YAŞAR UNIVERSITY

GRADUATE SCHOOL

MASTER'S THESIS

SPEED-ORIENTED ELLIPTIC CURVE SCALAR MULTIPLICATION

BERKAN EGRICE

THESIS ADVISOR: ASSIST. PROF.(PH.D.) HÜSEYİN HIŞIL

COMPUTER ENGINEERING

PRESENTATION DATE: 11.08.2022

BORNOVA / İZMİR

AUGUST 2022

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Jury Members:	**Signature:**
Prof. (PhD) Institution	. . . . .
Prof. (PhD) Institution	. . . . .
Prof. (PhD) Institution	. . . . .
Prof. (PhD) Institution	. . . . .
Assoc. Prof. (PhD) Institution	. . . . .

Prof. (PhD) XX

Director of the Graduate School

ABSTRACT

Speed-oriented Elliptic Curve Scalar Multiplication

Egrice, Berkan
MSc, Computer Engineering
Advisor: Assist. Prof. Hüseyin HIŞIL, Ph.D.
August 2022

Because of the increasing importance of cyber security in today's world, modern cryptographic applications is to be secure and fast. The aim of thesis is to explore secure and fast implementation techniques of elliptic curve scalar multiplication, the main performance bottleneck of implementations in Elliptic Curve Cryptography. The studies cover two main streams of work. The first one is to pinpoint primes which can be used to provide maximum speed. The other is to develop parallel versions of celebrated Montgomery ladder algorithm.

In order to achieve the first aim, a mathematical framework that will parameterize fast primes is proposed. With the help of this framework, the overlooked prime $2^{261}-2^{131}-1$ which allows fast multiplication is found. It is shown with our implementation that this new prime performs faster than the popular primes $2^{251}-9$ and $2^{255}-19$, which makes it a good candidate for efficient implementations. In addition, a 9-limb representation of the prime $2^{255}-19$ is developed. The new representation performs well on processors with slower integer multiplication circuits.

In order to achieve the second aim, the first 4-way parallel version of the Montgomery ladder algorithm is proposed. The proposed algorithm is inert to changes in the elliptic curve parameters such as curve constants and the base point. This makes it a perfect candidate for variable-base variable-scalar multiplication. The algorithm finds immediate application in the contemporary SIMD processors.

ÖZ

Speed-oriented Elliptic Curve Scalar Multiplication

Eğrice, Berkan
Yüksek Lisans, Bilgisayar Mühendisliği
Danışman: Dr. Öğr. Üyesi Dr. Hüseyin HIŞIL
Ağustos 2022

Siber güvenliğin günümüz dünyasında artan önemi ile birlikte modern kriptografik uygulamalar güvenli ve hızlı olmalıdır. Bu tezin amacı, eliptik eğri tabanlı kriptografik uygulamaların darboğazı olan eliptik eğri skalar çarpma işleminin güvenli ve hızlı gerçekleştirilmesini sağlamaktır. Bu tezdeki çalışmalar iki ana eksende yürütülmüştür. Birinci safhada, en yüksek hızı sağlamak için kullanılabilecek asal sayılar belirlenmiştir. İkinci safhada ise, Montgomery merdiven algoritmasının paralel versiyonu geliştirilmiştir.

Birinci amaca ulaşmak için hızlı asal sayıları belirleyecek matematiksel bir çerçeve önerilmiştir. Bu çerçeve yardımıyla hızlı çarpmaya izin veren ve gözden kaçan $2^{261} - 2^{131} - 1$ asal sayısı bulunmuştur. Bu yeni asalın, popüler $2^{251} - 9$ ve $2^{255} - 19$ asal sayılarından daha hızlı performans gösterdiği gerekçeleriyle gösterilmiştir. Ek olarak, $2^{255} - 19$ asalının 9 basamaklı bir temsili geliştirilmiştir. Yeni gösterim, daha yavaş tamsayı çarpma devrelerine sahip işlemcilerde iyi performans göstermektedir.

İkinci amaca ulaşmak için Montgomery merdiven algoritmasının ilk 4 yönlü paralel versiyonu önerilmiştir. Önerilen algoritma, eğri sabitleri ve taban noktası gibi eliptik eğri parametrelerindeki değişikliklerden bağımsız çalışmaktadır. Bu da, önerilen metodu değişken-taban değişken-skalar çarpma işlemi için iyi bir aday yapmaktadır. Algoritma, güncel SIMD işlemcilerinde uygulama bulmuştur.

ACKNOWLEDGEMENTS

First and foremost, I have to thank my research supervisors, Asst. Prof. Hüseyin Hışıl. Without their assistance and dedicated involvement in every step throughout the process, this thesis would have never been accomplished.

I would also like to show gratitude to my committee, including Asst. Prof. Erdem Alkım and Asst. Prof. Arman Savran.

Finally, I would like to express my deepest gratitude to my mother Güldem and my father Recai for their invaluable and endless support.

Berkan Egrice

İzmir, 2022

TEXT OF OATH

I declare and honestly confirm that my study, titled "Speed-oriented Elliptic Curve Scalar Multiplication" and presented as a Master's Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Berkan Egrice

Signature:

_____

August 11, 2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF CODES

CHAPTER 1
INTRODUCTION

Our lives changed dramatically in the past few decades with the advances in technology. The biggest role in this is, undeniably, the spread and development of the Internet. In the late 90's, the Internet has been involved in our lives and started shaping our lives everlastingly. This revolutionary change, however, also came with the era of security threats. It is far more important nowadays to keep personal information secure e.g. in online shopping, banking, social media interaction. With the emergence of threats, the efforts to reduce this risky side of the internet, which can be harmful to users, gained importance. As a result of the studies carried out in this direction, the security protocols that are currently in our lives were developed. The most famous of these, TLS (Transport Layer Security), ensures that the transactions we perform on the Internet are secure. Protocols take their importance from the reliability of the encryption systems behind them. Since late 90's various cryptography systems has been developed. AES (Rivest, Shamir, & Adleman, 1978) and Blowfish (Schneier, 1994) can be given as examples. However, in the context of this thesis, we focus only on public-key cryptography. In short, public-key encryption is an encryption system that uses a pair of keys. These keys are called public and private keys. Mathematically, public-key encryption uses a one-way trap-door function to provide this property. The following scenario is examined in order to explain such kind of encryption. For example, in scenario, the sender uses a public key which is published by the receiver in order to encrypt a secret message. This encrypted message can be transmittable on an untrusted channel. The recipient receives the message encrypted with the recipient's public key. The decryption of this message will be performed using the recipient's private key. There are several different techniques developed to realize Public Key Cryptography. These include the Diffie-Hellman (Diffie & Hellman, 1976) key exchange protocol, ElGamal, Digital Signature Algorithm and more. The Public-key encryption can be implemented using different techniques. In the context of the thesis, we are interested in Elliptic Curve Cryptography (ECC). Elliptic Curve Cryptography was proposed by (Koblitz, 1987) and (Miller, 1985). Cryptography relies on the power of number theory. ECC is no exception. As with other techniques, basic operations such as addition and multiplication play a significant role in their implementation. In this respect, there were many pioneering studies existed over the years. However, this thesis provides the establishment of an improvement over ECC with a focus on speed. More details are

given in the following sections, especially in Chapter 3 that expresses the basic idea of this thesis.

1.1. Motivation

An elliptic curve group is the building block of ECC. One typically fixes a finite field and then fixes an elliptic curve defined over this field. The elliptic curve is selected as to have a large prime order subgroup in which the group operation, namely the elliptic curve point addition, is computed. At this point, it is natural to ask how fast we can add points on an elliptic curve. A vast amount of work has been done in this direction. Two major research areas are: (i) speeding up the underlying field arithmetic, (ii) optimizing the point addition formulas. The main theme of this work is to focus on the first item. In particular, our motivations is to find/rework cryptographically interesting and fast prime(s) for ECC, and implement their arithmetic on modern processors. Speeding up the field arithmetic contributes to the performance of ECC applications eventually.

The most crucial step in building up fast arithmetic on finite fields is to define a fast way of integer multiplication. Various algorithms are available for performing fast integer multiplication, such as FFT, Karatsuba, Toom-Cook, and Schoolbook (naive-method). Within the scope of this thesis, we were mainly interested in schoolbook and Karatsuba methods. When implementing these algorithms, two important decisions are to be made:

- the number of limbs[1] to be used to represent a field element on the registers/memory of a processor,
- the number of bits that will reside inside each limb

We study all these technical decisions in detail in order to optimize field multiplication.

1.2. Aims and outcomes

This thesis aims to provide answers to the following research questions.

- TLS protocol uses two celebrated primes `p25519` and `p448`. Can we develop new and faster representations of these primes on modern processors?
- Perhaps there are overlooked primes which can be faster. Can we define a framework to look for such primes? Can we find any primes that can be faster than `p25519` and `p448` (in their context)?

[1] The word 'limb' comes from the nomenclature of the GNU GMP library. A limb is a digit of a number in radix representation.

The study on these questions led to the following outcomes.

- We developed a 4-way vectorizable Montgomery ladder step which suits well with processors with SIMD support such as processors having the AVX2 instruction extension. This outcome is not directly addressing the aforementioned research questions but the nature of research led to finding such a side product which is probably more interesting than having concrete answers to those questions.
- A 9-limb representation of elements of $GF(2^{255} - 19)$ is proposed. This new representation uses one less number of limb in comparison with the common 10-limb representation. The new method have potential to be competitive when used inside elliptic curve variable-point variable-scalar multiplication. No better representation of `p448` was found than the common 16-limb representation. These outcomes answers the first research question.
- A framework to find fast primes is found. Using this framework a new cryptographically interesting prime is discovered, which is $2^{261} - 2^{131} - 1$. The new prime has a distinctive property that it is Karatsuba-friendly. That is several optimizations can be made when Karatsuba multiplication method is employed over the field arithmetic. We present a fast elliptic curve variable-point variable-scalar multiplication implementation using this field. These outcomes answers the second research question. We admit that we were extremely lucky to find the Karatsuba friendly prime $2^{261} - 2^{131} - 1$. We present a 10-limb representation of this prime and implemented its arithmetic on a processors with AVX2 support.
- We searched and found cryptographically interesting elliptic curves defined over $GF(2^{261} - 2^{131} - 1)$. This part of the work is done to show the usefulness of our prime $2^{261} - 2^{131} - 1$ in the context of ECC.

The construction of the thesis was created in line with the desire to answer the above questions.

1.3. Outline of the thesis

The thesis structured is given as follows. Chapter 2 presents basic algorithms on how to perform multi-digit integer arithmetic in radix representation. Chapter 2 is informative. Readers who are familiar with topics such as radix representation, Karatsuba multiplication, Montgomery reduction, etc. can skip to other chapters. Yet, the content forms the basis of proceeding chapters. On the other hand, not all presented algorithms are used. Some other are used with modifications which are made clear in place. Chapter 3 is the heart of this thesis. This chapter includes the main idea of our thesis. This chapter has detailed valuable examples concerning chosen technique against the used prime

numbers properties. Also, the result of this chapter had expressed in a separate article given in Chapter 5. Chapter 4 represent a 4 way vectorization of the complete set of Montgomery curves. Chapter 4 is produced from a published article which is one of the main outcomes of this thesis. Chapter 4 represents a new way doing arithmetic over `p25519`. Chapter 5 is also produced from an article which is recently submitted for publication and is also one of the main outcomes of this thesis. Chapter 5 introduces the Karatsuba-friendly prime $2^{261} - 2^{131} - 1$ and shows its usefulness in the context of fast elliptic curve scalar multiplication.

CHAPTER 2
BACKGROUND

Applications of public-key cryptography are built on number theoretic constructions. One needs the arithmetic of large integers in order to realize such constructions. In order to implement basic arithmetic operations on large integers, one needs special algorithms that breaks down the large integer inputs into smaller chunks each of which can fit into registers residing inside a processor and then accomplish certain sequence of tasks that yields the desired output. Therefore, it is important to decide on how to represent integers on a processor. The most common one is the so-called radix representation where a non-negative integer a is represented by the sequence $A = [A_0, A_1, ..., A_{t-1}]$ satisfying

$$a = \sum_{i=0}^{t-1} A_i 2^{W_i}$$

for some W which is selected with respect to

1. The capabilities and restrictions of the underlying hardware:
    - register size: contemporary processors has either of 8, 16, 32, 64 bit registers. This size can grow further in processors with SIMD[1] support. More details will be provided on SIMD implementation in Chapters 4 and 5.
    - integer multiplier: some hardware can produce full-product e.g $32\times32 \rightarrow 64$ or $64 \times 64 \rightarrow 128$ bit multipliers. Some others produces higher and lower halves of the product in separate instructions. Yet, some others contain both. One typically uses the largest multiplier circuit available provided that the latency and throughput of that specific instructions is reasonable.
    - carry bit handling: it is reasonable to reduce dependency between instructions in order to achieve a better pipelining at executing time. Therefore, one should prevent carry bit propagation as much as possible. Some hardware such as SIMD processors do not even have instruction that allow handling overflows. As a solution to all, it is beneficial to select W to be smaller than the actual maximum bound supported by the underlying hardware. The "empty space" in each limb is then called nail bits. These nail bits typically varies between 1 to 7 depending on the implementation and usually have the

[1] Single Instructions Multiple Data

side benefit of providing extra space for multiplication with small constants when necessary.

2. The nature of cryptographic implementation: the shape of the chosen field arithmetic may structurally suits well with some instruction sets much better than others. Therefore, the process of deciding which type of instruction set to use in cryptographic applications plays an important role in performance.

Now, we present elementary algorithms for performing integer arithmetic. We closely follow the exposition in (Hankerson, Menezes, & Vanstone, 2003). We reproduce the algorithms in the notation of this thesis in order to keep the text self-contained. In the process, we used Magma language to present algorithms in ready to go code format. This makes it easy to present the algorithms. We start by providing auxiliary functions which are called by higher level routines. These algorithms are essential for computing with large integers. On the other hand, we never use any of these algorithms in the way they are proposed. Speed implementations requires several tweaks on such algorithms. Our modifications are presented in detailed in Chapter 3, 4 and 5.

2.1. Simulating instructions

The algorithms in this thesis are presented as scripts prepared in Magma language. Each digit of the numbers represented in these scripts must be `W`-bit delimited for some `W` which is determined with respect to technical properties of the underlying hardware. Our aim is not to run these codes on Magma but they provide useful prototypes before low-level implementation. There is no upper bound for integers defined in the Magma language. These integers can grow as large as the memory is sufficient. This situation deviates from the technical properties of the target hardware. In order to emulate the target hardware, we define the additional functions presented below. These additional auxiliary functions have given us the opportunity to develop using Magma as if we were developing at a low-level language like C. For this reason, these additional functions are used in all algorithms in the thesis.

The indices in Magma language starts from 1 whereas, the indices in C language starts from 0. In order to make the two worlds compatible with each other we do the following tweak given in Code 2.1. Now, for instance, the index `i` of an array, say `A`, can be accessed in the syntax `A[__+i]` which pretty much looks a C code as soon as `__+` is omitted.

The global variable `W` is used to define a radix-basis for arithmetic operations. In the remainder of thesis, `W` is fixed to 64 since contemporary processors use 64 bit registers. For other scenarios, one can simply alter the value of `W`.

```
__:=1;
```

Code 2.1. C-style indexing in Magma

```
W:=64;
```

Code 2.2. Radix base

These additional auxiliary functions are designed to realize *add* and *adc* instructions which are common most hardware. Code 2.3 is able to produce a sum of given 2 one-word integer on radix-basis W, and Code 2.4 handles the carry bit that can occur when adding two one-word integer of size $2^W - 1$. Thoroughly, Code 2.3 takes two integer values as the input, and then adds them by using built-in + (add) operator. To get the carry bit, the result of the addition is divided by 2^W, the division result will be CF[2]. The resulting CF is used to provide the portion of the sum that exceeds $2^W - 1$. The auxiliary function add returns CF and the sum. The Code 2.4 is works as same as Code 2.3, additionally adds the CF flag while performing addition.

```
add := function (a0, a1)
  zz := a0+a1;
  CF := zz div 2^W;
  z0 := zz-CF*2^W;
  return CF, z0;
end function;
```

Code 2.3. Single digit addition without carry

```
adc := function (CF, a0, a1)
  zz := a0+a1+CF;
  CF := zz div 2^W;
  z0 := zz-CF*2^W;
  return CF, z0;
end function;
```

Code 2.4. Single digit addition with carry

Similarly, Code 2.5 and 2.6 are created to perform *sub* and *sbb*. The subtraction function takes two integers as input, then returns the borrow bit and the difference. We note that negative integers are represented in their two's complement form. The Code 2.5 finds the difference using the mod operator. Here the mod operator is responsible of masking the positive outputs and taking the two's complement form. The carry flag (CF) is determined by logical controls which checks whether the first input is smaller than the second. The Code 2.6 differs from Code 2.5 only where the subtraction occurs, the difference being that the CF bit value passed to sbb is subtracted from the subtraction result.

```
sub := function (a0, a1)
  z0 := (a0 - a1) mod 2^64;
  CF := 0;
  if a0 lt a1 then
    CF := 1;
  end if;
  return CF, z0;
end function;
```

Code 2.5. Single digit subtraction without borrow

```
sbb := function (CF, a0, a1)
  z0 := (a0 - a1 - CF) mod
      2^64;
  CF := 0;
  if a0 lt a1 then
    CF := 1;
  end if;
  return CF, z0;
end function;
```

Code 2.6. Single digit subtraction with borrow

[2]Carry flag indicates when an arithmetic carry or borrow is created during the operation.

As described earlier, there is no size restriction for variables in the Magma language. Therefore, an auxiliary function(s) is needed to perform the multiplication, just like addition and subtraction. The Code 2.7 covers the built-in `* (mul)` Magma language operator. The helper `mul` function takes two one-word integers and multiplies them using the $\star$ operator. The result is divided by the base 2^W to get the lower part of the multiplication. The modulus operator is used for obtaining the high part of the multiplication. As a result, the Code 2.7 is returns `h` (high part) and `l` (low part) of the multiplication.

```
mul := function (a, b)
  z := a*b;
  h := z mod 2^W;
  l := z div 2^W;
  return h, l;
end function;
```

Code 2.7. Single digit multiplication

```
mulAddAdd := function (a, b,
    c, d)
  z := a*b+c+d;
  h := z mod 2^W;
  l := z div 2^W;
  return h, l;
end function;
```

Code 2.8. Single digit multiplication with two additions

Eventually, as explained earlier, the magma language is capable of doing arithmetic without the register size limitations. However, in the remainder of the thesis, each algorithm was developed with the assumption that this limitation exists. Therefore, these auxiliary functions were created to provide a link between Magma language and algorithmic requirements. As a result, the developed auxiliary functions are called in each of the following algorithms.

2.2. Addition and Subtraction

Adding the two numbers $a, b \in [0, 2^W)$ produces at most one bit of carry. Code 2.9 performs multi-digit integer addition of a and b, finally returning the `CF` (carry bit flag) and the result of addition. The algorithm 2.9 iterates through each limb of the given arrays, starting at index 1. The values of each iterated array are passed to the `adc` function 2.4, and returned CF variable continues to be used in the further iteration. The index 0 is specially processed, these index values are added using the `add` function 2.3 and the CF output is passed to the running `adc` process for the second index.

```
addn := function(a, b, t)
  c := [];
  CF, c[__+0] := add(a[__+0], b[__+0]);
  for i:=1 to t do
       CF, c[__+i] := adc(CF, a[__+i], b[__+i]);
  end for;
  return CF, c;
end function;
```

Code 2.9. Multiprecision addition

Multi-digit integer subtraction is performed using Code 2.10. It has a similar logic with Code 2.9. The only difference in this algorithm is the `CF` bit is called `BF` (Borrow bit flag).

```
subn := function(a, b, t)
  c := [];
  BF, c[__+0] := sub(a[__+0], b[__+0]);
  for i:=1 to t do
       BF, c[__+i] := sbb(CF, a[__+i], b[__+i]);
  end for;
  return BF, c;
end function;
```

Code 2.10. Multiprecision subtraction

In order to perform the operations in the algorithms that will be explained in the following parts of the thesis, auxiliary functions that can add and subtract in the n-limb are generated.

2.3. Integer multiplication

The total running time of most cryptographic primitives are dominated by the speed of unsigned integer multiplications. Therefore, one needs the fastest integer multiplication available. At this point, we start with a basic algorithm, namely schoolbook multiplication, with $O(n^2)$ running time were $n = \log_2(\max(a, b))$. The Code 2.11 provides an operand scanning strategy to multiply two unsigned integers. It also uses the `mulAddAdd` auxiliary function explicitly described in Code 2.8; this is developed assuming that the function never produces an output bigger than two words in base W, because $(W-1).(W-1)+(W-1)+(W-1) = (W^2-1)$. It cannot exceed two digits in base W.

This algorithm is handy for small sized inputs (i.e. inputs with a few digits, say up to 8 digits). The algorithm suits well with the cases where there is no nail bits in the registers (i.e. W is equal to the underlying register bit size). However, an implementer should be warned that the algorithm presented in Code 2.11 comes with the expense of long data dependencies. Therefore, using Code 2.11 may not be an optimal choice on pipelined processors.

```
integer_mul_osf := function(A, B, n, t)
  C := [];
  for i:=0 to (n+t+1) do
    C[__+i] := 0;
  end for;

  for i:=0 to t do
    U := 0;
    for j:=0 to n do
      U, V := mulAddAdd(A[__+j], B[__+i], C[__+i+j], U);
      C[__+i+j] := V;
    end for;
    C[__+i+n+1] := U;
  end for;

  return C;
end function;
```

Code 2.11. Integer multiplication (operand scanning form)

In this case, an alternative algorithm is given in Code 2.12. This algorithm uses a product scanning technique with less data dependency. We also note that the need for register R_2 in Code 2.12 is eliminated when there is enough nail bits. We always introduce nail bits and prefer the latter algorithm for implementations in this thesis.

```
integer_mul_psf := function(A, B, t)
  C := [];
  R_0 := 0; R_1 := 0; R_2 := 0;
  for k := 0 to 2*t-2 do
    for l in { car< Integers(), Integers() > | <i, j> : i in
        [0..t-1], j in [0..t-1] | i+j eq k } do
      U, V := mul(A[__+l[1]], B[__+l[2]]);
      E, R_0 := add(R_0, V);
      E, R_1 := adc(E, R_1, U);
      E, R_2 := add(R_2, E);
    end for;
    C[__+k] := R_0;
    R_0 := R_1;
    R_1 := R_2;
    R_2 := 0;
  end for;
  C[__+2*t-1] := R_0;
  return C;
end function;
```

Code 2.12. Integer multiplication (product scanning form)

As the number of digits grow, it is beneficial to switch to an asymptotically faster multiplication algorithm namely 2-way Karatsuba multiplication with $O(n^{\log 3/\log 2})$ running time. Most applications of Elliptic Curve Cryptography use 4 up to 20 digits to represent integers involved in arithmetic operations. The number of digits are determined with respect to the size of the prime that is used in that particular application and the register size on the target hardware. Karatsuba multiplication can be preferable when digits are more than or equal to 8. This is of course not a strict bound. An implementer should try both Schoolbook and Karatsuba multiplications and pick the fastest one. In comparison, Schoolbook multiplication (with product scanning) is less register hungry than Karatsuba multiplication. Therefore, multi-stage pipelining favors Schoolbook multiplication more in most implementations. Again, this is not a strict

inference. On the other hand, processors with slow multipliers and fast adders are likely to benefit more from Karatsuba method. Multiplication algorithms of even lower asymptotic complexity do exist. Typical examples are k-way Karatsuba multiplication ($k > 2$), Toom-Cook multiplication, and multiplication with Fast Fourier Transform. However, we do not use any of these methods as they are only interesting for primes which are larger than the primes used in Elliptic Curve Cryptography. We note that 3-way Karatsuba multiplication can still be interesting for large primes of Elliptic Curve Cryptography, e.g. primes having 448 up to 512 bits. We leave such an investigation as a future work because our primary focus is on the primes having 250 up to 270 bits. Also, we did not find any convenient prime numbers in this range using our proposed systematic search methodology.

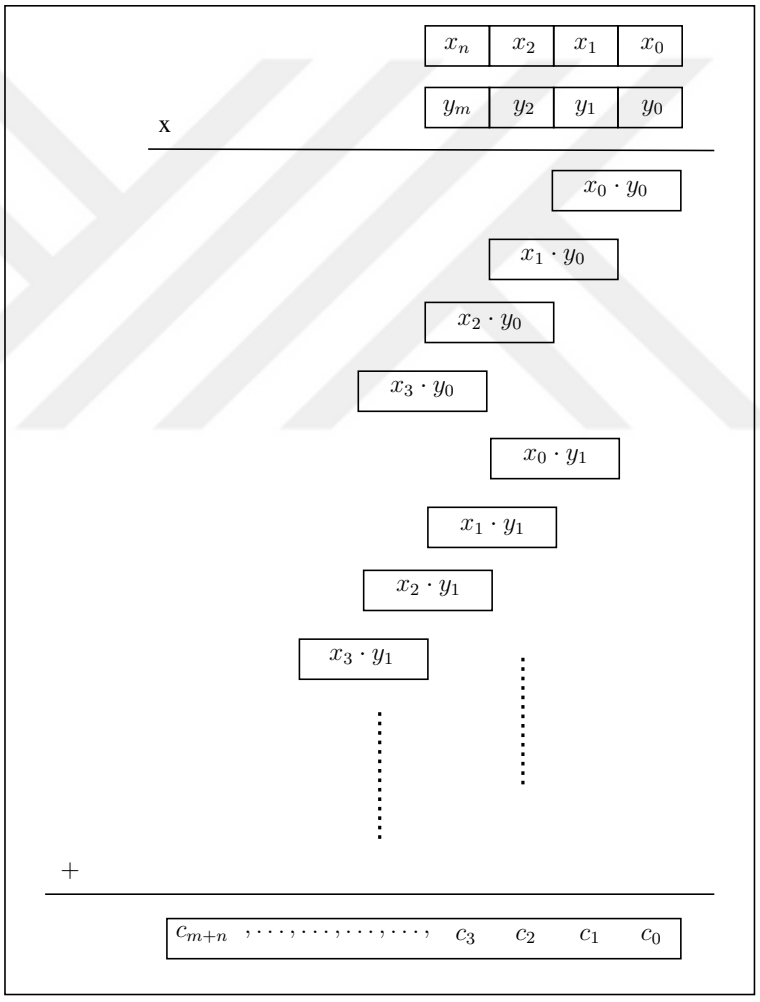

Figure 2.1. Schoolbook Multiplication Scheme(Operand scanning form)

Figure 2.1 shows the n-limb schoolbook multiplication algorithm for the Code 2.11 implementation.

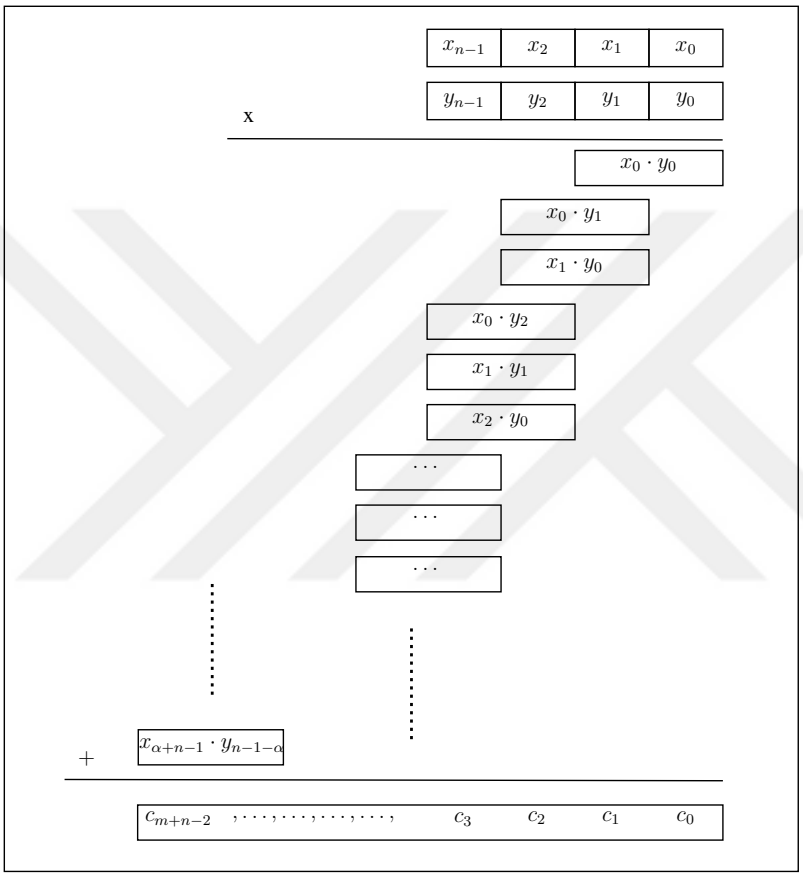

Figure 2.2. Schoolbook Multiplication Scheme(Product scanning form)

Also, figure 2.2 shows the multiplication algorithm for the Code 2.12 implementation, which also requires less data dependency.

Example

Let $a = 9274$ and $b = 847$ at base 10. Table 2.1 shows the algorithm values generated when executing Code 2.11 for each iteration. See Section 2.3

Table 2.1.. Multi-digit integer multiplication

i	j	$c_i + j + a_j \cdot b_i + U$	U	V	C_6	C_5	C_4	C_3	C_2	C_1	C_0
0	0	$0 + 28 + 0$	2	8	0	0	0	0	0	0	8
	1	$0 + 49 + 2$	5	1	0	0	0	0	0	1	8
	2	$0 + 14 + 5$	1	9	0	0	0	0	9	1	8
	3	$0 + 63 + 1$	6	4	0	0	6	4	9	1	8
0	0	$1 + 16 + 0$	1	7	0	0	6	4	9	7	8
	1	$9 + 28 + 1$	3	8	0	0	6	4	8	7	8
	2	$4 + 8 + 3$	1	5	0	0	6	5	8	7	8
	3	$6 + 36 + 1$	4	3	0	4	3	5	8	7	8
0	0	$8 + 32 + 0$	4	0	0	4	3	5	0	7	8
	1	$5 + 56 + 4$	6	5	0	4	3	5	0	7	8
	2	$3 + 16 + 6$	2	5	0	4	5	5	0	7	8
	3	$4 + 72 + 2$	7	8	7	8	5	5	0	7	8

Karatsuba Algorithm

As explained in Section 2.3, the Karatsuba algorithm generally performs better when limb sizes exceed a limit, usually when there are more than 8 to 20 limbs. The Karatsuba algorithm was first discovered in 1962 by Karatsuba and Ofman (Karatsuba & Ofman, 1962). The Karatsuba algorithm performs the product of two given integers using divide-and-conquer approaches. Within the idea of the divide and conquer technique, the Karatsuba algorithm divides the required multiplication by two then perform three multiplication instead four. As a final step, the final multiplication is operated to calculate the product. The Equation 2.1 shows the steps of how the Karatsuba algorithm works for integers u, v of n-bit length.

$$\begin{aligned} a &= u_H v_H \\ d &= u_L v_L \\ e &= (u_H + u_L)(v_H + v_L) - a - d \\ uv &= uv^n + eb^{\frac{n}{2}} + d \end{aligned} \tag{2.1}$$

Each sub-multiplying can be processed repeatedly using Karatsuba techniques, but is not necessary in the context of this thesis because of the length of the multiplied variables. Since the Karatsuba technique adds additional operations to perform itself, it will be computationally more costly to use the Karatsuba technique multiple times when the multiplied variable can be defined directly on the register on the target hardware. The reason for this decision will be understood more clearly when Chapter 3, the core of this thesis, is explained. However, to complete the explanation of the Karatsuba algorithm, Code 2.13 of the naive implementation of the Karatsuba algorithm, which is recursive in nature, is given.

```
karatsuba_algorithm := function(u, v)
  if u lt 2^W and v lt 2^W then
    return u*v;
  end if;
  n := Max(Ceiling(Log(2^W, u)), Ceiling(Log(2^W, v)));
  m := Ceiling(n div 2);
  u_H := u div (W^m);
  u_L := u mod (W^m);
  v_H := v div (W^m);
  v_L := v mod (W^m);
  //recursive steps
  a := karatsuba_algorithm(u_H, v_H);
  d := karatsuba_algorithm(u_L, v_L);
  e := karatsuba_algorithm(u_H + u_L , v_H + v_L) - a - d;
  return (a*(W^(m*2)) + e*(W^m) + d);
end function;
```

Code 2.13. Karatsuba Algorithm

Note that we do not use the Karatsuba technique as implemented in a 2.13 due to speed optimization concerns. The version of the Karatsuba technique used in this thesis will be examined in the next sections.

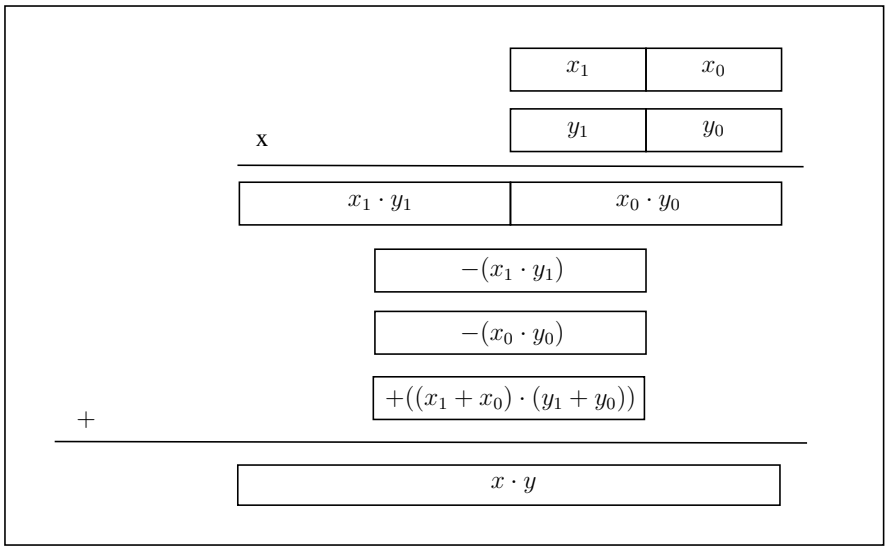

Figure 2.3. Karatsuba Multiplication Scheme

Figure 2.3 illustrates the two-way Karatsuba algorithm approach, also this figure follows the same approach as Code 2.13.

2.4. Integer squaring

As same as the field multiplication $a, b \in \mathbb{Z}^+$, the field squaring $a \in \mathbb{Z}^+$ performs by first taking squaring a, and then reducing the result modulo p. Code 2.14 shows the naive implementation of the field squaring a, the reducing the result modulo p.

```
integer_squaring := function(A, t)
  C := [];
  R_0 := 0; R_1 := 0; R_2 := 0;
  for k := 0 to 2*t-2 do
    for l in { car< Integers(), Integers() > | <i, j> : i in [0..t
        -1], j in [0..t-1] | i+j eq k } do
      U, V := mul(A[__+i], A[__+j]);
      if i lt j then
        E, V := add(V, V);
        E, U := adc(E, U, U);
        R_2 := add(R_2, E);
      end if;
      E, R_0 := add(R_0, V);
      E, R_1 := adc(E, R_1, U);
      E, R_2 := add(R_2, E);
    end for;
    C[__+k] := R_0;
    R_0 := R_1;
    R_1 := R_2;
    R_2 := 0;
  end for;
  return C;
end function;
```

Code 2.14. Integer squaring

The same concern with using Code 2.11 and Code 2.12 for integer multiplication applies to using Code 2.14 for integer squaring. As with integer multiplication, the Karatsuba algorithm described earlier in section 2.3 may also perform better at squaring integers. As with integer multiplication in Code 2.11 and Code 2.12, the way the squaring algorithm presented is for informational purposes only.

2.5. Reduction

The integer multiplication of $0 \leq a, b < p$ is most likely to produce a result greater than p. Therefore, the operation $(ab) \bmod p$ requires a reduction. Thus, there are two methodologies exist to performs a field multiplication, one of is realize a multiplication then reduce. The following multiplication algorithms strictly obey that methodology.

The schoolbook division (aka long division) algorithm involves frequent access to the division instruction (assuming that such a division instruction is present in the target hardware). We refer the reader to Knuth (Knuth, 1997), for an exposition of

long division algorithm. However, any use of division instruction is not preferred in cryptographic applications for two reasons: (i) the latency of the division instruction is high and the throughput is low. (ii) the instruction does not take constant time. Therefore, implementation of Long division was not implemented using Magma. However, to work around these problems, we have four alternatives: namely, (i) binary reduction; (2) Barrett reduction; (3) Montgomery reduction; (4) Special moduli.

We now explain these techniques noting that the special moduli technique is at the heart of this thesis.

2.5.1. Binary Reduction

The binary reduction can be used when a division instruction is not present in the hardware. The algorithm uses just basic linear operations such as shifts and additions. One reason to exclude this algorithm from cryptographic applications is that it is practically very slow. Yet another is that it is not constant time. For positive integers z and p, Code 2.15 produces a $z \bmod p$, following the Binary reduction algorithm.

```
RedBin:=function(z,p)
   pl:=Ceiling(Log(2,p)); // precomputed
   while z gt p do
      zl:=Ceiling(Log(2,z));
      q:=2^(zl-pl);
      t:=q*p; // Use muln
      if z lt t then
         t:=t/2;
      end if;
      z:=z-t;
   end while;
   return z;
end function;
```

Code 2.15. Binary Reduction

2.5.2. Barrett Reduction

Barrett reduction is a general purpose reduction algorithm introduced in 1986 by P. D. Barrett (Barrett, 1987). The algorithm is more efficient than the long division if precomputation on the prime moduli is allowed. This usually the case when several multiplications are to performed with a fixed prime, just as in Elliptic Curve Cryptography. In particular, Barrett reduction makes access to two multi-digit multiplications and some more linear operations. The first multi-digit multiplication requires access to higher half of the result while the second one accesses the lower half. Both of these half products can be optimized with lower level tweaks at the implementation. It can be proved that the while loop iterates at most 2 times, without causing a serious performance penalty.

```
b := 2^W;
k := Floor(Log(b, p)) + 1;
u := b^(2*k) div p;
q := (z div b^(k-1) * u) div b^(k+1);
r := (z mod b^(k+1)) - ((q * p) mod b^(k+1));
if r lt 0 then
  r := r + b^(k+1);
end if;
while r ge p do
  r := r - p;
end while;
return r;
```

Code 2.16. Barrett Reduction

The amount of work required for the amount of precomputation of the Barrett reduction adds a computational overhead that is easily negligible compared to modular multiplication. Also, note that in the Code 2.16 the `div` operation is used directly to perform the division, but in real life implementation these divisions are done using the `shift` operator.

2.5.3. Montgomery Reduction

Montgomery reduction algorithm proposed by P.L. Montgomery (P. L. Montgomery, 1985). The algorithm efficiently calculates $c = a \cdot b \pmod{n}$ where a, b and c are k-bit binary numbers. Like the Barrett reduction algorithm, division operations are replaced by simple shift operation. Let r be an integer such that $gcd(r, n) = gcd(2^k, n) = 1$, noting that n is always odd. The following steps are followed to define the Montgomery reduction algorithm.

Let a be an integer with property $a < n$, we can construct the following equation according to r as

$$\bar{a} = a \cdot r \bmod n.$$

It also equates to complete residue system. This property allows to develop a faster multiplication algorithm for two given integers in the n-residual system. Therefore, Montgomery multiplication with the given of two integer at n-residue, can be defined as:

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

r^{-1} shows the inverse of r modulo n, it's satisfies $r^{-1} \cdot r = 1 \pmod{n}$. If we use this property of modular inversion, we get

$$\begin{aligned} \bar{c} &= a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \\ &= a \cdot b \cdot r \pmod{n} \end{aligned}$$

To realize the Montgomery reduction additional properties are easily produced using

the extended Euclid algorithm, see 2.7.1, such as n' whose satisfies the property

$$r' \cdot r - n \cdot n' = 1.$$

The Montgomery modular multiplication inputs are accomplished using Code 2.17. To simplify the explanation of the algorithm, modular inversion is calculated using the built-in `Modinv` function instead of the Code 2.20 described in Section 2.7.1.

```
montgomery_reduction := function(z, p, R)
  c := 0;
  pd := Modinv(-p, R);
  c := c + (z + ((z*pd) mod R)*p) div R;
  if(c ge p) then
    c := c - p;
  end if;
  return c;
end function;
```

Code 2.17. Montgomery Reduction

Montgomery multiplication can be done using Code 2.18, this implementation also includes an correctness test written in Magma language. Built-in Magma functions were used to testing.

```
montgomery_multiplication := function(a, b, p, R)
    A := (a*R) mod p;
    B := (b*R) mod p;
    C := montgomery_reduction(A*B, p, R);
    c := (C*Modinv(R, p)) mod p;
    res := c eq (a * b) mod p;
    return res;
    end function;
```

Code 2.18. Montgomery Multiplication

Eventually, the Montgomery reduction is another general purpose reduction algorithm. It is specifically well-suited with modular exponentiation and elliptic curve scalar multiplication. Just like Barrett reduction, Montgomery reduction also requires the computation of two half products. Speed implementations always optimize these half products. For some specifically selected primes Montgomery reduction can be speed up significantly, see (Bos, Costello, Longa, & Naehrig, 2016).

2.5.4. Primes of a special modulus

Primes which are extremely close to a power of two are classified as special modulus. Such primes allows reduction with a few linear operations and sometimes multiplications by small constants. Some special primes are listed in Table 2.2.

Prime numbers listed in table 2.2 generally yield a better speed-oriented result because the nature of prime shapes allows arithmetic to be done using fewer instructions. This

Table 2.2.. List of selected primes of special modulus

p1271	$2^{127} - 1$
p192	$2^{192} - 2^{64} - 1$
p224	$2^{224} - 2^{96} + 1$
p2519	$2^{251} - 9$
p25519	$2^{255} - 19$
p256	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
p2663	$2^{266} - 3$
p384	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
p448	$2^{448} - 2^{224} - 1$
p521	$2^{521} - 1$

thesis makes another entry to table 2.2 with the prime $2^{261} - 2^{131} - 1$ the details are provided in Chapter 5.

2.6. Modular Exponentiation

Modular exponentiation, as the name suggests, performs exponentiation over a module. Modular exponentiation is one of the essential operations for cryptographic algorithms. This calculates $c = b^e \bmod p$, where $0 \leq c < p$. As we mentioned earlier, it is substantial to perform arithmetic operations rapidly for cryptographic algorithms. The same concern applies when calculating modular exponentiation, so various algorithms have been developed in this regard. The naive way to calculate modular exponentiation is to first calculate b^e for the given integers b, e, and then to compute this result in modulo p. The bit length of numbers significantly affects the computational speed. Therefore, this method becomes cumbersome as b and e increase further to provide better security.

2.6.1. Successive Squaring

This method provides a better algorithm for computing modular exponentiation. Unlike the naive way, it does not perform the reduction in modulo p after it produces the exponentiation result first. As a preliminary of this algorithm the exponent e should written as a binary form. This can be described as in 2.2

$$e = \sum_{i=0}^{n-1} a_i 2^i \tag{2.2}$$

In this form, n represents the bit length of e. Clearly, a_i can be either 0 or 1. In this

notation, modular exponentiation can be represented in 2.3

$$\prod_{i=0}^{n-1} b^{a_i 2^i} \pmod{p} \tag{2.3}$$

describing the operation in this notation indicates: It is adequate to perform the calculation only when $a_i \neq 0$. Code 2.19 has been developed in line with the method described.

```
successive_squaring := function(b, e, p)
   if p eq 1 then
      return 0;
   end if;
   res := 1;
   b := b mod p;
   while e gt 0 do
      if (e mod 2) eq 1 then
         res := (res*b) mod p;
      end if;
      b := (b * b) mod p;
      e := e div 2;
   end while;
   return res;
end function;
```

Code 2.19. Successive Squaring

The ability of Code 2.19 to perform faster arithmetic can also be useful for calculating the modular multiplicative inverse of an integer with Code 2.22.

2.7. Inversion

A modular multiplicative inverse of the integer a is an integer x represented by $ax \equiv 1 \pmod{p}$. The inverse of a can be simply noted as a a^{-1} or $a^{-1} \pmod{p}$ where the $a \in \mathbb{Z}$. There are several ways to calculate the inverse of a, different techniques (e.g., Extended Euclidean algorithm, Fermat's little theorem and Safe GCD) are studied in this section.

2.7.1. Extended Euclidean Algorithm

A Euclidean algorithm is useful for finding the greatest common divisor `(GCD)`. Simply calculates $d =$ `GCD(a,b)`, takes two integers number a, b where $a \leq b$, and then finds the number d, which is the greatest common divisor for a, b. The classical Euclidean algorithm can be modified to find x, y where the integers x, y are $ax + by = d$ and $d = gcd(a, b)$. The naive implementation of this algorithm is given in Code 2.20.

```
extended_gcd := function(a, b)
  u := a; v := b;
  x_1 := 1; y_1 := 0; x_2 := 0; y_2 := 1;
  while u ne 0 do
    q := v div u;
    r := v - q*u;
    x := x_2 - q*x_1;
    y := y_2 - q*y_1;
    v := u;
    u := r;
    x_2 := x1;
    x_1 := x;
    y_2 := y_1;
    y_1 := y;
  end while;
  d := v;
  x := x_2;
  y := y_2;
  return d, x, y;
end function;
```

Code 2.20. Extended GCD

2.7.2. Inversion with using the Extended Euclidean algorithm

Our aim is to compute the inverse of a, Code 2.20 can be used to calculate the inverse of a given integer such as a. Let p be prime and $a \in [1, p-1]$, and therefore $gcd(a, p) = 1$. Therefore, we have $ax + bp = 1 \iff ax \equiv 1 \pmod{p}$. Finally, the result of this Code 2.20 is equal to the inverse of a in $\mathbb{F}_p$. These restrictions on inputs produce slightly optimized Code 2.21.

```
inversion_with_extended_gcd := function(a, p)
  u := a; v := p;
  x_1 := 1; x_2 := 0;
  while u ne 1 do
    q := v div u;
    r := v-q*u;
    x := x_2 - q*x_1;
    v := u;
    u := r;
    x_2 := x_1;
    x_1 := x;
  end while;
  return (x_1 mod p);
end function;
```

Code 2.21. Inversion with Extended GCD

2.7.3. Fermat's little theorem

As with the Euclidean algorithm, Fermat's little theorem can be used to find the modular multiplicative inverse of any given integer a, p where $gcd(a, p) = 1$. Before we start explaining how Fermat's little theorem is used for the multiplicative inverse of integers. Recall, Fermat's little theorem is a special case of the Euclidean algorithm where a, p is coprime and p is a prime. Therefore, $a^{p-1} \equiv 1 \pmod{p}$. If the both sides are divided by a. This congruence becomes $a^{p-2} \equiv \frac{1}{a} \pmod{p}$ which implies $a^{p-2} \equiv a^{-1} \pmod{p}$. Thus, calculating the value of a^{p-2} when p is prime yields the modular multiplicative

inverse of a in $\mathbb{F}_p$. Code 2.22 shows the basic application of Fermat's little theorem algorithm to calculate the inverse of a in $\mathbb{F}_p$. For the sake of performance, the Code 2.22 is poorly implemented, but it explains the gist of the algorithm. As we mentioned earlier, these Magma codes are produced from an informative perspective only.

```
fermat_little_theorem := function(a, p)
  u := a;
  for i := 1 to p-3 do
    u := (u*x) mod p;
  end for;
  return u;
end function;
```

Code 2.22. Fermat's-little theorem

A better way might be to replace recurring multiplication on the Code 2.22 with modular exponentiation, see 2.6.1, which has a better speed oriented performance.

CHAPTER 3
FAST MODULAR MULTIPLICATION

Chapter 2 showed how to implement basic operations in finite field arithmetic. The total running time of most cryptographic primitives (such as elliptic curve scalar multiplication) depends heavily on the number of field multiplications performed over the underlying finite field. Therefore, we are particularly interested in modular multiplication in this chapter and in the remainder of this thesis.

Let p be a large prime. Let f and g be non-negative integers. Our task is to compute $(f \cdot g) \bmod p$. This task is named the modular multiplication. In the most naive way, modular multiplication is carried out by an integer multiplication (i.e. $f \cdot g$) followed by a reduction procedure which were detailed in Sections 2.3 and 2.5, respectively. Optimized speed implementations typically fix a NIST-like prime and merges the arithmetic of multiplication and reduction steps. The most common instances are the primes $2^{255} - 19$ and $2^{448} - 2^{224} - 1$. This chapter aims to investigate fast modular multiplication strategies with these primes and also build a systematic search for possible faster alternatives. Two concrete outcomes of this chapter are listed as follows:

1. We introduce a nine-limb representation the elements of $GF(2^{255} - 19)$, which have potential to be used in the place of classic ten-limb representation introduces in (D. Bernstein, 2006), also see (Chou, 2015).
2. We introduce the prime $2^{261} - 2^{131} - 1$ along with a ten-limb representation of it's elements, which outperforms the speed of the prime $2^{255} - 19$ in all settings.

We adopt this second methodology in this chapter. In particular, we are interested in two different strategies;

- Multiplication blended with reduction,
- Semi-reduced multiplication followed by reduction.

These strategies may have strengths depending on the conditions in which they have been used. However, we are heavily involved with this second strategy mentioned above.

The second strategy called "Semi-reduced multiplication followed by reduction", allows the items produced to be kept in an unreduced and redundant form. This is strictly limited by the register capacity. Hence, this strategy uses fewer limbs to do arithmetic

unlike the "Multiplication is blended with reduction". In this context, the our prime number `p261` represented in 10 limbs, and after multiplication it fits into 10 limbs. This hybridized technique is “Semi-reduced multiplication followed by reduction" allows you to do this. Further information about this technique is given in Chapter 4.

In addition, the multiplication techniques proposed in this thesis use a Karatsuba multiplication. Before we initiate to describe our proposed methods, it is beneficial to explain the classical Karatsuba multiplication.

As described earlier in Section 2.3, the classical Karatsuba multiplication produces the product fg. In this process, we split the operands into two ℓ-bit parts and perform three half sized multiplications. For instance, f and g can be represented $f = f_1 \cdot 2^{\frac{\ell}{2}} + f_0$, $g = g_1 \cdot 2^{\frac{\ell}{2}} + g_0$. The product formula of f and g can be described in the below.

$$fg \equiv (A + cB) + 2^{\ell} \cdot (C - A - B) \pmod{2^{2\ell} - c}$$

where $A = f_0 g_0$, $B = f_1 g_1$, and $C = (f_0 + f_1)(g_0 + g_1)$. Further formulas can have performed according to the properties of the selected prime. In this thesis, two types of multiplication and corresponding prime numbers have been studied, which had explained in 3.1 and 3.2 sections, respectively.

3.1. Karatsuba 2-way friendly primes

This section will explore the types of prime numbers which are suitable for using the Karatsuba 2-way modular multiplication technique. In order to obtain this information, all possible cases are explored on the most general suggested Karatsuba 2-way formula. At first, we should define a Karatsuba 2-way in the most general way. As described earlier, we use a semi-reduced multiplication technique. The classical Karatsuba multiplication with our settings has expressed as follows:

$$\begin{aligned} c_2 fg \equiv (\ & c_1(t_1) + c_2(t_2 - t_1 - t_0)\ ) \cdot \ell^1 + \\ (\ & c_0(t_1) + c_2(t_0)\ ) \cdot \ell^0 \end{aligned} \pmod{c_2\ell^2 - c_1\ell - c_0} \tag{3.1}$$

where $f = f_1\ell + f_0$, $g = g_1\ell + g_0$ and $p = c_2\ell^2 - c_1\ell - c_0$. Additional components consist of the Karatsuba formula $t_0 = f_0 g_0$, $t_1 = f_1 g_1$ and $t_2 = (f_0 + f_1)(g_0 + g_1)$, respectively. The explicitly illustrated formula shows us that the prime components (c_0, c_1, c_2) play an significant role in constructing a multiplication sequence. Each arithmetic operation used in the formula corresponds to at least one instruction, depending on the hardware used. So our goal is to eliminate as many linear operations as possible using prime components (c_0, c_1, c_2) to reduce instruction usage. This generated general formula

shows us some observations for making speed-oriented optimizations, they are as follows:

- If $c_0 = 0$, this case will never happen due to a prime number violation. Thus in the best-case scenario, we can eliminate a single multiplication when $c_0 = 1$. Also, $c_2 = 0$ cannot be happen.
- The most computationally cumbersome task performs in the ℓ^1 limb. Thus the optimization techniques should have focused on this step.

So representing the field multiplication in the most general way helped us understand the bottleneck of multiplication and constraints, keep in mind that the generated formulas are for multiplication only, but the same optimizations will be achieved in squaring within the same observation. Accordingly, there are several conditions for the selection of the prime components, they are examined below:

1. If $c_0 = 1$, the formula becomes

$$\begin{aligned} c_2fg \equiv (\quad & c_1(t_1) + c_2(t_2 - t_1 - t_0)\ ) \cdot \ell^1 + \\ (\ t_1 + \quad & c_2(t_0 \quad)\ ) \cdot \ell^0 \qquad (\mathrm{mod}\ c_2\ell^2 - c_1\ell - 1) \end{aligned} \tag{3.2}$$

   Ideally, we would look for a smaller c_0 due to our research to improve computational speed. But in this case, we have a better opportunity. In these settings, we have a chance to remove a linear operation.

2. If $c_1 = 0$, the formula becomes

$$\begin{aligned} c_2fg \equiv (\quad & c_2\ (t_2 - t_1 - t_0)\ ) \cdot \ell^1 + \\ (\ c_0(t_1) + \quad & c_2\ (t_0 \quad)\ ) \cdot \ell^0 \qquad (\mathrm{mod}\ c_2\ell^2 - c_0) \end{aligned} \tag{3.3}$$

   In this case, the operating cost is reduced by one multiplication. Prime numbers such as $2^{255} - 19$ and $2^{251} - 9$ are examples of this scenario. Most cryptographic applications have an application where such prime numbers are used.

3. If $c_1 = c_2$, the formula becomes

$$\begin{aligned} c_1fg \equiv (\quad & c_1(t_2 - t_0) \quad)\cdot \ell^1 + \\ (\ c_0(t_1) + c_1(t_0 \quad) \quad &)\cdot \ell^0 \qquad (\mathrm{mod}\ c_1\ell^2 - c_1\ell - c_0) \end{aligned} \tag{3.4}$$

   We can eliminate one subtraction and one multiplication operation. The computational cost has significantly reduced. Also, such prime numbers allow some

calculations have reused, such as reusing $c_1 t_0$ in the ℓ^1 limb.

4. If $c_2 = 1$, $c_1 = 0$, $c_0 = 1$ the formula becomes

$$\begin{aligned} fg \equiv (\quad & t_2 - t_1 - t_0 \quad) \cdot \ell^1 + \\ (\quad & t_1 + t_0 \quad) \cdot \ell^0 \qquad (\mathrm{mod}\ \ell^2 - 1) \end{aligned} \tag{3.5}$$

   Among all for combinations, the fourth one provides the fastest formulas. Unfortunately, there is no such prime having bit length around 256.

5. If $c_2 = c_1$, $c_0 = 1$ the formula becomes

$$\begin{aligned} c_1 fg \equiv (\quad & c_1(t_2 - t_0) \quad) \cdot \ell^1 + \\ (\quad & t_1 + c_1(t_0 \quad) \quad) \cdot \ell^0 \qquad (\mathrm{mod}\ c_1\ell^2 - c_1\ell - 1) \end{aligned} \tag{3.6}$$

   Ideally, we seek a for a prime of the form $(\mathrm{mod}\ c_1\ell^2 - c_1\ell - 1)$. An example of this is the well-known prime number $2^{448} - 2^{224} - 1$. Furthermore, we propose a new prime number with these properties, detailed information is explained in Chapter 5. Having these properties, the newly proposed prime has the possibility to compete with other well-known prime numbers in speed-oriented applications.

These generated formulas can provide a brief overview of how prime components should be selected when working with a finite field multiplication using the Karatsuba technique. The formula creation process below may vary depending on application constraints. Due to the focus of this thesis, only AVX2 instruction extension constraints have been considered when generating a lower-level domain multiplication formula. In line with the thesis focus, the possible formula representation is given in Section 3.1, note that this formula is also used in the implementation of $2^{261} - 2^{131} - 1$, these implementation details are separately examined in Chapter 5.

10-limb representation

At this point, we begin to take a closer look at the implementation details. As we said before, it is necessary to know how many limbs are used to represent the field elements in order to produce a formula that is closer to the target hardware representation. In these settings, field multipliers have represented in the 10 limb. Also, we assume that $c_2 = c_1$ in the following formula. Each limb of the field multiplier elements t_n could express in 64-bit registers. Therefore, this formula can be easily applicable to the target hardware.

$$
\begin{aligned}
c_1fg \equiv\ & (\ & & - c_1(t_{04} - t_{24}) &)\ ) \cdot \ell^9 + \\
& (\ c_0(t_{18}) & & - c_1(t_{03} - t_{23} - t_{28}) &)\ ) \cdot \ell^8 + \\
& (\ c_0(t_{17}) & & - c_1(t_{02} - t_{22} - t_{27}) &)\ ) \cdot \ell^7 + \\
& (\ c_0(t_{16}) & & - c_1(t_{01} - t_{21} - t_{26}) &)\ ) \cdot \ell^6 + \\
& (\ c_0(t_{15}) & & - c_1(t_{00} - t_{20} - t_{25}) &)\ ) \cdot \ell^5 + \\
& (\ c_0(t_{14}) & & + c_1(t_{04}) &)\ ) \cdot \ell^4 + \\
& (\ c_0(t_{13} - t_{08} + t_{28}) & & + c_1(t_{03}) &)\ ) \cdot \ell^3 + \\
& (\ c_0(t_{12} - t_{07} + t_{27}) & & + c_1(t_{02}) &)\ ) \cdot \ell^2 + \\
& (\ c_0(t_{11} - t_{06} + t_{26}) & & + c_1(t_{01}) &)\ ) \cdot \ell^1 + \\
& (\ c_0(t_{10} - t_{05} + t_{25}) & & + c_1(t_{00}) &)\ ) \cdot \ell^0 \\
& & & \pmod{c_2\ell^2 - c_1\ell - c_0} &
\end{aligned}
\tag{3.7}
$$

In the best scenario, when $c_0 = 1$ this formula will transform as follow:

$$
\begin{aligned}
c_1fg \equiv\ & (\ & & - c_1(t_{04} - t_{24}) &)\ ) \cdot \ell^9 + \\
& (\ (t_{18}) & & - c_1(t_{03} - t_{23} - t_{28}) &)\ ) \cdot \ell^8 + \\
& (\ (t_{17}) & & - c_1(t_{02} - t_{22} - t_{27}) &)\ ) \cdot \ell^7 + \\
& (\ (t_{16}) & & - c_1(t_{01} - t_{21} - t_{26}) &)\ ) \cdot \ell^6 + \\
& (\ (t_{15}) & & - c_1(t_{00} - t_{20} - t_{25}) &)\ ) \cdot \ell^5 + \\
& (\ (t_{14}) & & + c_1(t_{04}) &)\ ) \cdot \ell^4 + \\
& (\ (t_{13} - t_{08} + t_{28}) & & + c_1(t_{03}) &)\ ) \cdot \ell^3 + \\
& (\ (t_{12} - t_{07} + t_{27}) & & + c_1(t_{02}) &)\ ) \cdot \ell^2 + \\
& (\ (t_{11} - t_{06} + t_{26}) & & + c_1(t_{01}) &)\ ) \cdot \ell^1 + \\
& (\ (t_{10} - t_{05} + t_{25}) & & + c_1(t_{00}) &)\ ) \cdot \ell^0 \\
& & & \pmod{c_2\ell^2 - c_1\ell - 1} &
\end{aligned}
\tag{3.8}
$$

The reason we use 10-limb to represent field elements is because the newly discovered prime fits perfectly into the 10-limb notation when each limb is filled with about 26 bits. Therefore, this formula is used in Section 5. This formula may perform better at the newly suggested prime $2^{261} - 2^{131} - 1$ instead of `p25519`. Further information is given in Section 5.

3.2. Schoolbook 2-way friendly primes

As with Section 3.1, this section examines possible formulas for the schoolbook multiplication technique. The same methodology is followed here to explore possible formulas for the schoolbook multiplication technique. Firstly, the most general way of

technique is expressed as:

$$\begin{aligned} c_2fg \;\equiv (\quad & c_1(t_1) + c_2(t_2 + t_3)\ ) \cdot \ell^1 \;+ \\ (\ c_0(t_1) + \quad & + c_2(t_0 \quad)\ ) \cdot \ell^0 \qquad (\mathrm{mod}\ c_2\ell^2 - c_1\ell - c_0) \end{aligned} \tag{3.9}$$

where $f = f_1\ell + f_0$, $g = g_1\ell + g_0$ and $p = c_2\ell^2 - c_1\ell - c_0$. Other components come from the multiplication technique used, which are explained as $t_0 = f_0g_0$, $t_1 = f_0g_1$, $t_2 = f_1g_0$, $t_3 = f_1g_1$ respectively. As you noticed this technique requires more multiplication than the Karatsuba method. However, there may be situations where this schoolbook technique would be more beneficial. To investigate these cases, every possible formula that could be produced was investigated according to its prime components as follows. Before starting to explore formulas, some limitations should be explained. Those are given in a list form.

- $c_0 \not\equiv 0 \pmod 2$, also should satisfy $c_0 \neq 0$.
- $c_2 = \{\, x \mid x \in \mathbb{Z}^+, x > 0\}$

1. if $c_0 = 1$ the formula as follows:

$$\begin{aligned} c_2fg \;\equiv (\quad & c_1(t_3) + c_2(t_1 + t_2)\ ) \cdot \ell^1 \;+ \\ (\ \ t_3 + \quad & c_2(t_0 \quad)\ ) \cdot \ell^0 \qquad (\mathrm{mod}\ c_2\ell^2 - c_1\ell - c_0) \end{aligned} \tag{3.10}$$

   Such prime numbers can perform arithmetic using less linear operations compared to prime number types where $c_0 \neq 1$. This special case provides an opportunity to remove a multiplication that is $c_0(t_3)$ explicitly $c_0(f_1g_1)$.

2. if $c_0 = c_2$ the formula as follows:

$$\begin{aligned} c_2fg \;\equiv (\ c_0(t_1 + t_2) + c_1(t_3) \quad)\cdot\ell^1 \;+ \\ (\ c_0(t_0 + t_3) \qquad)\cdot\ell^0 \qquad (\mathrm{mod}\ c_2\ell^2 - c_1\ell - c_0) \end{aligned} \tag{3.11}$$

   In this case, using prime numbers with these properties helps us get t_0 and t_3 into a common multiplication group, so that if we use such primes to do arithmetic, we can produce a speed-optimized implementation. This observation gives us the opportunity to remove a one linear operation in arithmetic.

3. if $c_2 = 1$ the formula as follows:

$$\begin{aligned} c_2fg \;\equiv (\quad & c_1(t_3) + \ (t_1 + t_2)\ ) \cdot \ell^1 \;+ \\ (\ c_0(t_3) \ + \quad & (t_0 \quad)\ ) \cdot \ell^0 \qquad (\mathrm{mod}\ c_2\ell^2 - c_1\ell - c_0) \end{aligned} \tag{3.12}$$

The computational cost will be reduced dramatically. This situation where $c_2 = 1$ allows us to remove the two linear operations. A few examples can be given for this type of prime, such as $2^{192} - 2^{64} - 1$, $2^{224} - 2^{96} + 1$.

4. if $c_1 = 0$ the formula as follows:

$$\begin{aligned} c_2 fg \equiv (& \quad c_2(t_1 + t_2)\ ) \cdot \ell^1 + \\ (\ c_0(t_3) & + c_2(t_0 \quad)\ ) \cdot \ell^0 \qquad (\text{mod } c_2\ell^2 - c_0) \end{aligned} \tag{3.13}$$

The case where the property of the prime components is $c_1 = 0$ allows to remove a linear operation where t_3 is also equal to $f_1 g_1$.

5. if $c_2 = 1$, $c_1 = 0$, $c_0 = 1$ the formula as follows:

$$\begin{aligned} fg \equiv (& \quad t_2 - t_1 - t_0 \quad) \cdot \ell^1 + \\ (& \quad t_1 + t_0 \quad) \cdot \ell^0 \qquad (\text{mod } \ell^2 - 1) \end{aligned} \tag{3.14}$$

Ideally, we would look for a prime number with properties in this case. As it turns out, such prime numbers have better potential to produce a speed-optimized application.

These cases examined show the essential information about what kind of prime numbers should be used with the corresponding multiplication technique. Thus, we discovered that "Schoolbook algorithm" of the technique under review often performs better with NIST-like prime numbers, such as $2^{521} - 1$, $2^{255} - 19$. To summarize, we concluded that the prime components play an important role in the finite field of multiplication. These comparisons are extensively explored in the Section 3.1 and Section 3.2. Also, this systematic search for prime numbers shows that the multiplication techniques used must depend heavily on prime properties. With this information obtained, we showed which types of prime numbers would be useful. This systematic way of looking for prime numbers helped us gain this knowledge. Also, this search method helps to find the prime $2^{261} - 2^{131} - 1$. This newly discovered prime has been extensively researched on Chapter 5.

CHAPTER 4

CASE STUDY #1: *FAST 4 WAY VECTORIZED LADDER FOR THE COMPLETE SET OF MONTGOMERY CURVES*

This work introduces 4 way vectorization of Montgomery ladder on any Montgomery form elliptic curve. Our algorithm takes $2\mathbf{M}^4 + 1\mathbf{S}^4$ ($\mathbf{M}^4$: A vector of four field multiplications, $\mathbf{S}^4$: A vector of four field squarings) per ladder step for variable-scalar variable-point multiplication. This paper also introduces new formulas for doing arithmetic over $GF(2^{255} - 19)$.

DECLARATIONS:

1. This work has been published: *H. Hışıl , B. Eğrice and M. Yassı , "Fast 4 way vectorized ladder for the complete set of Montgomery curves", International Journal of Information Security Science, vol. 11, no. 2, pp. 12-24, Jun. 2022*.
2. This work is funded by Yasar University Scientific Research Project SRP-057.
3. Source code related to this project is publicly available at

   `https://github.com/crypto-ninjaturtles/montgomery4x`

4. We thank Erdem Alkım, Sedat Akleylek, and members of the Cyber Security and Cryptology Laboratory, Ondokuz Mayis University, for providing us access to `OMU-i9`, a Skylake i9-7900X machine. We developed the AVX-512 implementation on `OMU-i9`. The measurements are taken on `OMU-i9`.

4.1. Introduction

Elliptic curve cryptography was proposed by Miller (Miller, 1985) and Koblitz (Koblitz, 1987) in late 80s. In the past three decades, elliptic curves became one of the central objects in public key cryptography. The group law computations on elliptic curves are particularly interesting as they allow efficient arithmetic on computers. In addition, hard instances of discrete logarithm problem can be defined on elliptic curves over finite fields of fairly small size. These two properties of elliptic curves make them perfect candidates for many cryptographic primitives such as key exchange, key encapsulation mechanism, and digital signatures. In all of these primitives, the bottleneck operation is the multiplication of a point on an elliptic curve with a scalar. This operation is called

scalar multiplication. Optimizing scalar multiplication is one of the main challenges in elliptic curve cryptography.

An elliptic curve can be represented in several different forms. One of these forms was introduced by Peter L. Montgomery in his celebrated article (P. Montgomery, 1987) in 1987. An elliptic curve in Montgomery form is written as

$$By^2 = x^3 + Ax^2 + x$$

with constants A and B satisfying $B(A^2 - 4) \neq 0$. Let P be a point on this curve. Let $x(P)$ be the x-coordinate of P. Let k be a positive integer. Montgomery ladder algorithm which was also proposed in (P. Montgomery, 1987), computes $x(kP)$ by accessing a single point doubling and a single point addition operation per iteration of its main loop. In this setting, Montgomery provides doubling formulas to compute $x(2P)$ given $x(P)$, and differential addition formulas to compute $x(P+Q)$ given $x(P)$, $x(Q)$, and $x(P - Q)$. The auxiliary value $x(P - Q)$ is maintained naturally by the ladder. This regular structure of Montgomery ladder made it a perfect candidate to be used in elliptic curve cryptography.

In 2006, Bernstein (D. Bernstein, 2006) proposed an elliptic curve Diffie-Hellman key exchange function, Curve25519, which uses Montgomery ladder along with a twist-secure Montgomery curve over the field $GF(2^{255-19})$. Bernstein (D. Bernstein, 2006) also provided fast software which implements Curve25519, runs in constant-time, and can defend against timing-attacks. Bernstein's design is later re-specified by the Internet Research Task Force in RFC 7748 memorandum.

Montgomery ladder was also adapted to other elliptic curve forms. For example, Brier and Joye (Brier & Joye, 2002) presented formulas for any elliptic curve written in short Weierstrass form $y^2 = x^3 + a_4x + a_6$ covering all elliptic curves over a field k with char$(k) \neq 2, 3$. Analogous formulas over a field of characteristic 2 were given by Lopez and Dahab (López & Dahab, 1999). Additional alternative differential additions formulas can be found in (Castryck, Galbraith, & Farashahi, 2008), (D. J. Bernstein, Lange, & Rezaeian Farashahi, 2008), (R. Farashahi & Hosseini, 2016) and (R. R. Farashahi & Hosseini, 2017).

Building on an earlier work of Chudnovsky and Chudnovsky (Chudnovsky & Chudnovsky, 1986), Gaudry introduced doubling and differential addition analogues on genus 2 Kummer surfaces in (Gaudry, 2007). As a follow up work, Gaudry and Lubicz introduced genus 1 analogues of Kummer surfaces in (Gaudry & Lubicz, 2009). Their study covers both odd and even characteristics. We refer to these Kummer lines as canonical Kummer lines in this work following the language of (Renes & Smith, 2017).

Explicit formulas for squared Kummer lines appeared in EFD[1] with credits to Gaudry (Gaudry, 2007) and Gaudry, Lubicz (Gaudry & Lubicz, 2009).

Emerging hardware trend in single-instruction multiple-data (SIMD) circuits led researchers develop vectorized implementations of ladders. A SIMD implementation of Gaudry-Schost squared Kummer surface (Gaudry & Schost, 2012) was introduced by Bernstein, Chuengsatiansup, Lange, and Schwabe (D. Bernstein, Chuengsatiansup, Lange, & Schwabe, 2014). Their implementation is currently the speed leader in the genus 2 setting. The genus 1 setting is actively in development. Chou (Chou, 2015, Alg. 3.1) put forward a 2 way vectorized implementation of Montgomery ladder using the inherent 2 way parallelism in the classic formulas. Chou's implementation uses the 2 way vectorized $32 \times 32 \rightarrow 64$-bit multipliers on Sandy Bridge and Ivy Bridge. A 4 way vectorized implementation of squared Kummer lines were presented by Karati and Sarkar in (Karati & Sarkar, 2017). Their implementation uses the 4 way vectorized $32 \times 32 \rightarrow 64$-bit multipliers on Haswell and Skylake. Karati and Sarkar report that their implementation offers competitive performance in Kummer line based scalar multiplication for genus one curves over prime order fields using SIMD operations. Faz-Hernández and López provided a 2×2 way implementation of Montgomery ladder on Haswell and Skylake. The arithmetic of the underlying field is 2 way vectorized in their implementation (hence the notation 2×2).

Putting the vectorization option of the underlying field a side (which is also an option for squared Kummer lines), the sequence of recent advances in ladder implementations may lead to the illusion that Montgomery curves are less vectorization-friendly than Kummer lines. In this work,

- we show that Montgomery curves are efficiently 4 way vectorizable. See Section 4.3.
- we provide timings for our 4×1 way vectorized implementation on AVX2. See Section 4.4.
- we propose a new 9 limb representation of field elements which has potential to be faster than the widely applied 10 limb representation, in implementations without using field level vectorization. See Section 4.4.
- we provide timings for our 4×2 way vectorized implementation on AVX-512. See Section 4.5. This implementation sets the new speed record in variable-scalar variable-point multiplication over the field $GF(2^{255} - 19)$.

Results are provided in Section 4.6.

[1] `http://www.hyperelliptic.org/EFD/` (last accessed 2019-05-20)

4.2. Montgomery ladder

This section provides preliminaries on Montgomery ladder. We will skip detailed discussions on the group law, the pseudo-group structure, working solely on the x-line, point recovery etc. These are all very well understood and available in several texts in the literature, cf. (D. Bernstein & Lange, 2017, Chapter 4) and (Costello & Smith, 2018). Our approach will be more implementation oriented. Therefore, the treatment in this section is far from being comprehensive.

The abscissa $x(P)$ of a point P is represented in homogenous projective space $\mathbb{P}$ in the form $(x(P) : 1)$. In this projective representation, $(X : Z) = (\lambda X : \lambda Z)$ for all non-zero $\lambda \in \mathbb{K}$. The point $(1 : 0)$ is the pseudo-identity element. From now on, we update the definition of P and use the projective notation.

Given the points $(X_3 : Z_3)$, $(X_2 : Z_2)$, and $(X_1 : Z_1) = (X_3 : Z_3) - (X_2 : Z_2)$, we have $(X_5 : Z_5) = (X_3 : Z_3) + (X_2 : Z_2)$ and $(X_4 : Z_4) = 2(X_2 : Z_2)$. Montgomery provided the following explicit formulas in (P. Montgomery, 1987):

$$\begin{aligned}(X_5 : Z_5) &= \left(Z_1(X_2X_3 - Z_2Z_3)^2 : X_1(X_2Z_3 - Z_2X_3)^2\right), \\ (X_4 : Z_4) &= \left((X_2^2 - Z_2^2)^2 : 4X_2Z_2(X_2^2 + AX_2Z_2 + Z_2^2)\right). \end{aligned} \tag{4.1}$$

These differential addition and doubling formulas are the building blocks of the Montgomery ladder. Before providing the ladder, we simplify our notation and define the functions `DBLADD` and `SWAP`. The function `DBLADD` inputs three points where the third is the difference of the first two, and outputs the sum of the two initial points and the double of the second input point. The output is overwritten to $(X_3 : Z_3)$ and $(X_2 : Z_2)$, respectively. This is denoted as

$$\texttt{DBLADD}\left((X_3 : Z_3), (X_2 : Z_2), (X_1 : Z_1)\right).$$

The function `SWAP` inputs two points and a single bit. If `swap` is 0, then the output is identical to the input. If `swap` is 1, then the output is the swapped input points. The Montgomery ladder is provided succinctly in Algorithm 1.

In cryptographic applications, the output of Algorithm 1 is typically normalized as X_2/Z_2 in order to obtain a unique representative of the output. In addition, ℓ is fixed in order to fix the number of iterations. Moreover, one can force k to be multiple of a small power of 2 to surpass active attacks exploiting the existence of small subgroups. Cryptographic applications which are required to run in constant-time must have each sub-operation run in constant-time. We refer to `curve25519` specification for full detail, (D. Bernstein, 2006).

Algorithm 1: Montgomery ladder

Require: $P = (X : Z) \neq (1 : 0)$ and $k = \sum_{i=0}^{\ell-1} k_i 2^i$ with $k_{\ell-1} = 1$, $k_i \in \{0, 1\}$.
Ensure: kP.
1: $(X_3 : Z_3) \leftarrow P$,;
   $(X_2 : Z_2) \leftarrow (1 : 0)$,;
   $(X_1 : Z_1) \leftarrow P$
2: `prevbit` $\leftarrow 0$
3: **for** $i = \ell - 1$ down to 0 **do**
4: `swap` $\leftarrow$ `prevbit` $\oplus\, k[i]$
5: `prevbit` $\leftarrow k[i]$
6: `SWAP`$(\texttt{swap}, (X_3 : Z_3), (X_2 : Z_2))$
7: `DBLADD`$((X_3 : Z_3), (X_2 : Z_2), (X_1 : Z_1))$
8: **end for**
9: `SWAP`$(k[0], (X_3 : Z_3), (X_2 : Z_2))$
10: **return** $(X_2 : Z_2)$

4.3. 4 way Montgomery ladder

Montgomery's formulas (4.1) lie at the heart of `curve25519`. Several implementations of `curve25519` are available in public domain. Karati and Sarkar (Karati & Sarkar, 2017) commented for the ladder step used in `curve25519` specification (D. Bernstein, 2006, Appendix B):

> *"The structure of this ladder is not as regular as the ladder step on the Kummer line. This makes it difficult to optimally group together the multiplications for SIMD implementation."*

In this work, we aim to show that a higher level of parallelism can be achieved with new tweaks on the ladder step, see Figure 4.1. In the figure, $\mathcal{H}$ stands for Hadamard transformation which inputs two coordinates X and Z and outputs $X + Z$ and $X - Z$. The point doubling side of Figure 4.1 is recognizably different than Bernstein's diagram. Specifically, the squaring step now utilizes all 4 channels in vectorized form. On the other hand, an inspection on Figure 4.1 reveals that the outputs X_4, Z_4, X_5, and Z_5 agree with (4.1) up to a multiplication of the coordinates by a constant with no effect on the correctness of `DBLADD` routine.

The ladder step in Figure 4.1 takes $2\mathbf{M}^4 + 1\mathbf{S}^4$. In comparison, Karati and Sarkar's 4 way vectorized ladder step (Karati & Sarkar, 2017, Fig. 1) takes $2\mathbf{M}^4 + 1\mathbf{S}^4 + 1\mathbf{d}^4$ ($\mathbf{d}^4$: A vector of four field multiplications by four small constants). There is a speed trade-off between these two approaches, which is not clear immediately from the high level operation counts:

- *Multiplication with constants:* A squared Kummer line requires one multiplication by $[a^2 + b^2, a^2 - b^2, a^2 + b^2, a^2 - b^2]$ followed by reduction (denoted $\mathbf{d}^4$), per

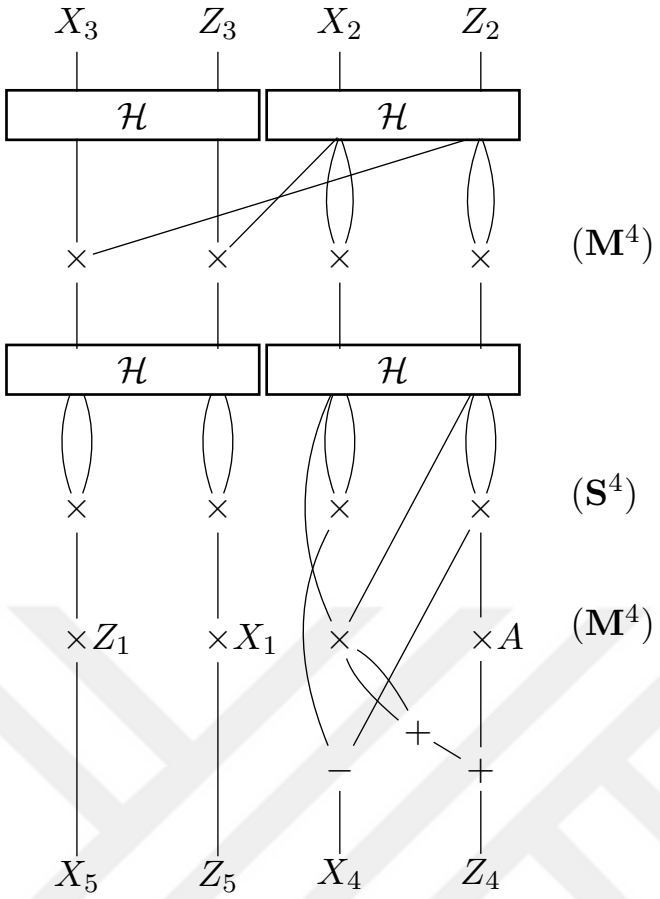

Figure 4.1. DBLADD: 4 way vectorized ladder step for the curve $By^2 = x^3 + Ax^2 + x$.

ladder step. Such a multiplication-reduction does not occur in Figure 4.1.

- *Extra permutations:* Data transfers between SIMD channels occur in Hadamard transform and constant-time conditional point swap operations in both types of ladder steps. Our algorithm requires additional transfers and linear operations following the second Hadamard transform.

These two items constitute a speed trade-off (even if $a^2 + b^2$ and $a^2 - b^2$ are extremely small). This trade-off depends heavily on the comparative throughput of SIMD multiplication and data transfer instructions, which can significantly vary depending on the micro-architecture. In any case, the overall timings can be expected to be close in optimized instantiations since neither of the operations is a speed bottleneck.

4.4. Implementation on AVX2

This section provides implementation details for 4 way vectorization of Montgomery ladder. Implementers are not limited to the specification of this section because Figure 4.1 is independent of choices made here. The same applies to Section 4.5.

We fix $p = 2^{255} - 19$ and work over $GF(p)$. We start by explaining field multiplication. The discussion is narrowed to a single field multiplication. On the other hand, the implementation computes 4 field multiplications simultaneously in vector form. We refer to (D. Bernstein et al., 2014) for a comprehensive explanation of the concept. We use core ideas from (D. Bernstein & Schwabe, 2012), (D. Bernstein et al., 2014), (Chou, 2015), and (Karati & Sarkar, 2017). Yet, we made different implementation choices.

Multiplication. We represent reduced field elements in 9 limbs rather than 10 and keep unreduced products in 11 limbs rather than 10. We provide justifications for how intermediate values always fit into 64 bit registers, without producing any overflow. This is a hybridization of two commonly followed methods:

- doing the $255 \times 255 \rightarrow 510$ bit multiplication first and then reducing to 255 bits, cf. (Karati & Sarkar, 2017) and
- merging reduction with integer multiplication and keeping elements always in specified number of limbs, cf. (D. Bernstein, 2006).

These scenarios are not in the context of the 4 way ladder (Figure 4.1) and thus omitted in this work.

We designed a two-layer implementation to carry out field multiplications with a redundant representation of elements. Both layers use a 3 way splitting strategy. Therefore, a field element is represented by 9 limbs each of which can accommodate non-negative values smaller than 2^{64}.

The higher layer is described as follows. A field element u is represented by integers u_0, u_1, and u_2 such that $u = u_0 + 2^{85}u_1 + 2^{170}u_2$. We note that this is not a unique representation. Let v be an integer also represented in the same way. We then have

$$\begin{aligned} uv \equiv \quad & 2^0(\ u_0v_0 + \ 19u_1v_2 + \ 19u_2v_1 \ ) + \\ & 2^{85}(\ u_0v_1 + \quad u_1v_0 + \ 19u_2v_2 \ ) + \\ & 2^{170}(\ u_0v_2 + \quad u_1v_1 + \quad u_2v_0 \ ) \quad (\mathrm{mod}\ p)\,. \end{aligned}$$

The congruence $255 \equiv 0 \pmod 3$ helps greatly in obtaining simple formulas. If we did not have this condition the given formulas would have contained several multiplications by 2 in addition to multiplications by 19. Such a situation would have added more linear operations to the ladder step.

The nine long multiplications in the form u_iv_j are reduced to six by three Karatsuba optimizations which are capable of sharing the sub-expressions u_iv_i as follows:

$$\begin{aligned} 2^0(\quad & u_0v_0 + 19((u_1+u_2)(v_1+v_2) - u_1v_1 - u_2v_2) \ ) + \\ 2^{85}(\ & 19u_2v_2 + \quad (u_0+u_1)(v_0+v_1) - u_0v_0 - u_1v_1 \ ) + \\ 2^{170}(\quad & u_1v_1 + \quad (u_0+u_2)(v_0+v_2) - u_0v_0 - u_2v_2 \ )\,. \end{aligned}$$

This variant leads to an increased number of additions/subtractions some of which can be shared. We eliminated these repeating operations at the cost of using more registers in our implementation. The additions of the form $u_i + u_j$ are 3-limb additions. All other additions and subtractions are 5-limb additions.

These high level operations do not provide low level details. For instance, we do not have hardware multipliers that can accommodate $85 \times 85 \rightarrow 170$-bit integer multiplications. Therefore, we further split each digit in the higher layer into three limbs:

$$\begin{aligned} u_0 &= a_0 + 2^{29}a_1 + 2^{57}a_2, \quad v_0 = b_0 + 2^{29}b_1 + 2^{57}b_2, \\ u_1 &= a_3 + 2^{29}a_4 + 2^{57}a_5, \quad v_1 = b_3 + 2^{29}b_4 + 2^{57}b_5, \\ u_2 &= a_6 + 2^{29}a_7 + 2^{57}a_8, \quad v_2 = b_6 + 2^{29}b_7 + 2^{57}b_8. \end{aligned}$$

Now, for instance, u_0v_0 can be computed with the following formulas

$$\begin{aligned} u_0v_0 = \quad & 2^{0}(\ a_0b_0 & & &) + \\ & 2^{29}(\ a_0b_1 + & a_1b_0 & &) + \\ & 2^{57}(\ a_0b_2 + & a_2b_0 + & 2a_1b_1 &) + \\ & 2^{86}(\ a_1b_2 + & a_2b_1 & &) + \\ & 2^{114}(\ a_2b_2 & & &) . \end{aligned}$$

These operations take 9 multiplications and 5 additions all of which can be directly carried out by the target hardware. Karatsuba optimization is not used here since the trade-off between multiplications and additions do not provide a practical speed-up at this level. The registers a_0, a_1, a_2 are bounded carefully as to prevent overflowing of the 64 bit registers and allow the final carries to be delayed to the end of the field operation. More explicitly, the multiplication algorithm inputs 9-limb integers and produces the following 11 limbs

- $w_0 = a_0b_0 + 19(a_3b_6 + a_6b_3)$,
- $w_1 = a_0b_1 + a_1b_0 + 19(a_3b_7 + a_4b_6 + a_6b_4 + a_7b_3)$,
- $w_2 = a_0b_2 + 2a_1b_1 + a_2b_0 + 19(a_3b_8 + a_8b_3 + 2(a_4b_7 + a_7b_4) + a_5b_6 + a_6b_5)$,
- $w_3 = a_0b_3 + a_3b_0 + 2(a_1b_2 + a_2b_1) + 19(a_6b_6 + 2(a_4b_8 + a_5b_7 + a_7b_5 + a_8b_4))$,
- $w_4 = a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 19(a_5b_8 + a_6b_7 + a_7b_6 + a_8b_5)$,
- $w_5 = a_0b_5 + a_2b_3 + a_3b_2 + a_5b_0 + 2(a_1b_4 + a_4b_1) + 19(a_6b_8 + 2a_7b_7 + a_8b_6)$,
- $w_6 = a_0b_6 + a_3b_3 + a_6b_0 + 2(a_1b_5 + a_2b_4 + a_4b_2 + a_5b_1 + 19(a_7b_8 + a_8b_7))$,
- $w_7 = a_0b_7 + a_1b_6 + a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2 + a_6b_1 + a_7b_0 + 19a_8b_8$,

- $w_8 = a_0b_8 + a_2b_6 + a_3b_5 + a_5b_3 + a_6b_2 + a_8b_0 + 2(a_1b_7 + a_4b_4 + a_7b_1)$,
- $w_9 = 2(a_1b_8 + a_2b_7 + a_4b_5 + a_5b_4 + a_7b_2 + a_8b_1)$, and
- $w_{10} = a_2b_8 + a_5b_5 + a_8b_2$

which satisfy in turn the following congruence

$$\begin{aligned} uv \equiv w \equiv (w_0 + 2^{29}w_1 + 2^{57}w_2)+ \\ 2^{85}(w_3 + 2^{29}w_4 + 2^{57}w_5)+ \\ 2^{170}(w_6 + 2^{29}w_7 + 2^{57}w_8)+ \\ 2^{255}(w_9 + 2^{29}w_{10}) \pmod{2^{255} - 19}. \end{aligned}$$

We do not perform all of these $9 \times 9 = 81$ multiplications but just $9 \times 6 = 54$. This is due to the shared-Karatsuba approach explained earlier.

Input/output specification. We set important bounds

$$0 \leq a_0, a_3, a_6 < 2^{29} + k,$$
$$0 \leq a_1, a_2, a_4, a_5, a_7, a_8 < 2^{28} + k$$

for the input and output limbs. $k = 173$ is a constant that will become clear in the reduction step. We always ensure the accuracy of these bounds after a reduction step which provide an easy-to-follow input/output specification.

The limbs w_i are displayed explicitly (in the item list) in order to help check the boundaries on the output easily. In particular, we need to show that these limbs cannot exceed 2^{64}. Now, inputting the largest possible values for each limb of u and v and evaluating on the formulas provided in the item list, we get

$$\begin{aligned} &w_0 < 2^{63.29}, \quad w_1 < 2^{63.29}, \quad w_2 < 2^{63.88}, \\ &w_3 < 2^{63.91}, \quad w_4 < 2^{62.95}, \quad w_5 < 2^{62.98}, \\ &w_6 < 2^{62.59}, \quad w_7 < 2^{61.05}, \quad w_8 < 2^{60.17}, \\ &w_9 < 2^{59.59}, \quad w_{10} < 2^{57.59}. \end{aligned}$$

Clearly, all of these values can be accommodated without overflow in 64-bit registers w_i.

Even if we have computed all w_i, we are not quite done yet. We only have a semi-reduced w satisfying

$$w \equiv uv \pmod{2^{255} - 19}. \tag{4.2}$$

We need to do the carries in order to get rid of w_9, w_{10} and also match the output requirements

$$0 \leq w_0, w_3, w_6 < 2^{29} + k,$$

$$0 \leq w_1, w_2, w_4, w_5, w_7, w_8 < 2^{28} + k$$

which agree with the input specification of u and v.

Carries (Reduction after multiplication). This operation is composed of several steps. Each step transforms w towards satisfying the input/output specification without violating the congruence in display (4.2) and without producing an overflow. We go as follows:

$$\begin{aligned}
&\textbf{Step 1}: t \leftarrow \lfloor w_9/2^{29} \rfloor,\ w_9 \leftarrow w_9 \bmod 2^{29},\ w_{10} \leftarrow w_{10} + t,\\
&\textbf{Step 2}: w_0 \leftarrow w_0 + 19w_9,\ w_9 \leftarrow 0,\\
&\textbf{Step 3}: w_1 \leftarrow w_1 + 19w_{10},\ w_{10} \leftarrow 0,\\
&\textbf{Step 4}: t \leftarrow \lfloor w_0/2^{29} \rfloor,\ w_0 \leftarrow w_0 \bmod 2^{29},\ w_1 \leftarrow w_1 + t,\\
&\textbf{Step 5}: t \leftarrow \lfloor w_1/2^{28} \rfloor,\ w_1 \leftarrow w_1 \bmod 2^{28},\ w_2 \leftarrow w_2 + t,\\
&\textbf{Step 6}: t \leftarrow \lfloor w_2/2^{28} \rfloor,\ w_2 \leftarrow w_2 \bmod 2^{28},\ w_3 \leftarrow w_3 + t,\\
&\textbf{Step 7}: t \leftarrow \lfloor w_3/2^{29} \rfloor,\ w_3 \leftarrow w_3 \bmod 2^{29},\ w_4 \leftarrow w_4 + t,\\
&\textbf{Step 8}: t \leftarrow \lfloor w_4/2^{28} \rfloor,\ w_4 \leftarrow w_4 \bmod 2^{28},\ w_5 \leftarrow w_5 + t,\\
&\textbf{Step 9}: t \leftarrow \lfloor w_5/2^{28} \rfloor,\ w_5 \leftarrow w_5 \bmod 2^{28},\ w_6 \leftarrow w_6 + t,\\
&\textbf{Step 10}: t \leftarrow \lfloor w_6/2^{29} \rfloor,\ w_6 \leftarrow w_6 \bmod 2^{29},\ w_7 \leftarrow w_7 + t,\\
&\textbf{Step 11}: t \leftarrow \lfloor w_7/2^{28} \rfloor,\ w_7 \leftarrow w_7 \bmod 2^{28},\ w_8 \leftarrow w_8 + t,\\
&\textbf{Step 12}: t \leftarrow \lfloor w_8/2^{28} \rfloor,\ w_8 \leftarrow w_8 \bmod 2^{28},\ w_0 \leftarrow w_0 + 19t\\
&\textbf{Step 13}: t \leftarrow \lfloor w_0/2^{29} \rfloor,\ w_0 \leftarrow w_0 \bmod 2^{29},\ w_1 \leftarrow w_1 + t.
\end{aligned}$$

In this sequence of operations, we are accumulating on registers w_i which contain values potentially very close to 2^{64}. Once more, we need to justify that these additions do not constitute any overflow.

- **Step 1:** $t = \lfloor w_9/2^{29} \rfloor < 2^{59.59-29} = 2^{30.59}$. So, $w_{10} + t < 2^{57.59} + 2^{30.59} < 2^{57.60}$. Therefore, the updated value of w_{10} still fits into 64 bits. A bit of care is needed now to track the updated w_9. Although we computed $w_9 \leftarrow w_9 \bmod 2^{29}$ for maximum possible inputs, the updated value of w_9 can still get values as large as $2^{29} - 1$ for some other input. Therefore, we assume for the sake of our inspection that we take $w_9 = 2^{29} - 1$ from here.
- **Step 2:** Now, we must have $w_0 + 19w_9 < 2^{63.29} + 19(2^{29} - 1) < 2^{63.30}$. Multipli-

cation by 19 here is performed with $32 \times 32 \to 64$ bit multiplication instruction `vpmuludq` since both 19 and w_9 are smaller than 2^{32}.

- **Step 3:** Similarly, we must have $w_1 + 19w_{10} < 2^{63.29} + 19(2^{57.60}) < 2^{63.75}$. We note that $19w_{10}$ is computed as $19w_{10} = 16w_{10} + 2w_{10} + w_{10}$ by using `vpaddq` and `vpsllq` instructions because w_{10} can exceed 2^{32}, and thus, is not suitable to be inputted to `vpmuludq`. We note that $w_9 \leftarrow 0$ and $w_{10} \leftarrow 0$ are displayed just for mathematical correctness.
- **Steps 4-11:** Repeating the same inspection by computing each step sequentially, we get $w_{1,\ldots,8} < 2^{64}$ after additions as expected. Limbs $w_{0,\ldots,7}$ obey the input/output specification after reducing $w_{1,2,4,5,7}$ modulo 2^{28} and $w_{3,6}$ modulo 2^{29}. Again, we assume for the sake of our inspection that $w_{0,3,6} = 2^{29} - 1$ and $w_{1,2,4,5,7,8} = 2^{28} - 1$ after the modular reductions are performed for these digits.
- **Step 12:** We get $t = \lfloor w_8/2^{28} \rfloor < 2^{60.17-28} = 2^{32.17}$. So, $w_0 + 19t < (2^{29} - 1) + 19(2^{32.17}) < 2^{36.43}$. Now, w_8 also obeys the input/output specification after being reduced modulo 2^{28}. We note that $19t$ is computed as $19t = 16t + 2t + t$ since w_8 can exceed 2^{32}.
- **Step 13:** We get $w_1 + t < (2^{28} - 1) + (2^{36.43-29}) < 2^{28} + 173$. This upper bound explains the value of k. We note that a lower upper bound can be found with an increased precision in calculations. Moreover, much larger values for k works without producing overflow in reduction[2] but 173 is adequate to test the stability of limbs.

Now, all w_i agrees with the input/output specification of u_i and v_i. We intentionally added k to all limbs in the input/output specification rather than adding just to w_1 because

- this simplifies the notation, and
- we need such extra additions when designing parallel carry chains.

The reduction step can be summarized as $h_9 \to h_{10}$ followed by the very long sequence

$$h_8 \to h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6 \to$$
$$h_7 \to h_8 \to h_0 \to h_1.$$

We do faster by computing two sequences

[2]We reiterate that we use a redundant representation. Therefore, reduction does not produce a unique representative. Nevertheless, we still call it *reduction* since we can do arithmetic in this form.

$$h_4 \longrightarrow h_5 \to h_6 \to h_7 \to h_8 \to h_0 \to h_1,$$

$$h_9 \to h_{10}, \quad h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5$$

in parallel at processor's ports. We refer to (D. Bernstein & Schwabe, 2012) and (Chou, 2015) for similar optimizations.

In this parallel reduction, not only w_1 but also w_5 can exceed $2^{28} - 1$ by k. But we have already relaxed w_5 (like all other limbs) by additions of k in our inspection.

Squaring. Squaring can be explained as a simplified multiplication routine.

$$\begin{aligned} 2^0(\quad & u_0^2 + \ 19((u_1+u_2)^2 - u_1^2 - u_2^2) \ ) + \\ 2^{85}(\ & 19u_2^2 + \quad (u_0+u_1)^2 - u_0^2 - u_1^2 \ ) + \\ 2^{170}(\quad & u_1^2 + \quad (u_0+u_2)^2 - u_0^2 - u_2^2 \ ). \end{aligned}$$

The nine long multiplications in the form $u_i v_j$ are reduced now to six squares. In addition, the computation of u_0^2 can be further optimized at the lower level in the form

$$\begin{aligned} u_i^2 = \quad & 2^0(\ a_0^2 \qquad\qquad\quad) + \\ & 2^{29}(\ (2a_0)a_1 \qquad\quad) + \\ & 2^{57}(\ (2a_1)a_1 + \ (2a_0)a_2 \ ) + \\ & 2^{86}(\ (2a_1)a_2 \qquad\quad) + \\ & 2^{114}(\ a_2^2 \qquad\qquad\quad). \end{aligned}$$

Similar applies to the other squarings. Our implementation delays multiplication by twos and pushes them towards the higher layer.

Squeeze/Unsqueeze. A field element w satisfying the input/output specification can be squeezed from 9 limbs to 5 by computing

$$w_{i+4} \leftarrow w_{i+4} \oplus 2^{32}\, w_i \quad \text{for } \ i = 0, 1, 2, 3.$$

Now, w is represented by w_4, w_5, w_6, w_7, w_8 only. Linear operations such as (field) additions and subtractions can be handled in this form provided that computed values do not exceed $2^{32} - 1$. This is always the case in our implementation.

A squeezed field element is unsqueezed into the original form by computing

$$w_i \leftarrow w_{i+4}/2^{32} \quad \text{for } i = 0, 1, 2, 3 \quad \text{and}$$

$$w_{i+4} \leftarrow w_{i+4} \bmod 2^{32} \quad \text{for } i = 0, 1, 2, 3$$

at multiplication, squaring, and reduction moments. We note that we skip computing $w_{i+4} \leftarrow w_{i+4} \bmod 2^{32}$ before multiplication and squaring since the higher 32 bits are not taken into consideration by `vpmuludq` instruction. See also (D. Bernstein et al., 2014).

This squeeze/unsqueeze method is adapted from the software introduced in (D. Bernstein et al., 2014). The difference is that we group together the limbs of a field element where Bernstein, Chuengsatiansup, Lange, and Schwabe group together points on a genus 2 Kummer surface.

Despite the added cost of squeezing and unsqueezing, linear operations in squeezed form can be done faster and save cycles in total.

Double Hadamard. This step can be put in 4 way vectorized form in modulus $2^{255} - 19$ as follows;

$$\begin{aligned}
&(\mathcal{H} \times \mathcal{H})(X_3, Z_3, X_2, Z_2) = \\
&\quad (X_3 + Z_3, X_3 - Z_3, X_2 + Z_2, X_2 - Z_2) = \\
&\quad (X_3 + Z_3, X_3 + (3p - Z_3), X_2 + Z_2, X_2 + (3p - Z_2)).
\end{aligned}$$

The additions of $3p$ are to ensure that $\mathcal{H} \times \mathcal{H}$ (double Hadamard) produces non-negative values for output limbs. We drop the word "double" for simplicity. This $3p$ needs to be prepared with some care as follows

$$\begin{aligned}
&2^0([3(2^{29} - 19)] + 2^{29}[3(2^{28} - 1)] + 2^{57}[3(2^{28} - 1)])+ \\
&2^{85}(\ [3(2^{29} - 1)] + 2^{29}[3(2^{28} - 1)] + 2^{57}[3(2^{28} - 1)])+ \\
&2^{170}(\ [3(2^{29} - 1)] + 2^{29}[3(2^{28} - 1)] + 2^{57}[3(2^{28} - 1)])\ .
\end{aligned}$$

Observe that each limb[3] is greater than the corresponding maximum bound in the input/output specification.

All of the limbs of $X_3 + Z_3$, $X_3 + (3p - Z_3)$, $X_2 + Z_2$, and $X_2 + (3p - Z_2)$ are always less than 2^{32} after the first Hadamard operation in Figure 4.1. To show this, we concentrate to the linear operations appearing at the right of the bottom of the figure.

- Z_4 is computed as the sum of three values. In order to simplify our analysis, we assume that all inputs to these additions take largest possible values. Then, $w_{0,3,6} = 3((2^{29} - 1) + k) < 2^{31}$ and $w_{1,2,4,5,7,8} = 3((2^{28} - 1) + k) < 2^{30}$.
- X_4 is computed as the difference of two values. We assume that minuend takes the largest and the subtrahend takes the smallest possible value. Then, $w_{0,3,6} =$

[3]The value of each limb appears in square brackets.

$((2^{29}-1)+k)+(2(2^{29}-1)-0) < 2^{31}$ and $w_{1,2,4,5,7,8} = ((2^{28}-1)+k)+(2(2^{28}-1)-0) < 2^{30}$. Observe that we added $2p$ rather than $3p$ this time, which is adequate because $2(2^{29}-1) > (2^{29}-1)+k$ and likewise $2(2^{28}-1) > (2^{28}-1)+k$. So, even if the subtrahend takes the maximum possible value, the limbs are still non-negative.

Up to this point, we showed that w_i of both X_4 and Z_4 fit into 31 bits. We now feed these extreme values[4] to the first Hadamard operation. Clearly, we have $0 \leq w_i < 2^{32}$ for $X + Z$. Separately, assuming that $w_i = 0$ for Z, we have $0 \leq w_i < 2^{32}$ for $X + (3p - Z)$. Analyzing the second Hadamard is even simpler since its inputs are already reduced values.

Fast carries (Fast reduction after Hadamard). Following a Hadamard step, a reduction operation must be applied to the output to match the input/output specification. This time, reduction can be performed faster since we do not have limbs w_9 and w_{10}. Therefore, fast reduction can be defined as a trimmed version of the reduction after multiplication as follows,

$$
\begin{aligned}
\textbf{Step 1} &: t \leftarrow \lfloor w_0/2^{29} \rfloor,\ w_0 \leftarrow w_0 \bmod 2^{29},\ w_1 \leftarrow w_1 + t,\\
\textbf{Step 2} &: t \leftarrow \lfloor w_1/2^{28} \rfloor,\ w_1 \leftarrow w_1 \bmod 2^{28},\ w_2 \leftarrow w_2 + t,\\
\textbf{Step 3} &: t \leftarrow \lfloor w_2/2^{28} \rfloor,\ w_2 \leftarrow w_2 \bmod 2^{28},\ w_3 \leftarrow w_3 + t,\\
\textbf{Step 4} &: t \leftarrow \lfloor w_3/2^{29} \rfloor,\ w_3 \leftarrow w_3 \bmod 2^{29},\ w_4 \leftarrow w_4 + t,\\
\textbf{Step 5} &: t \leftarrow \lfloor w_4/2^{28} \rfloor,\ w_4 \leftarrow w_4 \bmod 2^{28},\ w_5 \leftarrow w_5 + t,\\
\textbf{Step 6} &: t \leftarrow \lfloor w_5/2^{28} \rfloor,\ w_5 \leftarrow w_5 \bmod 2^{28},\ w_6 \leftarrow w_6 + t,\\
\textbf{Step 7} &: t \leftarrow \lfloor w_6/2^{29} \rfloor,\ w_6 \leftarrow w_6 \bmod 2^{29},\ w_7 \leftarrow w_7 + t,\\
\textbf{Step 8} &: t \leftarrow \lfloor w_7/2^{28} \rfloor,\ w_7 \leftarrow w_7 \bmod 2^{28},\ w_8 \leftarrow w_8 + t,\\
\textbf{Step 9} &: t \leftarrow \lfloor w_8/2^{28} \rfloor,\ w_8 \leftarrow w_8 \bmod 2^{28},\ w_0 \leftarrow w_0 + 19t\\
\textbf{Step 10} &: t \leftarrow \lfloor w_0/2^{29} \rfloor,\ w_0 \leftarrow w_0 \bmod 2^{29},\ w_1 \leftarrow w_1 + t.
\end{aligned}
$$

We do better by computing these operations in squeezed form and computing

$$
\begin{aligned}
&h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \longrightarrow h_4 \rightarrow h_5,\\
&h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_0 \rightarrow h_1
\end{aligned}
$$

in parallel on two 32 bit SIMD channels. We do not further exploit processor's port level parallelism since the sequence is short enough to produce low latency.

[4]Noticed that all these operations can be performed in squeezed form.

4.5. Implementation on AVX-512

`AVX-512` provides 8 way SIMD multiplication with the `vpmuludq` instruction. This provides twice as much $32 \times 32 \rightarrow 64$ bit multipliers in comparison to `AVX2`. Therefore it is reasonable to question whether the 4 way vectorized ladder can be computed faster on `AVX-512`. Since Figure 4.1 supports up to 4 way vectorization, additionally, we need to vectorize the field arithmetic in 8/4=2 way form to get a 4×2 way ladder.

Although, our 9 limb multiplication fits nicely on 4×1 ladder, it does not seem to be the best choice for its 4×2 counterpart. Yet, there is room for research in finding a fast 2 way vectorization of 9 limb multiplication described in Section 4.5. We do not pursue this idea further here.

As a practical solution, we decided to use a 2 way vectorized version of the 10 limb multiplication algorithm using Radix-$2^{25.5}$ from (D. Bernstein, 2006). This algorithm was previously used with minor modifications in (D. Bernstein & Schwabe, 2012) and (Chou, 2015). Fortunately, we were able to reuse optimized codes freely available in public domain. In particular, we used the 2 way `AVX2` targeted `intmul` and `intsqr` functions from

`hp-ecc-vec/src/eltfp25519_2w_redradix.c`

by Faz Hernández, López, Dahab [5] and have those functions run on `AVX-512`. Then, we applied the ladder step in Figure 4.1 to get a $4 \times 2 = 8$ way vectorized implementation of Montgomery ladder over the field $GF(2^{255} - 19)$. The speed comparison is given in Section 4.6.

4.6. Results

The final inversion. Our implementation reduces the output of scalar multiplication to a unique representative in the underlying field in radix 256. Therefore, we compute X_2/Z_2 after the main loop. We integrated Nath and Sarkar's (Nath & Sarkar, 2018) freely available and optimized inversion software without further modification. In particular, we used

`pmp-inv-master/p25519/SL-DCC/1`

which requires `BMI2` instruction set. Nath and Sarkar reports 9301 Skylake cycles for this inversion.

[5] `https://github.com/armfazh/hp-ecc-vec` (last accessed 2019-05-20)

Measuring cycles. We measure cycles for variable-scalar variable-point multiplication only. Our code changes base point and scalar at each iteration and excludes extra cycles coming from this randomization. Our implementation chains the outputs to prevent the compiler removing portions of the code. Measured cycle counts are given in Table 4.1 along with selected results from literature. The table is limited to our results and recently published measurements available for the Skylake micro-architecture.

Table 4.1.. Skylake cycles for variable-scalar variable-point multiplication.

ladder method	**instr. set**	**limbs**	**cycles (median)**
sq.Kum., 4×1	`AVX2`	10	123 102, (Karati & Sarkar, 2017)
Montg., 4×1	`AVX2`	10	116 654, *this work*
Montg., 1×1	`BMI2`	4	113 874, (Oliveira, López, Hışıl, Faz-Hernández, & Rodríguez-Henríquez, 2017)
Montg., 2×2	`AVX2`	5	99 400, (Faz-Hernández, López, & Dahab, 2019)
Montg., 4×1	`AVX2`	9	98 484, *this work*
Montg., 4×1	`AVX2`	10	95 437, (Nath & Sarkar, 2022a)
Montg., 2×4	`AVX-512`	5	81 600, (Faz-Hernández et al., 2019)
Montg., 4×2	`AVX-512`	5	74 368, *this work*

Table 4.1 justifies our motivation in proposing the 9 limb representation in Section 4.5. The 9 limb method is solidly faster than 10 *in the context* of our 4 way ladder and specified implementation platform. See (Nath & Sarkar, 2022a)[6] for new results on an other Skylake CPU with different microarchitecture. Nath and Sarkar's fastest implementation uses an extremely small curve constant where our proposed algorithm does not require such an assumption.

Figure 4.1 shows its real potential in our AVX-512 implementation. The reported 74368 cycles sets the new record among `curve25519` family of implementations, to the best of our knowledge.

Variable-scalar fixed-base multiplication. Our implementation can be used directly in a fixed-base multiplication without further modification. Nevertheless, one can make precomputation on fixed-base point to get additional speed up. In that case, we refer to Algorithm 5 of (Oliveira et al., 2017).

Apart from architecture dependent discussions, we expect that our 4 way ladder will gradually become even more useful if the current trend of increasing the level of SIMD parallelism in hardware continues. We reiterate that the speeds we achieve are common for all Montgomery curves; not specific to ones with small constants.

[6]Nath and Sarkar's work appeared at the time when this work was under review.

CHAPTER 5

CASE STUDY #2: *A KARATSUBA FRIENDLY PRIME FOR FAST ELLIPTIC CURVE ARITHMETIC*

We point to the cryptographic significance of the overlooked prime $2^{261} - 2^{131} - 1$ which we dubbed `p261`. We explain our motivation behind searching for such a prime. We present cryptographically secure elliptic curves over $GF(\texttt{p261})$. We provide our speed oriented implementation of variable-base variable-scalar elliptic curve scalar multiplication using the Montgomery ladder. In this setting, a single scalar multiplication implemented with AVX2 instructions takes 82720 on a i7-6500U Skylake processor, respectively. This gives similar performance in comparison with the previous record 83424 Skylake cycles by Nath and Sarkar, which was achieved with a prime having 10 less bits than `p261`.

5.1. The jungle of primes and curves

Fast integer arithmetic plays an important role in real-world applications of asymmetric cryptography. The performance of most popular cryptosystems such as RSA and ECC is centered around how fast we can multiply integers and reduce the product modulo some fixed integer. For arbitrary modulus one can use the long division algorithm which makes access to a built-in division instruction. However, using the division instruction usually leads to poor performance and non-constant running time. A faster choice is to use a general purpose modular arithmetic technique such as Barrett (Barrett, 1987) or Montgomery (P. L. Montgomery, 1985) reduction. ECC applications can further benefit from special prime modulus. These primes allows extremely fast reduction. Some of the fastest primes that facilitate 128-bit conjectured security level are listed in Table 5.1.

Table 5.1.. List of selected fast primes

Name	Prime	Ref.
`p2519`	$2^{251} - 9$	(Karati & Sarkar, 2017)
`p25519`	$2^{255} - 19$	(D. Bernstein, 2006)
`p2663`	$2^{266} - 3$	(Karati & Sarkar, 2017)

There are several works in the literature which use these primes in order to provide speed recording instances of cryptographic primitives such as Diffie-Hellmann Key

Exchange. These implementations fix cryptographically interesting elliptic curves. See (D. Bernstein, 2006), (Costigan & Schwabe, 2009), (Nath & Sarkar, 2022c), (Nath & Sarkar, 2022b), (Nath & Sarkar, 2022a), (Karati & Sarkar, 2017), (Hamburg, 2015), (Chou, 2015), (Hisil, Egrice, & Yassi, 2022). A list of selected curves are provided in Table 5.2.

Table 5.2.. List of selected elliptic curves and DH Key Exchange cycle counts

Elliptic curve	Prime	Security	*Skylake*
$\mathcal{K}_{683,18}$ (Nath & Sarkar, 2022b)	`p2663`	132	(Nath & Sarkar, 2022b) 105328
$\mathcal{M}_{486662}$ (D. Bernstein, 2006)	`p25519`	126	(Nath & Sarkar, 2022a) 95437
$\mathcal{K}_{838,831}$ (Nath & Sarkar, 2022b)	`p25519`	126.5	(Nath & Sarkar, 2022b) 91151
$\mathcal{M}_{4698}$ (Nath & Sarkar, 2022c)	`p2519`	124.5	(Nath & Sarkar, 2022c) 87807
$\mathcal{K}_{81,20}$ (Karati & Sarkar, 2017)	`p2519`	124.5	(Nath & Sarkar, 2022b) 83424
$\mathcal{K}_{276,5}$ (this work)	`p261`	128.5	82720

5.2. The Karatsuba friendly prime `p261`

Let f and g be integers written in radix 2^ℓ in the form $f_0 + 2^\ell f_1$ and $g_0 + 2^\ell g_1$ for some non-negative integer ℓ, respectively. The classic Karatsuba method for multiplying integers modulo a prime of the form $2^{2\ell} - t$ is given as

$$fg \equiv (A + tB) + 2^\ell \cdot (C - A - B) \pmod{2^{2\ell} - t}$$

where $A = f_0 g_0$, $B = f_1 g_1$, and $C = (f_0 + f_1)(g_0 + g_1)$. Such a method requires one multiplication by t and several linear operations. In comparison, the goldilock prime $p_{448} = 2^{448} - 2^{224} - 1$ has a structure which eliminates the multiplication by t and one linear operation. In particular, write f and g in the form $f_0 + 2^{224} f_1$ and $g_0 + 2^{224} g_1$, respectively. Hamburg (Hamburg, 2015) showed that

$$fg \equiv (A + B) + 2^{224} \cdot (C - A) \pmod{p_{448}}.$$

Building on this observation, one may question whether it is possible to do the same at the conjectured 128-bit security level. At this stage, we need a prime of the form $2^{2\ell} - 2^\ell - t$ with $t = 1$ and 2ℓ near 256. However, no such prime exist for $216 < 2\ell < 322$. One can pick $t \neq 1$ as an alternative. For instance, $2\ell = 256$ and $t = 79$ gives a prime. But in such a situation, one needs to bear the cost of several multiplications by 79. We note that the popular prime $2^{255} - 19$ requires several multiplications by 19 in a similar way. To this end, we simply cannot find a Karatsuba-friendly prime with $t = 1$ in the aforementioned scenario.

We now explain our approach to the problem and present our solution. We let our

prime to be of the form $r \cdot 2^{2\ell} - s \cdot 2^{\ell} - t$ for extremely small integers r, s and t. Now, the modular multiplication formulas read

$$fg \equiv (A + tB/r) + 2^{\ell} \cdot ((C - A - B) + sB/r) \pmod{r \cdot 2^{2\ell} - s \cdot 2^{\ell} - t}.$$

One obstacle in such a calculation is that division by r's are unlikely to be exact. To solve this problem we scale the congruence by r which gives

$$rfg \equiv (rA + tB) + 2^{\ell} \cdot (r(C - A - B) + sB) \pmod{r \cdot 2^{2\ell} - s \cdot 2^{\ell} - t}.$$

We arrived at an algorithm which produces rfg efficiently but not fg. This is of no problem for elliptic curve arithmetic since the representation of points in homogeneous projective coordinates allows non-zero scaling. For instance, a point $(X : Z)$ in $\mathbb{P}$ can also be represented by $(rX : rZ)$ with $r \neq 0$.

Now, we can search for primes satisfying this form. Since we target efficiency, it is reasonable to make the following additional assumptions: $r = s = 2$ and $t = 1$. The formula then simplifies to

$$2fg \equiv (2A + B) + 2^{\ell} \cdot (2C - 2A) \pmod{2 \cdot 2^{2\ell} - 2 \cdot 2^{\ell} - 1}. \tag{5.1}$$

At this stage, the question is whether there exist a prime of this form for 2ℓ close to 256. The answer is yes: $\ell = 130$ gives the Montgomery-friendly prime

$$\mathtt{p261} = 2^{261} - 2^{131} - 1.$$

The overhead of computing $2A$ and $2C$ are minor in comparison with the overhead of several multiplications by small constants. We emphasize that since $r = s$, we also eliminated an extra linear operation which is not possible for Karatsuba implementations of multiplication modulo primes of the form $2^{\alpha} - t$. In fact, even more optimizations are possible, which are not obvious at this level. See Section 5.4 for clarification.

To sum up, we introduced a cryptographically interesting prime which suits perfectly with fast elliptic curve arithmetic. We now investigate cryptographically secure elliptic curve over $GF(\mathtt{p261})$ in different elliptic curve forms.

5.3. Cryptographically interesting curves

We present cryptographically interesting elliptic curves over $GF(\mathtt{p261})$ in this section. All of the proposed curves are twist-secure and have large CM discriminants. The numbers given in hexadecimal form are prime. The dash character is used to describe

the non-trivial quadratic twists of the defined curve.

$$\mathcal{W}_{16417} \ : \ y^2 = x^3 - 3x + 16417 \quad \text{(Weierstrass)}$$
$$\#\mathcal{W}_{16417} = \texttt{1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash$$
$$\texttt{674A29D591C4954F6E0E4B49E39D45597}$$
$$\#\mathcal{W}'_{16417} = \texttt{1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash$$
$$\texttt{98B5D62A6E3B6AB091F1B4B61C62BAA69}$$

$$\mathcal{K}_{276,5} : \ y^2 = x^3 + \left(-\frac{276}{5} - \frac{5}{276}\right)x^2 + x \quad \text{(Kummer)}$$
$$\#\mathcal{K}_{276,5} = 16 \times \texttt{1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash$$
$$\texttt{73483AE694E73DDF48915F88662E6DAD}$$
$$\#\mathcal{K}'_{276,5} = 16 \times \texttt{1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash$$
$$\texttt{8CB7C5196B18C220B76EA07799D19253}$$

$$\mathcal{M}_{4318} \ : \ y^2 = x^3 + 4318x^2 + x \quad \text{(Montgomery)}$$
$$\#\mathcal{M}_{4318} = 4 \times \texttt{7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash$$
$$\texttt{CD18A213CBB39E6C7CECB70A57CD1B49B}$$
$$\#\mathcal{M}'_{4318} = 4 \times \texttt{7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash$$
$$\texttt{F2E75DEC344C6193831348F5A832E4B65}$$

Some implementations use $A_{24} = (A-2)/4$ instead of A. For our curve $A_{24} = (4318-2)/4 = 1079 = 13 \times 83$.

5.4. Implementation

There several different ways to carry out a modular multiplication. One classic approach is to compute the integer product fg with Karatsuba multiplication and then reduce the product modulo the prime. This approach is not able to benefit from the elimination of the linear operation explained in Section 5.2. Therefore, we closely follow the outline in therein.

10-limb representation. We provide implementation details in this part targeting $32 \times 32 \rightarrow 64$ bit multipliers and 64 bit adders without carry handling. The typical hardware platforms that suit this configuration are AVX, AVX2 and AVX-512. We

set $k = 2^{26}$. We start by writing $f = f_0 + k^5 f_1$ where f_0 and f_1 are further specified in the form $f_0 = f_{00} + kf_{01} + k^2 f_{02} + k^3 f_{03} + k^4 f_{04}$ and $f_1 = f_{10} + kf_{11} + k^2 f_{12} + k^3 f_{13} + k^4 f_{14}$. Therefore, a field element is represented with 10 limbs. We note here that f_{14} accommodates the single bit that would reside in f_{15} in a standard radix 2^{26} representation.

Modular multiplication. We want to compute $2fg \bmod p$ for 10-limb integers f and g, see Section 5.2. First, we carry out the three integer multiplications $A = f_0 g_0$, $B = f_1 g_1$, $C = (f_0 + f_1)(g_0 + g_1)$. We use schoolbook multiplication algorithm where both A and B take $5 \times 5 = 25$ multiplications and 16 additions, and C takes $5 \times 5 = 25$ multiplications and $5 + 5 + 16 = 26$ additions. At this stage, each of A, B, and C are composed of 9 limbs where each limb fits into a 64 bit register. We access these limbs with subscripts in the obvious way. E.g. A_0 is the least significant limb of A. Now, reducing $(2A + B) + 2^{130} \cdot (2C - 2A)$ modulo $\texttt{p261}$, the following the congruence is obtained:

$$
\begin{aligned}
2fg \equiv\ & (\ 2A_0 + B_0 + C_5 \ - A_5 \ )\, k^0 + \\
& (\ 2A_1 + B_1 + C_6 \ - A_6 \ )\, k^1 + \\
& (\ 2A_2 + B_2 + C_7 \ - A_7 \ )\, k^2 + \\
& (\ 2A_3 + B_3 + C_8 \ - A_8 \ )\, k^3 + \\
& (\ 2A_4 + B_4 \ )\, k^4 + \\
& (\ 2A_5 + B_5 + 2C_0 - 2A_0 + 2C_5 - 2A_5 \ )\, k^5 + \\
& (\ 2A_6 + B_6 + 2C_1 - 2A_1 + 2C_6 - 2A_6 \ )\, k^6 + \\
& (\ 2A_7 + B_7 + 2C_2 - 2A_2 + 2C_7 - 2A_7 \ )\, k^7 + \\
& (\ 2A_8 + B_8 + 2C_3 - 2A_3 + 2C_8 - 2A_8 \ )\, k^8 + \\
& (\ + 2C_4 - 2A_4 + \ )\, k^9 \pmod{\texttt{p261}}.
\end{aligned}
$$

The congruence reflects a redundant reduction modulo $\texttt{p261}$. In addition to the linear simplification provided in (5.1), it is readily observed that several more additions can be removed. In particular, we delete the terms[1] $2A_5$, $2A_6$, $2A_7$, $2A_8$, $-2A_5$, $-2A_6$, $-2A_7$, and $-2A_8$.

The remaining single digit operations take only 27 additions and 9 subtractions. To this end, we use $3 \times 25 = 75$ multiplications and $16 + 16 + 26 + 27 + 9 = 94$ additions (or subtractions). In comparison, Chou (Chou, 2015) reported 109 multiplications and

[1] We note that all of the multiplications by 2 can be moved to the computation phase of A and C, which saves even more additions if $2f_0$, $2f_1$, $2f_2$, $2f_3$, $2f_4$, $2(f_0 + g_0)$, $2(f_1 + g_1)$, $2(f_2 + g_2)$, $2(f_3 + g_3)$, and $2(f_4 + g_4)$ are precomputed. Then, modular multiplication with a constant can be performed faster. We do not exploit this property because our implementation uses the 4-way Montgomery ladder in (Hisil et al., 2022), which does not require any such specialized multiplication. On the other hand, this optimization can be useful in other settings where multiplication by curve constants and base points are to be computed.

95 additions for modular multiplication in $GF(2^{255} - 19)$. So, we do one less addition and save $109 - 75 = 34$ multiplications for `p261`; a prime having 6 more bits than `p25519` and 10 more bits than `p2519`. Table 5.3 provides more detailed comparison of modular multiplication for different primes.

Table 5.3.. Comparison of modular multiplication of different primes without carries.

	p2519	p25519	p2663	p261
Multiplication (vpmuludq)	89	109	109	75
Addition (vpaddq)	80	95	90	85
Subtraction (vpsubq)	0	0	0	9
Data movement (vmovdqa)	11	20	27	18
Cycles (Skylake)	67.10	78.76	82.20	63.25

Table 5.4.. Comparison of modular squaring of different primes without carries.

	p2519	p25519	p2663	p261
Multiplication (vpmuludq)	45	60	55	45
Shift left (vpsllq)	18	0	18	0
Addition (vpaddq)	55	53	62	52
Subtraction (vpsubq)	0	0	0	13
Data movement (vmovdqa)	3	10	12	10
Cycles (Skylake)	44.82	43.24	53.64	41.67

Table 5.5.. Comparison of doing the carries.

	p2519	p25519	p2663	p261
Shift left (vpsllq)	2	2	0	0
Shift left (vpsrlq)	11	12	11	11
Addition (vpaddq)	12	14	13	13
Subtraction (vpand)	11	12	11	11
Cycles (Skylake)	17.48	16.14	24.75	16.45

Finally, we do the adjustments between the limbs in order to make them 26 bits, where we do not have any multiplications by small constants.

Modular squaring. In a specialized squaring routine both A and B takes 15 multiplications, 6 additions, and 4 multiplications by 2. C takes 15 multiplications, 11 additions, and 4 multiplications by 2. These provide some saving in comparison with the modular multiplication routine. Furthermore, one can delay some of the multiplications by 2 until reduction which save even more time. More precisely, our implementation uses 45 multiplications, 52 additions, 13 subtractions, and 10 data movement instructions in total. We note that our implementation produces $2f^2$ rather than f^2, see Section 5.2.

4-way parallel Montgomery ladder. Our DH instances use the curves $\mathcal{M}_{4318}$ and $\mathcal{K}_{276,5}$. The implementation with $\mathcal{M}_{4318}$ is constructed with the 4-way ladder algorithm from (Hisil et al., 2022), which makes no assumption on the underlying field, base points, and curve constants. An alternative algorithm is given in (Nath & Sarkar, 2022a), which works fast if the squaring step can be efficiently distributed to SIMD channels. It is not clear how to perform such a distribution with `p261` which requires frequently horizontal data transfers. The implementation with $\mathcal{K}_{276,5}$ is constructed with the 4-way ladder algorithm from (Karati & Sarkar, 2017).

Doing the carries. Since the arithmetic of `p261` does not involve multiplications with small constants, the limbs are allowed to accommodate some more bits before entering the modular multiplication (or squaring) routine. These extra bits comes from linear operations such as the Hadamard transform. We always delay doing the carries which occur only inside multiplication and squaring routines. This is a side benefit of using the prime `p261` in the Montgomery ladder. The costs regarding doing carries is depicted in Table 5.5.

CHAPTER 6
CONCLUSIONS

This thesis proposes two concrete results. Firstly, a 4-way Montgomery ladder algorithm is proposed, providing a new, convenient, and vectorizable arithmetic for the AVX2 and AVX512 instruction extensions of Intel processors. Also, within this newly proposed ladder, a 9-limb representation of the elements of $GF(2^{255} - 19)$ is proposed. This new representation uses one less number of limbs compared to the common 10-limb notation. Our representation can be interesting on processors with slow multipliers.

Secondly, a framework is proposed to find fast prime numbers. Using this framework, a cryptographically interesting new prime is found, which is $2^{261} - 2^{131} - 1$. The new prime has the distinctive feature of being Karatsuba friendly. That is, several optimizations can be made when Karatsuba multiplication method is used on finite field arithmetic. We put forward a 10-limb representation of this prime and implemented its arithmetic on an AVX2 supported processor. Moreover, related finite field multiplication algorithms are reviewed. In its most general form, the Karatsuba and Schoolbook algorithms have been explored to find a better formula for performing a finite field multiplication.

REFERENCES

Barrett, P. (1987). Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko (Ed.), *Advances in cryptology — crypto' 86* (pp. 311–323). Springer Berlin Heidelberg.

Bernstein, D. (2006). Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, & T. Malkin (Eds.), *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* (Vol. 3958, pp. 207–228). Springer. Retrieved from https://doi.org/10.1007/11745853_14 doi: 10.1007/11745853_14

Bernstein, D., Chuengsatiansup, C., Lange, T., & Schwabe, P. (2014). Kummer strikes back: New DH speed records. In P. Sarkar & T. Iwata (Eds.), *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* (Vol. 8873, pp. 317–337). Springer. Retrieved from https://doi.org/10.1007/978-3-662-45611-8_17 doi: 10.1007/978-3-662-45611-8_17

Bernstein, D., & Lange, T. (2017). Montgomery curves and the montgomery ladder. In J. W. Bos & A. K. Lenstra (Eds.), *Topics in computational number theory inspired by Peter L. Montgomery* (pp. 82–115). Cambridge University Press. doi: 10.1017/9781316271575.005

Bernstein, D., & Schwabe, P. (2012). NEON crypto. In E. Prouff & P. Schaumont (Eds.), *Cryptographic hardware and embedded systems – CHES 2012* (Vol. 7428, pp. 320–339). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bernstein, D. J., Lange, T., & Rezaeian Farashahi, R. (2008). Binary edwards curves. In E. Oswald & P. Rohatgi (Eds.), *Cryptographic hardware and embedded systems – ches 2008* (pp. 244–265). Springer Berlin Heidelberg.

Bos, J. W., Costello, C., Longa, P., & Naehrig, M. (2016). Selecting elliptic curves for cryptography: An efficiency and security analysis. *Journal of Cryptographic Engineering*, *6*(4), 259-286.

Brier, E., & Joye, M. (2002). Weierstraß elliptic curves and side-channel attacks. In D. Naccache & P. Paillier (Eds.), *Public key cryptography, 5th international workshop on practice and theory in public key cryptosystems, PKC 2002, paris, france, february 12-14, 2002, proceedings* (Vol. 2274, pp. 335–345). Springer. Retrieved from https://doi.org/10.1007/3-540-45664-3_24 doi:

10.1007/3-540-45664-3_24

Castryck, W., Galbraith, S., & Farashahi, R. R. (2008). *Efficient arithmetic on elliptic curves using a mixed edwards-montgomery representation.* Cryptology ePrint Archive, Paper 2008/218. Retrieved from https://eprint.iacr.org/2008/218 (https://eprint.iacr.org/2008/218)

Chou, T. (2015). Sandy2x: New Curve25519 speed records. In O. Dunkelman & L. Keliher (Eds.), *Selected areas in cryptography - SAC 2015 - 22nd international conference, sackville, nb, canada, august 12-14, 2015, revised selected papers* (Vol. 9566, pp. 145–160). Springer. Retrieved from https://doi.org/10.1007/978-3-319-31301-6_8 doi: 10.1007/978-3-319-31301-6_8

Chudnovsky, D., & Chudnovsky, G. (1986). Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, *7*(4), 385–434.

Costello, C., & Smith, B. (2018). Montgomery curves and their arithmetic - The case of large characteristic fields. *J. Cryptographic Engineering*, *8*(3), 227–240. Retrieved from https://doi.org/10.1007/s13389-017-0157-6 doi: 10.1007/s13389-017-0157-6

Costigan, N., & Schwabe, P. (2009). Fast elliptic-curve cryptography on the Cell Broadband Engine. In B. Preneel (Ed.), *Progress in cryptology – africacrypt 2009* (pp. 368–385). Berlin, Heidelberg: Springer Berlin Heidelberg.

Diffie, W., & Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *22*(6), 644-654. doi: 10.1109/TIT.1976.1055638

Farashahi, R., & Hosseini, G. (2016, 03). Differential addition on binary elliptic curves. In (p. 21-35).

Farashahi, R. R., & Hosseini, S. G. (2017). Differential addition on twisted edwards curves. In J. Pieprzyk & S. Suriadi (Eds.), *Information security and privacy* (pp. 366–378). Springer International Publishing.

Faz-Hernández, A., López, J., & Dahab, R. (2019, jul). High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, *45*(3). Retrieved from https://doi.org/10.1145/3309759 doi: 10.1145/3309759

Gaudry, P. (2007). Fast genus 2 arithmetic based on Theta functions. *Journal of Mathematical Cryptology (JMC)*, *1*(3), 243–265.

Gaudry, P., & Lubicz, D. (2009). The arithmetic of characteristic 2 Kummer surfaces

and of elliptic Kummer lines. *Finite Fields and Their Applications*, *15*(2), 246 - 260. Retrieved from http://www.sciencedirect.com/science/article/pii/S1071579708000804 doi: https://doi.org/10.1016/j.ffa.2008.12.006

Gaudry, P., & Schost, E. (2012). Genus 2 point counting over prime fields. *J. Symb. Comput.*, *47*(4), 368–400. Retrieved from http://dx.doi.org/10.1016/j.jsc.2011.09.003 doi: 10.1016/j.jsc.2011.09.003

Hamburg, M. (2015). *Ed448-Goldilocks, a new elliptic curve.* Cryptology ePrint Archive, Paper 2015/625. Retrieved from https://eprint.iacr.org/2015/625 (https://eprint.iacr.org/2015/625)

Hankerson, D., Menezes, A. J., & Vanstone, S. (2003). *Guide to elliptic curve cryptography*. Springer-Verlag.

Hisil, H., Egrice, B., & Yassi, M. (2022). Fast 4 way vectorized ladder for the complete set of Montgomery curves. *International Journal of Information Security Science*, *11*(2), 12 - 24.

Karati, S., & Sarkar, P. (2017). Kummer for genus one over prime order fields. In T. Takagi & T. Peyrin (Eds.), *Advances in cryptology – asiacrypt 2017* (pp. 3–32). Cham: Springer International Publishing.

Karatsuba, A., & Ofman, Y. (1962, 12). Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, *7*, 595.

Knuth, D. E. (1997). The art of computer programming, volume 2 (3rd ed.): Seminumerical algorithms. In (p. 265-294). Addison-Wesley Longman Publishing Co., Inc.

Koblitz, N. (1987, January). Elliptic curve cryptosystems. *Mathematics of Computation*, *48*(177), 203–209.

López, J., & Dahab, R. (1999). Fast multiplication on elliptic curves over GF(2^m) without precomputation. In Ç. Koç & C. Paar (Eds.), *Cryptographic hardware and embedded systems, first international workshop, ches'99, worcester, ma, usa, august 12-13, 1999, proceedings* (Vol. 1717, pp. 316–327). Springer. Retrieved from https://doi.org/10.1007/3-540-48059-5_27 doi: 10.1007/3-540-48059-5_27

Miller, V. (1985). Use of elliptic curves in cryptography. In *CRYPTO'85* (Vol. 218, pp. 417–426). Springer.

Montgomery, P. (1987). Speeding the Pollard and elliptic curve methods of factorization.

Mathematics of computation, *48*(177), 243–264.

Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of Computation*, *44*(170), 519–521.

Nath, K., & Sarkar, P. (2018). *Efficient arithmetic in (pseudo-)mersenne prime order fields.* Cryptology ePrint Archive, Paper 2018/985. Retrieved from https://eprint.iacr.org/2018/985 (https://eprint.iacr.org/2018/985)

Nath, K., & Sarkar, P. (2022a). Efficient 4-way vectorizations of the Montgomery ladder. *IEEE Transactions on Computers*, *71*(3), 712-723. doi: 10.1109/TC.2021.3060505

Nath, K., & Sarkar, P. (2022b). Kummer versus Montgomery face-off over prime order fields. *ACM Trans. Math. Softw.*, *48*(2), 1-28. Retrieved from https://doi.org/10.1145/3503536 doi: 10.1145/3503536

Nath, K., & Sarkar, P. (2022c). Security and efficiency trade-offs for elliptic curve Diffie-Hellman at the 128-bit and 224-bit security levels. *Journal of Cryptographic Engineering*, *12*, 107-121.

Oliveira, T., López, J., Hışıl, H., Faz-Hernández, A., & Rodríguez-Henríquez, F. (2017). How to (pre–)compute a ladder – improving the performance of X25519 and X448. In C. Adams & J. Camenisch (Eds.), *Selected areas in cryptography - SAC 2017 - 24th international conference, ottawa, on, canada, august 16-18, 2017, revised selected papers* (Vol. 10719, pp. 172–191). Springer. Retrieved from https://doi.org/10.1007/978-3-319-72565-9_9 doi: 10.1007/978-3-319-72565-9_9

Renes, J., & Smith, B. (2017). qDSA: small and secure digital signatures with curve-based Diffie-Hellman key pairs. In T. Takagi & T. Peyrin (Eds.), *Advances in cryptology - ASIACRYPT 2017* (Vol. 10625, pp. 273–302). Springer. Retrieved from https://doi.org/10.1007/978-3-319-70697-9_10 doi: 10.1007/978-3-319-70697-9_10

Rivest, R., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*, 120–126.

Schneier, B. (1994). Description of a new variable-length key, 64-bit block cipher (Blowfish). In R. Anderson (Ed.), *Fast software encryption* (pp. 191–204). Springer Berlin Heidelberg.

APPENDIX A

SUPPLEMENTARY CODE

The following Maple script checks the multiplication formulas modulo `p261`.

```
k:=2^26:

f:=f0+k^5*f1:
g:=g0+k^5*g1:

f0:=f00+f01*k+f02*k^2+f03*k^3+f04*k^4:
f1:=f10+f11*k+f12*k^2+f13*k^3+f14*k^4:

g0:=g00+g01*k+g02*k^2+g03*k^3+g04*k^4:
g1:=g10+g11*k+g12*k^2+g13*k^3+g14*k^4:

######## Multiplication starts ########
A0:=f00*g00:
A1:=f00*g01+f01*g00:
A2:=f00*g02+f01*g01+f02*g00:
A3:=f00*g03+f01*g02+f02*g01+f03*g00:
A4:=f00*g04+f01*g03+f02*g02+f03*g01+f04*g00:
A5:=f01*g04+f02*g03+f03*g02+f04*g01:
A6:=f02*g04+f03*g03+f04*g02:
A7:=f03*g04+f04*g03:
A8:=f04*g04:

B0:=f10*g10:
B1:=f10*g11+f11*g10:
B2:=f10*g12+f11*g11+f12*g10:
B3:=f10*g13+f11*g12+f12*g11+f13*g10:
B4:=f10*g14+f11*g13+f12*g12+f13*g11+f14*g10:
B5:=f11*g14+f12*g13+f13*g12+f14*g11:
B6:=f12*g14+f13*g13+f14*g12:
B7:=f13*g14+f14*g13:
B8:=f14*g14:

f20:=f00+f10: g20:=g00+g10:
f21:=f01+f11: g21:=g01+g11:
f22:=f02+f12: g22:=g02+g12:
f23:=f03+f13: g23:=g03+g13:
f24:=f04+f14: g24:=g04+g14:

C0:=f20*g20:
C1:=f20*g21+f21*g20:
C2:=f20*g22+f21*g21+f22*g20:
C3:=f20*g23+f21*g22+f22*g21+f23*g20:
C4:=f20*g24+f21*g23+f22*g22+f23*g21+f24*g20:
C5:=f21*g24+f22*g23+f23*g22+f24*g21:
C6:=f22*g24+f23*g23+f24*g22:
C7:=f23*g24+f24*g23:
C8:=f24*g24:

Z0:=B0+2*A0+C5-A5:
Z1:=B1+2*A1+C6-A6:
Z2:=B2+2*A2+C7-A7:
Z3:=B3+2*A3+C8-A8:
Z4:=B4+2*A4:
Z5:=B5+2*(C5+C0-A0):
Z6:=B6+2*(C6+C1-A1):
Z7:=B7+2*(C7+C2-A2):
Z8:=B8+2*(C8+C3-A3):
Z9:= 2*( C4-A4):

######## Multiplication ends ########

expand(2*f*g-(
   Z0*k^0+Z1*k^1+Z2*k^2+Z3*k^3+Z4*k^4+
   Z5*k^5+Z6*k^6+Z7*k^7+Z8*k^8+Z9*k^9
)) mod (2*k^10-2*k^5-1); # Check.
```

Code A.1. Maple scripts for modulo p261 - multiplication