\cdot,-3/2)$
256-bit	p506-45	M$[\![996558]\!]$	$(4,4)$	$(\frac{\ell-1}{17}, \ell_T-1)$	252	$(3,\cdot)$	E$[\![\frac{249140}{249139}]\!]$	$(\cdot,2)$
	p510-75	M$[\![952902]\!]$	$(8,4)$	$(\frac{\ell-1}{17}, \ell_T-1)$	253.5	$(4,\cdot)$	$\widetilde{\text{E}}[\![\frac{-238225}{238226}]\!]$	$(\cdot,3/5)$
	p521-1	M$[\![1504058]\!]$	$(4,4)$	$(\frac{\ell-1}{2}, \frac{\ell_T-1}{4})$	259.5	$(8,\cdot)$	E$[\![\frac{376015}{376014}]\!]$	$(\cdot,9/7)$

Table 8.1: Parameters of the curves. In the table, $M[\![486662]\!]$ is Curve25519 and $M[\![156326]\!]$ is Curve448. See Section 2.2.1 for the definition of the parameters.

in a CFRG mailing list [ma], but neither the curve M$[\![p444\text{-}17, 4058]\!]$ nor E$[\![p444\text{-}17, 1015/1014]\!]$ have been considered in the literature.

Table 8.1 compares the parameters of the newly proposed curves with those in RFC-7748. Note that the curve coefficients of the new curves are quite small. Also, the fixed base points for the new Montgomery and Edwards curve are also very small. In fact, the fixed base point over three new Edwards curves is $(\cdot, 2)$. As we explain later, this has an effect on the speed of fixed base point scalar multiplication over such curves. From Table 8.1, we observe that M$[\![p506\text{-}45, 996558]\!]$, M$[\![p510\text{-}75, 952902]\!]$ and M$[\![p521\text{-}1, 1504058]\!]$ provide 29, 30.5 and 36.5 bits more security compared to M$[\![p\text{-}448\text{-}224\text{-}1, 156326]\!]$ (i.e., Curve448).

Curves at 128-bit and 224-bit security levels. For 64-bit implementations of the Montgomery ladder, scalar multiplication over the new curves at 128-bit and 224-bit security levels is faster than that over Curve25519 and Curve448. The improvement arises due to working with a slightly smaller prime. Suppose $m = \lceil \log_2 p \rceil$ and elements of $\mathbb{F}_p$ are represented using κ 64-bit words. We show that if $64\kappa - m \geq 3$, then it is possible to omit performing the reduction step on the outputs of all the addition/subtraction operations in the ladder step. This is the major reason for obtaining faster ladder computation modulo p251-9 compared to p255-19 and for obtaining faster ladder computation modulo p444-17 compared to p448-224-1. Using 64-bit implementations, for both the 128-bit and 224-bit security levels, compared to Curve25519 and Curve448, the new curves provide about 10% speed-up on the Haswell and the Skylake processors.

Recently, a 4-way vectorization strategy of the Montgomery ladder has been proposed in [HEY20]. We have provided an improved 4-way vectorized algorithm of the Montgomery ladder in Chapter 7. We have provided several vectorized implementations for the new curves. At the 128-bit level, M$[\![4698]\!]$ provides about 4% speed-up over Curve25519. The situation for the 224-bit level is different. For 4-way vectorized implementation using 256-bit registers, the representation of field elements has 16 limbs. This makes Karatsuba multiplication more efficient than schoolbook multiplication. Since p448-224-1 is particularly efficient for Karatsuba multiplication, the vectorized implementation of Curve448 turns out to be faster than that of M$[\![4058]\!]$. We note that future availability of wider vector operations would reduce the number of limbs and would possibly lead to schoolbook becoming faster than Karatsuba and consequently, M$[\![4058]\!]$

being faster than Curve448. So overall, the new curves provide efficient alternatives to Curve25519 and Curve448.

Motivation for the new curves at 128-bit and 224-bit security levels. The adoption of Curve-25519 and Curve448 as part of TLS version 1.3 will lead to widespread implementation and deployment of these curves. Nevertheless, the scientific spirit of looking for alternative curves endures. It is in this spirit that the present work has been undertaken. The new curves that have been introduced here provide an engineering trade-off of improved speed at the cost of about two bits of security. This leads to an expansion of the portfolio of secure curves that may be used in practice. In our opinion, the question of whether the improved speed is worth the two bits loss in security is best left to the judgment and discretion of the users and standardization bodies. Standard are not rigid documents and can expand over time. One such example is the addition of Ed25519 and Ed448 to the signature standards of NIST of USA[1]. Further, international standardization bodies (such as IEEE or ISO) other than the IETF as well as standardization bodies of various nations may consider all available options when determining their respective standards. We are hopeful that future work by researchers will further highlight the attractiveness of the new curves which may lead standardization bodies to expand existing standards to include these curves, or consider these curves for inclusion in future standards.

Curves at 256-bit security level. The prime $p521\text{-}1$ has been considered earlier in the work [ABGR13] which introduced the curve $\mathrm{E}[\![p521\text{-}1, -376014]\!]$ (and named it E-521) as part of a suite of general purpose high security elliptic curves. This prime provides a few bits more security than our target 256-bit security level. So, we considered some pseudo-Mersenne primes which are less than 2^{512}. For efficiency reasons, we wished to have δ small. Due to this reason, we chose not to work with the prime $2^{511} - 187$ suggested in [ABGR13]. We found two other pseudo-Mersenne primes less than 2^{512} which may be considered for 256-bit security. These are $p506\text{-}45$ and $p510\text{-}75$. Using the isogenies given in [CN15], it can be shown that E-521 is 4-isogenous to $\mathrm{M}[\![1504058]\!]$ shown in Table 8.1. To the best of our knowledge, neither of the curves $\mathrm{M}[\![1504058]\!]$ or $\mathrm{E}[\![376015/376014]\!]$ appear earlier in the literature. Also, to the best of our knowledge, the primes $p506\text{-}45$ and $p510\text{-}75$ have not been considered earlier in the literature and so the question of proposing curves over the corresponding fields do not arise.

The curve $\mathrm{E}[\![p506\text{-}45, 249140/249139]\!]$ which is birationally equivalent to the curve $\mathrm{M}[\![p506\text{-}45, 996558]\!]$ can be used for the key generation phase. The small base point on the curve $\mathrm{E}[\![p506\text{-}45, 249140/249139]\!]$ is helpful for fixed base scalar multiplication. Also, the curve $\mathrm{E}[\![p506\text{-}45, 249140/249139]\!]$ can be used to implement a signature scheme following the approach used for EdDSA [BDL+12]. Similarly, if $\mathrm{M}[\![p510\text{-}75, 952902]\!]$ is used for shared secret computation of the DH protocol, then $\widetilde{\mathrm{E}}[\![p510\text{-}75, -238225/238226]\!]$ which is birationally equivalent to $\mathrm{M}[\![p510\text{-}75, 952902]\!]$ can be used for key generation and also for instantiation of a signature scheme following [BDL+12].

We have also made sequential and vectorized implementations of the ladder for all the newly proposed Montgomery curves at the 256-bit security level. If security around

[1] https://csrc.nist.gov/CSRC/media/Presentations/NIST-Status-Update-on-Elliptic-Curves-and-Post-Qua/images-media/moody-dustin-threshold-crypto-workshop-March-2019.pdf.

this level is desired, the curves M⟦p506-45, 996558⟧, M⟦p510-75, 952902⟧ or M⟦p521-1, 1504058⟧ provide us with an idea of the trade-off between speed and security in comparison to the standard curve at the 224-bit security level, Curve448. A proper study of this idea have not appeared earlier in literature. We have made the source code of all our implementations publicly available at the links

`https://github.com/kn-cs/nice-curves`,
`https://github.com/kn-cs/mont256-dh`,
`https://github.com/kn-cs/mont256-vec`.

8.3.1 Related Work

In this work, we consider elliptic curves over large prime order fields. We note that elliptic curves over composite order fields have been proposed in the literature [CL15, HKM09]. Cryptography over hyper-elliptic curves was proposed by Koblitz [HKM09] and there have been concrete proposals for cryptography in genus 2 [BCLS14, BCHL16, GS12]. For the same security level, computations over these proposals are faster than over genus one prime order field curves. On the other hand, the security perception for composite order fields and genus two curves is different from that of elliptic curves over prime order fields. It is perhaps due to this perception issue that elliptic curves over prime order fields remain to be of primary interest.

Variable base scalar multiplication over Kummer lines associated with Legendre form elliptic curves have been proposed in the literature [GL09]. These have very efficient vectorized implementations [KS20]. So, if applications are targeted primarily for vector implementations, then the curves proposed in [KS20] will be the primary choice. On the other hand, for non-vectorized implementations, Montgomery curves will be faster.

8.4 Montgomery and (Twisted) Edwards Form Elliptic Curves

Let p be a prime and $\mathbb{F}_p$ be the finite field of p elements. Following TLS, Version 1.3, we consider elliptic curves over $\mathbb{F}_p$, where p is a large prime. Montgomery curve $E_{M,A,B}$ and twisted Edwards curve $E_{E,a,d}$ have already been defined. In our applications, we will have $B = 1$ and a to be either 1 or -1. If $a = 1$, then the corresponding curve is simply called an Edwards form curve (instead of twisted Edwards form curve). If a is a square and d is not a square in $\mathbb{F}_p$, then the addition formula in $E_{E,a,d}$ is complete [BBJ+08]. In this case, $E_{E,a,d}$ is called a complete twisted Edwards curve. Further, if $a = -1$, then particularly efficient addition formulas are known [HWCD08].

If $p \equiv 1 \bmod 4$, -1 is a square modulo p. In this case, if d is a non-square, the addition formula over $E_{E,-1,d}$ is both complete and the fastest. On the other hand, if $p \equiv 3 \bmod 4$, -1 is a non-square modulo p and so the addition formula over $E_{E,-1,d}$ is not guaranteed to be complete. In this case, one considers the Edwards curve $E_{E,1,d}$ with d a non-square so that the addition formula is complete. It is not, however, the fastest. If the base point on $E_{E,1,d}$ is small, then the difference in the number of operations between the addition formulas on $E_{E,-1,d}$ and $E_{E,1,d}$ is small. More concretely, if the base point on $E_{E,1,d}$ is $(\cdot, 2)$, then this difference is just two left shifts.

For $p \equiv 3 \bmod 4$, addition formula over $E_{E,-1,d}$ is not guaranteed to be complete making constant time implementation of scalar multiplication problematic. On the other

hand, for the verification phase of a signature scheme based on the EdDSA template [BDL+12], constant time implementation is not an issue. For this application, one may move from $E_{E,1,d}$ to $E_{E,-1,d'}$, for some d' (see below) using a birational equivalence and perform the main computation of signature verification over $E_{E,-1,d'}$.

We refer to [Mon87, BL17, CS18] for background theory and further details about Montgomery form curves. For (twisted) Edwards curves, we refer to [Edw07, BL07a, BBJ+08].

In the following discussion, a full field multiplication (resp. squaring) in $\mathbb{F}_p$ will be denoted as [M] (resp. [S]); if one of the multiplicands is a constant, the resulting multiplication will be denoted as [C].

8.4.1 Addition on Complete (Twisted) Edwards Curves

Following [HWCD08], the extended affine coordinate system is (u, v, t) with $t = uv$. The projective version of this coordinate system is (U, V, T, W) where $u = U/W$, $v = V/W$ and $t = T/W$. Suppose, it is required to add $(U_1 : V_1 : T_1 : W_1)$ and $(U_2 : V_2 : T_2 : W_2)$ to obtain $(U_3 : V_3 : T_3 : W_3)$. The formulas for U_3, V_3, T_3 and W_3 are as follows [HWCD08].

$$\left.\begin{array}{rcl} U_3 &=& (U_1V_2 + V_1U_2)(W_1W_2 - dT_1T_2) \\ V_3 &=& (V_1V_2 - aU_1U_2)(W_1W_2 + dT_1T_2) \\ T_3 &=& (U_1V_2 + V_1U_2)(V_1V_2 - aU_1U_2) \\ W_3 &=& (W_1W_2 + dT_1T_2)(W_1W_2 - dT_1T_2). \end{array}\right\} \tag{8.1}$$

1. Computing V_1V_2 and U_1U_2 and then computing $U_1V_2 + V_1U_2$ as $(U_1 + V_1)(U_2 + V_2) - (U_1U_2 + V_1V_2)$ leads to an algorithm for computing U_3, V_3, T_3 and W_3 using 9[M]+2[C] operations, where the multiplications by the two constants are by a and d. If $a = 1$, then the number of operations is 9[M]+1[C].

2. If $a = -1$, then by first computing $\alpha = (V_1 + U_1)(V_2 + U_2)$, $\beta = (V_1 - U_1)(V_2 - U_2)$ and then computing $2(V_1V_2 + U_1U_2) = \alpha + \beta$ and $2(V_1U_2 + U_1V_2) = \alpha - \beta$, the number of operations can be brought down to 8[M]+1[C] [HWCD08], where 1[C] corresponds to a multiplication by d. The relevant formula becomes the following.

$$\left.\begin{array}{rcl} 4U_3 &=& 2(U_1V_2 + V_1U_2)(2W_1W_2 - 2dT_1T_2) = (\alpha - \beta)(2W_1W_2 - 2dT_1T_2) \\ 4V_3 &=& 2(V_1V_2 + U_1U_2)(2W_1W_2 + 2dT_1T_2) = (\alpha + \beta)(2W_1W_2 + 2dT_1T_2) \\ 4T_3 &=& 2(U_1V_2 + V_1U_2)2(V_1V_2 + U_1U_2) = (\alpha - \beta)(\alpha + \beta) \\ 4W_3 &=& (2W_1W_2 + 2dT_1T_2)(2W_1W_2 - 2dT_1T_2). \end{array}\right\} \tag{8.2}$$

   If $W_1 = 1$, the number of operations required is 7[M]+1[C] [HWCD08].

3. For $a = -1$, suppose $(U_1 : V_1 : T_1 : W_1)$ is a fixed base point with $W_1 = 1$. By pre-computing and storing $(V_1 - U_1, V_1 + U_1, 2dT_1)$ the number of operations can be brought down to 7[M] [BDL+12]. The multiplication by d becomes part of the pre-computed quantity $2dT_1$. In this formula, since the multiplication by d is part of the pre-computed quantity $2dT_1$, the efficiency of the computation is not affected by whether d is small or large. Also, the efficiency of the computation is not affected by whether V_1 (or U_1) is small or large.

Consider (8.1) for $a = 1$ and suppose $(U_1 : V_1 : T_1 : W_1)$ is a fixed base point where $W_1 = 1$. Further suppose that V_1 is small and $U_1 + V_1$ and dT_1 are pre-computed and

stored as part of $(U_1, V_1, U_1 + V_1, dT_1)$. In (8.1), by directly computing U_1V_2, V_1U_2, U_1U_2 and V_1V_2, $(dT_1)T_2$ along with the other four multiplications, the formulas in (8.1) can be computed using 7[M]+2[C], where 2[C] counts the multiplications V_1U_2 and V_1V_2. The efficiency of the computation following this strategy is not affected by whether d is small or large. For the curves that we introduce, V_1 is equal to 2 as can be seen from Table 8.1. So, for fixed base multiplication, the difference in the cost between $a = -1$ and $a = 1$ is essentially two multiplications by very small constants.

For dedicated (not unified) addition in $\widetilde{\mathrm{E}}[\![p, d]\!]$, it has been shown in [HWCD08] that 8[M] operations are sufficient without the assumption that $(U_1 : V_1 : T_1 : W_1)$ is a fixed base point. The corresponding formulas do not involve d. Further, Section 4.3 of [HWCD08] shows how to perform efficient scalar multiplication using fast formulas for dedicated addition and dedicated doubling that do not involve d. The resulting scalar multiplication is not necessarily constant time and can be used only when the scalars are not secret.

Summary:

Role of d: For the fastest formulas, the size of d does not play a role.

- For fixed base point scalar multiplication, the fastest complete addition formulas over both $\mathrm{E}[\![d]\!]$ and $\widetilde{\mathrm{E}}[\![d]\!]$ do not depend on the size of d.
- For scalar multiplication with non-secret scalars, the fastest formulas do not involve d.

Size of fixed base point:

- For $\widetilde{\mathrm{E}}[\![p, d]\!]$, the fastest formula for complete and unified addition does not depend on the size of any of the components of the fixed base point. The number of operations required is 7[M].
- For $\mathrm{E}[\![p, d]\!]$, the fastest formula for complete and unified addition is achieved when V_1 is small. The number of operations required is 7[M]+2[C], where 2[C] counts two multiplications by very small constants. In particular, for both $\mathrm{E}[\![p251\text{-}9, \frac{1175}{1174}]\!]$ and $\mathrm{E}[\![p444\text{-}17, \frac{1015}{1014}]\!]$, $(\cdot, 2)$ is a base point. So, the multiplication by constant is the operation of multiplying an element of $\mathbb{F}_p$ by 2.

8.4.2 Birational Equivalences of Montgomery and Edwards Curves

RFC7748 [LH16] of TLS, Version 1.3 specifies both Montgomery and Edwards form curves for a given security level. In the present state of knowledge, the shared secret computation of the DH key agreement is performed best on a Montgomery form curve. On the other hand, the key generation phase as well as the computations required for an elliptic curve signature scheme based on the template in [BDL+12] are performed best on an Edwards form curve.

Edwards and Montgomery curves can be connected by either birational equivalences or by isogenies. For example, for the 128-bit security level, Curve25519 and Ed25519 are birationally equivalent. Similarly, at the 224-bit security level, Curve448 (i.e., $\mathrm{M}[\![p448\text{-}$

224-1, 156326⟧) and E⟦p448-224-1, 39082/39081⟧ are birationally equivalent. Additionally, Curve448 is 4-isogenous to E⟦p448-224-1, -39081⟧ [LH16]. The curve E⟦p448-224-1, -39081⟧ was proposed in [Ham15] where it was named Ed448-Goldilocks and it has been called Edwards448 in [LH16].

We provide below some explicit birational equivalences between Montgomery and Edwards form curves. These can be obtained by composing the elementary birational equivalences provided in [BL07a, BBJ+08]. The verification of these birational equivalences, on the other hand, can be done by direct substitution.

Case $p \equiv 3 \bmod 4$**.** Let $E_{M,A,B} : By^2 = x^3 + Ax^2 + x$ be a Montgomery curve and $E_{E,1,d} : u^2 + v^2 = 1 + du^2v^2$ be an Edwards curve over $\mathbb{F}_p$. Note that -1 is not a square in $\mathbb{F}_p$.

1. If $(A+2)/B$ is a square in $\mathbb{F}_p$, then the map

$$(x, y) \mapsto (u, v) = (\delta x/y, (x-1)/(x+1)), \tag{8.3}$$

   where $\delta^2 = (A+2)/B$, is a birational equivalence from $E_{M,A,B}$ to $E_{E,1,d}$ with exceptional points $y = 0$ and $x = -1$. Conversely, the map

$$(u, v) \mapsto (x, y) = ((1+v)/(1-v), \delta(1+v)/(u(1-v))), \tag{8.4}$$

   is a birational equivalence from $E_{E,1,d}$ to $E_{M,A,B}$ with exceptional points $u = 0$ and $v = 1$. The relation between A and d is $(A+2)/4 = 1/(1-d)$.

2. If $(A-2)/B$ is a square in $\mathbb{F}_p$, then the map

$$(x, y) \mapsto (u, v) = (\delta x/y, (x+1)/(x-1)), \tag{8.5}$$

   where $\delta^2 = (A-2)/B$, is a birational equivalence from $E_{M,A,B}$ to $E_{E,1,d}$ with exceptional points $y = 0$ and $x = 1$. Conversely, the map

$$(u, v) \mapsto (x, y) = ((v+1)/(v-1), \delta(v+1)/(u(v-1))), \tag{8.6}$$

   is a birational equivalence from $E_{E,1,d}$ to $E_{M,A,B}$ with exceptional points $u = 0$ and $v = 1$. The relation between A and d is $(A-2)/4 = 1/(d-1)$.

Suppose that d is not a square so that the addition formula over $E_{E,1,d}$ is complete. Since both d and -1 are not squares, $-d$ is a square. So, the map

$$(u, v) \mapsto (\hat{u}, \hat{v}) = (\gamma u, 1/v), \tag{8.7}$$

where $-\gamma^2 = d$, is a birational equivalence with exceptional points $v = 0$ from the Edwards curve $E_{E,1,d} : u^2 + v^2 = 1 + du^2v^2$ to the twisted Edwards curve $E_{E,-1,-1/d} : -\hat{u}^2 + \hat{v}^2 = 1 + (-1/d)\hat{u}^2\hat{v}^2$.

Case $p \equiv 1 \bmod 4$**.** Let $E_{M,A,B} : y^2 = x^3 + Ax^2 + x$ be a Montgomery curve and $E_{E,-1,d} : -u^2 + v^2 = 1 + du^2v^2$ be an Edwards curve over $\mathbb{F}_p$. Note that -1 is a square in $\mathbb{F}_p$.

1. If $(A+2)/B$ is a square in $\mathbb{F}_p$, then the map

$$(x,y) \quad \mapsto \quad (u,v) = (\delta x/y, (x-1)/(x+1)), \tag{8.8}$$

where $-\delta^2 = (A+2)/B$, is a birational equivalence from $E_{M,A,B}$ to $E_{E,-1,d}$ with exceptional points $y=0$ and $x=-1$. Conversely, the map

$$(u,v) \quad \mapsto \quad (x,y) = ((1+v)/(1-v), \delta(1+v)/(u(1-v))), \tag{8.9}$$

is a birational equivalence from $E_{E,-1,d}$ to $E_{M,A,B}$ with exceptional points $u=0$ and $v=1$. The relation between A and d is $(A+2)/4 = 1/(1+d)$.

2. If $(A-2)/B$ is a square in $\mathbb{F}_p$, then the map

$$(x,y) \quad \mapsto \quad (u,v) = (\delta x/y, (x+1)/(x-1)), \tag{8.10}$$

where $-\delta^2 = (A-2)/B$, is a birational equivalence from $E_{M,A,B}$ to $E_{E,-1,d}$ with exceptional points $y=0$ and $x=1$. Conversely, the map

$$(u,v) \quad \mapsto \quad (x,y) = ((v+1)/(v-1), \delta(v+1)/(u(v-1))), \tag{8.11}$$

is a birational equivalence from $E_{E,-1,d}$ to $E_{M,A,B}$ with exceptional points $u=0$ and $v=1$. The relation between A and d is $(A-2)/4 = -1/(d+1)$.

The above birational equivalences can be obtained using the elementary birational equivalences in [BL07a, BBJ+08]. On the other hand, verification of these birational equivalences can be done by direct substitution.

8.5 Concrete Curves

The parameters of the new curves are given below.

8.5.1 Curves over $\mathbb{F}_{2^{251}-9}$

Let $p = 2^{251}-9 \equiv 3 \bmod 4$. The minimum positive value of A for which the curve M$[\![p251\text{-}9, A]\!]$ attains the optimal value of (h, h_T) is $A = 4698$. We have that $A-2$ is a square in $\mathbb{F}_p$. Using the birational equivalences given by (8.5) and (8.6), we obtain the pair $(M[\![4698]\!], E[\![1175/1174]\!])$ of birationally equivalent curves. The quantity $1175/1174$ is a non-square modulo $p251\text{-}9$ and so the addition formula over $E_{E,1,1175/1174}$ is complete. The parameters for M$[\![p251\text{-}9, 4698]\!]$ are given in Figure 8.1.

The point $(\cdot, 2)$ is a point of order ℓ on $E_{E,1,1175/1174}$; the corresponding point on $E_{M,4698,1}$ is $(3, \cdot)$.

The set of scalars for $E_{M,4698,1}$ is set to be $4(2^{248} + \{0, 1, \ldots, 2^{248}-1\})$. Given a 32-byte scalar a, the clamping function $\mathsf{clamp}(a)$ is defined as follows (assuming that the first byte is the least significant byte of a): clear bits 0 and 1 of the first byte; set bit number 2 of the last byte and clear bits numbered 3 to 7 of the last byte.

$$
\begin{aligned}
n &= 3618502788666131106986593281521497120369356141117981896093957\backslash \\
\ell &= 0470945719024049046256971665327767466483203803742800923390352\backslash \\
&\quad 7949547402348926173642975601, \\
\log_2 \ell &= 249, \\
h &= 4, \\
k &= \ell - 1, \\
n_T &= 3618502788666131106986593281521497120460017900484553356372141\backslash \\
\ell_T &= 9533999987000769046256971665327767466483203803742801150044751\backslash \\
&\quad 2113833909303548834999967 5019, \\
\log_2 \ell_T &= 249, \\
h_T &= 4, \\
k_T &= (\ell_T - 1)/2, \\
D &= -1241912250180399745034327778701567247379997146229047842147 76\backslash \\
&\quad 46400945935050060, \\
\lceil \log_2(-D) \rceil &= 253.
\end{aligned}
$$

Figure 8.1: Parameters for the curve M$[\![p251\text{-}9, 4698]\!]$

8.5.2 Curves over $\mathbb{F}_{2^{444}-17}$

Let $p = 2^{444} - 17 \equiv 3 \bmod 4$. The minimum positive value of A for which the curve M$[\![p444\text{-}17, A]\!]$ attains the optimal value of (h, h_T) is $A = 4058$. We have that $A - 2$ is a square in $\mathbb{F}_p$. Using the birational equivalences given by (8.5) and (8.6), we obtain the pair $(M[\![4058]\!], E[\![1015/1014]\!])$ of birationally equivalent curves. The quantity $1015/1014$ is a non-square modulo $p444\text{-}17$ and so the addition formula over $E_{E,1,1015/1014}$ is complete. The parameters for M$[\![p444\text{-}17, 4058]\!]$ are given in Figure 8.2.

The point $(\cdot, 2)$ is a point of order ℓ for $E_{E,1,1015/1014}$; the corresponding point on $E_{M,4058,1}$ is $(3, \cdot)$.

The set of scalars is set to be $4(2^{441} + \{0, 1, \ldots, 2^{441} - 1\})$. Given a 56-byte scalar a, the clamping function $\mathsf{clamp}(a)$ is defined as follows (assuming that the first byte is the least significant byte of a): clear bits 0 and 1 of the first byte; set bit number 3 of the last byte and clear bits numbered 4 to 7 of the last byte.

Remark 8.1. *Using the isogenies given in [CN15], it can be shown that M$[\![p444\text{-}17, 4058]\!]$ is 4-isogenous to E$[\![p444\text{-}17, -1014]\!]$. Also, it has been mentioned earlier that M$[\![p251\text{-}9, 4698]\!]$ is 4-isogenous to Curve1174. Connecting Montgomery and Edwards using these isogenies can be a problem, since a small base point on one of these curves does not translate to a small base point on the other.*

$$
\begin{aligned}
n &= 4542742026847543065933273799300028339710258504295737876759313\backslash \\
&\quad 7448788478822109994887784723325457774857125204145126361050201\backslash \\
&\quad 810186649452, \\
\ell &= 1135685506711885766483318449825007084927564626073934469189828\backslash \\
&\quad 4362197119705527498721946180831364443714281301036281590262550\backslash \\
&\quad 452546662363, \\
\log_2 \ell &= 442, \\
h &= 4, \\
k &= (\ell - 1)/3, \\
n_T &= 4542742026847543065933273799300028339710258504295737876759313\backslash \\
&\quad 7448791432192064745527989158013762670837970111055656911191490\backslash \\
&\quad 015016927348, \\
\ell_T &= 1135685506711885766483318449825007084927564626073934469189828\backslash \\
&\quad 4362197858048016186381997289503440667709492527763914227797872\backslash \\
&\quad 503754231837, \\
\log_2 \ell_T &= 442, \\
h_T &= 4, \\
k_T &= (\ell_T - 1), \\
D &= -1795290825514957729951691737953552054640775333171791787428 76\backslash \\
&\quad 8650737402994958967759616343419869098585308883585466645039376\backslash \\
&\quad 73912260606892, \\
\lceil \log_2(-D) \rceil &= 446.
\end{aligned}
$$

Figure 8.2: Parameters for the curve M$[\![p444\text{-}17, 4058]\!]$

8.5.3 Curves over $\mathbb{F}_{2^{506}-45}$

Let $p = 2^{506} - 45$. We ran a search program to find Montgomery curves M$[\![p, A]\!]$ satisfying the security criteria given in Section 2.2.1. The minimum positive value of A for which $(h, h_T) = (4, 4)$ and the other parameters mentioned in Section 2.2.1 are large is $A = 996558$. This gives the curve M$[\![p506\text{-}45, 996558]\!]$. The curve E$[\![p506\text{-}45, 249140/249139]\!]$ is birationally equivalent to M$[\![p506\text{-}45, 996558]\!]$ using the birational equivalences given by (8.5) and (8.6). The quantity $249140/249139$ is a non-square modulo p506-45 and so the addition formula over E$[\![p506 - 45, 249140/249139]\!]$ is complete. The parameters for M$[\![p506\text{-}45, 996558]\!]$ are given in Figure 8.3.

The point $(3, \cdot)$ is a point of order ℓ on the Montgomery curve M$[\![p506\text{-}45, 996558]\!]$; the corresponding point on the Edwards curve E$[\![p506\text{-}45, 249140/249139]\!]$ is $(\cdot, 2)$. The set of scalars is defined to be $4(2^{503} + \{0, 1, \ldots, 2^{503} - 1\})$. Given a 64-byte scalar a, assuming the least significant byte ordering, the clamping function $\mathsf{clamp}(a)$ is defined as follows: clear bits 0 and 1 of the first byte; set bit number 1 of the last byte and clear bits numbered 2 to 7 of the last byte.

$$
\begin{aligned}
n &= 2094969989053530796808441405969663457418650909467561465269306\backslash \\
&\quad 4755815256296991875915250634273539623584422884898906005755971\backslash \\
&\quad 98262455620557286697553856857 88, \\
\ell &= 5237424972633826992021103514924158643546627273668903663173266\backslash \\
&\quad 1889538140742479689788126585683849058961057212247265014389929\backslash \\
&\quad 95656139051393216743884642 1447, \\
\log_2 \ell &= 504, \\
h &= 4, \\
k &= (\ell - 1)/17, \\
n_T &= 2094969989053530796808441405969663457418650909467561465269306\backslash \\
&\quad 4755815256296987958387255222908231949627108464657926763152945\backslash \\
&\quad 99825923112745821562961457542 52, \\
\ell_T &= 5237424972633826992021103514924158643546627273668903663173266\backslash \\
&\quad 1889538140742469895968138057270579874067771161644816907882364\backslash \\
&\quad 99564807781864553907403643856 3, \\
\log_2 \ell_T &= 504, \\
h_T &= 4, \\
k_T &= (\ell_T - 1), \\
D &= -4543123557506175626449202213951075823120804418252321163949549\backslash \\
&\quad 4734779419592882800858550177100850366751591345062685609028019 0\backslash \\
&\quad 1741018690827259888055710502 52, \\
\lceil \log_2(-D) \rceil &= 508.
\end{aligned}
$$

Figure 8.3: Parameters for the curve M$[\![p506\text{-}45, 996558]\!]$

Remark 8.2. *Let $\alpha = (A+2)/4 = 249140$. The curves $M[\![p506\text{-}45, 4\alpha - 2]\!]$ and $E[\![p506\text{-}45, 1-\alpha]\!]$ can be shown to be 4-isogenous using the isogenies given in [CN15]. Further, using the fact that $-\alpha$ is a square in $\mathbb{F}_p$, the curves $M[\![p506\text{-}45, 2 - 4/\alpha]\!]$ and $E[\![p506\text{-}45, 1-\alpha]\!]$ are birationally equivalent using the birational equivalences given by (8.5) and (8.6).*

8.5.4 Curves over $\mathbb{F}_{2^{510}-75}$

Let $p = 2^{510} - 75$. We ran a search program to find Montgomery curves M$[\![p, A]\!]$ satisfying the security criteria given in Section 2.2.1. The minimum positive value of A for which an optimal value of (h, h_T) is obtained is $A = 793638$. In this case, neither $(A+2)$ nor $(A-2)$ are squares in $\mathbb{F}_p$. So, the birational equivalences in Section 8.4.2 for connecting Montgomery and Edwards curves cannot be applied. One may consider a quadratic twist of $E_{M,A,1}$. Since 2 is not a square, $E_{M,A,2}$ is a quadratic twist of $E_{M,A,1}$. Then $E_{M,A,2}$ can be connected to $E_{E,-1,d}$ using either of the birational equivalences given by (8.8), (8.9) or, (8.10), (8.11). The form of d in these two cases are $(A-2)/(A+2)$

$$
\begin{aligned}
n &= 335195198248564927489350624955146153186984145514809834443089\backslash\\
&\quad 036093044100751840662869613465178025950119140505107956858789\backslash\\
&\quad 95333085656635076991016932459506 96,\\
\ell &= 418993997810706159361688281193932691483730181893512293053861\backslash\\
&\quad 295116305125939800828587016831472532437648925631384946073487\backslash\\
&\quad 44166357070793846238771165574383 7,\\
\log_2 \ell &= 507,\\
h &= 8,\\
k &= \ell - 1,\\
n_T &= 335195198248564927489350624955146153186984145514809834443089\backslash\\
&\quad 036093044100751836685970480249730319221265361087801367443752\backslash\\
&\quad 73436284030977727411513125709120 4,\\
\ell_T &= 837987995621412318723376562387865382967460363787024586107722\backslash\\
&\quad 590232610251879591714926200624325798053163402719503418609381\backslash\\
&\quad 83590821007744431852878281427280 1,\\
\log_2 \ell_T &= 508,\\
h_T &= 4,\\
k_T &= \ell_T - 1,\\
D &= -325310369051208815436075592806876377044871206601624247034270\backslash\\
&\quad 696834459740973354036373190534534468020293163287714643466196 6\backslash\\
&\quad 32005321502907801083234231911482 0,\\
\lceil \log_2(-D) \rceil &= 510.
\end{aligned}
$$

Figure 8.4: Parameters for the curve M$[\![p510\text{-}75, 952902]\!]$

and $(A+2)/(A-2)$ respectively. Since both $(A+2)$ and $(A-2)$ are not squares, both $(A-2)/(A+2)$ and $(A+2)/(A-2)$ are squares. Consequently, the completeness of the addition formula over $E_{E,-1,d}$ is not ensured. Since $p \equiv 1 \bmod 4$, it is desirable to use birational equivalences to connect a Montgomery curve to a twisted Edwards form curve having a complete addition formula. For $A = 793638$, this does not seem to be possible using the birational equivalences in Section 8.4.2.

The next value of A for which an optimal value of (h, h_T) is obtained is $A = 952902$. In this case, we obtain the curves M$[\![p510\text{-}75, 952902]\!]$ and $\widetilde{\text{E}}[\![p510\text{-}75, -238225/238226]\!]$ which are birationally equivalent using the birational equivalences given by (8.8) and (8.9). The quantity $-238225/238226$ is a non-square modulo $p510\text{-}75$ and so the addition formula over $\widetilde{\text{E}}[\![p510\text{-}75, -238225/238226]\!]$ is complete. The parameters for M$[\![p510\text{-}75, 952902]\!]$ are given in Figure 8.4.

The point $(4, \cdot)$ is of order ℓ on the Montgomery curve M$[\![p510\text{-}75, 952902]\!]$; the corresponding point on the twisted Edwards curve $\widetilde{\text{E}}[\![p510\text{-}75, -238225/238226]\!]$ is $(\cdot, 3/5)$. The set of scalars is set to be $8(2^{510} + \{0, 1, \ldots, 2^{510} - 1\})$. Given a 64-byte scalar a, as-

suming the least significant byte ordering, the clamping function $\mathsf{clamp}(a)$ is defined as follows: clear bits 0, 1 and 2 of the first byte; set bit number 5 of the last byte and clear bits numbered 6 and 7 of the last byte.

Remark 8.3. *Let $\alpha = (A+2)/4 = 238226$, which is a square. The curves $M[\![p510\text{-}75, 4\alpha - 2]\!]$ and $\widetilde{E}[\![p510\text{-}75, \alpha - 1]\!]$ can be shown to be 4-isogenous using the isogenies given in [CN15]. Further, $M[\![p510\text{-}75,\ 4/\alpha - 2]\!]$ and $\widetilde{E}[\![p510\text{-}75, \alpha - 1]\!]$ are birationally equivalent using the birational equivalences given by (8.8) and (8.9). $M[\![p510\text{-}75, 2 - 4/\alpha]\!]$ and $\widetilde{E}[p510\text{-}75, \alpha - 1]$ are birationally equivalent using the birational equivalences given by (8.10) and (8.11).*

8.5.5 Curves over $\mathbb{F}_{2^{521}-1}$

The curve E-521 [ABGR13] is same as the curve $E[\![p521\text{-}1, -376014]\!]$. Using the isogenies given in [CN15], the curve $E[\![p521\text{-}1, -376014]\!]$ is 4-isogenous to $M[\![p521\text{-}1, 1504058]\!]$. This gave us $M[\![p521\text{-}1, 1504058]\!]$. Since the birational equivalences in Section 8.4.2 are simpler than the isogenies in [CN15], we obtained the Edwards form curve $E[\![p521\text{-}1, 376015/376014]\!]$ which is birationally equivalent to $M[\![p521\text{-}1, 1504058]\!]$. The birational equivalences are given by (8.5) and (8.6). The quantity $376015/376014$ is a non-square modulo $p521\text{-}1$ and so the addition formula over $E[\![p521\text{-}1, 376015/376014]\!]$ is complete. The parameters for $M[\![p521\text{-}1, 1504058]\!]$ are given in Figure 8.5.

The point $(8, \cdot)$ is a point of order ℓ on the Montgomery curve $M[\![p521\text{-}1, 1504058]\!]$; the corresponding point on the Edwards curve $E[\![p521\text{-}1, 376015/376014]\!]$ is $(\cdot, 9/7)$.

The set of scalars for $E_{M,1504058,1}$ is set to be $4(2^{518} + \{0, 1, \ldots, 2^{518} - 1\})$. Given a 65-byte scalar a, assuming the least significant byte ordering, the clamping function $\mathsf{clamp}(a)$ is defined as follows: clear bits 0 and 1 of the first byte; set bit number 0 of the last byte and clear bits numbered 1 to 7 of the last byte.

Remark 8.4. *Let $\alpha = (A+2)/4 = 376015$. The curves $M[\![p521\text{-}1, 2 - 4/\alpha]\!]$ and $E[\![p521\text{-}1, 1 - \alpha]\!]$ are birationally equivalent using the birational equivalences given by (8.5) and (8.6).*

8.6 Implementation Details

We consider Montgomery form curves. Let P be a generator of the prime order cyclic subgroup of the elliptic curve over which cryptography is to be done. For all the curves considered in this work, the point P can be chosen such that its x-coordinate is small. Such a fixed point is called the base point of the corresponding curve. The base points of the curves considered in this paper are given in Table 8.1. For a point $Q \in \langle P \rangle$ and a non-negative integer a which is less than the order of P, the task of computing the a-fold product aQ is called scalar multiplication. In the case, $Q = P$, we will call the operation aP to be fixed base scalar multiplication, while when Q is an arbitrary element of $\langle P \rangle$, we will call the operation aQ to be variable base scalar multiplication. In the Diffie-Hellman protocol, variable base scalar multiplication is required for the shared secret phase, while fixed base scalar multiplication is required for the key generation phase. Our primary focus will be variable base scalar multiplication for the shared secret phase.

The Montgomery ladder [Mon87] is an x-coordinate only algorithm which can be used to compute the x-coordinate of the result of a scalar multiplication. The ladder computation is performed using projective coordinates and at the end, the result is converted to affine coordinates. When the scalar a in the scalar multiplication aQ is a secret,

$$
\begin{aligned}
n &= 6864797660130609714981900799081393217269435300143305409394463\backslash \\
&\quad 4591855431833976547019035066066546313985467746362609365704172\backslash \\
&\quad 7713179481016927197368517468043409 2, \\
\ell &= 1716199415032652428745475199770348304317358825035826352348615\backslash \\
&\quad 8647963857958494136754758766516636578496366936590652341426043\backslash \\
&\quad 1928294870254231799342129367010852 3, \\
\log_2 \ell &= 519, \\
h &= 4, \\
k &= \ell - 1, \\
n_T &= 6864797660130609714981900799081393217269435300143305409394463\backslash \\
&\quad 4591855431833976574023416126746682777114078179865220251456569\backslash \\
&\quad 6684420462311835317437140754968021 2, \\
\ell_T &= 1716199415032652428745475199770348304317358825035826352348615\backslash \\
&\quad 8647963857958494143505854031686670694278519544966305062864142\backslash \\
&\quad 4171105115577958829359285188742005 3, \\
\log_2 \ell_T &= 519, \\
h_T &= 4, \\
k_T &= \ell_T - 1, \\
D &= -2563609914934638872981081855263139865560965378352719883225 15\backslash \\
&\quad 6029512739349840149810402763183578852246400006757283129006946\backslash \\
&\quad 2218128904642355069855506040176465004, \\
\lceil \log_2(-D) \rceil &= 523.
\end{aligned}
$$

Figure 8.5: Parameters for the curve M$[\![p521\text{-}1, 1504058]\!]$

for secure computation, it is important that the computation be implemented in constant time. There are known ways to implement the Montgomery ladder in constant time. A detailed treatment of the Montgomery ladder has been addressed in [BL17, CS18].

Intel processors provide two kinds of 64-bit integer multiplication operations, namely `mul` and `mulx`, where `mul` modifies both the carry and overflow flags, but `mulx` does not modify either of these flags. The `add` and `adc` instructions perform addition and addition-with-carry using the carry flag respectively and modifies both the carry and the overflow flags; the instruction `adcx` performs addition-with-carry using the carry flag, but does not modify the overflow flag, while the instruction `adox` performs addition-with-carry using the overflow flag, but does not modify the carry flag. By `maa` we will denote implementations which use only the `mul`, `add` and `adc` instructions and not any of `mulx`, `adcx` or `adox`; `mxaa` will denote implementations which use `mulx`, `add` and `adc` instructions; while `maax` will denote implementations which use `mulx`, `adcx` and `adox`. The `maa` type implementations are supported across a wide range of Intel processors, the `mxaa` type implementations are supported from the Haswell processor onwards, while

the `maax` type instructions are supported on modern generation Intel processors such as Skylake.

The previously best known `maa`, `mxaa` and `maax` type implementations of Curve25519 are available in [OLH+17]. For all three types, we provide new implementations of Curve25519 which are faster than the implementations in [OLH+17]. For Curve448, `mxaa` and `maax` type implementations are available from [OLH+17]. Improved implementations of similar type have been discussed in Chapter 5 We provide a new `maa` type implementation for Curve448.

Intel processors from Haswell onwards provide `AVX2` instructions which support 4-way SIMD on 256-bit registers. This allows vectorized implementations. For the Montgomery ladder, 4-way vectorized algorithms have been described in [HEY20] and Chapter 7. Vectorized implementations of the Montgomery ladder for Curve25519 and Curve448 are known.

We provide `maa`, `mxaa` and `maax` type implementations for the new curves. We also provide vectorized implementations of the new curves which following the method discussed in Chapter 7.

8.6.1 64-bit Implementations

All 64-bit implementations of the Montgomery ladder are done using Algorithm 3.6. The implementations are done using both saturated limb and unsaturated limb representations. The details of the implementations are discussed below.

▶ Implementations Using Saturated Limb Representation

Let $m = \lceil \log_2 p \rceil$. Elements of $\mathbb{F}_p$ can be represented as m-bit strings which will be represented as κ 64-bit words. Conventionally, each such word is called a limb. We will consider packed or saturated limb representation. In this representation, m is written as $m = 64(\kappa - 1) + \nu$ with $1 \leq \nu \leq \eta \leq 64$. In other words, the first $(\kappa - 1)$ limbs are 64 bits long, while the size of the last limb is ν which lies between 1 and η.

Representation of field elements. The representations of the four primes of interest to this work are given in Table 8.2. Note that for p251-9. p444-17, p506-45 and p521-1, $\eta - \nu \geq 3$ (equivalently, the last limb has three or more "free" bits), for p510-75, $\eta - \nu = 2$ (equivalently, the last limb has two "free" bits), for p255-19, $\eta - \nu = 1$ (equivalently, the last limb has one "free" bit) and for p448-224-1, $\eta = \nu$ (equivalently, the last limb has no "free" bits). These have significant effect on the ladder computation as we will see below.

Integer multiplication/squaring. The `maax` operations can be used to perform fast integer multiplication using two independent carry chains. For multiplication/squaring of 256-bit numbers, this technique has been explained in the Intel white papers [OGG13, OGGF12]. A general algorithmic description for multiplication/squaring of 64κ-bit numbers, $\kappa \geq 4$ is given in Chpater 4. Saturated limb multiplication/squaring algorithms can also be implemented using a single carry chain with the help of the instructions mulx/add/adc. Such sequential implementations are applicable for the Haswell processor where the instruction mulx is available but the instructions adcx/adox are not.

Prime	m	κ	η	ν	$\eta - \nu$
p251-9	251	4	64	59	5
p255-19	255	4	64	63	1
p444-17	444	7	64	60	4
p448-224-1	448	7	64	64	0
p506-45	506	8	64	58	6
p510-75	510	8	64	62	2
p521-1	521	9	64	9	55

Table 8.2: Saturated limb representations of field elements.

Reduction. Integer multiplication/squaring of κ-limb quantities produces a 2κ-limb output. The reduction step reduces this output modulo the prime p. A full reduction will reduce the output to a value less than p. For the purposes of efficiency a full reduction is not carried out in the intermediate steps of the computation. Instead a size reduction is done. The size reduction can be of two types, namely, reduction to an $(m+1)$-bit integer and reduction to an m-bit integer (note than an m-bit integer is not necessarily fully reduced since it is not necessarily less than p). The former is more efficient than the later. Further, the reduced quantity should again be a κ-limb quantity. If $\nu < 64$, i.e., the last limb has at least one free bit, then reduction to an $(m+1)$-bit integer is a κ-limb quantity. On the other hand, if $\nu = 64$, i.e., the last limb has no free bits, then it is a necessity to reduce to an m-bit integer to obtain a κ-limb quantity. Among the primes in Table 8.2, the prime $2^{448} - 2^{224} - 1$ has no extra bits in the last limb and the reduction for this prime has to be to an m-bit integer. For the other primes, it is possible to reduce to an $(m+1)$-bit integer without any overfull. The size reductions to $(m+1)$ bits modulo $2^{251} - 9$ and $2^{444} - 17$ have been done following the algorithm reduceSLPMP of Chapter 4.

Addition and subtraction. Other than multiplication/squarings, the ladder algorithm also uses field addition and subtraction. In the ladder algorithm, the inputs to an addition/subtraction operation are outputs of multiplication/squaring operations and the outputs of addition/subtraction operations are inputs to multiplication/squaring operations. In particular, the outputs of addition/subtraction are never inputs to another addition/subtraction.

We have mentioned that the outputs of multiplication/squaring are size reduced to either m bits or to $(m+1)$ bits. So, the inputs to addition/subtraction operations are either m bits or $(m+1)$ bits. We require the outputs of the addition/subtraction operations to be κ-limb quantities so that the integer multiplication/squaring algorithm can be applied to these outputs. So, it is not always required to size reduce the outputs of addition/subtraction operations to m or $(m+1)$ bits. Depending upon the sizes of the inputs to the addition/subtraction operation and the relative values of η and ν, various cases may arise. We discuss the cases of addition and subtraction separately.

ADDITION. A field addition is typically an integer addition followed by a possible re-

duction operation. The integer addition operation increases the size of the output by one bit compared to the sizes of the inputs.

*Case p*448-224-1*:* In this case, there is no leeway in the last limb and the output of integer addition must necessarily be reduced to obtain a κ-limb quantity.

*Case p*255-19*:* If the inputs to the addition are m-bit quantities, then it is possible to omit applying the reduction step to the output of the integer addition operation. On the other hand, if the inputs to the addition are $(m+1)$-bit quantities, then the reduction step has to be applied to the output of the integer addition operation. The inputs to the addition operation are the outputs of previous multiplication/squaring operations. So, whether the output of the integer addition needs to be reduced depends on whether the outputs of the multiplication/squaring operation have been reduced to m bits or to $(m+1)$ bits.

*Cases p*251-9*, p*444-17*, p*506-45*, p*510-75 *and p*521-1*:* In these cases, it is possible to reduce the outputs of multiplication/squaring to $(m+1)$ bits *and* omit the reduction step after the integer addition operation.

SUBTRACTION. A field subtraction is of the type $a - b \bmod p$. To avoid handling negative numbers, a suitable multiple of p is added to a so that the result is guaranteed to be positive. Since the result will be reduced modulo p, the correctness of the result is not affected by adding a multiple of p.

*Cases p*255-19*, p*448-224-1 *and p*510-75*:* The reduction operation must be performed on the output of each subtraction operation to ensure that the result fits in κ limbs.

*Cases p*251-9*, p*444-17*, p*506-45 *and p*521-1*:* The operation $a - b \bmod p$ is performed as follows. Note that both a and b are $(m+1)$-bit quantities. The operation $4p + a - b$ is guaranteed to be an $(m+3)$-bit non-negative integer. So, instead of performing $a - b \bmod p$, the operation $(4p + a) - b$ is computed. Since the result is at most an $(m+3)$-bit quantity, it fits within κ limbs. Consequently, no reduction operation is performed on this result.

Remark 8.5. *We have discussed the issue of avoiding reduction with respect to 64-bit arithmetic. The general idea, on the other hand, holds for saturated limb representations using 32-bit (or, lower) arithmetic. The implementation benefits of p*251-9 *over p*255-19 *and of p*444-17 *over p*448-224-1 *also holds for 32-bit arithmetic.*

Optimizations of the ladder step. Based on the description in Section 8.6, the following strategy may be adopted for implementing the ladder step for the various primes.

*Case p*448-224-1*:* The outputs of all multiplication/squaring operations are to be size reduced to m bits. Outputs of all addition/subtraction operations are to be size reduced to m bits.

*Case p*510-75*:* The outputs of all multiplication/squaring/addition operations are to be size reduced to $m+1$ bits. Outputs of all subtraction operations are to be size reduced to m bits.

*Case p*255-19*:* The outputs of all multiplication/squaring operations are to be size reduced to $(m+1)$ bits. Outputs of all addition/subtraction operations are to be size reduced to m or $(m+1)$ bits.

*Cases p*251-9*, p*444-17*, p*506-45 *and p*521-1*:* The outputs of all multiplication/squaring operations are to be size reduced to $(m+1)$ bits. Outputs of all addition/subtraction operations are left unreduced.

The above strategy has direct consequences to the efficiencies of the ladder step for the various primes. We summarize these below.

4-*limb representations:* For both $\mathbb{F}_{2^{251}-9}$ and $\mathbb{F}_{2^{255}-19}$, field elements have 4-limb representations. So, the integer multiplication/squaring operations take the same time in both cases. Due to the ability to avoid reductions, the ladder step is significantly faster modulo $2^{251}-9$ compared to $2^{255}-19$.

7-*limb representations:* For the fields $\mathbb{F}_{2^{444}-17}$ and $\mathbb{F}_{2^{448}-2^{224}-1}$, field elements have 7-limb representations. So, the integer multiplication/squaring operations take the same time in both cases. Due to the ability to avoid reductions, the ladder step is significantly faster modulo $2^{444}-17$ compared to $2^{448}-2^{224}-1$.

8-*limb representations:* For both $\mathbb{F}_{2^{506}-45}$ and $\mathbb{F}_{2^{510}-75}$, field elements have 8-limb representations. So, the integer multiplication/squaring operations take the same time in both cases. Due to the ability to avoid reductions in the subtraction operation for $\mathbb{F}_{2^{510}-75}$, the ladder step is faster modulo $2^{506}-45$ compared to $2^{510}-75$.

► Implementations Using Unsaturated Limb Representation

Representation of field elements. 64-bit implementations of the Montgomery ladder using unsaturated limb representation of field elements have been explored for the Montgomery curves over $\mathbb{F}_{2^{251}-9}$, $\mathbb{F}_{2^{444}-17}$ and $\mathbb{F}_{2^{448}-2^{224}-1}$. The elements of $\mathbb{F}_{2^{251}-9}$ are represented using 5 words, i.e., we consider values of κ as 5. The elements of $\mathbb{F}_{2^{444}-17}$ and $\mathbb{F}_{2^{448}-2^{224}-1}$ are represented using 8 words, and here the value of κ is 8. The details of the representations are provided in Table 8.3. For the curves at 224-bit security level we have observed that vectorized implementations perform much better than the 64-bit unsaturated limb implementations. As a reason we have left out 64-bit implementations of the ladder using unsaturated limb representation for the curves over $\mathbb{F}_{2^{506}-45}$, $\mathbb{F}_{2^{510}-75}$ and $\mathbb{F}_{2^{521}-1}$ at 256-bit security level.

Prime	m	κ	η	ν	$\eta-\nu$
p251-9	251	5	51	47	4
p444-17	444	8	56	52	4
p448-224-1	448	8	56	56	0

Table 8.3: Unsaturated limb representations of field elements.

Field operations. We discuss below in brief the field operations for the primes provided in Table 8.3.

Integer multiplication/squaring: For the primes p251-9 and p444-17, integer multiplication and squaring have been done using the algorithm discussed in Section 4.7 and reduction after integer multiplication/squaring is done using Algorithm 4.8. For the prime p448-224-1 we use the algorithms discussed in Section 5.5.

Addition/subtraction: Addition and subtraction are performed limb-wise ignoring the carry and the results are kept unreduced.

Multiplication by a small constant: For the primes p251-9 and p444-17 multiplication by a small constant is done using the method applied in the unsaturated limb implementation of Curve25519 in [BDL+12]. A similar method has been applied for the prime p448-224-1.

8.6.2 Vectorized Implementation

4-way vectorized implementations of the Montgomery ladder are based on Algorithms 7.5 and 7.6. We have done vectorized implementations for all the new curves M⟦4698⟧, M⟦4058⟧, M⟦996558⟧, M⟦952902⟧, M⟦1504058⟧ and also for the standard curves Curve25519 and Curve448. The details of the field operations used within the ladder are as described in Chapter 6.

8.6.3 Inversion in $\mathbb{F}_p$

The final output of the ladder algorithm needs to compute a modular inverse. The computation of the inverse is done through exponentiation, which needs squaring and multiplication in $\mathbb{F}_p$. The relevant algorithms for field arithmetic from Chapter 4 have been used for computing the inversions.

8.7 Implementations and Timings

For the four curves at 128-bit and 224-bit security levels saturated limb implementations of both `maax`-type and `mxaa`-type of the ladder have been done, whereas for the three curves at 256-bit security level we have developed only `maax`-type implementations. The saturated limb implementations of `maa`-type have been developed only for the two curves at 128-bit security level. The vectorized implementations have been done using the `AVX2` instructions.

The timings of different implementations for variable base scalar multiplication are populated in Table 8.4. For comparison, we report the timings of the previously most efficient (to the best of our knowledge) and publicly available sequential and vectorized implementations. For 64-bit implementations of Curve25519 and Curve448 we refer to [OLH+17] and for vectorized implementations of Curve25519 we refer to [HEY20]. The timing results show that the `maa`-type implementations are uniformly slower than the `mxaa`-type and the `maax` type implementations. Due to this, we report the timings of the `maa` type implementations in seperately in Table 8.4. The performance analyses for variable base scalar multiplication at different security levels are described below.

Curve	Haswell	Skylake	κ	Strategy	Reference	Implementation Type
Curve25519	161045	148291	5	64-bit seq	[BDL+12]	`maa`, assembly
	179124	147823	4	64-bit seq	[BDL+12]	`maa`, assembly
	170381	137453	4	64-bit seq	[OLH+17]	`maa`, inline assembly
	167170	128843	4	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	143956	126940	4	64-bit seq	[OLH+17]	`mxaa`, inline assembly
	143369	113481	4	64-bit seq [Alg. 3.6]	this work	`mxaa`, assembly
	-	118231	4	64-bit seq	[OLH+17]	`maax`, inline assembly
	-	98694	4	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	140996	104519	9	4-way SIMD, [HEY20]	[HEY20]	`AVX2`, intrinsics
	121539	99898	9	4-way SIMD [HEY20]	Table 7.2	`AVX2`, assembly
	126521	97590	10	4-way SIMD [HEY20]	Table 7.2	`AVX2`, assembly
	120108	99194	9	4-way SIMD [Alg. 7.6]	Table 7.2	`AVX2`, assembly
	123899	95437	10	4-way SIMD [Alg. 7.6]	Table 7.2	`AVX2`, assembly
M⟦4698⟧	154455	132255	5	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	143282	118019	4	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	129732	102570	4	64-bit seq [Alg. 3.6]	this work	`mxaa`, assembly
	-	87807	4	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	114937	91203	9	4-way SIMD [Alg. 7.6]	this work	`AVX2`, assembly
Curve448	721044	558740	8	64-bit seq [Alg. 3.6]	Table 5.3	`maa`, assembly
	732013	587389	7	64-bit seq	[OLH+17]	`mxaa`, inline assembly
	719217	461379	7	64-bit seq [Alg. 3.6]	Table 5.1	`mxaa`, assembly
	-	530984	7	64-bit seq	[OLH+17]	`maax`, inline assembly
	-	434831	7	64-bit seq [Alg. 3.6]	Table 5.1	`maax`, assembly
	462277	373006	16	4-way SIMD [HEY20]	Table 7.2	`AVX2`, assembly
	441715	357095	16	4-way SIMD [Alg. 7.6]	Table 7.2	`AVX2`, assembly
M⟦4058⟧	681257	597240	8	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	644791	423042	7	64-bit seq [Alg. 3.6]	this work	`mxaa`, assembly
	-	384905	7	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	476866	401809	16	4-way SIMD [Alg. 7.6]	this work	`AVX2`, assembly
M⟦996558⟧	-	558957	8	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	674354	574985	18	4-way SIMD [Alg. 7.6]	this work	`AVX2`, assembly
M⟦952902⟧	-	566088	8	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	683419	580096	18	4-way SIMD [Alg. 7.6]	this work	`AVX2`, assembly
M⟦1504058⟧	-	689588	9	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	651934	545670	18	4-way SIMD [Alg. 7.6]	this work	`AVX2`, assembly

Table 8.4: CPU-cycle counts for variable base scalar multiplication on Montgomery curves at 128-bit, 224-bit and 256-bit security levels.

▶ Performances at 128-bit Security Level

New implementations of Curve25519. Table 8.4 shows the timing results for the new implementations of Curve25519 that have done in the context of the present work. The new `maax`-type implementation is about 17% faster over the previous best implementation [OLH+17] on Skylake. On Haswell, the performance improvement of the new `mxaa`-type implementation over the `mxaa`-type implementation of [OLH+17] is small. We would like to mention two issues.

1. For reduction we have used the algorithm reduceSLPMP of Chapter 4, while the algorithm used by [OLH+17] is the same as algorithm reduceSLPMPa of Chapter 4. To assess the effect of the reduction algorithm, we made an assembly implementation (note that the code of [OLH+17] uses inline assembly) using reduceSLPMPa. It turns out that using reduceSLPMP leads to a faster code.

2. The field operations in the implementations of [OLH+17] have been developed using inline assembly and then integrated through a high level function in the Montgomery ladder-step. In contrast, we have developed the entire Montgomery ladder-step as a single hand-optimized assembly code, in which we have judiciously used the available 64-bit registers to minimize the overall load/store operations. The timings indicate that such a strategy to develop the assembly code provides a substantial gain in efficiency on the Skylake processor, while on Haswell the gain is nominal.

From Table 8.4, we note that the performance of the new `maax` implementation of Curve25519 lies between the performances of the 9-limb and 10-limb 4-way SIMD implementations.

Comparison to other related works. In Table 8.4 we have compared the performance of the new 64-bit implementation of variable base ladder computation on Curve25519 to that of [OLH+17].An `AVX2` based implementation of the ladder computation has been reported in [FHD19]. This work predates the introduction of 4-way vectorization of the Montgomery ladder reported in Chapter 6 and [HEY20], and so the use of `AVX2` instructions in [FHD19] is sub-optimal. The presently best known `AVX2` implementation is available from our work given in Chapter 6.

A recent line of work has considered verified implementations of various cryptographic functionalities [EPG+19, PPF+20]. While the verified 64-bit implementation of the ladder for Curve25519 in [PPF+20] is reasonably fast, the timing does not appear to be competitive with the timing reported in Table 8.4. The paper [PPF+20] reports a timing of 113614 cycles on the Kaby Lake processor. In comparison, on the earlier generation Skylake processor our fastest 64-bit implementation requires 98649 cycles.

A different approach has considered the use of `AVX2` instructions for four simultaneous ladder computations [CGT+21]. This approach is useful for applications which require a number of ladder computations to be simultaneously performed. Since our focus is on a single scalar multiplication, it is not meaningful to compare our timings with that of [CGT+21].

Comparison between M⟦4698⟧ and Curve25519. From Table 8.4, we see that M⟦4698⟧ is faster than Curve25519 for all the types of implementations, though the speed-up percentages vary. For `mxaa` type implementations, the speed-ups on both Haswell and Skylake are about 9.5%; for `maax` type implementations the speed-up (on Skylake) is about 11%. The 64-bit `maax` type and `mxaa` type implementations of Curve25519 and $E_{M,4698,1}$ targeting the Skylake micro-architecture have been developed similarly using hand-written assembly. We have tried our best to optimize both the implementations fairly for a neat comparison of the performances of shared-secret computation between the two curves. The cost of the 64-bit assembly stubs used for field multiplication, field squaring and multiplication by a field constant operations are same over both the fields $\mathbb{F}_{2^{251}-9}$ and $\mathbb{F}_{2^{255}-19}$. The major reason for the speed gain of the curve $E_{M,4698,1}$ over Curve25519 is actually obtained due to the linear field operations involved in the Montgomery ladder which can be kept unreduced due to the sufficiently available free bits in the last limb of a field element, while working over the field $\mathbb{F}_{2^{251}-9}$. Unfortunately, we do not have this advantage, while working over $\mathbb{F}_{2^{255}-19}$. For the `AVX2` type implementations the speed-ups on both Haswell and Skylake are about 4.3%.

▶ Performances at 224-bit Security Level

Comparison between M⟦4058⟧ and Curve448. From Table 8.4, we observe that M⟦4058⟧ is faster than Curve448 for `mxaa` and `maax` implementations; for `mxaa` implementations, the speed-ups are 10.3% and 8.3% respectively on Haswell and Skylake, while for `maax`, the speed-up is 11.5 % (on Skylake). For `AVX2` implementations, compared to Curve448, M⟦4058⟧ has slowdowns of 7.4% and 11% on Haswell and Skylake respectively.

The explanation for the above observations is as follows. The number of limbs for both Curve448 and M⟦4058⟧ are the same. For `AVX2` implementations (using 256-bit registers) the number of limbs is 16, whereas for `mxaa` and `maax` type implementations the number of limbs is 7. As a result, for `AVX2` type implementations the underlying integer multiplication is faster using Karatsuba rather than schoolbook, while for the other two implementations schoolbook is faster than Karatsuba. The prime $p448\text{-}224\text{-}1$ on which Curve448 is based has been chosen such that Karatsuba is particularly efficient. So, whenever the underlying integer multiplication is faster using Karatsuba than schoolbook, Curve448 will be faster than M⟦4058⟧. In a similar vein, due to the reasons explained in Section 8.6, whenever the underlying multiplication is faster using schoolbook than Karatsuba, M⟦4058⟧ will be faster than Curve448. Future availability of wider vector operations would lead to a reduction in the number of limbs. This may result in schoolbook becoming faster than Karatsuba and consequently, M⟦4058⟧ being faster than Curve448.

Comparison between `maax` and `mxaa` type implementations. The `mxaa` type assembly implementations of Curve25519 and Curve448 which are suited for the Haswell micro-architecture have not demonstrated substantial improvements. It has to be noted that the same implementations provide a better speed-gain in the Skylake micro-architecture. We have found this behavior to be a little strange and have devoted a substantial amount of time to find out a good reason behind this. But unfortunately, we could not find any. It has to be noted that we have not developed the `mxaa` type implementations based on the Skylake micro-architecture. We believe that writing 64-bit programs using *hand-*

written assembly and *inline assembly* does not produce much difference in performance when executed in the Haswell micro-architecture.

Remark 8.6. *It is interesting to note that for both M⟦4698⟧ and M⟦4058⟧, the best timings are obtained for the* `maax` *implementation rather than the* `AVX2` *implementation. It is expected that the support for* `AVX2` *operations will improve in future processors which should lead to* `AVX2` *implementations outperforming* `maax` *implementations on such processors.*

Curve	Haswell	Skylake	κ	Strategy	Reference	Implementation Type
Curve25519	153754	119958	4	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	132162	106725	4	64-bit seq [Alg. 3.6]	this work	`mxaa`, assembly
	-	93247	4	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	100127	86885	9	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly
	106190	84047	10	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly
M⟦4698⟧	141965	121265	5	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	133346	109989	4	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	120599	96683	4	64-bit seq [Alg. 3.6]	this work	`mxaa`, assembly
	-	82116	4	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	96419	79770	16	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly
Curve448	644288	504808	8	64-bit seq [Alg. 3.6]	Table 5.3	`maa`, assembly
	653035	427058	7	64-bit seq [Alg. 3.6]	Table 5.2	`mxaa`, assembly
	-	396583	7	64-bit seq [Alg. 3.6]	Table 5.2	`maax`, assembly
	381417	317778	16	4-way SIMD [Alg. 7.7]	Table 7.3	`AVX2`, assembly
M⟦4058⟧	612225	540766	8	64-bit seq [Alg. 3.6]	this work	`maa`, assembly
	573782	389371	7	64-bit seq [Alg. 3.6]	this work	`mxaa`, assembly
	-	359597	7	64-bit seq [Alg. 3.6]	this work	`maax`, assembly
	403697	335314	16	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly
M⟦996558⟧	567250	481140	18	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly
M⟦952902⟧	573751	485203	18	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly
M⟦1504058⟧	556410	477765	18	4-way SIMD [Alg. 7.7]	this work	`AVX2`, assembly

Table 8.5: CPU-cycle counts for fixed base scalar multiplication on Montgomery curves at 128-bit, 224-bit and 256-bit security levels.

►Performances at 256-bit Security Level

Vectorized implementations of Montgomery curves at 256-bit security level. The timing results for the new implementations of variable base scalar multiplication of the Montgomery curves at 256-bit security level are shown in Table 8.4. We achieve the following speed-ups.

1. The vectorized implementation of M⟦1504058⟧ is 21% faster in Skylake over the 64-bit `maax`-type sequential implementation of M⟦1504058⟧.

2. The vectorized implementation of M⟦996558⟧ and M⟦952902⟧ are about 3% slower in Skylake over the 64-bit `maax`-type sequential implementation of the curves.

3. The vectorized implementations of the Montgomery curves at the 128-bit and 224-bit security levels outperform the sequential implementations in Haswell. On the basis of this we have left out exploring the sequential implementations of the Montgomery curves at 256-bit security level targeting the Haswell architecture.

8.7.1 Fixed Base Scalar Multiplication

Key generation. We have also developed assembly code for fixed base scalar multiplication using Montgomery ladder. This corresponds to the key generation phase of the Diffie-Hellman protocol. The timings are reported in Table 8.5.

Remark 8.7. *Sequential implementations of Algorithm 3.6 were not done to compute the fixed-base scalar multiplication over the curves at 256-bit security level. It can be observed from the timings reported in Table 8.5 that the vectorized implementations using Algorithm 7.7 outperform the sequential implementations using Algorithm 3.6 to compute the fixed-base scalar multiplications at 128-bit and 224-bit security levels.*

8.7.2 Complete Diffie-Hellman Protocol

Each user chooses a secret key and computes the corresponding public key using the key generation algorithm of the DH protocol. Two users agree upon a shared secret key by following the shared secret computation phase of the DH protocol. In static DH protocol, the same public key may be used with multiple shared secret computations. In such a protocol, the key generation is a less frequent operation compared to the shared secret computation. In ephemeral DH protocol, users choose new secret keys and compute corresponding public keys for each execution of the shared secret computation. So, for ephemeral DH protocol, each execution of the protocol between two parties require both the key generation and the shared secret computation phases. As a result, the computation time for ephemeral DH protocol consists of the sum of the times required for key generation and shared secret computation. Given a particular type of implementation (i.e., `maax`, `mxaa`, or `AVX2`), time estimate for a complete DH computation can be obtained by adding the figures in Table 8.4 with the corresponding figures in Table 8.5.

The timings for key generation phase that are provided in Table 8.5 are for fixed base Montgomery ladder computation. Key generation, on the other hand, will be faster if the scalar multiplication is done by moving to the corresponding birationally equivalent Edwards curve, performing fixed base scalar multiplication on Edwards curve, and then transferring the result back to the Montgomery curve. Since we have not implemented scalar multiplication on Edwards curve, we are unable to report the timings for key generation using this approach.

8.8 Conclusion

In this chapter we introduced three pair of Montgomery-Edwards curves for performing cryptography at the 128-bit, 224-bit and 256-bit security levels. Compared to the curves proposed in IETF RFC 7748, the new curves provide 1.5 to 2 bits less security. We have

performed various kinds of implementations of the Montgomery ladder for the new curves. At the 128-bit level, the new curve is faster than Curve25519 for all the types of implementations that we have considered. At the 224-bit level, the new curve is faster when the underlying integer multiplication is faster using schoolbook than Karatsuba. Our work provides a wider picture of the efficiency/security trade-off at the 128-bit, 224-bit and 256-bit security levels.

CHAPTER 9

Conclusion

9.1 Summary

In this thesis our broad focus has been in two areas - design and analysis of efficient field arithmetic and proposal of efficient curves at various security levels.

First, we have studied design of efficient sequential algorithms for multiplication and squaring over fields related to the primes of the form $2^m - \delta$ and the Goldilocks prime $2^{448} - 2^{224} - 1$. On the theoretical side, we provide various algorithms for multiplication/squaring and reduction. The correctness of the reduction algorithms have been rigorously proven. On the practical side, we provide efficient constant-time assembly implementation of the various algorithms for modern Intel processors. For well known primes our implementations are faster than the previous works. The algorithmic ideas proposed and discussed for the prime $2^m - \delta$ can be extended for the primes of the form $2^m + \delta$. Similarly, the ideas proposed for the Goldilocks prime $2^{448} - 2^{224} - 1$ can be used to design and analyze algorithms for the primes of the form $2^{2m} - 2^m - 1$. An example of a similar prime is $2^{480} - 2^{240} - 1$.

The Montgomery ladder is used to compute the shared secret over Montgomery curves through variable-base scalar multiplication. The ladder primarily consists of a ladder-step which contains a sequence of field operations. We have designed an efficient algorithm of the Montgomery ladder which vectorizes the field-operations of the ladder-step with 4-way vector instructions. Along with this we also propose a 4-way vectorized ladder which can compute the fixed-base scalar multiplication. Our designs are applicable to any Montgomery curve and efficient constant-time implementations of the ladder have been done for all the seven Montgomery curves considered in this work.

Second, we have proposed new Montgomery curves and Kummer lines at the 128-bit, 224-bit and 256-bit security levels and have made a wide range of sequential and vectorized implementations of the ECDH protocol over the proposed curves. For a very tight comparison of our curves we have also implemented the ECDH computation over the standard curves Curve25519 and Curve448. At 128-bit security level we provide Montgomery curves and Kummer lines which outperform the speed-performance of Curve25519 for shared secret computation. At 224-bit security level the sequential implementation of the proposed curve performs better than the sequential implementation

of Curve448 but the vectorized implementation is found to be slower. The ECDH computation over the Kummer lines at 128-bit security level has been found to be largely better than the previous work. Overall, the computation of ECDH over Kummer lines have been found to be better than Montgomery curves. We have made the source codes of all our implementations publicly available for use and further research.

9.2 Future Work

Efficient key-generation and signature generation/verification can also be done on the curves which have been proposed in this thesis. Finding out how the new curves perform with respect to the existing state-of-art while computing these cryptographic primitives is left out for future work. The results of prime field arithmetic from this thesis can also be applied in other areas like, pairing-based-cryprography and isogeny-based-cryptography where primes of similar shapes are employed in the underlying design.

Bibliography

[ABGR13] Diego F. Aranha, Paulo S. L. M. Barreto, C. C. F. Pereira Geovandro, and Jefferson E. Ricardini. A note on high-security general-purpose elliptic curves. *IACR Cryptology ePrint Archive*, 2013:647, 2013.

[Aum] Jean-Philippe Aumasson. Guidelines for low-level cryptography Software. Available at `https://github.com/veorq/cryptocoding`. Accessed on 25 February, 2021.

[AVX] AVX. Advanced Vector Extensions. Available at `https://en.wikipedia.org/wiki/Advanced_Vector_Extensions`. Accessed on 28 February, 2021.

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, pages 389–405, 2008.

[BCC+15] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: A white paper for the black hat https://bada55.cr.yp.to. In Liqun Chen and Shin'ichiro Matsuo, editors, *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, volume 9497 of *Lecture Notes in Computer Science*, pages 109–139. Springer, 2015.

[BCHL16] Joppe W. Bos, Craig Costello, Hüseyin Hisil, and Kristin E. Lauter. Fast cryptography in genus 2. *J. Cryptology*, 29(1):28–60, 2016.

[BCL14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2014.

[BCLN16] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *J. Cryptogr. Eng.*, 6(4):259–286, 2016.

[BCLS14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer strikes back: New DH speed records. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2014. Code available at `https://github.com/floodyberry/supercop/tree/master/crypto_scalarmult/kummer/avx2`.

[BD20] Jean-Claude Bajard and Sylvain Duquesne. Montgomery-friendly primes and applications to cryptography. *IACR Cryptol. ePrint Arch.*, page 665, 2020.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012. Code for 5-limb implementation available at https://github.com/floodyberry/supercop/blob/master/crypto_sign/ed25519/amd64-51-30k and the code for 4-limb implementation available at https://github.com/floodyberry/supercop/tree/master/crypto_sign/ed25519/amd64-64-24k.

[Ber06a] Daniel J. Bernstein. Can we avoid tests for zero in fast elliptic-curve arithmetic? https://cr.yp.to/ecdh/curvezero-20060726.pdf, 2006. Accessed on March 10, 2020.

[Ber06b] 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.

[BHH+14] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 157–175. Springer, 2014.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013.

[BHLM01] Michael Brown, Darrel Hankerson, Julio César López-Hernández, and Alfred Menezes. Software implementation of the NIST elliptic curves over prime fields. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2001.

[BLa] Daniel J. Bernstein and Tanja Lange. Failures in NIST's ECC standards. http://koclab.cs.ucsb.edu/teaching/ecc/eccPapers/Bernstein2016.pdf, 2016.

[BLb] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to/equation.html, Accessed on March 10, 2020.

[BL07a] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.

[BL07b] Daniel J. Bernstein and Tanja Lange. Inverted Edwards coordinates. In Serdar Boztas and Hsiao-feng Lu, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 17th International Symposium, AAECC-17, Bangalore, India, December 16-20, 2007, Proceedings*, volume 4851 of *Lecture Notes in Computer Science*, pages 20–27. Springer, 2007.

[BL17] 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.

[BM17] Joppe W. Bos and Peter L. Montgomery. Montgomery arithmetic from a software perspective. *IACR Cryptology ePrint Archive*, 2017:1057, 2017.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.

[Braa] Brainpool. ECC standard. http://www.ecc-brainpool.org/ecc-standard.htm.

[Brab] ECC Brainpool. ECC Brainpool Standard Curves and Curve Generation. https://www.teletrust.de/fileadmin/files/oid/oid_ECC-Brainpool-Standard-curves-V1.pdf.

[BS12] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.

[BY19] Daniel Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, May 2019.

[CFA+05] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman and Hall/CRC, 2005.

[CGT+21] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In Orr Dunkelman and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography, 2020*, Lecture Notes in Computer Science, 2021. to appear.

[Cho15] Tung Chou. Sandy2x: New curve25519 speed records. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2015. Code available at https://tungchou.github.io/sandy2x/.

[CHS14] Craig Costello, Hüseyin Hisil, and Benjamin Smith. Faster compact diffie-hellman: Endomorphisms on the x-line. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2014.

[CL15] Craig Costello and Patrick Longa. FourQ: Four-dimensional decompositions on a Q-curve over the Mersenne prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2015. Code available at https://www.microsoft.com/en-us/download/details.aspx?id=52310.

[CLN15] Craig Costello, Patrick Longa, and Michael Naehrig. A brief discussion on selecting new elliptic curves. Technical Report MSR-TR-2015-46, June 2015. Position paper presented at the NIST Workshop on Elliptic Curve Cryptography Standards (http://www.nist.gov/itl/csd/ct/ecc-workshop.cfm).

[CN15] Craig Costello and Michael Naehrig. Isogenies between (twisted) Edwards and Montgomery curves. https://cryptosith.org/papers/isogenies_tEd2Mont.pdf, 2015. Accessed on 16 September, 2019.

[CS09] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the cell broadband engine. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, pages 368–385, 2009.

[CS18] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic - the case of large characteristic fields. *J. Cryptographic Engineering*, 8(3):227–240, 2018.

[Cur] Curve25519. Wikipedia page on Curve25519. https://en.wikipedia.org/wiki/Curve25519.

[Cur99] NIST Curves. Recommended elliptic curves for federal government use. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf, 1999.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions of Information Theory*, 22(6):644–654, 1976.

[DHH+15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptogr.*, 77(2-3):493–514, 2015.

[Edw07] Harold M. Edwards. A Normal Form for Elliptic Curves. *Bulletin of the American Mathematical Society*, 44:393–422, 2007.

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219. IEEE, 2019.

[FA17] Hayato Fujii and Diego F. Aranha. Curve25519 for the Cortex-M4 and beyond. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology - LATINCRYPT 2017 - 5th International Conference on Cryptology and Information Security in Latin America, Havana, Cuba, September 20-22, 2017, Revised Selected Papers*, volume 11368 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2017.

[FHD19] Armando Faz-Hernández, Julio López Hernandez, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, 45(3):25:1–25:35, 2019.

[FHLD19] Armando Faz-Hernández, Julio López, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, 45(3), July 2019.

[FL15] Armando Faz-Hernández and Julio López. Fast implementation of curve25519 using AVX2. In *LATINCRYPT*, volume 9230 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2015.

[FLS15] Armando Faz-Hernández, Patrick Longa, and Ana H. Sánchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on GLV-GLS curves (extended version). *J. Cryptogr. Eng.*, 5(1):31–52, 2015.

[FPPR12] Jean-Charles Faugère, Ludovic Perret, Christophe Petit, and Guénaël Renault. Improving the complexity of index calculus algorithms in elliptic curves over binary fields. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2012.

[fSTa] National Institute for Standards and Technology. Digital signature standard. Federal Information Processing Standards Publication 186-4, 2013, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[fSTb] National Institute for Standards and Technology. Digital signature standard. Federal Information Processing Standards Publication 186-2, 2000, https://csrc.nist.gov/csrc/media/publications/fips/186/2/archive/2000-01-27/documents/fips186-2.pdf.

[GK15] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptogr. Eng.*, 5(2):141–151, 2015.

[GL09] P. Gaudry and D. Lubicz. The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009.

[GS12] Pierrick Gaudry and Éric Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012.

[GS15] Robert Granger and Michael Scott. Faster ECC over $\mathbb{F}_{2^{521}-1}$. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2015. Code available at http://indigo.ie/~mscott/ws521.cpp and http://indigo.ie/~mscott/ed521.cpp.

[Ham15] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive*, 2015:625, 2015.

[Has36] H. Hasse. Zur theorie der abstrakten elliptischen funktionenkorper. i, ii and iii. *Crelle*, 175, 1936.

[HEY20] Hüseyin Hisil, Berkan Egrice, and Mert Yassi. Fast 4 way vectorized ladder for the complete set of montgomery curves. *IACR Cryptology ePrint Archive*, 2020:388, 2020.

[HKM09] Darrel Hankerson, Koray Karabina, and Alfred Menezes. Analyzing the galbraith-lin-scott point multiplication method for elliptic curves over binary fields. *IEEE Trans. Computers*, 58(10):1411–1420, 2009.

[HMV03] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg, 2003.

[HS13] Michael Hutter and Peter Schwabe. Nacl on 8-bit AVR microcontrollers. In Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013.

[HWCD08] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, pages 326–343, 2008.

[K63] A. Karatsuba and Yu. Ofman (1962). Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences. 145: 293-294. Translation in the academic journal Physics-Doklady*, 7:595–596, 1963.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, 1987.

[Kob89] Neal Koblitz. Hyperelliptic cryptosystems. *J. Cryptology*, 1(3):139–150, 1989.

[KS20] Sabyasachi Karati and Palash Sarkar. Kummer for genus one over prime-order fields. *J. Cryptol.*, 33(1):92–129, 2020.

[LH16] Adam Langley and Mike Hamburg. Elliptic curves for security. Internet Research Task Force (IRTF), Request for Comments: 7748, https://tools.ietf.org/html/rfc7748, 2016. Accessed on 16 September, 2019.

[LS14] Patrick Longa and Francesco Sica. Four-dimensional gallant-lambert-vanstone scalar multiplication. *J. Cryptol.*, 27(2):248–283, 2014.

[ma] CFRG/IETF mail archive. https://mailarchive.ietf.org/arch/msg/cfrg/LQIyeCFGgoROzsx_UBf9cjlsS-A.

[Mil85] 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.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[Moo15] Andrew Moon. Implementations of a fast elliptic-curve diffie-hellman primitive. https://github.com/floodyberry/curve25519-donna/tree/2fe66b65ea1acb788024f40a3373b8b3e6f4bbb2, 2015.

[MOV93] Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Inf. Theory*, 39(5):1639–1646, 1993.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NS19] Kaushik Nath and Palash Sarkar. Reduction modulo $2^{448} - 2^{224} - 1$. *IACR Cryptology ePrint Archive*, 2019:1304, 2019.

[NUM] Nums: Nothing up my sleeve. https://tools.ietf.org/html/draft-black-tls-numscurves-00.

[OGG13] 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.

[OGGF12] E. Ozturk, J. Guilford, V. Gopal, and W. Feghali. 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.

[OLAR13] Thomaz Oliveira, Julio César López-Hernández, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2013.

[OLAR14] Thomaz Oliveira, Julio César López-Hernández, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *J. Cryptogr. Eng.*, 4(1):3–17, 2014.

[OLH+17] Thomaz Oliveira, Julio López, 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. Code available at https://github.com/armfazh/rfc7748_precomputed.

[OLR16] Thomaz Oliveira, Julio César López-Hernández, and Francisco Rodríguez-Henríquez. Software implementation of koblitz curves over quadratic fields. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 259–279. Springer, 2016.

[Pol78] John M. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.

[PPF+20] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 983–1002. IEEE, 2020.

[Res10] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. http://www.secg.org/sec2-v2.pdf, 2010.

[SA98] T. Satoh and Kiyomichi Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Commentarii Math. Univ. St. Pauli.*, 47:81–92, 1998.

[SEC] Optimized c library for ec operations on curve secp256k1. https://github.com/bitcoin-core/secp256k1.

[Sem98] Igor A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Math. Comput.*, 67(221):353–356, 1998.

[She21] Timothy Shelton. A note on "reduction modulo $2^{448} - 2^{224} - 1$". Cryptology ePrint Archive, Report 2021/804, 2021. https://ia.cr/2021/804.

[Sil86] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate texts in mathematics*. Springer, 1986.

[Sma99] Nigel P. Smart. The discrete logarithm problem on elliptic curves of trace one. *J. Cryptol.*, 12(3):193–196, 1999.

[Sol] J. A. Solinas. Generalized Mersenne numbers, Technical Report CORR 99-39, Centre for Applied Cryptographic Research, University of Waterloo. http://cacr.uwaterloo.ca/techreports/1999/corr99-39.ps, 1999.

[TP18] Version 1.3 TLS Protocol. RFC 8446. https://datatracker.ietf.org/doc/rfc8446/?include_text=1, 2018. Accessed on 16 September, 2019.

[Was08] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman and Hall/CRC, 2 edition, 2008.

[Zon] Intel Developer Zone. CADO-NFS - Crible Algbrique: Distribution, Optimisation - Number Field Sieve. Available at http://cado-nfs.gforge.inria.fr. Accessed on 25 February, 2021.