Studies on Efficient and Secure Implementation of Elliptic and Hyperelliptic Curve Cryptosystems

Thesis submitted to Indian Statistical Institute

by

Pradeep Kumar Mishra
Applied Statistics Unit
Indian Statistical Institute

2004

Studies on Efficient and Secure Implementation of Elliptic and Hyperelliptic Curve Cryptosystems

Thesis submitted to Indian Statistical Institute in partial fulfillment
of the requirements for the award of the degree of
Doctor of Philosophy

by

Pradeep Kumar Mishra
Applied Statistics Unit
Indian Statistical Institute
203 B. T. Road, Kolkata - 700 108, INDIA
e-mail : pradeep_t@isical.ac.in

under the supervision of

Dr. Palash Sarkar
Applied Statistics Unit
Indian Statistical Institute
203 B. T. Road, Kolkata - 700 108, INDIA
e-mail : palash@isical.ac.in

To Jyotsna

Acknowledgments

Several people have inspired me while working for this thesis in last couple of years. It gives me ample pleasure to thank them all after its completion. The first name which comes to my mind at this juncture is that of my teacher, friend and philosopher, Prof Bimal Roy, father of Cryptology research in India. He has remained a great source of inspiration for me throughout. I am greatly indebted to him for all kinds of support (both academic and non-academic) I have received from him while working for this dissertation.

I am also grateful to Prof Rana Barua, who initiated me into elliptic and hyperelliptic curve cryptography. He had given me the full oppertunity of taking advantage of his erudition.

I am profoundly grateful to my supervisor Dr Palash Sarkar, who has given me ample scope to make use of his expertise and experience while I was working for this dissertation. Particularly I am thankful to him for keeping his door open for me whenever I found some stumbling blocks on my way. His busy schedule has never come in my way of getting my doubts cleared.

I thank Dr. Subhamoy Maitra and Dr Pinakpani Pal for their help and encouragement.

I am grateful to Prof Alfred Menezes, who acted for me like an on line Guru. I have learnt a great deal from him through the numerous emails we have exchanged. Also, I am grateful to him for inviting me to University of Waterloo for a short but meaningful interaction in 2003. I can not forget the encouragement and help I have received from Dr Tanja Lange, who always requested me to keep her updated on my work. Her generous help has contributed significantly in enhancing the content and presentation of the work [107]. I am thankful to Prof Darrel Hankerson, whom I consulted several times while working for this thesis.

Also, the inspirations received from my “friend-in-need” (Dr K.) Srini(vas) will ever remain fresh in my mind.

I thank all my co-authors, working with whom was not only enlightening but also real fun. I must thank all anonymous referees of my published papers, whose comments have always added a new dimension to my works.

I also thank all members of Cryptology Research Group, Indian Statistical Institute for their enlightening and friendly interactions and to all the staff of ASU for providing all kinds of facilities whenever needed.

I am greatly indebted to all in my family and relations who have always supported my endeavours. Their support and encouragement has been an unfailing means to reach this end.

In Hindu tradition wife is considered *ardhangini*, one’s other half. So thanking my wife is tantamount to thanking myself. But I can not help mentioning here that this work could not have been a reality without the tremendous support from my wife Jyotsna. My son Siddhartha has always remained very nice and co-operative. (I am not sure, if this thesis is not a culmination of his prayers.)

Also, I must thank all my friends – Kishan, Sandeepan, Tanmay, Avishek, Sourav (both versions, Mukherjee and Benerjee), Joydeep, Sanjit, Partha, Deepak, Rajasekhar, Ratna, Sagarnil, Madhu, Chandan, $\cdots$ who selflessly cooperated with me in all possible ways and stood by me in hard times. I must thank them for all the fun and frolic we had together in last couple of years. Bishwa, Parashar, Mundbhai, Veena, Sarif, Sanjay, Prasanta, Ganesh, Venku, Kummi, Sangame, Sujit, Sudeshna are constant sources encouragement and inspiration.

Contents

Chapter 1

Introduction

In this thesis we study efficient and secure implementation of elliptic and hyperelliptic curve cryptosystems. These are two important classes of public key cryptosystems, which can be implemented efficiently over a variety of platforms, starting right from small handheld devices like smart cards and mobile phones to large capacity work-stations.

All known public key cryptosystems (PKC) derive their security from some computationally hard mathematical problems. The most popular ones are based mainly over two such problems:

1. Integer factorization problem,
2. Discrete logarithm problem.

The RSA cryptosystem is based on the hardness of integer factorization problem. Elliptic and hyperelliptic curve cryptosystems are based on discrete logarithm problem over their respective groups.

Let G be a large cyclic multiplicatively written group with generator g. Given any element $y = g^{\alpha}$ in the group, computing α is called the discrete logarithm problem (DLP) over G. Many groups have been proposed in literature over which the DLP is believed to be hard. Once such a group is fixed, an ElGamal [40] type cryptosystem can be built over it.

In 1985, Koblitz [76] and Miller [106] independently proposed the set of elliptic curve points over a finite field to be used for cryptographic purposes. Prior to that, elliptic curves had a wide literature developed for over a century by mathematicians purely to satisfy their aesthetic sense. Only recently various applications of elliptic curves are being used in many applied areas like coding theory, number theoretic algorithms for integer factorization and primality testing and cryptography. The main advantage of elliptic curve cryptography is that for suitably chosen curves there is no known subexponential algorithm (like Number

Field Sieve (NFS) algorithm for integer factorization) to solve ECDLP (elliptic curve discrete logarithm problem). This leads to smaller key length in comparison to other PKC's like RSA and other systems based on DLP with equivalent security. The smaller key length, in turn, leads to faster encryption and decryption, savings in bandwidth and efficient implementation. Thus ECC is a very suitable cryptosystem for resource constrained devices like smart cards and other mobile and wireless devices.

In 1987, Koblitz [77] showed the Jacobians of hyperelliptic curves over finite fields to be a rich source of finite cyclic groups suitable for cryptographic use. Hyperelliptic curves are elliptic curves of higher genera. As with ECDLP, there is no known subexponential time algorithm to solve HCDLP for suitably chosen curve parameters (specifically for curves of lower genera). So hyperelliptic curve cryptosystems (HECC) have the same advantages as ECC. Moreover, in HECC the underlying finite field is smaller and there are many more curves to chose from. However, [2] showed that curves of higher genera (≥ 5) are not suitable for cryptography. Also, arithmetic in HECC is much more complex than that in ECC. Due to this inherent complexity in the group operations in the Jacobian of the hyperelliptic curves, HECC, with all its advantages has not yet been able to establish itself as a feasible commercial solution to current cryptographic needs. Several groups of researchers in different part of the globe are working in this area in order to successfully employ hyperelliptic curves in the implementation of various cryptographic primitives. Recently, in an EU sponsored research project (see [7]) researchers from seven organizations from European and Asian countries have been engaged to work on implementation of HECC and they have unearthed many interesting results.

Initially, cryptanalysis was in the realm of mathematicians. The underlying computationally hard problems, which provide the security to each of the public key cryptographic schemes, pose a challenge to the genius of mathematicians. In 1996, Kocher [80] changed the cryptanalytic scenario. He extended cryptanalysis to application domain from the mathematical domain. In this path breaking work, he proposed side channel attacks (SCA). SCA instead of attacking the underlying hard problem, attacks the specific implementation by measuring and analyzing side channel information like timing, power consumption and electro-magnetic (EM) radiation traces of the computation. Side-channel attacks are believed to be the most potential threat to ECC and HECC, which are employed in hand-held mobile devices and used in hostile outdoor environments.

Thus modern security threat to ECC and HECC are two-fold: one based on the generic cryptanalytic attacks and the others based on side-channel attacks. The generic attacks can be avoided by carefully choosing the security parameters of the system, like the curve itself, the underlying field and its representation, the group order, the cofactor, the base point etc. Any naive implementation which neglects security aspects of these parameters can be broken by the generic attacks. For example, if the group order is large, but is divisible by only small primes, the DLP can be broken by Pohlig-Hellman attack [134].

The second threat to ECC and HECC is from side-channel attacks. As mentioned earlier, these were first proposed by Kocher in 1996. A stronger power based attack was proposed by Kocher et al [81]. Many attacks based on analysis of side-channel information were proposed by various authors (see [126] for example). Kocher, in his original work, had proposed timing attacks, which works on the basis of analysis of the timing of computation of various protocols in the system. Okeya and Sakurai [124] observed that even if a system is secure against timing attacks, it may fail to withstand power attacks. In [5], the authors have observed that systems secure against power analysis attacks may not be secure against EM based attacks. Goubin [57] proposed a new attack based on naive choice of the base point. In [169, 170] the authors have discussed that fault analysis can be used for side-channel attacks and countermeasures.

To immunize a cryptosystem from SCA the various algorithms and protocols constituting the cryptosystem are to be designed carefully. Many proposals have come from the research community to immunize ECC and HECC against SCA.

In the design of any cryptosystem, efficiency and security are two main concerns. Our contributions are in these two main areas. In the efficiency front, our strategies consists of both modification of existing schemes and proposing new algorithms. For the first time we propose parallel implementation of hyperelliptic curve arithmetic [109] and pipelined implementation of ECC [107]. Use of Frobenius map to compute scalar multiplication in ECC and HECC has been studied in the literature. In [140], we have proposed a new table look-up method to speed-up computations in ECC and HECC.

Group arithmetic in affine coordinates in both ECC and HECC involves field element inversions. Field element inversions are considered very costly operation (see [103, 45, 66]) particularly over prime fields. In [111], we have proposed one algorithm for ECC and HECC with a fixed base point, which uses affine arithmetic and achieves significant speed-up in most cases. The algorithm uses a small look-up table and hence uses slightly more memory. In [139], we have used similar technique to design an parallel algorithm which uses affine coordinates and computes the scalar multiplication in ECC efficiently.

1.1 Plan of the Thesis

This thesis is based on eight articles [109, 111, 110, 139, 140, 107, 89, 108] and is organized as follows.

Our contribution can be broadly divided into two parts. The first part consists of our contributions for efficient and secure point arithmetic. The articles [109, 89, 108] fall in this category. We have described them in Chapters 3, 8 and 4 respectively. The other part of the thesis (Chapters 5, 6, 7 and 9) is devoted to efficient and secure computation of scalar multiplication in (H)ECC.

In *Chapter 1*, an introduction to ECC and HECC has been presented. The material of the section is intended to serve as a motivation for the rest of the thesis.

In *Chapter 2*, we present the necessary preliminary material required in the later chapters. It provides a brief description of group operation, group structure, group order in ECC and HECC and other issues related to the efficiency and security of the cryptosystem. The operation of scalar multiplication is very fundamental in any implementation. We discuss various methods proposed in literature to compute the scalar multiplication efficiently and securely. It also briefly describes various attacks proposed in literature against elliptic and hyperelliptic curve cryptosystems and proposed countermeasures. Side-channel attacks are believed to be the most potential threat against ECC and HECC. We devote one section of this chapter in describing various side channel attacks and the countermeasures proposed against such attacks.

One of the recent thrust areas in research on hyperelliptic curve cryptography has been to obtain explicit formulae for performing arithmetic in the Jacobian of such curves. In *Chapter 3*, we continue this line of research by obtaining parallel versions of such formulae. Our first contribution in this direction is to develop a general methodology for obtaining parallel algorithm for any explicit formula. Any parallel algorithm obtained using our methodology is provably optimal in the number of multiplication rounds. We next apply this methodology to Lange's explicit formula [84, 85, 86] for arithmetic in genus 2 hyperelliptic curve – both for the affine coordinate and inversion-free arithmetic versions. Since encapsulated add-and-double algorithm is an important countermeasure against side channel attacks, we develop parallel algorithms for encapsulated add-and-double for both of Lange's versions of explicit formula. For the case of inversion free arithmetic, we present parallel algorithms using 4, 8 and 12 multipliers. All parallel algorithms described in this chapter are optimal in the number of parallel rounds. One of the conclusions from our work is the fact that the parallel version of inversion free arithmetic is more efficient than the parallel version of arithmetic using affine coordinates.

Elliptic (ECC) and hyperelliptic (HECC) curve cryptosystems have emerged as the cryptosystem for the small handheld and mobile devices. A lot of research has been devoted to their secure and efficient implementation. As all such devices come with very limited amount of resources, efficient memory management is an important issue in all such implementations. HECC is now generally implemented via so called explicit formula. In literature there is no result which focuses on the exact amount of memory requirement for implementation of these formulae. In *Chapter 4*, we report such minimum memory requirement. Also, we provide a general methodology to find such minimum memory requirement for any explicit formula. Applying this methodology we settle the issue for some formulae appearing in literature.

In *Chapter 5*, we propose a sequential scalar multiplication algorithm for elliptic and hyperelliptic curve cryptosystems, which uses affine arithmetic and is resistant against simple power attacks. Also, using a modification of known techniques the algorithm can be made immune against differential power attacks. The algorithm uses Montgomery's trick and a precomputed table consisting of multiples of the base point. Consequently, the algorithm is useful in a scenario where the base point is fixed, like ElGamal encryption or signature generation [40]. Under such circumstances, for hyperelliptic curves, the algorithm compares favourably with other known algorithms over all fields. For elliptic curves, under similar circumstances, the algorithm performs better than other algorithms over prime fields. The increase in speed is due to a proper application of Montgomery's trick to efficiently perform the simultaneous inversion of several field elements.

Montgomery's trick is a well known technique for performing simultaneous inversions of several field elements. However, this technique is a strictly sequential algorithm. In *Chapter 6*, we introduce two parallel algorithms for performing simultaneous inversions of several finite field elements [139, 110]. First of the algorithms [139] uses a binary tree and can perform inversions of 2^r elements using $3 \times 2^{r-1}$ multipliers in $(r+1)$ multiplication rounds and one inversion round. Although this performs a larger number of multiplications than Montgomery's trick, it takes much less number of parallel multiplication rounds if sufficient number of multipliers are available. We provide another parallel algorithm [110] which does not increase the number of multiplications in comparison to Montgomery's algorithm, but does not provide as much scope for parallelism as the first algorithm. We also describe how to modify the algorithms when less number of multipliers are available. These parallel algorithms are used to obtain new parallel algorithms for elliptic curve scalar multiplication using a fixed base point. The scalar multiplication algorithm is resistant against simple power analysis and can be implemented with different number of multipliers $(2, 4, 8, \ldots)$. Results show that implementation with 2 multipliers can lead to almost 40% speed-up over previously best known sequential SPA resistant algorithm.

In *Chapter 7*, we propose a pipelining scheme for implementing Elliptic Curve Cryptosystems (ECC). The scalar multiplication is the dominant operation in ECC. It is computed by a series of point additions and doublings. The pipelining scheme is based on a key observation: to start the subsequent operation one need not wait until the current one exits. The subsequent operation can begin while a part of the current operation is still being processed. The scheme requires slightly more hardware support than generally available to ECC. The pipelining scheme is resistant to side-channel attacks (SCA). On the performance front, it compares favourably against all known SCA-resistant sequential and parallel methods of computing the scalar multiplication.

As mentioned earlier, hyperelliptic curve cryptosystem is a suitable cryptosystem for resource constrained devices like smart cards and side-channel attacks are one of the most potential threats against HECC. So efficient algorithms resistant against side channel attacks are the need of the hour. In *Chapter 8*, we propose addition and doubling algorithms for

HECC immunized against side-channel attacks. The basic explicit formulae we use, were proposed by Lange in [84], [86] and are the most economic ones among all the formulae occurring in literature for general hyperelliptic curves of genus 2. Also side-channel atomicity [23] is the most efficient countermeasure against simple power attacks (SPA). We combine these two ideas to propose hyperelliptic curve divisor addition and doubling algorithms. To make the formulae resistant against DPA, various countermeasures like curve randomization or private key randomization can be used. The beauty of these algorithms is that they can be implemented in parallel.

In *Chapter 9*, we describe a new scalar multiplication algorithm for elliptic and hyperelliptic curve cryptosystems. The algorithm is obtained by combining Koblitz's idea of using Frobenius automorphism along with a very special kind of look-up table. In the case where the base point is unknown, we present an efficient algorithm to compute the look-up table online. Our algorithm applies to prime power fields $GF(p^n)$. One important subclass of such fields are Optimal Extension Fields (OEF's) which are believed to be ideal for efficient implementation of cryptographic primitives. Over prime power fields, our algorithm compares favourably to other known algorithms for scalar multiplication.

This area of research has many open problems whose solutions have ample practical value. The discovery of side-channel attacks has thrown an open challenge to cryptographic community. Although many attacks and countermeasures have been proposed in literature, no general security model has yet been developed. Besides, combining side-channels is another challenging avenue which has not been explored fully. In fact, our studies has also opened many new avenues to work on. They are not only interesting from a theoretical point of view, but also are of much practical significance. We list a few of them below.

1. The parallelization idea described in Chapter 3 can be extended to curves of higher genus and also to Picard Curves. Explicit formula for addition and doubling over such curves are already available in literature.

2. Extending the pipelining idea to hyperelliptic curve cryptosystems and designing of a elliptic and/or hyperelliptic curve coprocessor which utilizes this idea in hardware.

3. The addition and doubling algorithm in side-channel atomic blocks presented in Chapter 8 are capable of being implemented in parallel by two processors. Can it be extended to 4 or more processors?

4. Many efficiently computable endomorphisms have been identified which are capable of enhancing the efficiency of computation of scalar multiplication in ECC and HECC. It is an interesting idea to see how the new table look up method described in Chapter 9 can be extended to these endomorphisms.

In *Chapter 10*, we describe some of the open problems in this area and conclude the dissertation.

Chapter 2

Basics of Elliptic and Hyperelliptic Curve Cryptosystems

2.1 Introduction

The single most important discovery in the history of modern cryptography is probably the celebrated work by Diffie and Hellman [37]. It gave birth to public key cryptography (PKC). The Diffie-Hellman key exchange protocol proposed in [37] enables two people to share a secret key through an insecure channel. The scheme uses a finite cyclic group. Given a cyclic group G of large order with generator g and a group element g^x, to compute x is considered to be a computationally hard problem, known as *Discrete Logarithm Problem (DLP)*. The Diffie-Hellman key exchange protocol derives its security from the hardness of DLP. Many groups have been identified on which the DLP is believed to be hard. Various variants of DLP has been proposed later. In Computational Diffie-Hellman problem (CDH), for example, one is to compute g^{xy} given g^x and g^y. In Decision Diffie-Hellman problem (DDH), given group elements g^x, g^y and g^z, one is required to decide if $g^{xy} = g^z$. Note that any algorithm to solve CDH can be used to solve DDH. Groups on which one of these problems is computationally hard is used for implementing various cryptographic primitives.

Groups suitable for cryptographic use need to possess three important characteristics. First, each element of the group should have a compact representation, preferably in $\log n$ bits, where n is the group order. Secondly, there should be an efficient algorithm to compute the group operation, where the group elements have compact representation. Thirdly, the DLP must be computationally hard over the group. ElGamal [40] described how DLP can be used for encryption and digital signature schemes.

Since the discovery of Diffie-Hellman, many groups have been proposed in the literature for implementation of cryptographic schemes. These include the multiplicative group of a binary finite field (see [4] for example), the group of units of $\mathbb{Z}_n$ where n is an composite integer [98], the multiplicative subgroup of the prime field $\mathbb{Z}_p$ [144], the group of points on an elliptic curve defined over a finite field ([76] and [106]), the Jacobian of an hyperelliptic curve over a finite field [77]. The reason for searching for newer and newer groups is to find a group, which outperforms others in some or all three respects mentioned in the last paragraph.

The security of a cryptographic primitive defined over a group rests on the strength of DLP over that group. One advantage of the elliptic and hyperelliptic curve based cryptography is that for suitably chosen security parameters (like the curve, the underlying field, the group order etc), there is no known subexponential algorithm to solve the DLP. This leads to the fact that a high level of security can be attained over a group of comparatively much smaller order. Groups of smaller order lead to smaller key length, which in turn leads to faster implementation, bandwidth saving and smaller chip size for a hardware design. Thus elliptic and hyperelliptic curve cryptography are the ideal ones for resource constrained devices like smart cards and other handheld small devices.

Another beauty of elliptic and hyperelliptic curve cryptography is that there are many parameters involved and there are ample choices for each of them. A short list of these parameters: the underlying field, the representation of field element, the curve, the base point and its representation, algorithms for point arithmetic, algorithm for scalar multiplication and so on. Of course arbitrary choices may compromise security. There are restrictions. In spite of these constraints the implementer has a lot of scope to use his discretion. In the next sections we will briefly introduce elliptic and hyperelliptic curve cryptography and provide an overview of all choices in regard to these parameters. In the following sections elliptic curve cryptography (or cryptosystem) will be always abbreviated as ECC, hyperelliptic curve cryptography (or cryptosystem) will be always abbreviated as HECC and when both of them are meant we will use the abbreviation (H)ECC.

2.2 The Underlying Field

In this section, we will briefly describe fields used in (H)ECC. Efficient implementation of field arithmetic is an essential prerequisite of an implementation of (H)ECC and a vast literature exists on the subject. In this dissertation we do not consider efficient algorithms for field arithmetic. For details on the subject discussed in this section the reader can refer to [92, 105, 99, 11, 12].

In (H)ECC, an elliptic or hyperelliptic curve is always involved. The finite field K from which the curve derives its defining coefficients is called the underlying field. The rational points of the curve are members of $L \times L$, where L is an extension field of K. The underlying field and its representation play an important role in the security and efficiency of an implementation. Even the form of the equation representing the curve depends upon the underlying field (see Sections 2.3, 2.4).

Broadly speaking, the types of field used for cryptography are $\mathbf{F}_{p^n}$, where p is prime and $n = 1$ or n is prime. The fields for which $p = 2$ are called binary fields and is an attractive choice (See [21, 95]). In binary fields squaring and inversion are quite cheaper in comparison to fields for which $p > 2$.

The fields for which $p > 2$ and $n = 1$ are called prime fields $\mathbf{F}_p$ $(= \mathbb{Z}_p)$ and form another class of fields popularly used in cryptography. The field operations are carried out modulo the prime p. The efficiency of modular reduction can be enhanced by using primes of some special form. NIST has standardized some of such primes suitable for ECC (e.g. $p = 2^{192} - 2^{64} - 1$).

Another class of fields used in (H)ECC are $\mathbf{F}_{p^n}, p > 2, n > 1$. A special subclass of these fields are called optimal extension fields (see [11, 12]). An OEF is a finite field of the form $\mathbf{F}_p^n$, where (i) p is a pseudo-Mersenne prime and (ii) an irreducible binomial $P(x) = x^n - \omega$ exists over $GF(p)$. The prime p is generally chosen to be very close to the word size of the processor, so that each machine word can accommodate one element of the subfield F_p and each element of the OEF $\mathbf{F}_p^n$, can be accommodated in n words, with minimum wastage of memory. Also, OEF's allow efficient modular reduction for arithmetic in the extension field.

An extension field of degree n is an n-dimensional vector space over the extended field. Like any vector space it can have several bases for representation of its elements. Two popular representations are polynomial basis representation and normal basis representation. Polynomial basis representation is obtained by using the basis $\{1, x, \cdots, x^{n-1}\}$ and working modulo an irreducible polynomial $p(x)$. A field F_{p^n} is said to have a normal basis if it has a basis (over F_p) of the form $\{\alpha, \alpha^p, \cdots, \alpha^{p^{n-1}}\}$, where α is a primitive element of the field F_p. Any element of the field can be represented as $x = \sum_{j=0}^{n-1} a_j \alpha^{q^j}$ or briefly as an ordered n-tuple $x = (a_0, \cdots, a_{n-1})$. In the field F_{q^n}, we have, $x^q = (\sum_{j=0}^{n-1} a_j \alpha^{q^j})^q = (\sum_{j=0}^{n-1} a_j \alpha^{q^{j+1}})$. Thus if x is represented by the tuple $(a_0, \cdots, a_{n-1})$ then x^q is represented by $(a_{n-1}, a_0, \cdots, a_{n-2})$, as $\alpha^{q^n} = 1$. With a normal basis representation of elements, x^q can be computed from x by a circular shift operation only. Thus, over a binary field squaring becomes almost for free.

There have been several proposals for representation of the group elements in (H)ECC (see Section 2.8). The particular choice of a specific representation depends upon the choice of the underlying field. The ratio of cost of an inversion to that of a multiplication, known as the *I/M ratio* affects this choice. The *I/M ratio* has been reported to be between 8 to 10 for binary fields and between 30 to 40 for prime fields [45]. In affine representation of the group elements in (H)ECC the group operations involve inversion. Therefore, for the fields over which *I/M ratio* is higher the affine coordinates are generally avoided in an implementation.

2.3 Introduction to Hyperelliptic Curves

Koblitz [77] in 1987 proposed that the Jacobians of hyperelliptic curves can be a rich source of abelian groups suitable for cryptography. Besides public key cryptography, hyperelliptic curves have diverse applications in several other areas of research activities like primality proving [3], design of error correcting codes [19] and integer factorization problem [90]. In this section we give a brief introduction to the theory of hyperelliptic curves. For details readers can refer to [102, 83, 79].

2.3.1 Definition and Elementary Concepts

Definition 1 *Let K be a field and let $\overline{K}$ be the algebraic closure of K. A hyperelliptic curve C of genus g ($g \geq 1$)over K is an equation of the form*

$$C : y^2 + h(x)y = f(x) \tag{2.1}$$

where $h(x)$ in $K[x]$ is a polynomial of degree at most g, $f(x)$ in $K[x]$ is a monic polynomial of degree $2g+1$ and there is no solution (x, y) in $\overline{K} \times \overline{K}$, which simultaneously satisfies the equations

$$y^2 + h(x)y = f(x) \tag{2.2}$$

$$2y + h(x) = 0 \tag{2.3}$$

$$h'(x)y - f'(x) = 0. \tag{2.4}$$

A singular point on C is a solution $(x, y) \in \overline{K} \times \overline{K}$ which simultaneously satisfies the equations 2.2, 2.3 and 2.4. Thus, a hyperelliptic curve, by definition, does not have any singular points. Over fields of odd characteristic, the polynomial $h(x)$ is zero or can be made zero by a suitable transformation.

Definition 2 *Let L be an extension field of K. The set of $L-$rational points on C, denoted $C(L)$, is the set*

$$\{(x, y) \in L \times L : y^2 + h(x)y = f(x)\} \cup \{\mathcal{O}\}$$

where $\mathcal{O}$ is a special point, called the point at infinity .

When there is no confusion, the set of points $C(L)$ will simply be denoted by C. The points in C other than $\mathcal{O}$ are called finite points. The opposite point of a finite point $P(x, y)$ on the curve C is defined to be the point $\tilde{P}(x, -y - h(x))$. We also define the opposite of $\mathcal{O}$ to be itself. A point is said to be a special point if it is equal to its opposite. $\mathcal{O}$ is a special point.

Definition 3 *Let I be the ideal of $K[x,y]$ generated by the polynomial $x^2+h(x)v-f(x)$ i.e. $I=(x^2+h(x)y-f(x))$. The quotient ring of $K[x,y]/I$ is called the coordinate ring of C over K, denoted by $K[C]$. Similarly, one can define coordinate ring or the set of polynomial functions of C over $\overline{K}$.*

It is simple to check that every polynomial $G(x,y) \in K[x,y]$ has a unique representation of the form

$$G(x,y) = a(x) - b(x)y$$

for some polynomials $a(x), b(x) \in K[x]$.

It can be easily seen that the ring of polynomial functions on C is an integral domain. So it can be embedded on the field of its rationals. The elements of this field are called rational functions. More formally,

Definition 4 *The function field $K(C)$ (resp $\overline{K}(C)$) of C over K (resp $\overline{K}$) is the field of all fractions of polynomial functions in $K[C]$ (resp $\overline{K}[C]$). An element of $K(C)$ is called a rational function on C. A polynomial function is also a rational function. That is so as $K[C]$ is a subring of $K(C)$.*

We define order of a rational function R to be non-zero at the points where it vanishes (*zeros*) or where it becomes infinity (*poles*). Order of R at these points determine what is called the divisor of R. The divisors of rational functions play an important role in hyperelliptic curve cryptography.

Definition 5 *Let $R \in K(C)$, be a rational function on C and let $P \in C$, be a point on C. The point P is said to be a zero of R if $R(P) = 0$ and P is a pole of R if $R(P) = \mathcal{O}$.*

The order of a polynomial function at a point is defined in terms of a rational function, called the uniformising parameter of the corresponding point. The next theorem states the basic result on uniformising parameter.

Theorem 6 *Let $P \in C$. A rational function $U \in K(C)$ with $U(P) = 0$ is said to be a* uniformising parameter *for P if the following condition holds: let $G \in K[C]^*$ be any non-zero polynomial function on C. Then there exists an integer d and a function $S \in K(C)$ which neither has a pole nor a zero at P and $G = U^d S$. Moreover, the value of d is independent of the choice of U.*

The existence of uniformising parameter can be proved by construction. If $P(a,b)$ is an ordinary point then it can be seen that $U = (x-a)$ can be a uniformising parameter. For a special point $P(a,b)$, $U = (y-b)$ and if $P = \infty$ then $U = (x^g)/y$ do the same.

Uniformising parameters are used to define the order of rational functions.

Definition 7 *Let $G \in K[C]^*$ be a polynomial over C. The order of G at P is defined to be d, if $G = U^d S$, where U is a uniformising parameter of P and S is a rational function such that $S(P) \neq 0$ nor $S(P) \neq \infty$. If $R = G/H$ then order of R at P is denoted by $ord_P(R)$ and is defined to be $ord_P(R) = ord_P(G) - ord_P(H)$.*

2.3.2 Divisors

Unlike the points on an elliptic curve, the rational points of an hyperelliptic curve do not form a group. The group provided by hyperelliptic curves for cryptography is a subgroup of the free abelian group $\mathcal{D}$ generated by the set of points on the curve. Let C be a hyperelliptic curve of genus g over a finite field K. The elements of $\mathcal{D}$ are called *divisors*. Divisors are of the form

$$D = \sum_{P \in C} m_P P, \qquad m_P \in \mathbb{Z} \text{ and } P \in C$$

and only finitely many of m_P's are nonzero. We define,

Definition 8 *The degree of D is the integer $\sum_{P \in C} m_P$ and order of D at P is, $ord_P(D) = m_P$.*

It is simple to see that the set $\mathcal{D}_0$ of all divisors of degree 0 forms a subgroup of $\mathcal{D}$. Let R be a rational function on C. The divisor of R is defined to be $div(R) = \sum_{P \in C} m_P P$, where $m_P = ord_P(R)$. A divisor D is called a principal divisor if $D = div(R)$ for some rational function R. A principal divisor is a divisor of degree 0 and this is a simple consequence of the fact that the sum of orders over all zeros and poles of a rational function is 0. The set of all principal divisors $\mathcal{P}$ is a subgroup of $\mathcal{D}_0$. The quotient group $\mathcal{D}_0/\mathcal{P}$ is called the Jacobian of the curve C. Two divisors D_1 and D_2 are said to be equivalent (written as $D_1 \sim D_2$) if they belong to the same coset in $\mathcal{D}_0/\mathcal{P}$.

The support of a divisor D, denoted by Supp(D) is the set of all points P on the curve C at which the weight of the divisor m_P is non-zero.

Definition 9 *A semi-reduced divisor D is a divisor of the form $\sum m_i P_i - (\sum m_i)\mathcal{O}$, where,*

1. *each $m_i \geq 0$*
2. *P_i's are finite points such that when $P_i \in supp(D)$ then $\widetilde{P_i} \notin supp(D)$*
3. *if $P_i = \tilde{P_i}$ then $m_i = 1$.*

It can be proved that, to every divisor D is equivalent to a semi-reduced divisor $D' = \sum m_i P_i - (\sum m_i)\mathcal{O}$ such that $\sum m_i \le g$, where g is the genus of the curve C. Such a divisor is called a *reduced divisor*. Thus each coset of the quotient group $\mathcal{J} = \mathcal{D}_0 \, / \mathcal{P}$ has exactly one reduced divisor. Hence we can identify each coset with its reduced divisor.

Let C be a hyperelliptic curve of genus g defined over a finite field K and let $\mathcal{J}$ be the Jacobian of C. If $P = (x, y) \in C$, and σ is an automorphism of $\overline{K}$ over K, then P^σ, defined as (x^σ, y^σ), is also a point on C. A divisor $D = \sum m_P P$ is said to be defined over K if $D^\sigma = \sum m_P P^\sigma$ is equal to D for all automorphism σ of $\overline{K}$ over K. It is not difficult to see that a principal divisor is defined over K if and only if it is the divisor of a rational function that has coefficients in K. Let $J_C(K)$ be the set of all divisor classes in $\mathcal{J}$ that have a representative that is defined over K. $J_C(K)$ is a subgroup of $\mathcal{J}$. $J_C(K)$ is the group which is used for cryptographic purposes.

The next theorem specifies an elegant way to represent semi-reduced divisors as a pair of polynomials of small degree over K. It uses the following definition:

Definition 10 *The gcd of two divisors $D_1 = \sum_{P \in C} m_P P$ and $D_2 = \sum_{P \in C} n_P P$ is the divisor $D = \sum_{P \in C} min(m_P, n_P) P$.*

Theorem 11 *Let $D = \sum m_i P_i - (\sum m_i)$ be a semireduced divisor, where $P_i = (x_i, y_i)$. Let $u(x) = \prod (x - x_i)^{m_i}$. Let v(x) be the unique polynomial satisfying:*

1. *$deg\ v < deg\ u$,*
2. *$v(x_i) = y_i$ for all i for which $m_i \neq 0$,*
3. *$u(x)$ divides $v(x)^2 + v(x)h(x) - f(x)$.*

Then

$$D = gcd(div(u(x)), div(v(x) - y))$$

For sake of brevity $gcd(div(u(x)), div(v(x) - y))$ is written as $div(u(x), v(x) - y)$ or, simply as $div(u, v)$. Thus every semi-reduced divisor can be effectively represented by two polynomials of small degree in $K[x]$. This representation was proposed by Mumford and hence is popularly known as *Mumford's representation* [119].

Koblitz, in his pioneering paper [77] had prescribed an algorithm for addition of two divisors represented in this cannonical form, although he had not given proof of correctness of his algorithms. Later, Cantor [22] and Menezes et al [102] have provided the proofs of correctness of the algorithms.

Algorithm 1 (Divisor Addition)

Input: *Reduced divisors $D_1 = div(u_1, v_1)$ and $D_2 = div(u_2, v_2)$ both defined over K.*
Output: *A semi–reduced divisor $D = div(u, v)$ defined over K such that $D \sim D_1 + D_2$.*

1. *Use the extended Euclidean algorithm to find polynomials $d_1, e_1, e_2 \in K[x]$ where $d_1 = gcd(u_1, u_2)$ and $d_1 = e_1u_1 + e_2u_2$;*
2. *Use the extended Euclidean algorithm to find polynomials $d, c_1, c_2 \in K[x]$ where $d = gcd(d_1, v_1 + v_2 + h)$ and $d = c_1d_2 + c_2(v_1 + v_2 + h)$;*
3. *Let $s_1 = c_1e_1, s_2 = c_1e_2$, and $s_3 = c_1$, so that $d = s_1u_1 + s_2u_2 + s_3(v_1 + v_2 + h)$;*
4. *Set $u = u_1u_2/d^2$ and*
$$v = \frac{s_1u_1v_2 + s_2u_2v_1 + s_3(v_1v_2 + f)}{d} \pmod u$$
5. *Output $div(u, v)$.*

However, the output of Algorithm 1 is not necessarily a reduced divisor. It is a semi-reduced one. Algorithm 2 is used to convert a semi-reduced divisor to the equivalent reduced divisor.

Algorithm 2 (Divisor Reduction)

Input: *A semireduced divisor $D = div(u, v)$ defined over K.*
Output: *The (unique) reduced divisor $D_1 = div(u_1, v_1)$ such that $D_1 \sim D$.*

1. *Set*
$$u_1 = (f - vh - v^2)/u$$
and
$$v_1 = (-h - v) \pmod{u_1}$$
2. *If deg $u_1 > g$ then set $u \leftarrow u_1, v \leftarrow v_1$ and go to step 1;*
3. *Make u_1 monic;*

4. Output (u_1, v_1).

A hyperelliptic curve cryptosystem is built on the hardness of the discrete logarithm problem in the Jacobian of reduced divisors of the curve. The hyperelliptic curve discrete logarithm problem (HCDLP) can be stated as follows:

Definition 12 *Let C be a hyperelliptic curve of given genus and let D_1 be an element of order n of the Jacobian of C. Let D_2 be another element in the subgroup generated by D_1 . The problem is to find an integer $\lambda \pmod{n}$ such that $D_2 = \lambda \times D_1$.*

2.4 Introduction to Elliptic Curves

Elliptic curves are hyperelliptic curves of genus 1. The Jacobian of a genus 1 hyperelliptic curve is in one-to-one correspondence with the set of all points on the curve. All the concepts described in the last section are greatly simplified for elliptic curves. In this section we briefly describe elliptic curve cryptosystems. In this thesis we do not deal with curves over fields of characteristic 3. We will treat only those part of the subject which are relevant to this dissertation. Interested readers can refer to [100, 154, 155, 42, 18, 62, 166] for details.

Definition An elliptic curve E over a field K is defined by an equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.5}$$

where $a_1, a_2, a_3, a_4, a_5, a_6 \in K$.

The discriminant Δ of E is defined as

$$\Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \tag{2.6}$$

where,

$$\left.\begin{array}{lcl} d_2 & = & a_1^2 + 4a_2 \\ d_4 & = & 2a_4 + a_1a_3 \\ d_6 & = & a_3^2 + 4a_6 \\ d_8 & = & a_1^2a_6 + 4a_2a_6 + a_1a_3a_4 + a_2a_3^2 + a_4^2 \end{array}\right\} \tag{2.7}$$

Equation 2.5 is called Weierstrass form of equation of an elliptic curve. The condition $\Delta \neq 0$ implies that there is no singular point on the curve.

The set of all L-rational points on any extension field L of K is defined as in the case of hyperelliptic curves and is generally denoted by $E(L)$. It also contains the point at infinity.

Over binary fields the Weierstrass equation of an elliptic curve can be transformed into two simpler forms. If $a_1 \neq 0$, then Equation 2.5 can be transformed into

$$y^2 + xy = x^3 + ax^2 + b \tag{2.8}$$

where $a, b \in K$ and the discriminant of the curve is b. Such curves are called *non-supersingular* curves.

If $a_1 = 0$, then Equation 2.5 can be reduced to the form

$$y^2 + cy = x^3 + ax^2 + b \tag{2.9}$$

where $a, b, c \in K$. Such curves have discriminant c^4 and are said to be *supersingular curves.*

If the characteristic of the underlying field is not 2 or 3, then the Weierstrass equation of an elliptic curve can be transformed to the following simple form:

$$y^2 = x^3 + ax + b \tag{2.10}$$

Such curves have discriminant $\Delta = -(4a^3 + 27b^2)$.

2.4.1 Group Law in EC

The set of rational points on an elliptic curve forms an additive group with respect to a composition defined by the *rule of secant and tangent.* If (x, y) is a point on the non-supersingular curve curve over binary fields, then we define the opposite of (x, y) to be $(x, x + y)$. For a curve over a field of characteristic > 3, the opposite point of (x, y) is $(x, -y)$. We define the sum of any point with infinity to be the point itself and the sum of a point with its opposite to be the point at infinity. Let P and Q be two distinct points on the elliptic curve with $Q \neq \pm P$. The line through P and Q intersects the curve at a third point, say R. The sum of P and Q is the opposite point of R. If P and Q are the same point, then instead of the secant line the tangent line to the curve at that point is used in the group law. The point at infinity plays the role of identity.

Analytically, let the characteristic of the underlying field K be greater than 3 and let the equation of the elliptic curve E be $y^2 = x^3 + ax + b$. For any point P on the curve $P + \mathcal{O} = P$. If P has coordinates (x_1, y_1) then coordinates of $-P$ are $(x_1, -y_1)$ and $P + (-P) = \mathcal{O}$. Let Q be another point on the curve with coordinates (x_2, y_2). The sum of P and Q has coordinates (x_3, y_3), where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2$$

and

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

If $P = Q$ and $P \neq -P$ then the sum is $2P$ and it has coordinates (x_4, y_4) with

$$x_4 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1$$

and

$$y_4 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_4) - y_1$$

Let the E be a non-supersingular elliptic curve over a binary field $K = F_{2^n}$ with equation $y^2 + xy = x^3 + ax^2 + b$. On the group $E(K)$, again the point at infinity $\mathcal{O}$ plays the role of identity. The inverse of the point $P(x_1, y_1)$ is now $-P = (x_1, x_1 + y_1)$. Let $Q(x_2, y_2)$ be another point on the curve with $Q \neq \pm P$. Then $P + Q$ is the point (x_3, y_3), where

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$$

and

$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

with $\lambda = \frac{y_1+y_2}{x_1+x_2}$.

The double of the point P is (x_4, y_4) where,

$$x_4 = \lambda^2 + \lambda + a$$

and

$$y_4 = x_1^2 + \lambda x_4 + x_4$$

with $\lambda = x_1 + y_1/x_1$.

Similar formulae for addition and doubling of elliptic curve points exist for supersingular curves also. We do not consider them in this thesis.

Under the operation defined above, the set of points on an elliptic curve defines a group. For a proof of this fact the reader can refer to any standard text on the subject.

2.5 Elliptic and Hyperelliptic Curve Cryptosystems

Elliptic and hyperelliptic curves over finite fields is a rich source of finite abelian groups, in which for carefully chosen curve parameters the DLP is believed to be hard. For these instances of the DLP no subexponential algorithm has been proposed yet. That leads to the fact that a high level of security can be ensured in the corresponding cryptosystems with much smaller key length. A variation of index calculus attack to solve DLP over a finite field (FFDLP) $\mathbf{F}_q$ has subexponential running time $O(exp((1.923 + o(1))(\log q)^{1/3}(\log\log q)^{2/3})$ (see [34] for $q = 2^k$, [55, 143] for prime q). The fastest algorithm to solve (H)ECDLP with suitable security parameters has running time $O(q^{1/2})$. Hence a cryptosystem based on a (hyper)elliptic curve group of order around 2^{160} can provide the same level of security as a cryptosystem based on the multiplicative group of a finite field of order 2^{1024}. This leads to a 84% reduction in the key length.

The (hyper)elliptic curve cryptosystems are ElGamal type cryptosystems built over the abelian group provided by the (hyper)elliptic curves. In this section, we describe what is called the “textbook” ElGamal cryptosystem over an arbitrary group.

Let G be an arbitrary cyclic group of order n over which the DLP is intractable.

ElGamal Key Generation: Let α be a generator of the group. An user A randomly chooses an integer m between 1 and $n-1$ and computes $\beta = \alpha^m$ and publishes β in a public directory. β is A's public key. Her secret key is m. The group G, its order n, representation of group element etc. are also made public.

ElGamal Encryption: Suppose an user, say B, wishes to send a message γ to A. She first embeds the message uniquely in the group G, so that the message itself becomes a group element. Without loss of generality, we assume that γ is a group element. B chooses a random integer k in $[1, n]$ and computes $C_1 = \alpha^k$ and $C_2 = \beta^k \times \gamma$. The pair (C_1, C_2) is the ciphertext corresponding to the plaintext γ. B sends the ciphertext to A.

ElGamal Decryption: After receiving the ciphertext (C_1, C_2), A decrypts it as follows: She computes $\delta = (C_1^m)^{-1}$. Note that δ is nothing but $(\beta^k)^{-1}$. Hence multiplying C_2 by δ she retrieves the plaintext.

Note that decryption is easy as A knows her private key. If an eavesdropper without the knowledge of A's private key tries to decrypt the message, he has to solve an instance of the Computational Diffie-Hellman (CDH) Problem. In fact, the encryption function is an example of a beautiful mathematical entity, so called *trapdoor one-way functions*. These are functions which are easy to compute in the forward direction and easy to invert if one has the knowledge of the “trapdoor”. Without the knowledge of the trapdoor, it is computationally infeasible to invert the function. The private key of the user is the trapdoor for the one-way function used for encryption.

ElGamal also proposed a signature scheme in [40] based on the intractability of DLP over the multiplicative group of a finite field.

Note that, in ElGamal cryptosystem in particular and any DLP based cryptosystem in in general, the computation of exponentiation of group elements is the dominant operation.

In (H)ECC, the groups are additive groups, the exponentiation translates into computing the *m-folds*. That is, given a group element X and an integer m, computing the point mX. This operation is called *the point multiplication* or *scalar multiplication* and is the dominant operation in (H)ECC. The point whose m-fold is computed is called the *base point*.

In ElGamal cryptosystem, in the encryption phase, the computation of the exponentiation β^k uses the public key of the receiver. This is an example of the situation when in the computation of the scalar multiplication the base point is fixed. In the decryption phase, the sender has to compute C_1^m, where C_1 varies from one encrypted message to another. It is an example of computation of scalar multiplication with variable base point.

Thus, the computation of scalar multiplication has two broad scenarios, one with a fixed base point and the other with a variable base point.

2.6 Group Order

The order of the elliptic curve group over the underlying field is an important security parameter. There are attacks (for example Pohlig-Hellman attack [134]) which can be launched on (H)ECC if the group order is not divisible by a very large prime. In fact the Pohlig-Hellman attack dictates that the group order for (H)ECC should be product of a large prime multiplied by a small positive integer less than 4. This small number is called *cofactor* of the curve. Various algorithms have been proposed in literature (for example Kedlaya's algorithm [72] for HECC and Schoof's algorithm [145] for ECC) for efficiently counting the group order.

The group order of an elliptic curve is given by *Hasse' theorem.*

Theorem 13 *Let E be an elliptic curve over a finite field F_q of order q. Then the order $\#E(F_q)$ of the elliptic curve group is given by*

$$\#E(F_q) = q + 1 - t$$

where $|t| \leq 2q^{1/2}$.

The parameter t is called trace of E over F_q. An interesting fact is that given any integer $|t| \leq 2q^{1/2}$, there exists an elliptic curve E over F_q such that $\#E(F_q) = q + 1 - t$.

The next theorem is about the structure of an elliptic curve group.

Theorem 14 *Let E be an elliptic curve over a finite field F_q. Then $E(F_q)$, the elliptic curve group over F_q is isomorphic to $\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$, where n_1 and n_2 are uniquely determined integers such that n_2 divides both n_1 and $q - 1$.*

Note that $\#E(F_q) = n_1 \times n_2$. If $n_2 = 1$, then the group $E(F_q)$ is cyclic.

For hyperelliptic curves the group order is computed using the zeta function of the curve. It has been proved that :

Theorem 15 *Let C be a hyperelliptic curve over F_{p^n}, a degree-n extension of the finite field F_p and let N_n denote the order of the Jacobian $J(F_{p^n})$ of C. Then*

$$(p^{n/2} - 1)^{2g} \leq N_n \leq (p^{n/2} + 1)^{2g}$$

Hence $N_n \approx p^{ng}$.

Thus in (H)ECC the group order is intimately connected to the size of the underlying field. The current state of the computer technology dictates that for an implementation requiring a medium term security, the group order should be around 2^{160}. In view of theorems on group order stated above, one can say that, for medium term security the underlying field should have a size of around $2^{160/g}$, where g is the genus of the curve.

2.7 Isomorphic Curves

Definition 16 *Let E_1 and E_2 be two elliptic curves defined over the field K by the equations*

$$\begin{aligned} E_1 &: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \\ E_2 &: y^2 + \bar{a_1}xy + \bar{a_3}y = x^3 + \bar{a_2}x^2 + \bar{a_4}x + \bar{a_6} \end{aligned} \tag{2.11}$$

are said to be isomorphic if there exist $\alpha, \beta, \gamma, \delta \in K, \alpha \neq 0$ such that the transformation

$$(x, y) \rightarrow (\alpha^2 x + \beta, \alpha^3 y + \alpha^2 \gamma x + \delta)$$

changes equation of E_1 to that of E_2.

If K is a field of characteristic $\neq 2, 3$, it is proved that two curves E_1 and E_2 over the field K given by the equations

$$\begin{aligned} E_1 &: y^2 = x^3 + a_1x + b_1 \\ E_2 &: y^2 = x^3 + a_2x + b_2 \end{aligned} \tag{2.12}$$

are isomorphic if there exist $u \in K^*$ such that $u^4 a_2 = a_1$ and $u^6 b_2 = b_1$.

If K is a binary field (characteristic = 2), it is proved that two non-supersingular curves E_1 and E_2 over the field K given by the equations

$$\begin{aligned} E_1 &: y^2 + xy = x^3 + a_1x^2 + b_1 \\ E_2 &: y^2 + xy = x^3 + a_2x^2 + b_2 \end{aligned} \tag{2.13}$$

are isomorphic if there exist $\gamma \in K^*$ such that the transformation

$$(x, y) \to (x, y + sx)$$

changes equation of E_1 to equation of E_2.

If E_1 and E_2 are isomorphic over K then groups of K-rational points formed by them, namely, $E_1(K)$ and $E_2(K)$ are also isomorphic. This property is exploited in an effective countermeasure against differential power attacks (see Section 2.10).

Let C and C' be two hyperelliptic curves of genus g over the field K given by the equations

$$C : y^2 + h(x)y = f(x) \tag{2.14}$$

$$C' : y^2 + \bar{h}(x)y = \bar{f}(x) \tag{2.15}$$

All K-isomorphisms $\sigma : C \to C'$ are of the type

$$\sigma : (x, y) \to (s^{-2}x + b, s^{-2g+1}y + A(x))$$

for some $0 \neq s \in K$, $b \in K$ and $A(x)$ is a polynomial of degree at most g over K (see [93]). Substituting $s^{-2}x + b$ and $s^{-2g+1}y + A(x)$ in place of x and y, we see that the curves C and C$'$ are isomorphic if there exists a nonzero s and a b in K and a polynomial $A(x)$ of degree at most g such that,

$$h(x) = s^{2g+1}(\bar{h}(s^{-2}x + b) + 2A(x)), f(x) = s^{2(2g+1)}(\bar{f}(s^{-2}x + b) - A(x)^2 - \bar{h}(s^{-2}x + b)A(x))$$

In [41], the authors have estimated the number of isomorphic curves of genus 2 over various fields.

2.8 Point Representation and Cost of Group Operations

Point addition and point doubling are two important operations in (H)ECC. We have seen in Section 2.4.1 that computing these operations in affine coordinates involves field inversion.

Table 2.1: Cost of Group Operations in ECC for Various Point Representations for Characteristic > 3

Coordinates	Cost(Addition)	Coordinates	Cost(Doubling)
$\mathcal{A}+\mathcal{A} \to \mathcal{A}$	$1[i]+2[m]+1[s]$	$2\mathcal{A} \to \mathcal{A}$	$1[i]+2[m]+2[s]$
$\mathcal{P}+\mathcal{P} \to \mathcal{P}$	$12[m]+2[s]$	$2\mathcal{P} \to \mathcal{P}$	$7[m]+3[s]$
$\mathcal{J}+\mathcal{J} \to \mathcal{J}$	$12[m]+4[s]$	$2\mathcal{J} \to \mathcal{J}$	$6[m]+4[s]$
$\mathcal{C}+\mathcal{C} \to \mathcal{C}$	$11[m]+3[s]$	$2\mathcal{C} \to \mathcal{C}$	$5[m]+4[s]$

Inversion in a finite field is an expensive operation. To avoid these inversions, several point representations have been proposed in literature. The cost of point addition and doubling varies depending upon the representation of the group elements. In the current section, we will briefly deal with some point representations commonly used. Let $[i], [m], [s], [a]$ stand for cost of a field element inversion, a multiplication, a squaring and an addition respectively. Field element addition is considered to be a very cheap operation. In binary fields, squaring is also quite cheaper than a multiplication. If the underlying field is represented in normal basis then squaring is almost for free. Inversion is considered to be 8 to 10 times costlier than a multiplication in binary fields. In prime field the *I/M ratio* is even more. It is reported to be between 30 and 40 (see [45]).

2.8.1 Elliptic Curves

Point representation in ECC is a well studied area. In the following two sections we describe some of the point representation popularly used in implementations.

Fields of Characteristic > 3

Elliptic curves over fields of characteristic > 3 have equations of the form $y^2 = x^3 + ax + b$. For such curves the following point representation methods are mostly used.

1. In *Standard Projective Coordinates* the curve has equation of the form $Y^2Z = X^3 + aXZ^2 + bZ^3$. The point $(X : Y : Z)$, with $Z \neq 0$ in projective coordinates is the point $(X/Z, Y/Z)$ in affine coordinates. The point at infinity is represented by the point $(0 : 1 : 0)$ and the inverse of $(X : Y : Z)$ is the point $(X : -Y : Z)$.

2. In *Jacobian Projective Coordinates* the curve has equation of the form $Y^2Z = X^3 + aXZ^4 + bZ^6$. The point $(X : Y : Z), Z \neq 0$ in Jacobian coordinates correspond to the

affine point $(X/Z^2, Y/Z^3)$. The point at infinity is represented by the point $(1:1:0)$ and the inverse of $(X:Y:Z)$ is the point $(X:-Y:Z)$. Point doubling becomes cheaper in Jacobian coordinates if the curve parameter $a=-3$.

3. In *Chudonovski Jacobian Coordinates*, the Jacobian point $(X:Y:Z)$ is represented as $(X:Y:Z:Z^2:Z^3)$. Cost of point addition in Chudonovski Jacobian coordinates is the minimum among all representations.

In Table 2.1, we present the cost of addition and doubling in the coordinate systems described above. In the table we use $\mathcal{A}$, $\mathcal{P}$, $\mathcal{J}$, $\mathcal{C}$ for affine, projective, Jacobian and Chudnovski Jacobian respectively. By $2\mathcal{A} \rightarrow \mathcal{A}$ we mean the doubling formula in which the input is in affine and so is the output. Similarly for addition and other coordinate systems.

Fields of Characteristic 2

We will consider only non-supersingular curves. Elliptic curves (non-supersingular) over binary fields have equations of the form $y^2+xy=x^3+ax^2+b$. For such curves the following point representation methods are mostly used.

1. In *Standard Projective Coordinates* the curve has equation of the form $Y^2Z+XYZ=X^3+aX^2Z+bZ^3$. The point $(X:Y:Z)$, with $Z\neq 0$ in projective coordinates is the point $(X/Z, Y/Z)$ in affine coordinates. The point at infinity is represented by the point $(0:1:0)$ and the inverse of $(X:Y:Z)$ is the point $(X:X+Y:Z)$.

2. In *Jacobian Projective Coordinates* the curve has equation of the form $Y^2+XYZ=X^3+aX^2Z^2+bZ^6$. The point $(X:Y:Z)$, with $Z\neq 0$ in Jacobian coordinates correspond to the affine point $(X/Z^2, Y/Z^3)$. The point at infinity is represented by the point $(1:1:0)$ and the inverse of $(X:Y:Z)$ is the point $(X:X+Y:Z)$.

3. In *Lopez-Dahab Coordinates*, the point $(X:Y:Z)$, with $Z\neq 0$ represents the affine point $(X/Z, Y/Z^2)$. The equation of the elliptic curve in this representation is $Y^2+XYZ=X^3Z+aX^2Z^2+bZ^4$. The point at infinity is represented by the point $(1:0:0)$ and the inverse of $(X:Y:Z)$ is the point $(X:X+Y:Z)$.

In Table 2.2 we present the cost of addition and doubling in the coordinate systems over binary fields. In the table we use $\mathcal{A}$, $\mathcal{P}$, $\mathcal{J}$, $\mathcal{L}$ for affine, projective, Jacobian and Lopez-Dahab respectively. The table follows the same notational convention as in last subsection. Note that in Table 2.2 we have neglected squarings also. That is because in binary fields squaring is a much cheaper operation than multiplication.

Table 2.2: Cost of Group Operations in ECC for Various Point Representations in Even Characteristics

Coordinates	Cost(Addition)	Coordinates	Cost(Doubling)
$\mathcal{A}+\mathcal{A}\rightarrow\mathcal{A}$	$1[i]+2[m]$	$2\mathcal{A}\rightarrow\mathcal{A}$	$1[i]+2[m]$
$\mathcal{P}+\mathcal{P}\rightarrow\mathcal{P}$	$13[m]$	$2\mathcal{P}\rightarrow\mathcal{P}$	$7[m]+3[s]$
$\mathcal{J}+\mathcal{J}\rightarrow\mathcal{J}$	$14[m]$	$2\mathcal{J}\rightarrow\mathcal{J}$	$5[m]$
$\mathcal{L}+\mathcal{L}\rightarrow\mathcal{L}$	$14[m]$	$2\mathcal{L}\rightarrow\mathcal{L}$	$4[m]$

It has been reported that [33, 94] if one point is in affine and the other is in projective or some other weighted co-ordinate, then point addition becomes relatively cheaper. This operation is called *addition in mixed coordinates or mixed addition*. In (H)ECC, the base point is generally stored in affine coordinates to take advantage of mixed additions.

2.8.2 Hyperelliptic Curves

The algorithms proposed by Koblitz [77] for divisor addition and doubling are known as Cantor's algorithms. Later it was found that the efficiency of group operation algorithm can be significantly enhanced by fixing the genus of the curve and computing the parameters of the resultant divisors explicitly from those of the input divisors. Such divisor addition and doubling algorithms are called *explicit formula*.

For curves of genus 2, the explicit formulae have gone through a long process of evolution. Lately, Lange has proposed various point representation for HECC of genus 2 and has proposed explicit formulae for those representations. To our knowledge these are the only result proposing newer representations. We discuss these issues in the next section.

Explicit Formulae

Spallek [162] made the first attempt to compute divisor addition by explicit formula for genus 2 curves over fields of odd characteristic. Harley [60] improved the running time of the algorithm in [162]. Gaudry and Harley [53] observed that one can derive different explicit formula for divisor operations depending upon the weight of the divisors. In 2000, Nagao [120] proposed two algorithms; one for polynomial division without any inversion and another for extended gcd computation of polynomials requiring only one inversion. Both these algorithms can be applied to Cantor's algorithm to improve efficiency. Lange [83] generalized Harley's approach to curves over fields of even characteristic. Takahashi [163]

Table 2.3: Complexity of Various Explicit Formulae

Genus	Name/Proposed in	Char($\mathbf{F}_q$)	Cost(Add)	Cost(Double)
Genus 2	Cantor [120]	All	$3[i] + 70[m/s]$	$3[i] + 76[m/s]$
	Nagao [120]	Odd	$1[i] + 55[m/s]$	$1[i] + 55[m/s]$
	Harley [60]	Odd	$2[i] + 27[m/s]$	$2[i] + 30[m/s]$
	Matsuo et al [96]	Odd	$2[i] + 25[m/s]$	$2[i] + 27[m/s]$
	Miyamoto et al [112]	Odd	$1[i] + 26[m/s]$	$1[i] + 27[m/s]$
	Takahashi [163]	Odd	$1[i] + 25[m/s]$	$1[i] + 29[m/s]$
	Lange [84]	All	$1[i] + 22[m] + 3[s]$	$1[i] + 22[m] + 5[s]$
	Lange [85]	All	$40[m] + 6[s]$	$47[m] + 4[s]$
	Lange [86]	Even	$48[m] + 4[s]$	$39[m] + 6[s]$
	Lange [86]	Odd	$47[m] + 7[s]$	$34[m] + 7[s]$
Genus 3	Nagao [120]	Odd	$2[i] + 154[m/s]$	$2[i] + 146[m/s]$
	Pelzl et al [131]	All	$1[i] + 70[m] + 6[s]$	$1[i] + 61[m] + 10[s]$
Genus 4	Pelzl et al [132]	All	$2[i] + 160[m] + 4[s]$	$2[i] + 193[m] + 14[s]$

and Miyamoto, Doi, Matsuo, Chao and Tsujii [112] achieved further speed-up using Montgomery's trick to reduce the number of inversions to 1. The fastest version of explicit formula for genus 2 curves is given in [84].

Following ECC, Lange in [85] has proposed group arithmetic in the Jacobian of genus 2 curves in "projective coordinates". Introducing a new variable in the structure of a divisor, she has shown that the inversions in group operations can be avoided. Taking a cue from the Chudonovski-Jacobian coordinates in ECC, Lange in [86] has proposed group operations in her "new" coordinates, weighted coordinates for HECC of genus 2. In the new coordinates group operations are inversion-free. The doubling, which is more frequently used operation is more efficient than in other coordinates. Lange's papers are the only work known so far in the direction of using various coordinate systems to reduce the complexity of group operations in HECC so far.

Pelzl et al have proposed explicit formula for performing arithmetic in the Jacobian of hyperelliptic curves of genus 3 [131] and genus 4 [132].

We summarize the complexity of various explicit formulae proposed in literature in Table 2.3. In the cost column, $[i]$, $[m]$, $[s]$ stand for the time taken by an inversion, a multiplication and a squaring in the underlying field respectively. The notation, $[m/s]$ stands for time of a square or multiplication. In the corresponding papers, multiplications and squarings have been treated to be of the same complexity.

Algorithm DBL-AND-ADD (Left-to-right binary method)	**Algorithm DBL-AND-ADD (Right-to-left binary method)**
Input : $X, m\ (m_{k-1}, \cdots m_1, m_0)$	*Input :* $X, m\ (m_{k-1}, \cdots m_1, m_0)$
Output : mX.	*Output :* mX.
1. $E = m_{k-1}X$	1. $E_0 = X, E_1 = 0$
2. for $i = k-2$ down to 0	2. for $i = 0$ to $k-1$
3. E = DBL(E)	3. if $m_i = 1$
4. if $m_i = 1$	4. E_1 = ADD(E_0, E_1)
5. E = ADD(E, X)	5. E_0 = DBL(E_0)
6. return E	6. return(E_1)

Table 2.4: Left-to-right and Right-to-left Binary Algorithm

2.9 Scalar Multiplication

In ECC and HECC, computationally the most expensive operation is scalar multiplication. It is also very important from security point of view. The implementation attacks (see Section 2.10) generally target the computation of this operation to break the cryptosystem. Given a point X and an positive integer m, computation of $m \times X = X + \cdots$ (m times) $\cdots + X$ is called the operation of scalar multiplication. In this section we briefly outline various scalar multiplication algorithms proposed in literature. We do not include multi scalar multiplication methods (i.e. methods to compute $lP + mQ$). Also, due to the vastness of the subject and space constraints we will elaborate only those methods which are discussed in depth in this dissertation. For an excellent review of the subject reader can refer to [62].

The basic algorithms to compute the scalar multiplication are the age old binary algorithms. They are believed to have been known to the Egyptians two thousand years ago. In Table 2.9 we describe these algorithms. These algorithms invoke two functions ADD and DBL. ADD takes as input two points X_1 and X_2 and returns their sum $X_1 + X_2$. DBL takes as input one point X and computes its double $2X$.

Both the algorithms first convert the scalar multiplier m into binary. Suppose m has a n-bit representation with hamming weight h. Then, mX can be computed by $n-1$ invocations of DBL and $h-1$ invocations of ADD. Hence cost of the scalar multiplication is $(n-1) \times cost(\text{DBL}) + h \times cost(\text{ADD})$. As the average value of h is $n/2$, on the average these algorithms require $(n-1)$ doublings and $n/2$ additions. As doublings are required more often than additions, attempts are made to reduce complexity of the doubling operation.

The scalar multiplication is the dominant operation in (H)ECC. Extensive research has been carried out to compute it efficiently and a lot of results have been reported in literature. We will devote much of our time in this thesis for efficient and secure implementation of the scalar multiplication in (H)ECC.

To compute the scalar multiplication efficiently there are three main approaches. As is seen in the basic binary algorithms the efficiency is intimately connected to the efficiency of ADD and DBL algorithms. So the first approach is to compute group operations efficiently. The second approach is to use a representation of the scalar such that the number of invocation of group operation is reduced. The third approach is to use more hardware support (like memory for precomputation) to compute it efficiently. In some proposals these have approaches have been successfully combined to yield very efficient algorithms.

As noted in the Sections 2.8, the cost of ADD and DBL depend to a large extent on the choice of underlying field and the point representation. Hence the cost of scalar multiplication also depend upon these choices. Based on the underlying field more efficient operations have been proposed. Over binary fields for ECC, using a point halving algorithm instead of DBL has been proved to be very efficient. Over fields of characteristic 3, point tripling has been more efficient. There are proposals for using more fancier algorithms like the ones efficiently computing $2P + Q$, $3P + Q$ etc. instead of ADD and DBL.

The second approach has also been extensively studied and techniques based on this idea have been successfully employed. We will discuss three techniques of this approach, namely NAF, w-NAF and base-ϕ expansion of the scalar, where ϕ is the Frobenius map.

The third approach is to use more hardware resources to compute the scalar multiplication efficiently. This includes methods using precomputations, parallel methods and pipelining. Many algorithms use a precomputed table of values to compute the scalar multiplication. These algorithms use more memory for efficiency and are ideal when the base point is fixed. Many parallel algorithms for (H)ECC have been proposed. Recently pipelining techniques have been proposed. All these methods require more hardware support than the basic binary algorithms.

2.9.1 NAF and w-NAF

As is seen in the last section, the efficiency of computation of scalar multiplication depends upon the hamming weight of the scalar multiplier. The lesser the hamming weight the more efficient is the method. Also, for curves over prime fields, the computation of negation of a point is virtually for free. Hence point addition and point subtraction have almost the same cost. Therefore there are proposals for representing the multiplier in signed binary representation with lesser hamming weight. One such representation is Non Adjacent Form (NAF). We formally define it below.

Definition 17 *A representation of m as $\Sigma_i m_i 2^i$ is in Non Adjacent Form if and only if $m_i m_{i+1} = 0$ for all i.*

The following theorem ensures that every integer has a unique NAF representation. A proof of the theorem can be found in [136, 8, 117, 31].

Theorem 18 *Every integer has a unique NAF representation. This representation has the lowest weight among the signed digit representations and its length is at most one bit longer than the binary representation. The average density of NAF is one third.*

The following algorithm computes the NAF of a given integer.

Algorithm 3 (Computation of NAF)

```
Input: An integer $m$.
Output: NAF representation $\Sigma d_i 2^i$ of $m$.
1. $i \leftarrow 0$
2. while $m \geq 1$ do
3.     if $m$ is odd $d_i \leftarrow 2 - (m \bmod 4)$
4.     else $d_i = 0$
5.     $m \leftarrow m/2, i \leftarrow i + 1$
6. return $(d_0, d_1, \cdots, d_{i-1})$
```

Average hamming weight of an integer of n bits in NAF is $n/3$. Hence, the computation of scalar multiplication requires n doublings and $n/3$ additions on the average. The scalar multiplication algorithm using NAF is similar to the left-to-right and right-to-left algorithm described above, except that the addition step now becomes:

if $d_i = \pm 1$, then $Q = Q \pm P$.

The concept of NAF has been generalized to reduce the complexity of the scalar multiplication algorithm further. The general concept is that of w-NAF. It can be defined as follows:

Definition 19 *Let w be a positive integer. The width w NAF (or briefly w-NAF) representation of a positive integer m is an expression $\Sigma_{i=0}^{l-1} d_i 2^i$, where d_i are odd integers in the range $[-2^{w-1}, 2^{w-1} - 1]$, $d_j \neq 0$ and at most one of any w consecutive integers is nonzero.*

The w-NAF representation of an integer m satisfies the following properties:

Theorem 20 *1. The w-NAF representation of an integer in unique.*

2. The NAF representation is a special case of w-NAF, with $w = 2$.

3. The length of the w-NAF representation of an integer is at most one more than the NAF representation.

4. The average hamming weight (number of nonzero digits in the representation) of an integer is $n/(w + 1)$, where n is the length of representation.

The following algorithm calculates the w-NAF representation of a given integer m.

Algorithm 4 (Computation of w-NAF)

```
Input: An integer $m$.
Output: $w$-NAF representation $\Sigma d_i 2^i$ of $m$.
1. $i \leftarrow 0$
2. while $m \geq 1$ do
3.     if $m$ is odd $d_i \leftarrow 2^{w-1} - (m \bmod 2^w)$
4.     else $d_i = 0$
5.     $m \leftarrow m/2, i \leftarrow i + 1$
6. return $(d_0, d_1, \cdots, d_{i-1})$
```

The cost of computation of scalar multiplication goes down drastically if some precomputation is done, particularly when the base point is fixed. If the base point is not fixed the precomputation can be done online. One precomputes the points $P_i = iP$ for odd i in the range $1 \leq i \leq 2^{w-1} - 1$. Note that the computation of negation of these points is very cheap(virtually "for free" over prime fields). So P_i for negative i's need not be computed, which also reduces the storage requirement. The following algorithm computes the scalar multiplication.

Algorithm 5 (Scalar Multiplication using w-NAF)

```
Input: An integer $m = \Sigma_{i=0}^{n-1} d_i 2^i$ in $w$-NAF.
Output: $mP$.
1. precompute $P_i \leftarrow iP$ for odd $i$ in $[-2^{w-1}, 2^{w-1} - 1]$
2. set $Q = P$
3. for $i = n - 1$ to 0
3.     $Q = 2Q$
4.     if $d_i > 0$ then $Q = Q + P_{d_i}$
```

```
5.      if $d_i < 0$ then $Q = Q - P_{d_i}$
6. return Q
```

The above algorithm requires 1 doubling and $2^{w-2} - 1$ additions for the precomputation. The scalar multiplication computation requires n doublings and $n/(w+1)$ additions on the average.

2.9.2 Frobenius Map

In [78], Koblitz had suggested the use of Frobenius map to speed up scalar multiplication algorithm. This idea has later been developed by several authors [30, 50, 75, 118, 156, 159]. For hyperelliptic curves, it has been shown [83, 26] that the Frobenius map based method can be used over any field of finite characteristic.

Let q be a prime power, F_q be the finite field of order q and F_{q^n} an extension field of F_q. Let C be the curve of genus g to be used for the cryptosystem and we consider the F_{q^n}-rational points of C. The Frobenius map $\phi : F_{q^n} \rightarrow F_{q^n}$ is an automorphism of F_{q^n} and is defined as $\phi(x) = x^q$. The map is extended to points of an elliptic or hyperelliptic curve over F_{q^n} in the following manner: A point of an elliptic curve is represented using a pair of elements of F_{q^n}; similarly a reduced divisor of a hyperelliptic curve is represented using a tuple of elements of F_{q^n}. An application of the Frobenius map to a point is to actually apply the map individually to the field elements which represent the point. We note that ϕ^n is the identity map on F_{q^n}. If the field F_{q^n} is represented using a normal basis, then the computation of $\phi(x)$ is "for free". Further, as observed in [160, 159], in the case $q = 2$, the Frobenius map is $\phi(x) = x^2$ and hence can be computed using a field squaring which is a relatively cheap operation even if polynomial basis representation of elements is used.

Let m be an integer, X a point (either a point of an elliptic curve or a reduced divisor of a hyperelliptic curve) and we wish to compute mX. The base-ϕ expansion of m is $\sum_{i=0}^{n-1} u_i \phi^i$, where under reasonable assumptions each u_i is an integer in the range $[-q^g, q^g]$. It is possible to obtain the base-ϕ expansion for each integer m. Next we define the following parameters:

1. $A = \max\lfloor \log_2(|u_i|) \rfloor$.
2. For $i \in \{0, \ldots, n-1\}$ write $|u_i| = \sum_{j=0}^{A} u'_{i,j} 2^i$, where $u'_{i,j} \in \{0, 1\}$.
3. $u_{i,j} = \mathsf{sgn}(u_i) u'_{i,j}$, where $\mathsf{sgn}(u_i)$ is the sign of u_i.
4. For $0 \leq i \leq n-1$, define $X_0 = X$ and $X_i = \phi^i(X_0) = \phi^i(X)$.

The expression mX can be written as

$$\left.\begin{aligned} mX &= u_0X_0 + u_1X_1 + \cdots + u_{n-1}X_{n-1} \\ &= (u_{0,0} + u_{0,1}2 + \cdots + u_{0,A}2^A)X_0 \\ &\quad +(u_{1,0} + u_{1,1}2 + \cdots + u_{1,A}2^A)X_1 \\ &\quad + \cdots \\ &\quad +(u_{n-1,0} + u_{n-1,1}2 + \cdots + u_{n-1,A}2^A)X_{n-1} \end{aligned}\right\} \tag{2.16}$$

The following simple algorithm can be used to compute mX from (2.16) (see [26, 75, 83, 159, 160]).

Algorithm 6 (Scalar Multiplication Using Frobenius Map)

```
Input : integer $m = \sum_{i=0}^{n-1} u_i\phi^i$ and point $X$ .
Output : $mX$.
    1. For $0 \le i \le n-1$ and $0 \le j \le A$, compute $X_i$ and $u_{i,j}$;
    2. Set $Y = \sum_{i=0}^{n-1} u_{i,A}X_i$;
    3. For $j = A-1$ down to 0
    4.        $Y = 2Y$; $Y = Y + \sum_{i=0}^{n-1} u_{i,j}X_i$
    5. return $Y$.
```

Theorem 21 *In the above algorithm, the average numbers of additions and doublings needed to compute mX are $n(A+1)/2$ and A respectively.*

In Chapter 9 we will demonstrate a simple technique to speed-up the computation of scalar multiplication using Frobenius map over prime power fields $\mathbf{F}_{p^n}, p > 2$.

2.9.3 Window Based Methods

Window based methods come under the third category of scalar multiplication algorithms, which require higher amount of computational resources. Window based methods use higher amount of memory as they use a precomputed table and are more efficient if the base point is fixed. The target of these methods is to minimize the size of the look-up table and maximize the gain in performance. Algorithms using w-NAF are window based methods. Besides, many other efficient methods have been proposed in literature. Interested readers can see [113, 66, 62].

2.9.4 Parallel Methods

Many algorithms have been proposed in literature to compute the scalar multiplication in parallel. For elliptic curves, Koyama and Tsuruoka [82] had proposed a parallel algorithm as early as 1992, using a special hardware. Later many other algorithms (for example [44, 6, 64, 65, 51]) have been proposed some of which are also resistant against side channel attacks. For hyperelliptic curves we have proposed parallel versions of the explicit formulae for genus 2 curves in [109]. Later in [15] the authors have investigated about the most efficient parallel architecture for genus 2 curves.

2.10 Side-Channel Attacks

The discrete logarithm problem over (hyper)elliptic curve has been investigated at length (see for example [2, 13, 34, 46, 47, 49, 52, 54, 55, 68, 97, 101, 104, 135, 137]). Various attacks have been proposed. Some of them are general in nature in the sense that they can be applied to any DLP. Some have been proposed against ECDLP and there is possibility of generalizing them to HCDLP, where as some of them are exclusively against HCDLP. We will refer to these attacks as *generic attacks.* The one of the best of all these attacks is the Pollard's rho attack, which takes fully exponential time if the security parameters are chosen carefully. Thus, the generic attacks against ECC and HECC can be avoided by cleverly choosing the system parameters. For example, the attack proposed by Addleman, De Marraise and Huang [2] against HCDLP can be avoided by choosing hyperelliptic curves of small genus ($1 \leq g \leq 4$). However a straight forward implementation can be broken by the so called *side-channel attacks.*

Side-channel attacks are those attacks which can be launched against a cryptographic device exploiting unintentionally leaked information. While executing various steps of a cryptographic algorithm a device consumes different amount of power, takes different amount of time, emits different amount of electro-magnetic radiations, produces different sounds or different error messages. Analyzing these side-channel information it is possible to successfully guess the computation running inside. Attacks exploiting these side-channel data are called side-channel attacks. If the attacker only monitors the leaked information then it is said to be a passive side-channel attack. If the attacker also manipulates the device it is said to be an active side-channel attack.

First instance of side-channel attacks dates back to as early as 1956. In [168], the author reports that, the British Intelligence service M15 in 1956, broke a cryptographic device used by Egyptian embassy in London, by using the sound produced by the device. The encryption device was a rotor machine of Hagelin type. Wright observed that the position of the rotors can be guessed from the sound produced by the machine and the attack succeeded.

First side-channel attack against modern cryptographic primitives was reported by Paul Kocher in 1996. In [80, 81] Kocher et al reported how side-channel information like timing and power can be exploited to break modern cryptosystems. Since then many attacks based on side channel information and countermeasures against them have been proposed in literature.

2.10.1 Types of Side-channel

In this section, we will briefly discuss various side-channels exploited in launching attacks against cryptographic protocols.

Execution time: The execution of various steps of an algorithm may take different amount of time. That may be caused by different nature of the input, various optimization techniques used or due to conditional branching instructions in the algorithm. So by measuring the amount of time consumed for executing various steps of the algorithm by the computing device it may be possible to correctly guess what is being computed. Such attacks are said to be timing attacks. A successful timing attack against hyperelliptic curve cryptosystem has been reported in [71].

Power Consumption Leakage: Most cryptographic devices use CMOS logic. Whenever such a circuit clocks, all its gates change state. This leads to a simultaneous charging or discharging of the internal capacitors. This further leads to a current flow, which can be measured from outside by means of a digital oscilloscope. This surge in current flow is dependent on the type of instruction being processed by the device. So by measuring power consumption an adversary can correctly guess what is the sequence of instruction being processed by the device, which may lead to revealing the secrecy of the cryptosystem. Power analysis attacks are most popular among all side-channel attacks due to their simplicity and effectiveness.

Electromagnetic Radiations During the clocking of the CMOS circuits, the simultaneous charging or discharging of capacitors creates an electromagnetic field. The strength of the radiation increases due to miniaturization and complexity of the modern CMOS devices. Miniaturization leads to coupling effects between components in close proximity. Attacks exploiting electromagnetic emanations are briefly called EM-attacks.

Fault based Attacks: Error messages produced by a cryptographic device if monitored properly can be used to reveal the secret key of a cryptosystem. A common assumption is that there is a one bit feedback from the device whether or not a ciphertext could be decrypted or not. An adversary can gain some information about the secret decryption key by well-chosen ciphertexts to the device and monitoring the resulting error messages.

One interesting area of research on side-channel cryptanalysis and which has not been investigated properly so far is combining different side-channels. The only two side-channels, which have been combined together effectively to attack cryptosystems are timing and power consumption (see [142] [165]).

2.10.2 Countermeasures against SCA

In this dissertation we will primarily concentrate on power analysis attacks. In fact, power analysis attacks subsumes timing attacks. So our power analysis resistant algorithms are immunized against timing attacks also.

Power analysis attacks is divided into two class of attacks, namely, *Simple Power Analysis (SPA)* and *Differential Power Analysis (DPA)*. SPA uses power analysis attacks from one computation to reveal the secret, while DPA uses those of several computations with statistical tools to reveal the secret of the cryptosystem. In general, a system secure against SPA may not be secure against DPA and vice-versa. Hence a robust system has to be secure against both of them.

SPA and Countermeasures

As mentioned in Section 2.10, scalar multiplication algorithms are often the target of side-channel attacks. Suppose the scalar multiplication is computed by left-to-right or right-to-left binary algorithm described in Section 2.9. It is observed that the computation depends upon the bit pattern representing the scalar multiplier m. In both the algorithms, it is computed by a series of point doubling and point addition operations. If the adversary sampling the side-channel information can guess the correct order of these operations then he can guess the value of the m. That is because an addition is carried out if the corresponding bit is 1. So, if the adversary can distinguish an addition from a doubling from the side-channel information, then from a single observation of the power-consumption traces he can correctly find out the bit pattern representing m, i.e. m itself. Unfortunately, addition and doubling are two different algorithms, which vary in complexity for general curves. So their execution takes different amount of resources, which makes them distinguishable from the side-channel information. Attacks using this principle and using the power consumption traces are called SPA.

Thus there are two different ways to immunize the computation from SPA. First, making the addition and doubling operations indistinguishable from the side-channel. Second approach is instead of using the binary algorithms use a special computation pattern interleaving the addition and doubling operations in such a way that the side-channel information becomes independent of the bit pattern of m.

In [32], the authors have proposed a universal exponentiation algorithm as a countermeasure of first type. In [91], the authors have proposed indistinguishable addition and doubling algorithm for Jacobi form of elliptic curves. Similar algorithms for Hessian form of elliptic curves are proposed in [70]. However, these algorithms do not apply to general elliptic curves and particularly to the ones recommended by NIST, ANSI, SEC. Brier and Joye [20] proposed an indistinguishable addition and doubling algorithms for Weierstrass form of elliptic curves. But the method fails on certain inputs, making it vulnerable to attack (see [67]).

The second approach has been exploited extensively and several countermeasures have emerged in literature based on this approach. The simplest of them Coron's dummy addition method is based on the double-and-always-add approach. Okeya and Sakurai([125]) proposed a double and always add method for Montgomery form of elliptic curve, which was later generalized to general Weierstrass form curves by Brier and Joye [20] and Izu and Takagi [65]. Various addition chains have been proposed in literature to compute the scalar multiplication independent of the bit pattern of m. Two such chains are proposed by Möller [113] and Seysen [148].

All the countermeasures described above demand some computational overhead to immunize the computation against SPA. For example in Coron's dummy addition method one has to compute $n-1$ additions and $n-1$ doublings for a secure computation, where as the unprotected method needs $n/2$ additions and $n-1$ doublings on the average. Recently Cavalier-Mames, Ciet and Joye ([23]) have proposed a countermeasure which involves the minimum (almost negligible) computational overhead for the purpose. The authors use a new concept, namely, *side-channel atomicity.* Two set of instructions are said to be equivalent (denoted by $\sim$) if they are indistinguishable from the side-channel. The following definition is from [28].

Definition 22 *Given a set of processes* $\{\Pi_0, \Pi_1, \cdots, \Pi_k\}$ *a common side-channel atomic block* Γ *for* $\Pi_0, \Pi_1, \cdots, \Pi_k$ *is a side-channel equivalent sequence of instructions so that each process* $\Pi_j (0 \leq j \leq k)$ *can be expressed as the repetition of this block* Γ*, i.e. there exists sequences* $\gamma_{j,i} \sim \Gamma$ *such that* $\Pi_j = \gamma_{j,1}, \cdots, \gamma_{j,l_j}$.

A common side-channel atomic block can always be created, because one can insert artificial dummy computations to the processes $\Pi_0, \Pi_1, \cdots, \Pi_k$ make all of them indistinguishable. The problem resides in finding an atomic block Γ for the most efficient implementation. A possible rearranging and/or rewriting of the processes may shorten Γ or limit the dummy operations and thus improve the overall performance. In [23], authors have provided elliptic curve addition and doubling algorithm in atomic blocks. In [107], we have also provided algorithms for same, which are used for pipelined computation of scalar multiplication. In [89], we have provided algorithms for hyperelliptic curve divisor addition and doubling in atomic blocks.

DPA and Countermeasures

The differential power analysis attacks were introduced by Kocher, Jaffe and June in [81]. The attack utilizes power consumption traces from several computations and retrieves the secret by statistically analyzing them. We briefly summarize the attack principle below.

Let $P_1, P_2, \cdots, P_l$ be l points of the curve. We recall that by point we mean both, a point on an elliptic curve or a divisor in the Jacobian of a hyperelliptic curve. Using the scalar multiplication algorithm as a black box, we compute $Q_1 = mP_1, Q_2 = mP_2, \cdots, Q_l = mP_l$. Let $Pow_i(t)$ be the power consumption trace of the computation of $mP_i, 1 \leq i \leq l$ at the time t. After obtaining all these data the cryptographic device is no more needed.

Let $m = m_{n-1}2^{n-1} + \cdots + m_1 2 + m_0$ be a n bit integer. Then clearly, $m_{n-1} = 1$. So the next bit m_{n-2} is attacked. The attack is based on the following principle: if $m_{n-2} = 1$ then $4P$ is computed or $4P$ is never computed. Then for $1 \leq i \leq l$, $4P_i$ is computed and a decision boolean function f_i is chosen, for example any bit value of $4P_i$, for $1 \leq i \leq l$. Based on the values of f_i, the Q_i's are partitioned into two sets, say $\mathcal{S}_0$ and $\mathcal{S}_1$. Let $Exp_0(t)$ be the mean of all $Pow_i(t)$ for $Q_i \in \mathcal{S}_0$ and $Exp_1(t)$ be the mean of all $Pow_i(t)$ for $Q_i \in \mathcal{S}_1$. We define a function g by

$$g(t) = Exp_0(t) - Exp_1(t)$$

It now permits deciding the value of m_{n-2}. The computation of $4P$ if done takes place at time t_{n-2}, i.e. when the second bit is treated. If $m_{n-2} = 0$, then a peak appears in the graphic representation of g at t_{n-2} else no peak appears.

After the second bit is decided, the third and other subsequent bits are attacked on the same principle.

Various countermeasures against DPA have been proposed in literature. All are based on randomization. Clearly by randomizing various parameters in the computation, the results of the statistical tests can be randomized and the guessing of the bit pattern of m will be difficult. We discuss some of the popular countermeasures below.

Private Key Randomization: Coron in [36] proposed to randomize the private exponent in the following simple manner to beat DPA. He suggested to compute $(m + r \times \#(G)P$ instead of mP where $\#(G)$ is the order of the group and r is a randomly chosen small integer. As the order of P is a divisor of $\#(G)$, clearly $(m + r \times \#(G))P = mP$. Coron's countermeasure can be used in any exponentiation algorithm. The next countermeasure is discussed in the light of ECC only. Oswald et al [127] make use of a randomised form of the Morain et al [117] point multiplication algorithm to obscure the private key. Later Ha et al in [59] extended this approach by combining the randomization approach with the NAF. In [123] the authors have proposed a new attack strategy, which can break this countermeasure with the help of around 20 observations.

Blinding the Point: In [36], the author proposes the following scheme to resist DPA. To compute mP, the author recommends to compute $m(P + R)$ and then to subtract $S = mR$ from the result. In order to make the scheme efficient, it is proposed to precompute and store R and S and to update R and S by $(-1)^r R$ and $(-1)^r S$ for a random r respectively before each computation. It thus recommends storing of 2 points.

Point Randomization: As mentioned in Section 2.8 to avoid the field inversion in affine co-ordinates various coordinate systems have been proposed for ECC. In Jacobian coordinates, the point $P(X : Y : Z)$ represents the point $(X/Z^2, Y/Z^3)$ on the curve. This representation is not unique. For example, for any random r in the field, the point with co-ordinates $P'(r^2X, r^3Y, rZ)$ represents the same point. Hence, in [36] Coron suggests another countermeasure against DPA, known as *Coron's Projective Randomization*. This countermeasure prescribes to choose a random element r in the field and compute mP' instead of mP. Clearly results of both the computations are the same. Taking the lead from ECC, Lange in [85] has proposed inversion-free arithmetic for HECC. Avanzi [9] has generalized Coron's projective randomization scheme to HECC.

As seen in Section 2.8, the addition group operation in both ECC and HECC becomes significantly cheaper if one point is kept in affine coordinates. This operation is called mixed addition. For example in ECC a general addition in Jacobian coordinates takes $12[m] + 4[s]$, whereas mixed addition costs $8[m] + 3[s]$. As Coron's projective randomization can not be implemented in affine coordinates, the performance penalty is $4[m] + 1[s]$ for each point addition.

Curve Randomization: Let C be an elliptic or hyperelliptic curve used to implement a cryptographic primitive. Let C' be a randomly chosen curve isomorphic to C. Let $G(C)$ and $G(C')$ be the groups formed by these curves on which cryptography is to be done. We know the groups $G(C)$ and $G(C')$ are isomorphic. In [69], the authors have proposed an elegant countermeasure against DPA, based on the isomorphism σ from $G(C)$ to $G(C')$. They propose to shift the computation from $G(C)$ to $G(C')$ and transform the result back to the original curve C after the computation. Let P be the base point and m be the scalar. Let $P' = \sigma(P) \in G(C')$. We compute mP' using parameters of the curve C'. After the computation, we transform the result back to C by $mP = \sigma^{-1}(mP')$. This countermeasure is named after its proposers as *Joye-Tymen Countermeasure*. Avanzi [9] has generalized this countermeasure to hyperelliptic curves, both for even and odd characteristics.

Curve randomization for ECC as proposed in [69] can not be applied to elliptic curves over binary fields. Because, the x-coordinate of the base point is invariant under the transformation σ. A variation of the method applicable to ECC over binary fields can be found in [28] (Section V.6).

Let z be a random nonzero field element. The steps are as follows.

1. Compute z^2, z^3, z^4, z^6.
2. Transform the base point $P(x, y)$ to (z^2x, z^3y).
3. Transform the curve coefficients (a, b) to $a' = z^4a, b' = z^6b$.
4. Compute scalar multiplication with the new point on the new curve.
5. Transform the result (x, y) back to the original curve using $(x, y) \to (x/z^2, y/z^3)$.

The additional cost of obtaining DPA resistance is $4[m]$ for Step 1; $2[m]$ for Step 2; $2[m]$ for Step 3 and finally $1[i] + 2[m]$ for Step 5.

Recently, Avanzi [9] has generalized this technique to HECC. Briefly, we describe the curve randomization countermeasure which we will use in our algorithm. Let the underlying curve C of the cryptosystem be $y^2 + h(x)y = f(x)$ where $h(x) = h_2x^2 + h_1x + h_0$ and $f(x) = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$. Let $D = [u(x), v(x)]$ be the base divisor in C and let z be a random field element.

1. Compute $z^{-1}, z^{-2}, z^{-3}, z^{-4}, z^{-5}, z^{-6}, z^{-8}$ and z^{-10}.
2. Transform $h(x)$ and $f(x)$ into $\widetilde{h}(x)$ and $\widetilde{f}(x)$ as follows:
   $\widetilde{h}(x) = z^{-1}h_2x^2 + z^{-3}h_1x + z^{-5}h_0$ and
   $\widetilde{f}(x) = x^5 + z^{-2}f_4x^4 + z^{-4}f_3x^3 + z^{-6}f_2x^2 + z^{-8}f_1x + z^{-10}f_0$.
3. Transform D to $\widetilde{D} = [\widetilde{u}(x), \widetilde{v}(x)]$, where $\widetilde{D}$ is defined as follows :
   If $\deg(u) = 2$ and $u(x) = x^2 + u_1x + u_0$ and $v(x) = v_1x + v_0$ then
   $\widetilde{u}(x) = x^2 + z^{-2}u_1x + z^{-4}u_0$ and $\widetilde{v}(x) = z^{-3}v_1x + z^{-5}v_0$.
   If $\deg(u) = 1$ and $u(x) = x + u_0$ and $v(x) = v_0$ then $\widetilde{u}(x) = x + z^{-2}u_0$ and $\widetilde{v}(x) = z^{-5}v_0$.
4. Compute scalar multiplication using the new curve and the new divisor.
5. The result is transformed back to the original curve using the inverse of Step 3 and the relevant powers of z (rather than z^{-1}).

The additional cost of attaining DPA resistance is as follows: $1[i] + 7[m]$ for Step 1; $8[m]$ for Step 2; maximum $4[m]$ for Step 3 and $8[m]$ for Step 5. If the characteristic of the field is odd, then the polynomial $h(x)$ can be taken to be zero and hence the cost of Step 2 would come down to $5[m]$.

2.11 Conclusion

In this chapter we introduced elliptic and hyperelliptic curves and the groups provided by them for cryptographic purposes. We also briefly dealt with field arithmetic, which is a pre-requisite for any implementation of (H)ECC. We discussed the group arithmetic in (H)ECC and measures adopted to make these efficient. In Section 2.9, we briefly discussed some scalar multiplication algorithms. In the last section briefly dealt with side-channel attacks and countermeasures against them.

Chapter 3

Parallelizing Explicit Formula for Arithmetic in the Jacobian of Hyperelliptic Curves

3.1 Introduction

The basic algorithm for performing arithmetic in the Jacobian of hyperelliptic curves is due to Cantor [22]. However, this algorithm is not sufficiently fast for practical implementation. There has been extensive research on algorithms for efficient arithmetic. The main technique is to obtain so called "explicit formula" for performing addition and doubling. These explicit formulae are themselves composed of addition, multiplication, squaring and inversion operations over the underlying finite field. Moreover, these formulae are specific to a particular genus. Thus there are separate formulae for genus 2 and genus 3 curves. See Table 2.3 in Chapter 2 for more details.

In this chapter, we consider the problem of parallel execution of explicit formula. An explicit formula can contain quite a few field multiplications and squarings. (In certain cases, this can even be 50 or more.) On the other hand, the number of inversions is usually at most one or two. An explicit formula usually also contains many field additions; however, the cost of a field addition is significantly less than the cost of a field multiplication or inversion. Hence the dominant operation in an explicit formula is field multiplication.

On inspection of different explicit formulae appearing in the literature there appear to be groups of multiplication operations that can be executed in parallel. Clearly the ability to perform multiplications in parallel will improve the speed of execution of the algorithm. This gives rise to the following question: *Given an explicit formula, what is the best parallel* *algorithm for computing the formula?*

Our first contribution in this chapter is to develop a general methodology for obtaining parallel version of any explicit formula. The methodology guarantees that the obtained parallel version requires the minimum number of rounds. The methodology can be applied to any explicit formula appearing in the literature. (There could also be other possible applications.)

The most efficient explicit formula for performing arithmetic in the Jacobian of genus 2 curve is given in [84, 85]. In [84], the affine coordinate representation of divisors is used and both addition and doubling involve a field inversion. On the other hand, in [85] the explicit formula is developed for inversion free arithmetic in the Jacobian.

Our second contribution is to apply our methodology to both [84] and [85]. For practical applications, it is necessary to consider resistance to side channel attacks. One important countermeasure is to perform a so-called encapsulated add-and-double algorithm (see [36, 64, 66] for details). We develop parallel versions of encapsulated add-and-double algorithm for both [84] and [85]. In many situations, the number of parallel multipliers available may be limited. To deal with such situations we present the encapsulated add-and-double algorithm using inversion free arithmetic using 4, 8 and 12 multipliers. For the affine version we present an algorithm using 8 multipliers. All of our algorithms are optimal parallel algorithms in the sense that no other parallel algorithm can perform the computation in lesser number of rounds.

Some of the results that we obtain are quite striking. For example, using 4 multipliers, we can complete the inversion free encapsulated add-and-double algorithm in 27 rounds and using 8 multipliers we can complete it in 14 rounds. The algorithm involves 108 multiplications. In the case of arithmetic using affine coordinates, the 8 multiplier algorithm will complete the computation in 11 rounds including an inversion round. Usually inversions are a few times costlier than multiplications, the actual figure being dependent upon exact implementation details. However, from our results it is clear that in general the parallel version of arithmetic using affine coordinates will be costlier than the parallel version of inversion free arithmetic.

3.2 General Methodology for Parallelizing Explicit Formula

An explicit formula for performing doubling (resp. addition) in the Jacobian of a hyperelliptic curve is an algorithm which takes one (resp. two) reduced divisor(s) as input and produces a reduced divisor as output. Also the parameters of the curve are available to the algorithm.

The algorithm proceeds by a sequence of elementary operations, where each operation is either a multiplication or an addition or an inversion over the underlying field. In general the formulae involve one inversion. If there is one inversion, the inversion operation can be neglected and the parallel version can be prepared without it. Later, it can be plugged in as a separate round at an appropriate place. The same is true if the formula contains more than one inversions. Hence, we can assume that the formula is inversion-free. The cost of a field multiplication (or squaring) is significantly more than the cost of a field addition and hence the number of field multiplications is the dominant factor determining the cost of the algorithm. On inspection of the different explicit formulae available in the literature, it appears that there are groups of multiplication operations which can be performed in parallel. The ability to perform several multiplications in parallel can significantly improve the total computation time. So the key problem that we consider is the following: *Given an explicit formula, identify the groups of multiplication operations that can be performed in parallel.* In this section we develop a general methodology for solving this problem.

Let $\mathcal{F}$ be an explicit formula. Then $\mathcal{F}$ consists of multiplication and addition operations. Also several intermediate variables are involved. First we perform the following pre-processing on $\mathcal{F}$.

1. *Convert all multiplications to binary operation* : Operations which are expressed as a product of three or more variables are rewritten as a sequence of binary operations. For example, the operation $p_5 = p_1p_2p_3$ is rewritten as $p_4 = p_1p_2$ and $p_5 = p_3p_4$.

2. *Reduce multiplication depth* : Suppose we are required to perform the following sequence of operations: $p_3 = p_1^2p_2;\ p_4 = p_3p_2$. The straightforward way of converting to binary results in the following sequence of operations: $t_1 = p_1^2;\ p_3 = t_1p_2;\ p_4 = p_3p_2$. Note that the three operations *have* to be done sequentially one after another. On the other hand, suppose we perform the operations in the following manner: $\{t_1 = p_1^2;\ t_2 = p_2^2;\}\{p_3 = t_1p_2;\ p_4 = t_1t_2\}$. In this case, the operations within $\{\}$ can be performed in parallel and hence the computation can be completed in two parallel rounds. The total number of operations increases to 4, but the number of parallel rounds is less.

3. *Eliminate reuse of variable names* : Consider the following sequence of operations:

$$q_1 = p_1 + p_2;\ q_2 = p_3;\ \ldots;\ q_1 = p_4 + p_5;\ \ldots$$

   In this case, at different points of the algorithm, the intermediate variable q_1 is used to store the values of both $p_1 + p_2$ and $p_4 + p_5$. During the process of devising the parallel algorithm we rename the variable q_1 storing the value of $p_4 + p_5$ by a unique new name. In the parallel algorithm we can again suitably rename it to avoid the overhead cost of initializing a new variable.

4. *Labeling process* : We assign unique labels to the addition and multiplication operations and unique names to the intermediate variables.

Given a formula $\mathcal{F}$, we define a directed acyclic graph $G(\mathcal{F})$ in the following fashion.

- The nodes of $G(\mathcal{F})$ correspond to the arithmetic operations and variables of $\mathcal{F}$. Also there are nodes for the parameters of the input divisor(s) as well as for the parameters of the curve.
- The arcs are defined as follows: Suppose $\mathbf{id}: r = qp$ is a multiplication operation. The identifier $\mathbf{id}$ is the label assigned to this operation. Then the following arcs are present in $G(\mathcal{F})$: $(q, \mathbf{id}), (p, \mathbf{id})$ and $(\mathbf{id}, r)$. Similarly, the arcs for the addition operations are defined, with the only difference being the fact that the indegree of an addition node may be greater than two.

Proposition 23 *The following are true for the graph $G(\mathcal{F})$.*
1. The indegree of variable nodes corresponding to the parameters of the input divisors and the parameters of the curve is zero.
2. The indegree of any node corresponding to an intermediate variable is one.
3. The outdegree of any node corresponding to an addition or multiplication operation is one.

Note that the outdegree of nodes corresponding to variables can be greater than one. This happens when the variable is required as input to more than one arithmetic operation. Our aim is to identify the groups of multiplication operations that can be performed in parallel. For this purpose, we prepare another graph $G^*(\mathcal{F})$ from $G(\mathcal{F})$ in the following manner:

- The nodes of $G^*(\mathcal{F})$ are the nodes of $G(\mathcal{F})$ which correspond to multiplication operation.
- There is an arc $(\mathbf{id_1}, \mathbf{id_2})$ from node $\mathbf{id_1}$ to node $\mathbf{id_2}$ in $G^*(\mathcal{F})$ only if there is a path from $\mathbf{id_1}$ to $\mathbf{id_2}$ in $G(\mathcal{F})$ which does not pass through another multiplication node.

The graph $G^*(\mathcal{F})$ captures the ordering relation between the multiplication operations of $\mathcal{F}$. Thus, if there is an arc $(\mathbf{id_1}, \mathbf{id_2})$ in $G^*(\mathcal{F})$, then the operation $\mathbf{id_1}$ must be done before the operation $\mathbf{id_2}$. We now define a sequence of subgraphs of $G^*(\mathcal{F})$ and a sequence of subsets of nodes of $G^*(\mathcal{F})$ in the following manner.

- $G_1(\mathcal{F}) = G^*(\mathcal{F})$ and M_1 is the set of nodes of G_1 whose indegree is zero.

- For $i \geq 2$, G_i is the graph obtained from G_{i-1} by deleting the set M_{i-1} from G_{i-1} and M_i is the set of nodes of G_i whose indegree is zero.

Let r be the least positive integer such that G_{r+1} is the empty graph, i.e., on removing M_r from G_r, the resulting graph becomes empty.

Proposition 24 *The following statements hold for the graph $G^*(\mathcal{F})$.*
1. The sequence $M_1, \ldots, M_r$ forms a partition of the nodes of $G^*(\mathcal{F})$.
2. All the multiplications in any M_i can be performed in parallel.
3. There is a path in $G^*(\mathcal{F})$ from some vertex in M_1 to some vertex in M_r. Consequently, at least r parallel multiplication rounds are required to perform the computation of $\mathcal{F}$.

It is easy to obtain the sets M_i's from the graph $G^*(\mathcal{F})$ by a modification of the standard topological sort algorithm [35]. The sets M_i $(1 \leq i \leq r)$ represent only the multiplication operations of $\mathcal{F}$. To obtain a complete parallel algorithm, we have to organize the addition operations and take care of the intermediate variables. There may be some addition operations at the beginning of the formula. Since additions are to be performed sequentially, we can ignore these additions while deriving the parallelized formula, treating the sums they produce as inputs. Later, they can be plugged in at the beginning of the formula.

For $1 \leq i \leq r-1$, let A_i be the set of addition nodes which lie on a path from some node in M_i to some node in M_{i+1}. Further, let A_r be the set of addition nodes which lie on a path originating from some node in M_r. There may be more than one addition operation in a path from a node in M_i to a node in M_{i+1}. These additions have to be performed in a sequential manner. (Note that we are assuming that $\mathcal{F}$ starts with a set of multiplication operations and ends with a set of addition operations. It is easy to generalize to a more general form.)

Each multiplication and addition operation produces a value which is stored in an intermediate variable. We now describe the method of obtaining the set of intermediate variables required at each stage of computation. Let $I_1, \ldots, I_{2r}$ and $O_1, \ldots, O_{2r}$ be two sequences of subsets of nodes of $G(\mathcal{F})$, where each I_i and O_j contain nodes of $G(\mathcal{F})$ corresponding to variables. The parameters of the curve and the input divisor(s) are not included in any of the I_i and O_j's. These are assumed to be additionally present throughout the algorithm. For $1 \leq i \leq r$, these sets are defined as follows:

1. I_{2i-1} contains intermediate variables which are the inputs to the nodes in M_i.
2. I_{2i} contains intermediate variables which are the inputs to the addition nodes in A_i.
3. O_{2i-1} contains intermediate variables which are the outputs of the nodes in M_i.
4. O_{2i} contains intermediate variables which are the outputs of the addition nodes in A_i.

For $1 \leq j \leq 2r$, define

$$V_j = (\cup_{i=1}^{j} O_i) \cap (\cup_{i=j+1}^{2r} I_i). \tag{3.1}$$

If a variable x is in V_j, then it has been produced by some previous operation and will be required in some subsequent operation. We define the parallel version $\mathsf{par}(\mathcal{F})$ of $\mathcal{F}$ as a sequence of rounds

$$\mathsf{par}(\mathcal{F}) = (\mathcal{R}_1, \ldots, \mathcal{R}_r). \tag{3.2}$$

where $\mathcal{R}_i = (M_i, V_{2i-1}, A_i, V_{2i})$. In round i, the multiplications in M_i can be performed in parallel; the sets V_{2i-1} and V_{2i} are the sets of intermediate variables and A_i is the set of addition operations. Note that the addition operations are not meant to be performed in parallel. Indeed, in certain cases the addition operations in A_i have to be performed in a sequential manner. We define several parameters of $\mathsf{par}(\mathcal{F})$.

Definition 25 *Let* $\mathsf{par}(\mathcal{F}) = (\mathcal{R}_1, \ldots, \mathcal{R}_r)$*, be the r-round parallel version of the explicit formula $\mathcal{F}$. Then*
1. The total number of multiplications (including squarings) occuring in $\mathsf{par}(\mathcal{F})$ *will be denoted by* TM.
2. The multiplication width (MW) *of* $\mathsf{par}(\mathcal{F})$ *is defined to be* $\mathsf{MW} = \max_{1 \leq i \leq r} |M_i|$.
3. The buffer width (BW) *of* $\mathsf{par}(\mathcal{F})$ *is defined to be* $\mathsf{BW} = \max_{1 \leq i \leq 2r} |V_i|$.
4. A path from a node in M_1 to a node in M_r is called a critical path in $\mathsf{par}(\mathcal{F})$.
5. The value r is the critical path length (CPL) *of* $\mathsf{par}(\mathcal{F})$.

The parameter MW denotes the maximum number of multipliers that can operate in parallel. Using MW parallel multipliers $\mathcal{F}$ can be computed in r parallel rounds. The buffer width BW denotes the maximum number of variables that are required to be stored at any stage in the parallel algorithm.

3.2.1 Decreasing the Multiplication Width

The method described above yields a parallel algorithm $\mathsf{par}(\mathcal{F})$ for a given explicit formula $\mathcal{F}$. It also fixes the number of computational rounds r required to execute the algorithm using MW number of processor. By definition, MW is the maximum number of multiplications taking place in a round. However, it may happen that in many rounds the actual number of multiplications is less than MW. If we use MW multipliers, then some of the multipliers will be idle in such rounds. The most ideal scenario is $\mathsf{MW} \approx \lceil \mathsf{TM}/r \rceil$. However, such an ideal situation may not come about automatically. We next describe a method for making the distribution of the number of multiplication operations more uniform among various rounds.

We first prepare a *requirement table*. It is a table containing data about the intermediate variables created in the algorithm. For every variable it contains the name of the variables used in the expressions computing it, the latest round in which one of such variables is created and the earliest round in which the variable itself is used. For example, suppose an intermediate variable $v_x = v_y * v_z$ is computed in the j-th round. Of v_y and v_z, let v_z be the one which is computed later and in the i-th round. Let v_x be used earliest in the k-th round. Then in the requirement table we have an entry for v_x consisting of v_y, v_z, i, k. If both of v_x and v_y are input values then we may take $i = 0$. Note that we have $i < j < k$.

Now suppose, there are more than $\lceil \mathsf{TM}/r \rceil$ multiplications in the j-th round. Further suppose that for some j_1 $(i+1 \leq j_1 \leq k-1)$, the number of multiplications in the j_1^{th} round is less than $\lceil \mathsf{TM}/r \rceil$. Then we transfer the multiplication producing v_x to the j_1^{th} round and hence reduce the multiplication width of the j-th round. This change of position of the multiplication operation does not affect the correctness of the algorithm.

This procedure is applied as many times as possible to rounds which contain more than $\lceil \mathsf{TM}/r \rceil$ multiplications. As a result we obtain a parallel algorithm with a more uniform distribution of number of multiplication operations over the rounds and consequently reduces the value of MW.

3.2.2 Managing Buffer Width

The parameter BW provides the value of the maximum number of intermediate variables that is required to be stored at any point in the algorithm. This is an important parameter for applications where the amount of memory is limited. We justify that obtaining parallel version of an explicit formula does not substantially change the buffer width. Our argument is as follows.

First note that the total number of multiplications in the parallel version is roughly the same as the total number of multiplications in the original explicit formula. The only place where the number of multiplications increases is in the pre-processing step of reducing the multiplication depth. Moreover, the increase is only a few multiplications. The total number of addition operations remain the same in both sequential and parallel versions. Since the total numbers of multiplications and additions are roughly the same, the total number of intermediate variables also remains roughly the same.

Suppose that after round k in the execution of the parallel version, i intermediate variables have to be stored. Now consider a sequential execution of the explicit formula. Clearly, in the sequential execution, all operations upto round k has to be executed before any operation of round greater than k can be executed. The i intermediate variables that are required to be stored after round k are required as inputs to operations in round greater than k. Hence these intermediate variables are also required to be stored in the sequential execution of the explicit formula.

3.3 Application to Lange's Explicit Formulae

In [84] and [85], Lange presented explicit formulae for addition and doubling in the Jacobian of genus 2 hyperelliptic curves. In fact, there are many special cases involved in these explicit formulae and our methodology can be applied to all the cases. But to be brief, we restrict our attention to the most general and frequent case only. The formulae in [84] uses an inversion each for addition and doubling while the formulae in [85] does not require any inversion.

We apply the methodology described in Section 3.2 separately to the formulae in [84] and [85]. In the case of addition, the inputs are two divisors D_1 and D_2 and in the case of doubling the input is only one divisor D_1. We use the following conventions.

- We assume that the curve parameters $h_2, h_1, h_0, f_4, f_3, f_2, f_1, f_0$ are available to the algorithm.
- We do not distinguish between squaring and multiplication.
- The labels for the arithmetic operations in the explicit formula for addition start with **A** and the labels for the arithmetic operations in the explicit formula for doubling start with **D**. The second letter of the label (**M** or **A**) denotes (m)ultiplication or (a)ddition over the underlying field. Thus **AM23** denotes the 23rd multiplication in the explicit formula for addition.
- The intermediate variables for the explicit formula for addition are of the form p_i and the intermediate variables for the explicit formula for doubling are of the form q_j.
- In [84, 85], multiplications by curve constants are presented. However, during the total multiplication count, some of these operations are ignored, since for most practical applications the related curve constants will be 0 or 1. In this section, we include the multiplication by the curve parameters.
- The set of intermediate variables (V_i's) required at any stage is called the buffer state.

3.3.1 Inversion Free Arithmetic

In this section, we consider the result of application of the method of Section 3.2 to the inversion free formula for addition and doubling given in [85]. We present the details in the

Table 3.1: Parameters for parallel versions of explicit formula in [85].

	MW	BW	CPL	TM
Add	8	20	8	59
Double	11	15	8	65

Table 3.2: Parameters for parallel versions of explicit formula in [84].

	MW	BW	CPL	TM
Add	6	12	7^*	29^*
Double	5	13	8^*	34^*

* Including one inversion

Appendix A. The details of addition formula are presented in Section A.1 and the details of the doubling formula are presented in Section A.2. We present a summary of the parameters of the parallel versions in Table 3.1. Based on Table 3.1 and Proposition 24(3), we obtain the following result.

Theorem 26 *Any parallel algorithm for executing either the explicit formula for addition or the explicit formula for doubling presented in [85] will require at least 8 parallel multiplication rounds. Consequently, the parallel algorithms presented in Sections A.1 and A.2 are optimal algorithms.*

3.3.2 Arithmetic Using Affine Coordinates

The most efficient explicit formula for arithmetic using affine coordinates has been presented in [84]. Here we consider the result of applying the methodology of Section 3.2 to this formula. Again we present the details in the Appendix A. The parallel version of the addition formula is presented in Section A.3 and the parallel version of the doubling formula is presented in Section A.4. A summary of the results is presented in Table 3.2. We have the following result about parallel versions of the explicit formula in [84].

Theorem 27 *Any parallel algorithm for executing the explicit formula for addition (resp. doubling) presented in [84] will require at least 7 (resp. 8) parallel multiplication rounds. Consequently, the parallel algorithms presented in Sections A.3 and A.4 are optimal algorithms.*

3.4 Computation Charts for HECC Formulae

The methodology developed in Section 3.2 was used to derive the parallel version of some of the formulae for arithmetic in the Jacobians of genus 2 curves. Instead of applying the methodology to divisor addition and doubling formulae separately, we have encapsulated the addition and doubling formulae into one. The encapsulated add-and-double formula takes as input two divisors P_1 and P_2 and computes $P_1 + P_2$ and $2P_1$ simultaneously. This can be used in a Montgomery's ladder type scalar multiplication algorithm (see [65, 66]). In case of affine arithmetic at least one improvement can be easily achieved. Both divisor addition and doubling involve one field inversion each. In the encapsulated formula one can compute both the inversions by a single inversion using Montgomery's trick. Also, computation of scalar multiplication using such an encapsulated algorithm is secure against SPA.

The parallel version of encapsulated add-and-double algorithm for genus 2 curves using inversion-free arithmetic using 4, 8 and 12 multipliers have been presented in Tables 3.4, 3.5 and 3.6 respectively. The same algorithm using affine arithmetic using 8 multipliers has been presented in Table 3.7.

In Table 3.4, the encapsulated algorithm in inversion-free arithmetic has been presented with 4 processors. Each round of computation has a multiplication phase and an addition phase. The multiplication phase has at most 4 multiplications to be computed in parallel by 4 multipliers. The addition phase has some addition operations, which are proposed to be computed sequentially by one adder. The operations are identified by the identifiers used in the parallel versions of the addition and doubling formulae presented in Appendix A. Similarly for the other tables.

3.5 Application to Elliptic Curves

Elliptic Curves in affine co-ordinates involves only a few multiplications and squaring. In projective or Jacobian co-ordinates the number of multiplications in the encapsulated add and double algorithm involve substantial number of multiplications where one may use several multipliers to compute in parallel. We applied our method to the x-co-ordinate only encapsulated add and double formula presented in [64]. The results obtained are presented in Table 3.3.

Parallel version of x-coordinate only encapsulated add and double formula for elliptic curves using Jacobian coordinates is presented in Section A.5 of Appendix A.

Table 3.3: Parameters for parallel versions of encapsulated add and double formula in [64].

MW	BW	CPL	TM
5	8	5	19

3.6 Conclusion

In this work, we have developed a general methodology for deriving parallel versions of any explicit formula for computation of divisor addition and doubling. We have followed the methods to derive the parallel version of the explicit formula given in [85] and [84]. We have considered encapsulated add-and-double algorithms to prevent side channel attacks. Moreover, we have described parallel algorithms with different number of processors.

It has been shown that for the inversion free arithmetic of [85] and with 4, 8 and 12 field multipliers an encapsulated add-and-double can be carried out in 27, 14 and 10 parallel rounds respectively. All these algorithms optimal in the number of parallel rounds. In the case of arithmetic using affine coordinates [84], an eight multiplier algorithm can perform encapsulated add-and-double using 11 rounds including an inversion round. Since an inversion is usually several times costlier than a multiplication, in general the parallel version of inversion free arithmetic will be more efficient than the parallel version of arithmetic using affine coordinates.

We have applied our general methodology to explicit formula for genus 2 curves. The same methodology can also be applied to the explicit formula for genus 3 curves and to other explicit formulae appearing in the literature. Performing these tasks will be future research problems.

Table 3.4: Computation chart using four parallel multipliers for inversion free arithmetic of [85].

Rnd	Operation
1	AM01, AM02, AM03, AM04
2	AM05, AM06, AM07, AM08
	AA01, AA02, AA03, AA04
3	DM01, DM02, DM04, DM08
	DA01, DA02, DA03, DA04
4	DM09, AM09, AM10, AM11
	DA05, DA06, DA07, AA07, AA08, AA09
5	AM12, AM13, AM14, AM16
	AA05, AA06
6	DM12, DM13, DM14, DM15
	DA08
7	DM16, DM17, DM18, DM19
	DA09, DA10
8	DM20, DM22, AM17, AM18
	AA10, DA11, DA11, DA12, DA13
9	AM19, AM20, AM21, AM22
	AA12, AA13, AA14, AA15
10	DM23, DM24, DM25, DM26
	DA14, DA15, DA16, DA17, DA18, DA19
11	DM27, DM29, AM23, AM24
12	AM25, AM26, AM27, AM28
13	AM29, AM30, DM30, DM31
	AA16, AA17
14	DM32, DM33, DM34, DM35
	DA20, DA21
15	AM31, AM32, AM33, AM34
	AA18, AA19
16	AM35, AM37, AM38, DM36
17	DM37, DM38, DM39, DM41
18	DM43, AM39, AM40, AM41
19	AM42, AM43, AM44, AM46
	AA20, AA21, AA22, AA23, AA24, AA25
20	DM44, DM45, DM46, DM47
21	DM48, DM49, DM50, AM47
	DA22, DA23, DA24, DA25
22	AM48, AM49, AM50, AM51
23	AM52, AM53, DM51, DM52
	AA26, AA27
24	DM53, DM54, DM55, DM56
25	DM57, AM54, AM55, AM56
	DA26, DA27, DA28
26	AM57, DM58, DM59, DM60
	AA28, AA29, AA30, AA31
27	DM62, DM63, DM65, DM66
	DA29, DA30, DA31, DA32, DA33, DA34

Table 3.5: Computation chart with eight processors for inversion free arithmetic of [85].

Rnd	Operation
1	AM01, AM02, AM03, AM04, AM05, AM06, AM07, AM08
	AA01, AA02, AA03, AA04
2	DM01, DM02, DM04, DM08, DM09, AM09, AM10, AM11
	DA01, DA02, DA03, DA04, DA04, DA05, DA06, DA07
3	AM12, AM13, AM14, AM16, DM12, DM13, DM14, DM15
	AA05, AA06, AA07, AA08, DA09
4	DM16, DM17, DM18, DM19, DM20, DM22, AM17, AM18
	DA08, DA09, DA10, DA11, DA12, DA13
5	AM19, AM20, AM21, AM22, DM23, DM24, DM25, DM26
	AA10, AA11, AA12, AA13, AA14, AA15
6	DM27, DM29, AM23, AM24, AM25, AM26, AM27, AM28
	DA14, DA15, DA16, DA17, DA18, DA19
7	AM29, AM30, DM30, DM31, DM32, DM33, DM34, DM35
	AA16, AA17, DA20, DA21
8	AM31, AM32, AM33, AM34, AM35, AM37, AM38, DM36
	AA18, AA19
9	DM37, DM38, DM39, DM41, DM43, AM39, AM40, AM41
10	AM42, AM43, AM44, AM46, DM44, DM45, DM46, DM47
	AA21, AA22, AA23, AA24, AA25
11	DM48, DM49, DM50, AM47, AM48, AM49, AM50, AM51
	DA22, DA23, DA24, DA25
12	AM52, AM53, DM51, DM52, DM53, DM54, DM55, DM56
	AA26, AA27
13	DM57, AM54, AM55, AM56, AM57
	AA28, AA29, AA30, AA31, DA26, DA27, DA28
14	DM58, DM59, DM60, DM62, DM63, DM65, DM66
	DA29, DA30, DA31, DA32, DA33, DA34

Table 3.6: Computation chart with twelve processors for inversion free arithmetic of [85].

Rnd	Operation
1	AM01, AM02, AM03, AM04, AM05, AM06, AM07, DM01, DM02, DM04, DM08, DM09
	DA1, DA02, DA03, DA04, DA05, DA06, DA07
2	AM08, DM12, DM13, DM14, DM15, DM16, DM17, DM18, DM19, DM20, DM22
	AA01, AA02, AA03, AA04, DA8, DA9, DA10, DA11, DA11, DA12, DA13
3	AM09, AM10, AM11, AM12, AM13, AM14, DM23, DM24, DM25, DM26, DM27, DM29
	DA14, DA15, DA16, DA17, DA18, DA19
4	AM16, DM30, DM31, DM32, DM33, DM34, DM35
	AA05, AA06, AA07, AA08, AA09, DA20, DA21
5	AM17, AM18, AM19, AM20, AM21, AM22, DM36, DM37, DM38, DM39, DM41, DM43
	AA10, AA11, AA12, AA13, AA14, AA15
6	AM23, AM24, AM25, AM26, AM27, AM28, AM29, AM30, DM44, DM45, DM 46, DM47
	AA16, AA17
7	AM31, AM32, AM33, AM34, AM35, AM37, AM38, DM48, DM49, DM50
	AA18, AA19, DA22, DA23, DA24, DA25
8	AM39, AM40, AM41, AM42, AM43, AM44, AM46, DM51, DM52, DM53, DM54, DM55
	AA20, AA21, AA22, AA23, AA24, AA25
9	AM47, AM48, AM49, AM50, AM51, AM52, AM53, DM56, DM57
	AA26, AA27, DA26, DA27, DA28
10	AM54, AM55, AM56, AM57, DM58, DM59, DM60, DM62, DM63, DM65, DM66
	AA28, AA29, AA30, AA31, DA29, DA30, DA31, DA32, DA33, DA34

Table 3.7: Computation chart with eight processors for arithmetic using affine coordinates in [84].

Rnd	Operation
	AA01, AA02, AA03, AA04, DA01, DA02, DA03, DA04
1	AM01, AM02, AM03, DM01, DM02, DM03, DM04, DM05
	AA05, AA06, AA07, AA08, DA05, DA06, DA07, DA08
2	AM04, AM05, AM06, AM07, DM06, DM07, DM08, DM09
	AA09, DA09, DA10, DA11, DA12, DA13
3	AM08, AM09, DM10, DM11, DM12, DM13
	AA10, AA11, DA14, DA15
4	AM10, AM11, AM12, AM13, DM14, DM15, DM16, DM17
5	$\gamma = q_{17}p_{20}$
6	$\delta = \gamma^{-1}$
7	$q_{20} = \delta p_{20}, p_{24} = \delta q_{17}$
8	AM15, AM16, AM17, AM18, DM19, DM20, DM21, DM22
	AA12, AA13, AA14, AA15, DA16, DA17
9	AM19, AM20, AM22, AM23, DM23, DM24, DM25, DM26 AA16, AA17, AA18, AA19, DA18, DA19
10	AM21, AM24, AM25, AM26, DM27, DM28, DM29, DM30
	AA20, AA21, DA20, DA21
11	AM27, AM28, AM29, DM31, DM32, DM33, DM34
	AA22, AA23, AA24, DA22, DA23, DA24

Chapter 4

Towards Minimizing Memory Requirement for Implementation of Hyperelliptic Curve Cryptosystems

HECC arithmetic is now generally performed using so called explicit formulae. In literature, there is no result which focuses on the exact memory requirement for implementation these formulae. This is the first work to report such minimal memory requirement. Also, in the work we have provided a general methodology for realization of explicit formulae with minimal number of registers. Applying the methodology, this work settles the issue for some important explicit formula available in the literature. This is an attempt to experimentally solve a particular instance based on HECC explicit formulae of the so called "Register Sufficiency Problem", which is an NP-complete problem.

4.1 Introduction

ECC and HECC are considered to be the ideal cryptosystem for mobile devices. Mobile devices are generally equipped with much less computing power. Before trying to implement a cryptosystem on these devices one has to be sure that the resources, particularly memory, available on these devices are sufficient for the implementation. For ECC, the formulas are smaller involving 7 to 15 multiplications and squarings in the underlying field in non-affine arithmetic. So it is simple to calculate the numbers of registers required to store the inputs, outputs and intermediate variables. In many works reported in literature on ECC, the authors have provided the number of registers required for the computation. These figures are obtained by manual checking. However, to our knowledge, there is no result stating that a particular ECC algorithm cannot be implemented in less than a certain amount of memory.

There has been no study of exact memory requirement for an implementation of HECC. In [15] the authors have briefly touched the topic. Besides that there has been no mention of memory requirement in any work on HECC so far. The point addition and doubling algorithm for ECC can be found in many papers as a sequence of three address codes, like, $R_i = R_j \; op \; R_k$, where R_i, R_j, R_k are register names or constants and *op* is an arithmetic operation. We will refer to this format as *Explicit Register Specified Format (ERSF)*. Looking at a formula in ERSF, one can know exactly how many registers will be required for its implementation. Unfortunately, no HECC explicit formulae occurring in the literature has been described in ERSF. All are described as a set of mathematical equations. We will refer to this format of representing a formula as *raw* format.

Probably, the reason behind all HECC formulae appearing in raw format only is the fact that HECC formulae are relatively complex ones than those of ECC. An HECC (genus 2) formula involves around 25 to 50 multiplication with or without an inversion. The first step for expressing such a formulae in ERSF is to know how many registers will be required. For a long formula it is difficult to manually find out how many registers will suffice. It is nearly impossible to say what is the minimal requirement.

In fact, finding the minimum number of intermediate variables required for the execution of a formula is an NP-complete problem. This is called *Register Sufficiency Problem* and has been studied earlier in a very general framework. However, the results obtained earlier does not apply straight away to the scenario we are in now. In the current work, we try to provide an experimental solution to this particular instance of the problem. We believe that our methodology can be applied to many similar situations.

The question is: *Given an explicit formula what is the minimum number of registers required to compute it sequentially or in parallel*? In the current work, we provide a methodology (in Section 4.3) to address this issue. We used this methodology to compute the minimum memory requirement for some of the well known and widely used formulas. For elliptic curves, we checked for the general addition formula in Jacobian coordinates and found that it can not be executed with less than 7 registers. It is known that the elliptic curve addition can be done in 7 registers. Our finding ensures that it can not be done in less.

We have used our methodology to find the minimum register requirement for many formulae in HECC. The formulae proposed by Pelzl et al [133] for a special class of curves are most efficient ones. Their doubling formulae uses 10 registers and the addition uses 15. Similar formulae are proposed by Lange in [84]. These set of formulae use 11 and 15 registers for doubling and addition respectively. Thus for computing the scalar multiplication both set of formulae require 16 registers. Thus, although the formulae proposed in [133] are cheaper in number of operations, they use the same amount of memory as the ones in [84]. All our findings have been described in Section 4.5.

The rest of this chapter is organized as follows. In Section 4.2, we briefly describe the theoretical status of the register sufficiency problem. In Section 4.3, we describe our methodology. In Section 4.5, we provide the results we found. Section 4.6 concludes the paper. The detailed description of the formulae we consider are presented in ERSF in the appendices.

4.2 Theoretical Status of the Register Sufficiency Problem

The problem of minimizing the number of intermediate variables required for executing a set of arithmetic formulae has been studied earlier. The problem is called the register sufficiency problem and the decision version is known to be NP-complete [147]. See [48, page 272] for further details.

The minimization version has also been studied in the literature. According to the compendium of NP-optimization problems [1], there is an $O(\log^2 n)$ (where n is the number of operations) approximation algorithm for this problem [73]. This result is obtained in [73] using general results on flow problems and does not lead to a practical algorithm to solve the problem. Another issue is that one does not obtain any idea about the constant in the $O(\log^2 n)$ expression. Moreover, for the cases in which we are interested n is at most around 200. For such n, a performance guarantee of $O(\log^2 n)$ is not really useful.

In fact, in our implementation, in many cases we are able to obtain the minimum number of registers and in other cases we are able to show that the minimum is at most one or two less than the result we obtain. Thus, on the one hand, the theoretical status of the problem is not really useful for obtaining a practical algorithm and on the other hand, for the concrete situations in which we are interested, we obtain better performance guarantee than the known theoretical bound.

4.3 Our Methodology

Our primary aim in this work is to answer the question:
Problem: Given an explicit formula $\mathcal{F}$, what is the minimum number of intermediate variables required to be stored to execute $\mathcal{F}$?

Let $\mathcal{F}$ be an explicit formula. Let $p_1, \cdots, p_k$ be the inputs to $\mathcal{F}$. We can look at $\mathcal{F}$ as a sequence of arithmetic operations, each having a unique id, like; $Id_i : p_i = q_i \; op_i \; r_i, k \leq i \leq n$, where op_i is one of the binary operations $\{+, -, *, /\}$ and q_i, r_i are among the p_j's $j < i$. In fact, explicit formula in literature generally occur in raw format. We can convert them into this form by a simple parser program.

We will call a sequence $S = \{Id_{i_1}, Id_{i_2}, \cdots, Id_{i_{n-k+1}}\}$ or simply $S = \{i_1, i_2, \cdots, i_{n-k+1}\}$ of operation id's of $\mathcal{F}$ a *valid sequence* if $\mathcal{F}$ can be computed by executing its operations in the order as dictated by the sequence S. For example if $\mathcal{F} = \{Id_1, Id_2, Id_3, Id_4\}$, where $Id_1 : p_4 = x * y$, $Id_2 : p_5 = p_4 * z$, $Id_3 : p_6 = y * z$ and $Id_4 : p_7 = p_5 * p_6$, then there are only two valid sequences, namely, $\{1, 2, 3, 4\}$ and $\{3, 1, 2, 4\}$. $\mathcal{F}$ can not be executed in any other order.

Further, one may be interested to know which valid sequence needs the minimum number of intermediate variables for executing the explicit formula $\mathcal{F}$.

Let $\mathcal{F}$ be an explicit formula and let $\mathcal{A}_0$ be the set of inputs to it. In $\mathcal{F}$, there are certain computations which can be computed from the the set $\mathcal{A}_0$ of inputs to $\mathcal{F}$. After one or more these are executed we get some intermediate values which can trigger some more operations of $\mathcal{F}$. Let V_0 be the set of computations in $\mathcal{F}$, which can be computed from the set $\mathcal{A}_0$ of inputs to $\mathcal{F}$. Let $|V_0| = \alpha_0$ be the size of the set V_0. So one can begin the execution of $\mathcal{F}$ starting from any one of these α_0 operations. Suppose we choose the operation

$$Id_{i_1} : p_{i_1} = q_{i_1}\ op_{i_1}\ r_{i_1}$$

in V_0 to be executed first. After this operation we have the value of p_{i_1} available to us. So an operation involving p_{i_1} and some other known value in A_0 in its right hand side can also now be executed. Let $\mathcal{A}_{i_1} = \mathcal{A}_0 \bigcup\{p_{i_1}\}$. Let $V_1^{i_1}$ be the set of operations in $\mathcal{F}_1 = \mathcal{F} - \{Id_{i_1}\}$, which can be executed from the the values available in $\mathcal{A}_{i_1}$. Let $|V_1^{i_1}| = \alpha_{i_1}$. Note that the set $V_1^{i_1}$ and the value of α_{i_1} depend upon the choice of Id_{i_1}. Thus, we have α_{i_1} options for executing the next operation of $\mathcal{F}$. Suppose we choose

$$Id_{i_2} : p_{i_2} = q_{i_2}\ op_{i_2}\ r_{i_2}$$

We update the set of available values as $\mathcal{A}_{i_1,i_2} = \mathcal{A}_{i_1} \bigcup\{p_{i_2}\}$ and look for the set of operations in $\mathcal{F}_2 = \mathcal{F}_1 - \{Id_{i_2}\}$ which can be computed from the set of values available in $\mathcal{A}_{i_1,i_2}$ and proceed like this. In general, if k operations $Id_{i_1}, \cdots, Id_{i_k}$ are already computed and p_{i_k} is the output of the last computation, then we update the set of available values as $\mathcal{A}_{i_1,i_2,\cdots,i_{k+1}} = \mathcal{A}_{i_1,i_2,\cdots,i_k} \bigcup\{p_{i_{k+1}}\}$ and set $V_{k+1}^{i_1,\cdots,i_{k+1}}$ to be the set of computations in $\mathcal{F}_{k+1} = \mathcal{F}_k - \{Id_{i_k}\}$, which can be computed from $\mathcal{A}_{k+1}$. Note that Id_{i-k-1} is the Id of the last operation. Let $|V_k^{i_0,\cdots,i_k}| = \alpha_{i_0,\cdots,i_k}$. We choose one of the operations $\alpha_{i_0,\cdots,i_k}$ operations in $V_k^{i_0,\cdots,i_{k-1}}$ to continue.

The sets A_j contain the set of live variables at each step. To minimize the number of intermediate variables we can not afford to keep redundant values in this set. At each step before inserting a new value into $\mathcal{A}_{i_0,\cdots,i_j}$, we check if it contains any value which is not required in further computations in $\mathcal{F}_j$. All such redundant values are discarded from $\mathcal{A}_{i_0,\cdots,i_k}$.

We stop the procedure when for some $\bar{k}$, $V_{\bar{k}}^{i_0,\cdots,i_{\bar{k}}-1}$ becomes empty. If $\bar{k}$ is $n-k+1$, i.e. the total number of operations in $\mathcal{F}$, then the sequence of operations $\{Id_{i_1},\cdots,Id_{\bar{k}_1}\}$, is a valid sequence for $\mathcal{F}$.

After choosing a valid sequence, we check the sets

$$\mathcal{A}_{i_1,\cdots,i_j} - \mathcal{A}_0, \qquad\qquad 0 \le j \le n-k+1$$

If $\max_{0\le i\le n-k+1} |\mathcal{A}_i - \mathcal{A}_0| = \beta$, then for executing $\mathcal{F}$ by the obtained valid sequence the storage for β intermediate variables is necessary.

After obtaining one valid sequence, we backtrack to the last step where we had more choices for the next operation than the ones already undertaken. We choose a different operation from the corresponding set V_i there and proceed in a new path of computation. Following this method we obtain all valid sequences for $\mathcal{F}$ and find the one which requires the minimum number of intermediate variables. That valid sequence gives us the execution sequence of $\mathcal{F}$, which is the most memory efficient one.

Obviously, the method described above is an exhaustive search type. It looks for all possible path from a possible starting point (which may not be unique) for execution of $\mathcal{F}$ to the end. To find the minimum number of intermediate variables it looks for all possible paths from the beginning to end of $\mathcal{F}$. To bring down the running time we adopt the following four strategies:

1. **Neglecting the paths which requires same number of intermediate variables as the known one:** As we get the first valid sequence we count the number of intermediate variables required to be stored to execute $\mathcal{F}$ by that sequence and store it in a variable, say β. While looking for another path by backtracking, we check the size of the set of intermediate variables after each step. If the current size is equal to the value stored in β, then we need not proceed along this path further. It is not going to yield a more economical path. So we abandon this path and look for another one. If a particular valid sequence needs less than β intermediate values, then we replace the value of β by this new value.

2. **Avoiding the count of the number of intermediate variables at each step:** Counting the number of variables at each step of the algorithm is a time consuming operation. Suppose the value stored currently in β is t. While looking for a new path, we save time by not counting the number of minimum variable till the path is t operations long. Because, if less than t operations have been executed then the number of intermediate variables can not be more than t.

3. **Backtracking several steps at a time:** After finding a valid sequence, instead of backtracking one step (to the last step) at a time, we can go back until a step b such that $\max_{0\leq i\leq b}|\mathcal{A}_i - \mathcal{A}_0| = \beta - 1$. This will reduce the task of going to each level and hence will aid to efficiency. Note that this does not affect the optimality of the final result. That is because the paths which we are by passing need at least β intermediate variables.

4. **Using ordered sets in place of V_j's:** Before starting the first step, we scan the explicit formula and make a frequency table of all the inputs and intermediate variables. Against name of each variable it contains the number of times it has been used in the formula, i.e the number of times it appears in the right hand side of some equation in $\mathcal{F}_j$. We treat the sets V_j's as ordered sets and ordered according as priorities assignicned to these equations. The higher priority is assigned to those, in which the inputs variables have lower frequency. Each time we choose an equation for computation, we update the frequency table by reducing the frequencies of the involved input variables by 1.

   Here we describe how we assign priorities to the operations in V_j. Observe that any equation $r = p\ op\ q$ takes two inputs and computes an intermediate value. If the variable p and q are used again later i.e. their frequencies are greater than 1 before this operation, then we have to store all three variables. But if frequencies of p and q are one then we are required to store the result only. Thus, we can reduce the width of the set of live variables by 1. We assign highest priorities to such computations and put them at the beginning of the sets V_j's. If the frequency of one of p and q is 1, then we need not save that variable. We are required to store the other variable and the computed value and the thus the number of live variables does not increase. So we assign second highest priority to such equations and put them next in the set V_j. So are the equations which have a constant and a variable of frequency 1. Thirdly, squarings and doublings involve one variable only and produce a new variable. So a squaring or a doubling of a variable of frequency 1 also does not increase the number of live variables. We keep such type of operations next. Other equations of $\mathcal{F}_j$ are kept in such an order that the ones involving a variable of minimum frequency precedes others.

These optimization techniques for running time of the algorithm has paid high dividends. It is observed that an implementation using these techniques runs much faster than the one without them. These techniques however, do not guarantee that an implementation will terminate in reasonable time for any large explicit formula. An explicit formula may contain a huge number of equations. In that case the program may run for a considerable duration of time and the last value of β may be accepted as the good minimal value. The actual minimum may be lesser.

Let us put the above discussion in the form of an algorithm. In fact, we present two algorithms below, the first one only initializes and calls the second one. The second is a recursive one implementing the techniques described above.

Algorithm 7 (MinVar)

```
Input : The number of inputs to $\mathcal{F}$ and all the operations in $\mathcal{F}$.
Output : Minimum number of intermediate variables required to be stored if $\mathcal{F}$ is executed sequentially.
1. (Initialization) Read the number of inputs $k$ to $\mathcal{F}$;
2. Read all computation in $\mathcal{F}$ and store them in an array indexed by their Id's. Let their number be $N$;
3. Let $k = 0$, minvar = the number of equations in $\mathcal{F}$;
4. Call Algorithm ProcVar(k);
5. Output minVar;
```

Algorithm 8 ProcVar(k).

```
Input : The number of computations (k) of already carried out.
Output : Recursively computes the number of intermediate variables and outputs their minimum.
1. Let $Count = N - k$;
2. If $k >$ minVar
    2.1 Count the number of variables in the current path;
    2.2 If the number of variables in the current path = minVar then return; /* There is no need to consider this path */
3. If $Count \neq 0$
    3.1 Find the corresponding ordered set $V_k$ and $\alpha_k$;
    3.2 for $i = 1$ to $\alpha_k$;
        3.2.1 Process the $i$th computation in $V_k$;
        3.2.2 Update the data structures for $\mathcal{A}_{k+1}$;
        3.2.3 Call Algorithm ProcVar($k+1$);
4. minvar = the number of variables in the current path;
5. end
```

Here is an important observation about running of the algorithm.

- **Observation** At any point of time during the execution of the algorithm in the search of a valid sequence, suppose the operations $S_1 = \{i_1, i_2, \cdots, i_k\}$ have been chosen. Also suppose that at the end of this last step the number of live variables is t. Let $S_2 = \{j_1, j_2, \cdots, j_k\}$ be any permutation of $\{i_1, i_2, \cdots, i_k\}$. If S_2 is a valid order in which the operations in S_1 can also be performed, then at the end of last step of S_2 one will have t number of intermediate variables as well. An important application of this observation is that if we have two valid sequences of an explicit formula, which have same k operations at the beginning, may be in a different order and if we know that the first requires t intermediate variables at the kth step, then we need not count the number of intermediate variables for first k steps of the second. It is t at the end of kth step.

4.3.1 The Forward and Reverse Programs

As said earlier, each explicit formula in raw format was modified to be a sequence of binary operations. This is the pre-processing done to each of the raw formula under consideration. This pre-processed formula was given as input to a program embodying the methodology described in the last section, which calculated the minimum number of intermediate variables required for sequential execution of the explicit formula. When the program terminates it outputs exactly how many intermediate variables are required for an execution of the formula and the corresponding valid sequence. As our methodology is an exhaustive search kind with some running time optimization measures, for a long input file it may take substantial amount of time to run. In order to get the results within a reasonable time, another program was employed. We will refer to this later program as the *reverse* program and the former as *forward* program. The *reverse* program initially takes $k = 1$ and using the same logic as *forward* checks if the explicit formula can be executed with k temporary locations. If not it reports this fact and tries again with k replaced by $k+1$. When it gets an affirmative answer it outputs the corresponding value of k and the corresponding valid sequence. The *forward* program after obtaining a path requiring l intermediate variables looks for path needing less than l intermediate locations. If during the search process it comes across a path which also requires l locations, then *forward* abandons this path and look for a newer one. The *reverse* program uses the same logic, taking $k = 1, 2 \cdots$. We run both the programs with the same input file on two different machines. If either of *forward* or *reverse* terminates we get the result. Otherwise, if at some point of time *forward* reports the formulae can be executed with k variables and at the same time *reverse* reports it can not be done with less than $k-1$ intermediate locations, then also the conclusion follows.

The employment of *reverse* helped us to get to the conclusions quite early. Some of the explicit formula under consideration have more that 100 lines of three address codes (i.e. total number of arithmetic operations is more than 100). In spite of the speed up measures described above, there is no guarantee that *forward* will terminate. In fact, we ran *forward* program without any speed up measure on some longer inputs and found that it did not terminate in a week. Even after the optimization methods described above are employed, for some of the formulae given in [86] it did not terminate for three days. Of course it ran much faster. Surprisingly (because they have quite less number of operation), for the doubling formulae in new coordinates in even characteristic, in both cases $h_2 \neq 0$ and $h_2 = 0$, *forward* did not terminate. So, we took help of the *reverse* program to derive our conclusions. With the help of *reverse* we could get conclusion on any formula in less than two days.

4.4 Pre- and Post-Processing

We implemented the methodology describe above and found out the minimum number of registers required for a select set of explicit formula. Each formula was pre-processed in the manner described below:

4.4.1 Pre-processing

We call the different explicit formulae which appear in research papers to be in raw form. Our first step is to pre-process such raw formulae and convert into a form suitable for input to our register minimization program. This conversion of explicit formulae from one form to another is done using a pre-processing program. We verify the correctness of the two forms using Mathematica, a symbolic computation software. We first describe this verification method and then describe the pre-processing algorithm.

Each explicit formula consists of a set of input and output variables and a sequence of arithmetic operations involving temporary intermediate variables. The arithmetic operations are addition, subtraction and multiplication. Thus each output variable can be written as a multivariate polynomial in the input variables. This polynomial can be computed by successively eliminating the intermediate variables. We say that two explicit formulae with the same set of input and output variables are equivalent if the multivariate polynomials corresponding to the output variables are same for both the formulae. We use Mathematica to perform this verification. Being a symbolic computation software, Mathematica can efficiently construct the multivariate polynomial corresponding to the output variable of an explicit formula. There is, however, a problem in this method. If an explicit formula reuses a variable, then Mathematica falls into an infinite loop and fails to construct the multivariate polynomial. There is another pitfall when variables are reused. According to Garey-Johnson [48, page 272], the decision version of the problem when variables can be reused, is known to be NP-hard but not known to be in NP. Thus our first step is to eliminate variable reuse from raw formulae.

We say that a variable is reused if it occurs more than once on the left hand side.There are three different kinds of variable – input, output and temporary. If an input variable is reused, then we want the first occurrence to have the identifier of the input variable, while subsequent ones will have new identifiers. On the other hand, if an output variable is reused, the last occurrence must have the identifier of the output variable while the others will have new identifiers. Reuse of intermediate variables can be tackled either as an input or as an output variable – we tackle them as input variables. While eliminating reuse of input variables, we have to scan the formula from the start to the end and while eliminating reuse of output variables, we have to scan the formula from the end to the start. We do not provide further details, as they are fairly routine and technical in nature.

The second step is to convert the raw formula after elimination of variable reuse into a sequence of binary operations. For this we initially convert the formula to postfix form and then use a stack to obtain the sequence of binary operations. This step is also quite routine and is given in most data structure textbooks. However, we note that we attempt no optimization at this step. Compiler construction based techniques of using directed acyclic graphs (DAG's) to identify common minimal subexpressions could be could be applied at this point. Thus the minimum we report report are really minimum after conversion to three address codes. The actual minimum could be lesser, though we think this to be unlikely.

The third step is to perform a series of checks: no variable is reused, each output variable occurs once on the left side, each intermediate variable should occur exactly once on the left side and at least once on the right side. Finally, we also check if each input variable is used at least once. We do not consider the failure of this check to be a serious error, since some of the curve parameters are sometimes not used in some formulae. However, we do report this failure and if any of the other checks fail, we do not proceed with further processing of the formulae. The output of the third step is verified to be equivalent to the raw formula after eliminating variable reuse. Once this verification succeeds, we provide this as input to our register minimization algorithm.

4.4.2 Post-processing

As mentioned in the last section, the register minimization programs, both determine the minimum number of intermediate variables required in the implementation of the explicit formulae and also output the corresponding valid sequence. If number of inputs to a formula is α, minimum number of intermediate variables required is β and γ is the number of outputs of the formula, then it can surely be implemented with $\alpha + \beta + \gamma$ locations in memory. For optimal usage of memory, the registers occupied by input variables can be reused after last usage of the input. Also, in some situations one may not like reusing the locations storing the input variables as they may be required later. For example in scalar multiplication algorithms, whenever a HCADD is called one argument is the base point. So, if the locations storing the parameters of the base point are reused, one has to load them again into these registers when HCADD is invoked again by the scalar multiplication algorithm. Thus there may be some inputs such that the register containing them can not be reused. This leads to two cases,

1. All registers are reused.
2. Some selected registers containing some vital inputs are not reallocated.

In our investigation, we allowed reuse of all registers for HCDBL. For HCADD, the inputs are the parameters of two points (divisors) to be added. In our implementation we allowed reuse of the registers containing the parameters of the first divisor. We assumed that the second divisor is the base divisor and its contents should not be destroyed by reusing these registers. For mixed addition algorithm, the divisor which is in affine coordinates is the base point. We did not allow the reuse of the registers containing the point in affine coordinates. Also, the registers containing the curve parameters are never reused nor counted. These are the registers containing the 'vital' inputs.

For sake of generality, we also ran our programs allowing reuse of all the registers (except the ones containing the curve parameters) for all addition formulae. We found that the register usage level can be brought down by reusing all the registers. If in a device, the base point is stored in some permanent memory like some kind of ROM and transferring data from ROM to registers is fast enough, then reuse of all registers is preferable. However, if such data transfer takes significant time, then the time for an addition may go up significantly relative to doubling and the implementation risks being prone to timing attacks.

A register allocation program was implemented which converted the binary instructions of the valid sequences obtained from the minimum intermediate variables programs into the explicit register specified format (ERSF). A register containing a non-vital input is reallocated as soon as the variable it is containing is not required any more (is not a 'live' register).

The output of this program for an explicit formula is in the ready-to-implement form. We applied this methodology to several important formulae in ECC and HECC. For HECC, these outputs are the first explicit formula in ERSF. Hopefully, these formulae will be of importance for an implementation in software or in hardware.

4.5 Results

The addition (ECADD) and doubling (ECDBL) formulae in ECC have received much attention from the researcher community and the formulae are quite simpler in comparison to those in HECC. It has been reported by many researcher that the ECADD in Jacobian coordinates for most general curves over fields of odd characteristic needs 7 registers for implementation. To our knowledge, there is no result stating that it can not be executed in less than seven registers. This aroused our curiosity for testing these formulae in our methodology. We experimented with ECADD and ECDBL in Jacobian coordinates and found that ECADD and ECDBL can be implemented with no less than 7 and 6 registers respectively. Both *forward* and *reverse* program reported this fact. As we intended this work to focus on HECC, we did not pay attention to other ECC algorithms.

We applied our methodology to many formula in HECC. First of all, we applied our methodology to Lange's formulae in new co-ordinates [86]. These formulae are the most efficient ones for general hyperelliptic curves of genus 2. All these formulae are inversion-free. However, the cost of avoiding the inversions is more than an inversion in binary fields. Hence for an implementation over binary fields, affine arithmetic still looks quite attractive. So we used our methodology to calculate the minimum number of registers required for implementing HCDBL and HCADD in affine coordinates also. We used the formulae presented in [84] which are the most efficient ones in affine coordinates for general curves of genus 2. Pelzl et al [133], have proposed a very efficient HCDBL formula for a special class of curves. Over such curves the formulae presented by them are quite lucrative for implementation. So we investigated the memory requirement of the HCADD and HCDBL formulae presented in [133] also.

Note that HCADD and HCDBL for genus 2 curves have many special cases. The most general and also the most frequent case is the one in which the divisor(s) are of full weight, i.e. if $D = (u, v)$ is the divisor, then $deg(u) = 2, deg(v) = 1$. In the current work we concentrate on the most general and the frequent case only. The same methodology can be applied to other special cases as well. Also, the cost of various operations we have given in the Table 4.1 below does not corroborate with the costs provided in the corresponding papers. That is because authors generally avoid counting the multiplication and squaring of/with curve constants. In some formulae such operations occur in significant numbers. For example in even characteristic doubling formula ($h_2 \neq 0$), there are 21 such multiplications/squarings. Many of the curve constants can be made 0 or 1. Also in implementations, these are generally chosen to be 0 or 1. For sake of generality, we have accounted for these multiplications and squarings as well.

We use the following naming convention for the name of various algorithms. The formulae presented in [84] and in [133] are in affine coordinates. We use a superscript $\mathcal{A}$ for them,

Table 4.1: Register Requirement for Various Explicit Formulae

Algorithm	Proposed in	Char($\mathbf{F}_q$)	Cost	#Registers
HCADD$^{\mathcal{A}}$	[84]	All	$1[i] + 21[m] + 3[s] + 44[a]$	15
HCDBL$^{\mathcal{A}}$	[84]	All	$1[i] + 22[m] + 5[s] + 56[a]$	11
HCADD$^{\mathcal{N}o}$	[86]	Odd	$49[m] + 7[s] + 34[a]$	23
mHCADD$^{\mathcal{N}o}$	[86]	Odd	$36[m] + 5[s] + 35[a]$	16
HCDBL$^{\mathcal{N}o}$	[86]	Odd	$36[m] + 7[s] + 41[a]$	20
HCADD$^{\mathcal{N}e}_{h_2 \neq 0}$	[86]	Even	$52[m] + 4[s] + 35[a]$	27
mHCADD$^{\mathcal{N}e}_{h_2 \neq 0}$	[86]	Even	$42[m] + 5[s] + 34[a]$	17
HCDBL$^{\mathcal{N}e}_{h_2 \neq 0}$	[86]	Even	$54[m] + 8[s] + 29[a]$	20
HCADD$^{\mathcal{N}e}_{h_2 = 0}$	[86]	Even	$47[m] + 6[s] + 37[a]$	27
mHCADD$^{\mathcal{N}e}_{h_2 = 0}$	[86]	Even	$37[m] + 6[s] + 30[a]$	22
HCDBL$^{\mathcal{N}e}_{h_2 = 0}$	[86]	Even	$40[m] + 6[s] + 27[a]$	16
HCADD$^{\mathcal{A}}$	[133]	Even	$1[i] + 21[m] + 3[s] + 30[a]$	15
HCDBL$^{\mathcal{A}}$	[133]	Even	$1[i] + 9[m] + 6[s] + 24[a]$	10

e.g. HCADD$^{\mathcal{A}}$ and HCDBL$^{\mathcal{A}}$. The formulas in [86] are in Lange's new coordinates. For the formulas in these new coordinates over fields of even characteristic we will use the superscript $\mathcal{N}e$ and for those over fields of odd characteristic we will use superscript $\mathcal{N}o$. Divisor addition algorithms in mixed coordinates will be denoted by a suffix 'm' e.g mHCADD$^{\mathcal{N}o}$.

We summarize our findings in Table 4.1. In Appendix B, we present all these formulae in Explicit Register Specified Format.

Observing Table 4.1, one can conclude that the formulae presented in [133] are the best for an implementation over binary fields. However, these are based on a special class of curves. An implementation of the formulae in [84], which are more general in nature, needs only one more register in doubling. In the scalar multiplication algorithm, sets of explicit formulae will require 15 registers each. The former has no curve parameter which is not zero or 1. The later requires storing of at most 6 curve parameters. The computation can be made secure against simple power attacks by using Coron's dummy addition method without any extra registers. Two popular countermeasures against DPA are Coron's point randomization [36] and Joye-Tymen's Curve randomization [69]. Both have been extended to hyperelliptic curves by Avanzi [9]. In affine representation point randomization countermeasure can not be implemented. As there is no curve constant involved in the explicit formulae in [133], curve randomization can not be applied. Hence for a secure implementation the formulae of [84] is suitable. It will require at most 21 registers (at most 6 registers for curve parameters).

Table 4.2: Register Requirement: Register Reuse vs No Reuse

Algorithm	Proposed in	#Registers (all reused)	#Registers (selective reuse)
HCADD$^{\mathcal{A}}$	[84]	13	15
HCADD$^{\mathcal{N}o}$	[86]	19	23
mHCADD$^{\mathcal{N}o}$	[86]	19	20
HCADD$^{\mathcal{N}e}_{h_2\neq 0}$	[86]	23	27
mHCADD$^{\mathcal{N}e}_{h_2\neq 0}$	[86]	18	20
HCADD$^{\mathcal{N}e}_{h_2=0}$	[86]	23	27
mHCADD$^{\mathcal{N}e}_{h_2=0}$	[86]	19	22
HCADD$^{\mathcal{A}}$	[133]	14	15

For an implementation over fields of odd characteristic, clearly the formulae in [86] are the most suitable. In this representation mixed addition requires 20 registers and doubling requires 16. Besides 2 curve parameters are to be stored. So the scalar multiplication can be computed in 22 registers. For a secure implementation, Coron's dummy addition method and point randomization can be used without increasing the number of registers.

For addition formulae we do not reuse the registers containing the parameters of the base point. As stated above we also experimented reusing all registers. We provide our results in Table 4.2. It can be seen that number of registers goes down significantly if all registers are reused.

4.6 Possible Improvements and Conclusion

Although our register minimization technique produces minimum number of registers required for any explicit formula, its output depends upon the nature of the input file. The input file is generally a sequence of three address codes. There is a vast literature in compiler construction studies on efficient methods for converting an arithmetic formula into three address codes. Our parsing program which converted the explicit formulae into three address codes may not be the optimal one. Therefore there is still some scope for improvement. Besides, the explicit formulae used for finding the minimum register requirements are best known algorithms. In future, researchers may come out with more efficient formulae. Thus the minimum register requirements reported in the current work may not be the best for hyperelliptic curve cryptosystems.

In a memory constrained small device, we may sacrifice a small amount efficiency for efficient memory usage. That is, instead of keeping a memory location occupied with a computed value which will be required much later, we can free the corresponding location to store other intermediate values and recompute the the earlier value once again exactly when it is required. For example, suppose in an algorithm at step k a value $x = y \; op \; z$ is computed and used at Steps $k+1$ and $k+k_1$, where k_1 is not small. Also, suppose that at Step $k+k_1$, both y and z are alive. Then if memory is a concern, instead of storing the value of x for k_1 steps, one may prefer to free that memory at Step $k+2$ and recompute x just before the Step $k+k_1$. Thus one saves a memory location for some steps by recomputing one operation. It can be a cheap operation like field addition. In this way one can trade-off memory for some extra operations. In the current work, we have not gone for such optimizations. In an implementation on a small device, this kind of optimization can lead to better utilization of memory.

Chapter 5

Reducing the Number of Inversions in Affine Coordinates

In this chapter we propose a scalar multiplication algorithm for elliptic and hyperelliptic curve cryptosystems, which uses affine arithmetic and is resistant against simple power attacks. Also, using a modification of known techniques the algorithm can be made immune against differential power attacks. The algorithm uses Montgomery's trick and a precomputed table consisting of multiples of the base point. Consequently, the algorithm is useful in a scenario where the base point is fixed, like Elgamal encryption or signature generation. Under such circumstances, for hyperelliptic curves, the algorithm compares favourably with other known algorithms over all fields. For elliptic curves, under similar circumstances, the algorithm performs better than other algorithms over prime fields. The increase in speed is due to a proper application of Montgomery's trick to efficiently perform the simultaneous inversion of several field elements.

5.1 Introduction

In this paper, we will use the term point to mean both a point on an elliptic curve and a divisor in the Jacobian of a hyperelliptic curve. The most important and computationally costly operation in (H)ECC is the scalar multiplication. Scalar multiplication is the operation of multiplying a point X with a scalar (an integer) m i.e. computing mX.

In this chapter, we present a new algorithm for computing the scalar multiplication for both elliptic and hyperelliptic curve cryptosystems. Our approach uses arithmetic with inversion and performs better than algorithms using inversion-free arithmetic in prime fields, where the cost of inversion is much higher than in binary fields. Moreover, the proposed algorithm is resistant against side-channel attacks.

The efficiency of the algorithm is derived from the fact that, while computing the scalar multiplication, instead of scanning one bit at a time, several bits can be scanned from different locations in the binary representation of the multiplier and the point additions and doublings can be done simultaneously. As each of the additions and doublings involve one inversion, all these inversions can be computed simultaneously by Montgomery's trick (see Section 5.2.2) with only one inversion and some extra multiplications. The partial results so obtained are added by another point addition algorithm, TreeADD, which computes addition of several points in a tree structure. The partial sums at the nodes of a particular level of the tree are computed together and the involved inversions are computed simultaneously by Montgomery's trick. This yields a very efficient scalar multiplication algorithm.

Our algorithm uses a precomputed table. So, it is useful in applications where the base point is fixed, like Elgamal encryption and signature generation etc. Also, the use of Montgomery's trick requires storage of several points which increases the memory requirement. In Section 5.5, it can be seen that for HECC over prime fields, when the base point is fixed, the algorithm is 77% faster than DPA resistant version of Coron's dummy addition method. Over binary fields the speed enhancement is about 28.1%. The performance is lower due to the fact that inversion is cheaper over such fields. For ECC over fields of characteristic > 3, under similar assumptions, the performance of our algorithm compares favourably against all SCA-resistant algorithms. The speed up is around 10% in the best scenario (window-size $= 5$).

5.2 Preliminaries

In this section we will briefly review various aspects of hyperelliptic/elliptic curve cryptosystems ((H)ECC), which are relevant to the material of this chapter. Most of the material of this section has been presented in Chapter 2. In this section we recollect them for sake of completeness.

5.2.1 Cost of Field Operations

Let $[i], [m]$ and $[s]$ denote the amount of time required to compute an inversion, a multiplication and a squaring respectively in the underlying field. We will use the notation $\mathbf{i}$ to denote the ratio $[i]/[m]$, which represents the relative cost of an inversion compared to a multiplication. The value of $\mathbf{i}$ depends on the choice of the underlying field. For binary fields this value has been reported to be between 3 and 10 and for prime fields it is somewhere between 30 and 40. In prime fields the cost of a squaring is known to be somewhat less than the cost of a multiplication. *For simplicity, in the current work we assume [m] = [s].*

5.2.2 Computing Inverses Simultaneously

Our scalar multiplication algorithm derives its efficiency from the fact that inversions of several field elements can be computed simultaneously by one inversion and some extra multiplications. One well known technique for doing this is Montgomery's trick [151], [116] which works as follows. Let $x_1, \cdots, x_n$ be the elements to be inverted. The algorithm first computes $a_1 = x_1, a_2 = x_1x_2, \cdots, a_n = x_1 \cdots x_n$, by $(n-1)$ multiplications. Then it inverts a_n. Now, x_n^{-1} is computed by multiplying a_n^{-1} by a_{n-1}. Also, $a_{n-1}^{-1} = a_n^{-1}x_n$ and x_{n-1}^{-1} is $a_{n-1}^{-1}a_{n-2}$. Similarly, it computes inverse of other elements. It is not difficult to see that the algorithm uses only $3(n-1)$ multiplications and one inversion. We will denote the cost of computing the inverses of n field elements by $\mathcal{I}(n)$. Montgomery's trick shows that we can take $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$.

5.2.3 Scalar Multiplication Methods for HECC

Many scalar multiplication algorithms immunized against SCA have been proposed for ECC. We discuss some specific methods for genus 2 HECC using Lange's formula (see Table 5.1) along with their costs below.

(a) The usual add and double algorithm: For an n-bit multiplier such an algorithm requires n doublings and $n/2$ additions on the average (though this computation is not SCA resistant). So cost of computing the scalar multiplication using [85] is $n \times 46[m] + (n/2) \times 51[m] = 71.5n[m]$. Using affine arithmetic [84] the cost of n doublings and $n/2$ additions is $n \times (1[i] + 22[m] + 5[s]) + (n/2) \times (1[i] + 22[m] + 3[s]) \approx ((3/2)\mathbf{i} + 39.5)n[m]$. However, this computation is not SCA resistant.

(b) To make the computation SPA resistant we can resort to Coron's countermeasure for ECC. That is we can make some dummy additions if the corresponding bit in the binary representation is 0. In this case, we have to compute n additions and n doublings. Cost of the computation in inversion-free arithmetic will be $51n[m] + 46n[m] = 97n[m]$. This computation is costlier than that of (a), but is SPA resistant. But, again in binary fields the affine arithmetic will be preferable. In binary fields, the cost will be $n(1[i] + 22[m] + 5[s]) + n(1[i] + 22[m] + 3[s]) \approx (2\mathbf{i} + 52)n[m]$. It can be made resistant against DPA using the methods described above.

(c) We can encapsulate the addition and doubling formula of affine co-ordinates to obtain a more efficient formula. Suppose, we wish to compute an addition an doubling simultaneously.

Table 5.1: Complexity of different algorithms for HECC.

ALGORITHM	**i**	COMPLEXITY
(a)	30	$71.5n[m]$ (avg case)
	8	$51.5n[m]$
(b)	30	$97n[m]$
	8	$68n[m]$
(c)	30	$(84n + 57)[m]$
	8	$(63n + 35)[m]$

In affine co-ordinates, both will involve one inversion. Instead of computing two inversions, we can compute them by Montgomery's trick with 1 inversion and 3 multiplications. So cost of one addition and doubling is $1[i] + 3[m] + 52[m] \approx (\mathbf{i} + 55)[m]$. We can now use this algorithm in Montgomery's ladder type scalar multiplication algorithm to compute the scalar multiplication. The method involves one doubling at the outset. Amount of computations involved in computing the scalar multiplication would be $((\mathbf{i} + 55)n + \mathbf{i} + 27)[m]$. Again the computation will be SPA resistant. It can also be made resistant against DPA.

We produce the summary of this discussion in Table 5.1. In the table we have considered two specific values of **i**, 8 and 30. It is clear from the Table that, (c) is better than (b). Although, the average case complexity of (a) is better than (c), the former is not SCA resistant. Note that both (b) and (c) can be made DPA resistant (at an additional cost) by using Avanzi's countermeasure as discussed in Section 5.4.2.

5.3 New Algorithm for Scalar Multiplication

Before describing the proposed algorithm, let us have a close look at the addition and doubling algorithms in affine co-ordinates.

5.3.1 Addition and Doubling in Affine Co-ordinates

Let us consider the addition (HCADD) and doubling (HCDBL) algorithms for HECC in [84] in the most general and frequent case. These can be divided into three parts. In part one, some multiplications and squarings of the underlying field elements are carried out. In part two, a field element, generated in part one is inverted. The inverse so obtained in part two is used in part three along with some more multiplications and squarings of field elements. The output of part three provides the required divisor. See Figure 5.1.

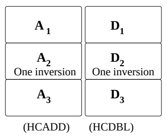

Figure 5.1: HCADD and HCDBL algorithms proposed in [84]

Let us name the modules of these algorithms as $\mathbf{A_1}, \mathbf{A_2}, \mathbf{A_3}$ (parts of addition algorithms) and $\mathbf{D_1}, \mathbf{D_2}, \mathbf{D_3}$ (parts of doubling algorithms). In each of $\mathbf{A_2}$ and $\mathbf{D_2}$, we compute only one inverse. Let the number of multiplications and squarings required in $\mathbf{A_1}$, $\mathbf{A_3}$, $\mathbf{D_1}$ and $\mathbf{D_3}$ be $\mathbf{a_1}$, $\mathbf{a_3}$, $\mathbf{d_1}$ and $\mathbf{d_3}$ respectively. We will use the notation $\mathbf{a}$ for $\mathbf{a_1} + \mathbf{a_3}$ and $\mathbf{d}$ for $\mathbf{d_1} + \mathbf{d_3}$. By $\mathbf{A_1}(D_1, D_2)$, we will mean the field element α created in module $\mathbf{A_1}$ of the addition algorithm and which is inverted in $\mathbf{A_2}$. Similarly, by $\mathbf{D_1}(D_1)$, we will mean the field element β created in module $\mathbf{D_1}$ of the doubling algorithm and which is inverted in module $\mathbf{D_2}$. By $\mathbf{A_3}(D_1, D_2, \alpha^{-1})$ (resp. $\mathbf{D_3}(D_1, \beta^{-1})$) we mean the divisor produced by the module $\mathbf{A_3}$ (resp. $\mathbf{D_3}$) as sum of D_1 and D_2 (resp. D_1).

The same can be said about addition and doubling algorithms for ECC in affine coordinates. In ECC, the values of $\mathbf{a_1}, \mathbf{a_3}, \mathbf{a}$ etc will be much smaller.

5.3.2 The Proposed Algorithm

Let $w \geq 2$ be a positive integer. We express m in the base 2^w. Let $m = c_0 + c_1 2^w + \cdots + c_{t-1}2^{w(t-1)}$, where each $c_j \in \{0, \ldots, 2^w - 1\}$. Then $mD = c_0 D + c_1 2^w D + \cdots + c_{t-1}2^{w(t-1)}D$. For all $j, 0 \leq j \leq t-1$ we precompute $2^{jw}D$ and store it in a table $T[\,]$. Thus $T[j] = 2^{jw}D$ for $0 \leq j \leq t-1$. This table is used to simultaneously compute $c_0 D, c_1 2^w D, \cdots, c_{t-1}2^{w(t-1)}D$ using the right-to-left binary method. (A similar algorithm can also be developed using the left-to-right binary method.) Finally we add them to obtain mD.

Let the n-bit representation of m be $m_{n-1}\ldots m_0$. Note that $t = \lceil (n/w) \rceil$. We express c_j in binary, i.e., we write $c_j = c_j^0 + c_j^1 2 + \cdots + c_j^{w-1}2^{w-1}$, where $c_j^i = m_{wj+i}$. We require $2t+1$ point type variables $R_0, \ldots, R_{t-1}$ and $Q_0, Q_1, \ldots, Q_t$. The variable R_j is initialized to $2^{jw}D$. Starting from the least significant bit of c_i, we scan a bit in each iteration. If the scanned bit is 1, then we add R_j to Q_{j+1}; if the scanned bit is 0, then we add R_j to Q_0; in either case we double R_j. After w iterations, Q_{j+1} is $c_j 2^{wj}D$. For $0 \leq j \leq t-1$, we compute all the expressions $c_j 2^{wj}D$ simultaneously. Each of the additions and doublings in each iteration will involve one inversion. While doing these additions and doublings, we carry out all the inversions simultaneously by Montgomery's trick. This yields an efficient algorithm for scalar multiplication. Our algorithm calls a routine INVERT, which simultaneously inverts a number of field elements using Montgomery's trick. Recall that by $\mathcal{I}(n)$ we denote the cost of inversion of n elements using INVERT and $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$.

Algorithm 9 (EFF-SCLR-MULT)

Input: $m, t, c_0, c_1, \cdots, c_{t-1}, D$.
Output: mD.

```
1.  For $j = 0$ to $t-1$ {$R_j = T[j]$;
2.      If $c_j^0 = 0$ then $b = 0, t_b = j+1$
        else $b = j+1, t_b = 0$, $Q_b = R_j$; }
2.  For $j = 0$ to $t-1$ let $\beta_j = \mathbf{D_1}(R_j)$;
3.  Let $(\beta_0^{-1}, \cdots, \beta_{t-1}^{-1})$ = INVERT$(\beta_0, \cdots, \beta_{t-1})$
4.  For $j = 0$ to $t-1$ let $R_j = \mathbf{D_3}(R_j, \beta_j^{-1})$;
5.  For $i = 1$ to $w-2$
6.      For $j = 0$ to $t-1$ { $\alpha_j = \mathbf{A_1}(R_j, Q_{j+1})$; $\beta_j = \mathbf{D_1}(R_j)$;}
9.      Let $(\alpha_0^{-1}, \cdots, \alpha_{t-1}^{-1}, \beta_0^{-1}, \cdots, \beta_{t-1}^{-1},)$ = INVERT$(\alpha_0 \cdots \alpha_{t-1}, \beta_0, \cdots, \beta_{t-1})$
10.     For $j = 0$ to $t-1$ {$Q_{c_j^i(j+1)} = \mathbf{A_3}(R_j, Q_{j+1}, \alpha_j^{-1})$; $R_j = \mathbf{D_3}(R_j, \beta_j^{-1})$;}
13. end do.
14. For $j = 0$ to $t-1$ let $\alpha_j = \mathbf{A_1}(R_j, Q_{j+1})$;
15. Let $(\alpha_0^{-1}, \cdots, \alpha_{t-1}^{-1})$ = INVERT$(\alpha_0, \cdots, \alpha_{t-1})$
```

```
16. For $j = 0$ to $t-1$ let $Q_{c_j^{w-1}(j+1)} = \mathbf{A_3}(R_j, Q_{j+1}, \alpha_j^{-1})$;
17. Let $RES$ = TreeADD$(Q_1, \cdots, Q_t)$
18. Return $(RES)$
```

Proposition 28 *The cost of the above algorithm is* $[t(w-1)(\mathbf{a}+\mathbf{d})+t\mathbf{a}][m]+(w-1)\mathcal{I}(2t)+\mathcal{I}(t)+$ *cost(TreeADD), where cost(TreeADD) is the cost of the TreeADD algorithm invoked by the algorithm.*

The algorithm TreeADD adds a number of points efficiently. It uses a tree structure for computation. Suppose $D_0, D_1, \cdots, D_{k-1}$ are the input points. For simplicity, assume $k = 2^r$. Imagine a tree of depth r with the input points at the leaf nodes. We pairwise add the points at the nodes with a common parent and put the sum at the parent node of each pair. There are 2^{r-1} nodes at level $r-1$ and to get the points at these nodes we have to perform 2^{r-1} additions. Note that, each of these additions needs one inversion. Instead of computing 2^{r-1} inversions separately, we can compute them with 1 inversion and $(3 \times 2^{r-1} - 1)$ multiplications using Montgomery's algorithm. This process is carried out at each level to the root. The root then contains the sum of all the points.

Algorithm 10 (TreeADD)

```
Input: $D_0, \cdots, D_{2^k-1}$
Output: $D_0 + D_1 + \cdots + D_{2^k-1}$
1. For $i = 0$ to $2^k - 1$ let $D_i^{(0)} = D_i$.
2. For $j = 1$ to $k$
3.        let $(D_0^{(j)}, D_1^{(j)}, \cdots, D_{2^{k-j}-1}^{(j)})$ = ADD$(D_0^{(j-1)}, D_1^{(j-1)}, \cdots, D_{2^{k-j+1}-1}^{(j-1)})$
4. Return $(D_0^{(k)})$
```

TreeADD invokes the algorithm ADD, which takes as input $2k$ points, $D_0, D_1, \cdots, D_{2k-1}$ and returns $D_0 + D_1, D_2 + D_3, \cdots, D_{2k-2} + D_{2k-1}$. ADD computes k additions at one invocation. Hence, the inversions at $\mathbf{A_2}$ step of all these additions can be done simultaneously using the Montgomery's algorithm.

Algorithm 11 (ADD)

```
Input: $D_0, \cdots, D_{2k-1}$.
Output: $D_0 + D_1, \cdots, D_{2k-2} + D_{2k-1}$.
```

```
1. For $i = 0$ to $k-1$, let $\alpha_i = \mathbf{A_1}(D_{2i}, D_{2i+1})$.
2. Let $(\alpha_0^{-1}, \alpha_1^{-1}, \cdots, \alpha_{k-1}^{-1})$ = INVERT($\alpha_0, \alpha_1, \cdots, \alpha_{k-1}$).
3. For $i = 0$ to $k-1$ let $E_i = \mathbf{A_3}(D_{2i}, D_{2i+1}, \alpha_i^{-1})$.
4. Return $(E_0, E_1, \cdots, E_{k-1})$.
```

Proposition 29 *ADD takes $2^r\mathbf{a}[m] + \mathcal{I}(2^r)$ computations to compute the $k = 2^r$ sums of $2k = 2^{r+1}$ input points.*

With $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$ (see Subsection 5.2.2), the cost of ADD becomes, $((2^r\mathbf{a} + 3(2^r - 1)[m] + 1[i]$.

Now we can compute the complexity of the algorithm TreeADD. TreeADD repeatedly calls the algorithm ADD, first with 2^r points, then with 2^{r-1} points and so on. Let the cost of ADD with k input points be $[kA]$. Then, cost of TreeADD with 2^r points is $[2^rA] + [2^{r-1}A] + \cdots + [1A]$. By Proposition 29, $[2^iA] = 2^{i-1}\mathbf{a}[m] + \mathcal{I}(2^{i-1})$. Hence computational cost of TreeADD is $\sum_{i=1}^{r-1}[2^iA] = \sum_{i=1}^{r-1} 2^{i-1}\mathbf{a}[m] + \mathcal{I}(2^{i-1}) = ((2^r - 1)\mathbf{a}[m] + \sum_{i=1}^{r-1}\mathcal{I}(2^{i-1})$. With $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$, we have:

Proposition 30 *TreeADD takes $(2^r - 1)\mathbf{a}[m] + \sum_{i=1}^{r-1}\mathcal{I}(2^i) = [(2^r - 1)\mathbf{a} + 3 \times 2^r - 3r - 3][m] + r[i]$ computations to compute the sum of 2^r input points.*

We now compute the complexity of the algorithm EFF-SCLR-MULT. In the Steps 2–4 we double t points, inverting t elements by INVERT. In each iteration of the loop in Steps 5–13, we add t points and double t points. So in each iteration we invoke INVERT with $2t$ field elements. In the Steps 14–16 we add t points, inverting t elements by INVERT. Finally in Step 17 the TreeADD algorithm is invoked.

Proposition 31 *EFF-SCLR-MULT takes $(w + r)[i] + [2^r(w-1)(\mathbf{a} + \mathbf{d} + 6) - 3w + (2^r - 1)\mathbf{a} + 3 \times 2^r - 3r - 3][m]$ computations to compute the scalar multiplication mD where, m is an n-bit integer, w is the window size and $t = \lceil n/w \rceil = 2^r$.*

The algorithm uses a table which must be precomputed. Online computation will be very costly. The table will store t points. An elliptic curve point is an ordered pair of field elements. Each field element is of 160 bits. So a point occupies 320 bits of memory. Similarly, a divisor of a hyperelliptic curve of genus 2 is a 4-tuple of field elements, where each field element is of around 80 bits. So, a divisor also occupies almost the same amount of memory. The algorithm needs to store t points means, it requires about $320t$ bits of storage.

5.4 Resistance Against SCA

In this section we discuss the resistance of our algorithm to side-channel attacks and the cost involved in achieving such resistance.

5.4.1 Resistance Against SPA

Algorithm EFF-SCLR-MULT is resistant against simple power attacks, the reason being the following. At each iteration in steps 2 to 10, we are scanning t bits from the binary representation of m and computing some additions and doublings. The numbers of additions and doublings are fixed and independent of the actual bit pattern scanned. Similarly in steps 12 to 17, the same number of additions are being computed irrespective of the actual bits scanned. Hence we conclude that the computations are resistant against SPA.

5.4.2 Resistance Against DPA

We discuss a method for making the algorithm resistant against DPA. Recall that we use a look-up table $T[\,]$ of t points. The steps for the counter-measure for both ECC and HECC are as follows.

1. Choose a random nonzero field element z.
2. Compute the relevant powers of z. (see Subsection 2.10)
3. Transform the curve parameters.
4. Transform each of the t points of $T[\,]$.
5. Perform scalar multiplication using Algorithm EFF-SCLR-MULT.
6. Transform the result back to the original curve.

The specific transformations are different for ECC and HECC. For ECC we use Joye-Tymen transformation while for HECC we use Avanzi's transformation. Accordingly the costs are also different. From the discussion in Section 2.10 we get the following costs.

ECC **:** $1[i] + 4[s] + (4 + 2t)[m] \approx 1[i] + (8 + 2t)[m]$ assuming $[m] = [s]$.
HECC **:** $1[i] + (23 + 4t)[m]$ $(1[i] + (20 + 4t)[m]$ for odd characteristic).

5.5 Results and Comparison

In this section, we present some results of our algorithm and compare it with other algorithms. We do it separately for HECC of genus 2 and ECC.

Table 5.2: HECC: Comparison of the number of multiplications for different values of the number of bits n required to represent the scalar multiplier m.

Parameters			$\mathbf{i} = 8$				$\mathbf{i} = 30$			
n	w	t	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
160	5	32	8240	10896	10129	8501	11440	15539	13665	8743
160	10	16	8240	10896	10129	8937	11440	15539	13665	9267
160	20	8	8240	10896	10129	9190	11440	15539	13665	9718
160	40	4	8240	10896	10129	9389	11440	15539	13665	10335
160	80	2	8240	10896	10129	9690	11440	15539	13665	11440

5.5.1 HECC

We compare the performance of Algorithm EFF-SCLR-MULT to the algorithms (a), (b) and (c) described in Section 5.2.3. Table 5.2 displays these calculations. In Table 5.2, columns (a)-(c) refer to the algorithms listed in rows (a)-(c) of Table 5.1. Cost of DPA resistance has been added to the cost of (b) and (c). Column (d) stands for our algorithm. The column n stands for the bit size of the multiplier m. The parameter w stands for the window size. The complexity of the algorithms (a)-(c) do not vary with w as they are not window-based. The parameter t stands for the size of the look up table and is equal to the number of points required to be stored. One can easily observe that, for certain window sizes over prime fields, the proposed algorithm (d) is better than even average case complexity of double-and-add (column (a)), which is not SPA resistant. In the best scenario, the new algorithm achieves a speed-up of around 77 percent if $\mathbf{i} = 30$ over usual SPA resistant double and always add approach (b). Over binary fields, the performance enhancement is lower, only 28.1%, due to the fact that $\mathbf{i}$ is lower. For other values of $\mathbf{i}$ similar comparisons can be made. We have observed that as $\mathbf{i}$ increases from 30 to 40, the cost of (b) goes up by around 10%, whereas the cost of our algorithm goes up by about 1% only.

5.5.2 Efficiency for elliptic curves

The algorithm EFF-SCLR-MULT can also be used for ECC over prime fields. This is because ECADD and ECDBL algorithms in affine co-ordinates have a structure similar to that of Figure 1. For 160 bits scalar multiplier, the amount of computation required by the algorithm to compute the scalar multiplication is shown in Table 5.3. To compare the performance of the algorithm with other algorithms proposed in the literature we show Table 5.4 which is taken from [66]. It shows efficiency of some other SCA resistant methods. Note that

Table 5.3: ECC: Number of multiplications required by EFF-SCLR-MULT assuming $\mathbf{i} = 30$.

n	w	t	Complexity
160	5	32	2222[m]
160	10	16	2410[m]
160	20	8	2693[m]
160	40	4	3226[m]

Table 5.4: ECC: Number of multiplications required by previous algorithms under the assumptions $[\mathbf{i}] = 30$ and $[m] = [s]$.

Method	(160-bit ECC)
Coron's dummy addition	3375[m]
Coron's dummy addition with $a = -3$	3057[m]
Improved Moller with $w = 2$	3220[m]
Improved Moller with $w = 2$ and $a = -3$	3064[m]
Improved Moller with $w = 3$	2543[m]
Improved Moller with $w = 3$ and $a = -3$	2429[m]
Improved Izu-Takagi	2758[m]
Improved Izu-Takagi with $a = -3$	2439[m]

the table does not exactly matches with the table presented in [66], as we have not taken additions into account and have taken $[s] = [m]$. Table 5.4 shows that for efficient and secure computation of scalar multiplication, Improved Moller's method with window size 3 and $a = -3$ is the best. It takes $2429[m]$ computations. Our algorithm takes $2222[m]$ in the best situation, when the window size is 5, nearly 10% performance enhancement.

5.5.3 Memory Requirement

The parameter t in Table 5.2 determines the size of the look-up table. It is equal to the number of points to be stored in the look-up table. It is natural that the efficiency of the algorithm goes up as we invert more and more elements together. If a window size of 5 is chosen then the table size will be 32. In hyperelliptic curve cryptosystem with reasonable security, a point size is around 320 bits. So, the table will occupy around 1.2 kilobyte of memory.

Additionally Algorithm EFF-SCLR-MULT requires some more intermediate points and field elements. The calculation for these is as follows.

- $2t + 1$ intermediate points including one dummy point (Q_0) for Coron's trick.
- $2t$ field elements $(\alpha_0, \ldots, \alpha_{t-1})$ and $(\beta_0, \ldots, \beta_{t-1})$ for applying Montgomery's trick.

This memory requirement might be costly for memory constrained applications (as in smart card applications). In such situations our algorithm cannot be used. However, we note that in situations where the amount of memory is not a constraint (as in desktops), our algorithm provides a speed-up over the known algorithms (for fixed base point).

5.6 Conclusion

In this chapter, we presented an algorithm for scalar multiplication for (H)ECC, which is SCA resistant. The algorithm uses Montgomery's trick and a precomputed table. So it is useful for applications where the base point is fixed. The memory requirement for the algorithm is somewhat higher, which makes it unsuitable for memory constrained devices. For applications with a fixed base point and over prime fields, the algorithm performs better than existing SCA resistant algorithms for both elliptic and hyperelliptic curves. Additionally, for hyperelliptic curves, it enhances performance over binary fields as well.

Chapter 6

Reducing the Number of Inversions in Affine Coordinates: Parallel Approach

Montgomery's trick is a well known technique for performing simultaneous inversions of several field elements. However, this technique is a strictly sequential algorithm. Here we introduce two parallel algorithm for performing simultaneous inversions of several finite field elements. The algorithms use binary tree structures for computation. The first one can perform inversions of 2^r elements using $3 \times 2^{r-1}$ multipliers in $(r+1)$ multiplication rounds and one inversion round. The second one can simultaneously invert 2^r elements in $2r$ multiplication rounds and one inversion round. We also describe how to modify these algorithms when less number of multipliers are available. These parallel algorithms are used to obtain a new parallel algorithm for elliptic curve scalar multiplication using a fixed base point. The scalar multiplication algorithm is resistant against simple power analysis and can be implemented with different number of multipliers $(2, 4, 8, \ldots)$. Our algorithm compares favourably against many previously known parallel algorithms.

6.1 Introduction

Finite fields play an important part in coding and cryptography. The basic operations on finite fields are addition, multiplication and inversion. Of these, addition is the cheapest and inversion is the costliest. The cost of an inversion can be as high as thirty (or more) times the cost of a multiplication for a finite field of cardinality a prime [103], whereas the cost of an addition is usually negligible compared to that of a multiplication. In many situations, the requirement is to compute inversions of several field elements (for example in SSL Handshake scheme [151]). Montgomery's trick (see for example [151]) is an elegant technique for *simultaneous* computation of the inverses of several field elements. Using this trick it is possible to compute the inverses of n elements using $3(n-1)$ multiplications and one inversion (see Section 5.2.2). However, Montgomery's trick is a strictly sequential algorithm with little scope for parallelism.

In this chapter, we introduce two new algorithms for simultaneous computation of the inverses of several field elements. A sequential execution of first of these algorithm leads to more multiplications than that of Montgomery's trick. On the other hand, our algorithms are nicely parallelizable and can be implemented quite easily using a binary tree. We show that by the first algorithm, the inverses of 2^r elements can be computed using $3 \times 2^{r-1}$ multipliers using $(r+1)$ multiplication round and one inversion round. When r is moderately large, the availability of $3 \times 2^{r-1}$ multipliers can be quite costly. In such a situation, we show how to implement the algorithm using a fixed number $(2, 4, 8, \ldots)$ of multipliers.

Our second algorithm uses only 2^{r-1} multipliers to invert 2^r field elements in $2r$ parallel rounds. It, like Montgomery's trick, requires only $3(2^r - 1)$ multiplications. Although the second algorithm requires lesser number of multiplication and lesser number of multipliers, still our first algorithm is better than the second in parallel implementations if there is no restriction on the number of multipliers. That is because it can compute the inverses in half the number of multiplication rounds required by the later.

The second part of this chapter is devoted to obtaining a new parallel algorithm for elliptic curve scalar multiplication. The algorithm is SPA resistant and is applicable to the situation where the base point P is fixed. The algorithm can be made DPA resistant using generic techniques, though we do not describe them here. The main idea behind the algorithm is to combine the new inverse computation algorithm with a table look-up and a binary tree based technique. The algorithm can be implemented using a fixed number $(2, 4, 8, \ldots)$ of multipliers. Our algorithm compares favourably against many previously known parallel algorithms.

6.1.1 Affine Arithmetic in ECC Revisited

Formulae for affine arithmetic in elliptic curve groups has been described in Section 2.4.1. In this section, we present elliptic curve point arithmetic in affine coordinates in algorithmic form. An elliptic curve point is represented using a pair of finite field elements. Addition and doubling of points are performed as shown in Table 6.1 (for the case where the characteristic of the field is greater than 3). Note that in the addition algorithm we assume $P \neq \pm Q$. Let $[i], [m]$ and $[s]$ be the times required for one inversion, multiplication and squaring in the underlying fields respectively. Then, ECADD has complexity $1[i] + 2[m] + 1[s]$ and ECDBL has complexity $1[i] + 2[m] + 2[s]$. In the current work, we do not distinguish between a multiplication and a squaring. This may not be a realistic assumption when the underlying

Table 6.1: ECADD and ECDBL Algorithm

Algorithm ECADD	**Algorithm ECDBL**
Input : $P(x_1, y_1), Q(x_2, y_2)$	Input : $P(x_1, y_1)$
Output : $P + Q = (x_3, y_3)$.	Output : $2P = (x_4, y_4)$.
A1. $t_1 = (x_2 - x_1)^{-1}$	D1. $T_1 = (2y_1)^{-1}$
A2. $\lambda = t_1 * (y_2 - y_1)$	D2. $T_2 = 3x_1^2 + a$
A3. $x_3 = \lambda^2 - x_1 - x_2$	D3. $\Lambda = T_1 * T_2$
A4. $y_3 = \lambda * (x_1 - x_3) - y_1$	D4. $x_4 = \Lambda^2 - x_1 - x_2$
A5. return (x_3, y_3)	D5. $y_4 = \Lambda * (x_1 - x_3) - y_1$
	D6. return (x_4, y_4)

field is represented using a normal basis; in such a situation, squaring is virtually free. However, for standard (or polynomial) basis representation the cost of a squaring is nearly equal to that of a multiplication. Hence we consider only standard basis representation of the finite field.

6.2 New Algorithms for Computing Simultaneous Inverses

In this section we describe our proposed parallel algorithms for simultaneous inversion.

6.2.1 First Algorithm

Suppose we wish to compute the inverses of the field elements $a_0, \cdots, a_{n-1}$. For simplicity, let us assume, $n = 2^r$ for some r. Consider the full binary tree with levels numbered 0 to r. At j-th node of the i-th level of the tree ($0 \le i \le r; 0 \le j \le 2^i - 1$), we associate a set S_j^i and a field element β_j^i. At the leaf (i.e. rth) level, we take $\beta_j^r = a_j$ and $S_j^r = \{1\}$ for all j. From S_j^i and β_j^i, we compute S_j^{i-1} and β_j^{i-1} as follows. We use the following notation: for any set of field elements A and any field element x, we define, $xA = \{xy : y \in A\}$. Then, $S_j^{i-1} = \beta_{2j+1}^i S_{2j}^i \bigcup \beta_{2j}^i S_{2j+1}^i$ and $\beta_j^{i-1} = \beta_{2j}^i \times \beta_{2j+1}^i$. We have the following.

Lemma 32 *$S_0^0 = \{\overline{a_0}, \overline{a_1}, \cdots, \overline{a_{n-1}}\}$ and $\beta_0^0 = a_0 a_1 \cdots a_{n-1}$, where $\overline{a_i} = a_0 \ldots a_{i-1} a_{i+1} \ldots a_{n-1}$.*
■

The following recursive algorithm is used to compute the (S, β) pair at each node.

Algorithm 12 (MERGELST)

```
Input: Field elements $a_0, a_1, \cdots, a_{n-1}$.
Output: $(a_0 \ldots a_{n-1}, \{\overline{a_0}, \ldots, \overline{a_{n-1}}\})$.
1. If $n = 2$ return $(a_0 a_1, \{a_0, a_1\})$
2. Let $(\beta_1, S_1) = MERGELST(a_0, \ldots, a_{\lfloor n/2 \rfloor})$
3. Let $(\beta_2, S_2) = MERGELST(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_{n-1})$
4. Return($\beta_1\beta_2, \beta_1 S_2 \bigcup \beta_2 S_1$).
```

Algorithm MERGELST can be used to compute the inverses in the following manner:
1. Let $(\beta, S) = \text{MERGELST}(a_0, \ldots, a_{n-1})$;
2. return $(\beta^{-1}S)$.

Proposition 33 *The cost of MERGELST invoked on n elements is $O(n \log n)$ multiplications. When $n = 2^r$, the exact number of multiplication required is $r2^r - 1$. The cost inverting n elements is $1[i] + O(n \log n)[m]$. When $n = 2^r$ the exact cost is $1[i] + ((r+1)2^r - 1)[m]$.* ■

As examples, we can compute inverses of 16 (resp. 64) elements by 1 inversion and 79 (resp. 447) multiplications. Using Montgomery's trick, the inverses of 16 and 64 elements can be obtained using one inversion and 45 and 183 multiplications respectively. Thus Montgomery's trick requires less number of multiplications. On the other hand, Montgomery's trick does not provide any scope for parallelism whereas Algorithm MERGELST does. Below we describe a parallel version of MERGELST.

First assume that there are $3 \times 2^{r-1}$ multipliers available and sufficient memory to store $3 \times 2^{r-1}$ field elements. We name the multipliers as $M_0, M_1, \cdots, M_{2^r-1}$ and $P_0, P_1, \cdots, P_{2^{r-1}-1}$. We also name the memory locations to be used by the algorithm by $u_0, u_1, \cdots, u_{2^r-1}$ and $v_0, v_1, \cdots, v_{2^{r-1}-1}$. Each location is capable of holding one field element. Before describing our algorithm, we introduce a notation: For $0 \leq i \leq 2^r - 1$ and $1 \leq j \leq r - 1$, define $\sigma(i, j) = (i \gg j) \oplus 2^{j-1}$. As an example, when $r = 4$, we have $\sigma(14, 2) = ((1110)_2 \gg 2) \oplus (0010)_2 = (0011)_2 \oplus (0010)_2 = (0001)_2 = 1$. (Here we use the notation $(x)_2$ to denote the integer whose binary representation is x.)

Algorithm 13 (PAR-INV1)

```
Initialization: For $0 \leq i \leq 2^r - 1$, set $u_i = a_i$.
Round 1:
```

```
For $0 \le i \le 2^{r-1} - 1$,
    $P_i$ computes $u_{2i}u_{2i+1}$ and stores it to $v_i$.
For $k = 2$ to $r$
Round k:
    For $0 \le t \le 2^{r-k} - 1$ and $0 \le j \le 2^k - 1$ do in parallel
        $P_{i_1}$ computes $v_{i_1}v_{i_2}$ and stores to $v_{i_1}$.
        $M_{i_3}$ computes $u_{i_3}v_{i_4}$ and stores to $u_{i_3}$.
Round r+1:
Invert the element in $v_0$ and store to $v_0$.
Round r+2:
For $0 \le i \le 2^r - 1$ do in parallel
    $M_i$ computes $u_iv_0$ and stores to $u_i$.
For $0 \le i \le 2^r - 1$, output $u_i = a_{i\oplus 1}^{-1}$.
```

The variables i_1, i_2, i_3 and i_4 are defined as: $i_1 = t2^{k-1}$, $i_2 = i_1 + 2^{k-2}$, $i_3 = 2i_1 + j$ and $i_4 = \sigma(i_3, k-1)$.

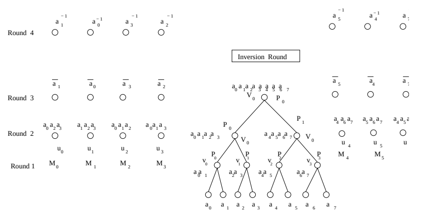
Round 4
Round 3
Round 2
Round 1
Inversion Round
a_1^{-1} a_0^{-1} a_3^{-1} a_2^{-1} a_5^{-1} a_4^{-1}
$\overline{a}_1$ $\overline{a}_0$ $\overline{a}_3$ $\overline{a}_2$ $\overline{a}_5$ $\overline{a}_4$
$a_0a_2a_3$ $a_1a_2a_3$ $a_0a_1a_2$ $a_0a_1a_3$ $a_4a_6a_7$ $a_5a_6a_7$
u_0 u_1 u_2 u_3 u_4 u_5
M_0 M_1 M_2 M_3 M_4 M_5
$a_0a_1a_2a_3a_4a_5a_6a_7$
V_0 P_0 P_1
$a_0a_1a_2a_3$ $a_4a_5a_6a_7$
v_0 v_1 v_2 v_3 P_0 P_1 P_2 P_3
a_0a_1 a_2a_3 a_4a_5 a_6a_7
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7

Figure 6.1: Algorithm PAR-INV1 in action for $n = 8$ field elements. The figure demonstrates various parallel rounds. The inversion in between rounds 3 and 4.

An example of the working of the algorithm is given in Figure 6.1. The action of the multipliers $P_0, \ldots, P_3$ are clear from the figure. We explain the action of the multipliers $M_0, \ldots, M_7$ in Rounds 2 and 3.

Round 2 : M_0 and M_1 computes u_0v_1 and u_1v_1 respectively; M_2 and M_3 computes u_2v_0 and

u_3v_0 respectively; M_4 and M_5 computes u_4v_3 and u_5v_3 respectively; M_6 and M_7 computes u_6v_2 and u_7v_2 respectively.
Round 3 : M_0, M_1, M_2 and M_3 computes u_0v_2, u_1v_2, u_2v_2 and u_3v_1 respectively; M_4, M_5, M_6 and M_7 computes u_4v_0, u_5v_0, u_6v_0 and u_7v_0 respectively.
From the description given above it is easy to verify that Algorithm PAR-INV correctly computes the inverses of 2^r elements in $r + 2$ rounds including one inversion round. With $3 \times 2^{r-1}$ multipliers PAR-INV can compute the inverses in $r + 2$ rounds.

6.2.2 Second Algorithm

Let $x_1, x_2, \cdots, x_n$ be the field elements to be inverted. Let $A[1, \cdots, (2n-1)]$ be an array of $(2n-1)$ elements, each capable of storing one field element. The following algorithm computes the inverses simultaneously.

Algorithm 14 (Simultaneous Inversion)

```
Input: Field elements $x_1, x_2, \cdots, x_n$.
Output: $x_1^{-1}, x_2^{-1}, \cdots, x_n^{-1}$.
1. For $i = n$ to $(2n-1)$, $A[i] \leftarrow x_{i-n+1}$;
2. For $i = (n-1)$ down to 1, $A[i] \leftarrow A[2i] * A[2i+1]$;
3. Invert $A[1]$, i.e. $A[1] \leftarrow A[1]^{-1}$;
4. For $i = 2$ to $2n-1$ step 2,
        $t \leftarrow A[i]$;
        $A[i] \leftarrow A[\lfloor i/2 \rfloor] * A[i \oplus 1]$;
        $A[i+1] \leftarrow A[\lfloor i/2 \rfloor] * t$;
5. Output $A[i], (n \leq i \leq (2n-1))$;
```

Proposition 34 *The cost of Algorithm 14 is* $3(n-1)$ *multiplications and one inversion.*

Proof : It is obvious that Step 2 and 4 of the algorithm require $n-1$ and $2(n-1)$ multiplications respectively. There is one inversion in Step 3. ■

The algorithm requires $2n$ memory locations, each capable of holding one field element each ($(2n-1)$ for $A[]$ and 1 for t). The elements $A[n]$ to $A[2n-1]$ in the array store the input data and $A[1]$ to $A[n-1]$ are used for storing intermediate variables. Montgomery's trick also demands same amount of memory. (The sequential version of Algorithm 14 was also independently discovered by Möller [115]). The beauty of Algorithm 14 is that it can be implemented in parallel.

Parallel Implementation

Let the elements to be inverted be $x_1, x_2, \cdots, x_n$ where $2^{r-1} \leq n \leq 2^r$. We assume that the algorithm is to be processed by 2^{r-1} multipliers and we have sufficient memory to store $2 \times n$ field elements. We name the multipliers as $P_1, P_2, \cdots, P_{2^{r-1}}$. In fact we do not need more than 2^{r-1} multipliers. The algorithm can also be run with less number of multipliers, but the number of parallel multiplication rounds will be more. We will discuss that scenario in details later. Next, we describe our algorithm.

Algorithm 15 (PAR-INV2)

```
Input: Field elements $x_1, x_2, \cdots, x_n$.
Output: $x_1^{-1}, x_2^{-1}, \cdots, x_n^{-1}$.
1. Initialisation: For $i = n$ to $2n-1$, $A[i] \leftarrow x_i$;
2. For $k \leftarrow 1$ to $r$ do in parallel
   (Round k:)
    For $i = 2^{r=k}$ to $\min\{2^{r-(k-1)} - 1, n-1\}$;
      $P_{i+1-2^{r-k}}$ computes $A[i] \leftarrow A[2i] * A[2i+1]$
3. Round r+1:
   Invert the element in $A[1]$ and store to $A[1]$;
4. For $k \leftarrow r+2$ to $2r+1$ do
   (Round k:)
    For $2^{k-(r+1)} \leq i \leq 2^{k-r} - 1$,
      $P_{i-2^{k-(r+1)}+1}$ computes in parallel $A[i] \leftarrow A[\lfloor i/2 \rfloor] * A[i \oplus 1]$;
Output $A[i]$, $n \leq i \leq (2n-1)$.
```

We present the next proposition, whose proof is obvious.

Proposition 35 *Algorithm 15 correctly computes the inverses of 2^r elements in $2r$ parallel multiplication rounds.* ■

Computing with Lesser Number of Multipliers

With 2^{r-1} multipliers Algorithm 15 can compute the inverses in $2r$ parallel rounds. Let the number of available multipliers be $t = 2^p$. Then the obvious way of carying out the computations is to allow the available multipliers to parallelly compute one round of Algorithm 15 possibly in more than one parallel rounds.

Table 6.2: Number of parallel rounds required for inverting $n = 8, 16, 32$ elements with $k = 8, 4, 2$ multipliers by Algorithm 15.

n/k	8	4	2
8	6	7	11
16	9	13	23
32	15	25	47

To carry out the computations of round $k, (1 \leq k \leq r)$ of Algorithm 15, the t processors will require $\lceil 2^{r-k}/2^p \rceil$ parallel rounds of computations. The $(r+1)$st round is an inversion round. Similarly, for round k, $r+2 \leq k \leq 2r+1$, the t processors will require $\lceil 2^{k-r-1}/2^p \rceil$ parallel rounds of computations. Hence we have,

Proposition 36 *With $t = 2^p$ multipliers Algorithm 15 can be computed in*

$$\sum_{k=1}^{r} \lceil \frac{2^{r-k}}{2^p} \rceil + \sum_{i=r+2}^{2r} \lceil \frac{2^{k-r-1}}{2^p} \rceil = 2 \sum_{k=1}^{r-1} \lceil \frac{2^k}{2^p} \rceil + 2^{r-p}$$

parallel rounds of computation besides one inversion round. ■

In the following we will denote by $\mathsf{PI}(r,p)$ *the number of multiplication rounds required by Algorithm PAR-INV2 to compute the inverses of* 2^r *elements using* 2^p *multipliers.* Hence $\mathsf{PI}(r,p) = 2\sum_{k=1}^{r-1} \lceil \frac{2^k}{2^p} \rceil + 2^{r-p}$

Performance and Comparison:
In the Table 6.2 we show the number of parallel rounds required for inverting n number of elements by k number of multipliers using Algorithm 15.

Memory Requirement

Here by a memory unit we mean a storage unit capable of storing a field element. It is easy to check that Algorithm PAR-INV2 and Montgomery's trick require n additional memory units to invert n elements. In memory requirement, algorithm PAR-INV1 is the best. It requires only $n/2$ additional memory units.

6.3 Application to Scalar Multiplication Algorithm in ECC

First we describe a parallel algorithm for simultaneous add-and-double of several elliptic curve points. The steps in the algorithm t-ECADDBL refer to the steps in Table 6.1. Also note that in Step 1 of the algorithm we invoke PAR-INV to invert several field elements simultaneously. Here by PAR-INV we mean PAR-INV1 or PAR-INV2, as any one of them can be used.

Algorithm 16 (t-ECADDBL)

```
Input: $P_1, Q_1; \cdots; P_t, Q_t$
Output: $P_1 + Q_1, 2P_1; \cdots; P_t + Q_t, 2P_t$
1. Use PAR-INV to perform the inversions in the
      steps $A1(P_1, Q_1)$, $D1(P_1)$; ...; $A1(P_t, Q_t)$, $D1(P_t)$.
2. For $1 \leq i \leq t$, do in parallel $A2(P_i, Q_i)$, $D2(P_i)$.
3. For $1 \leq i \leq t$, do in parallel $A3(P_i, Q_i)$, $D3(P_i)$.
4. For $1 \leq i \leq t$, do in parallel $A4(P_i, Q_i)$, $D4(P_i)$.
5. For $1 \leq i \leq t$, do in parallel $D5(P_i)$.
6. For $1 \leq i \leq t$, do in parallel $A5(P_i, Q_i)$, $D6(P_i)$.
```

It is easy to verify that, if $t = 2^r$ and $3t$ multipliers are available for computation then algorithm t-ECADDBL can be computed in $r + 5$ multiplication rounds and one inversion round if we use PAR-INV1 in place of PAR-INV. Using PAR-INV2 with $t/2$ multipliers, it can be computed in $2r + 4$ multiplication rounds and one inversion round. If $k = 2^p$ multipliers are available, then each of Steps 2 through 4 will take $\lceil 2^{r+1}/2^p \rceil$ multiplication rounds and Step 5 will take $\lceil 2^r/2^p \rceil$ multiplication rounds. Step 1 requires $\mathsf{PI}(r + 1, p)$ multiplication rounds. Hence we have

Proposition 37 *With $k = 2^p$ multipliers, Algorithm 2^r-ECADDBL can be computed in $3 \times \lceil 2^{r+1}/2^p \rceil + \lceil 2^r/2^p \rceil + \mathsf{PI}(2^{r+1}, p)$ multiplication rounds and one inversion round.*

In the following we will use algorithms t-ECADD and t-ECDBL. These algorithms are straightforward modifications of t-ECADDBL for t additions and t doublings respectively. The cost of 2^r-ECADD (resp. 2^r-ECDBL) using 2^p multipliers is $3 \times \lceil 2^r/2^p \rceil + \mathsf{PI}(r, p)$ (resp. $4 \times \lceil 2^r/2^p \rceil + \mathsf{PI}(r, p)$) multiplication rounds and one inversion round.

We now present our new scalar multiplication algorithm. Our algorithm incorporates a countermeasure to resist SPA which is based on Coron's technique of dummy addition [36].

Our algorithm computes the scalar multiplication in parallel and is suitable for the situation when the base point is fixed. Applications for such situations can be found in [62].

Let $w \geq 2$ be a positive integer. We express m in the base 2^w. Let $m = c_0 + c_1 2^w + \cdots + c_{t-1} 2^{w(t-1)}$, where each $c_j \in \{0, \dots, 2^w - 1\}$ and $t = 2^r$ for some r. Then $mP = c_0 P + c_1 2^w P + \cdots + c_{t-1} 2^{w(t-1)} P$. For all j with $0 \leq j \leq t-1$ we precompute $2^{jw}P$ and store it in a table $T[\,]$. Thus $T[j] = 2^{jw}P$ for $0 \leq j \leq t-1$. This table is used to simultaneously compute $c_0 P, c_1 2^w P, \cdots, c_{t-1} 2^{w(t-1)} P$ using the right-to-left binary method. Finally we add them to obtain mP. Let the n-bit binary representation of m be $m_{n-1} \dots m_0$. Note that $t = \lceil n/w \rceil$. We express c_j in binary, i.e., we write $c_j = c_j^0 + c_j^1 2 + \cdots + c_j^{w-1} 2^{w-1}$, where $c_j^i = m_{wj+i}$.

Algorithm 17 (PAR-SC)

```
Input : $c_j^i$ for $0 \leq i \leq w-1, 0 \leq j \leq t-1$; table $T[\,]$
Output : $mP$
1. For $j = 0$ to $t-1$
       $R_j = T[j]$; $Q_j^{(c_j^0)} = R_j$; $Q_j^{(1-c_j^0)} = 0$;
2. $(R_0, \dots, R_{t-1}) = t$-ECDBL$(R_0, \dots, R_{t-1})$;
3. For $i = 1$ to $w-2$
       $(Q_0^{(c_0^i)}, R_0, \dots, Q_{t-1}^{(c_{t-1}^i)}, R_{t-1})$
           $= t$-ECADDDBL$(R_0, Q_0^{(1)}, \dots, R_{t-1}, Q_{t-1}^{(1)})$;
4. $(Q_0^{(c_0^{w-1})}, \dots, Q_{t-1}^{(c_{t-1}^{w-1})})$
       $= t$-ECADD$(R_0, Q_0^{(1)}, \dots, R_{t-1}, Q_{t-1}^{(1)})$.
5. Let res $= t$-ADD$(Q_0^{(1)}, \cdots, Q_{t-1}^{(1)})$.
6. Return (res).
```

Note that the amount of computation is independent of the values of the c_j^is. Hence the algorithm is SPA resistant. Algorithm t-ADD takes as input t points and computes their sum.

Algorithm 18 (t-ADD)

```
Input:  $P_0, P_1, \cdots, P_{2^r-1}$.
Output: $P_0 + P_1 + \cdots + P_{2^r-1}$.
1. For $i = 1$ to $r$ do
2.   $k = 2^{r-i}$;
3.   $(P_0, P_{2^i}, P_{2 \cdot 2^i}, P_{3 \cdot 2^i}, \dots, P_{(k-1)2^i})$
```

Table 6.3: Number of rounds required for PAR-SC with PAR-INV2 for 160-bit scalar multiplier. Note $2^r = 160/w$.

$w \backslash 2^p$	8	4	2
5	(251, 10)	(464, 10)	(907, 10)
10	(273, 14)	(492, 14)	(955, 14)
20	(298, 23)	(506, 23)	(965, 23)

$=k$*-ECADD*$(P_0, P_{2^{i-1}}, P_{2\cdot 2^{i-1}}, P_{3\cdot 2^{i-1}}, \ldots, P_{(2k-1)2^{i-1}})$;
4. return P_0.

Algorithm 2^r-ADD invokes 2^i-ECADD for $0 \le i \le r-1$. Using the cost of 2^i-ECADD we obtain the cost for 2^r-ADD to be a total of $\sum_{i=0}^{r-1}(3\lceil 2^i/2^p \rceil + \mathsf{PI}(i,p))$ multiplication rounds and r inversion rounds. Thus we can now obtain the number of rounds required to compute algorithm PAR-SC.

Proposition 38 *Using 2^p multipliers, Algorithm PAR-SC requires $(r+w)$ inversion rounds and $(w+5)\lceil 2^r/2^p \rceil + 2\mathsf{PI}(r,p) + \sum_{i=0}^{r-1}(3\lceil 2^i/2^p \rceil + \mathsf{PI}(i,p)) + (w-2)(3\lceil 2^{r+1}/2^p \rceil + \mathsf{PI}(r+1,p))$ multiplication rounds to complete the scalar multiplication.*

Proof : Algorithm PAR-SC invokes 2^r-ECDBL, 2^r-ECADD and 2^r-ADD once each. Further, it invokes 2^r-ECADDBL a total of $(w-2)$ times. Adding up all the costs gives us the required result. ■

6.4 Results

In this section, we present some results for typical situations. For elliptic curve cryptosystem a group order of 160 bits is considered to be secure. We consider various window sizes (values of w) and calculate the number of multiplication and inversion rounds. The results are summarized in Table 6.3.

Each entry of Table 6.3 is a pair of the form (a, b), where a is the number of multiplication rounds and b is the number of inversion rounds. The cost of an inversion over prime (> 3) fields can be thirty or more times the cost of a multiplication [45, 103]. If we assume that an inversion is equal to 30 multiplications, then for window size $w = 5$ and using two multipliers, we will require around 1207 multiplication rounds using PAR-INV2. The parallel algorithm presented in [51] with two multipliers computes 160 point doublings and

40 point additions on average. Using Jacobian coordinates, the computation will take (10$[m]$ for a doubling and 11$[m]$ for a mixed addition) more than 2000 multiplication rounds. The scheme proposed in [44], uses a parallelized encapsulated-add-and-double algorithm using Montgomery arithmetic. This algorithm using two multipliers takes 10$[m]$ computations per bit of the scalar (1600[m] for a scalar of length 160 bits). Our algorithm does that in 1207 multiplications with 2 multipliers, which is a speed-up of around 25% over the algorithm in [44].

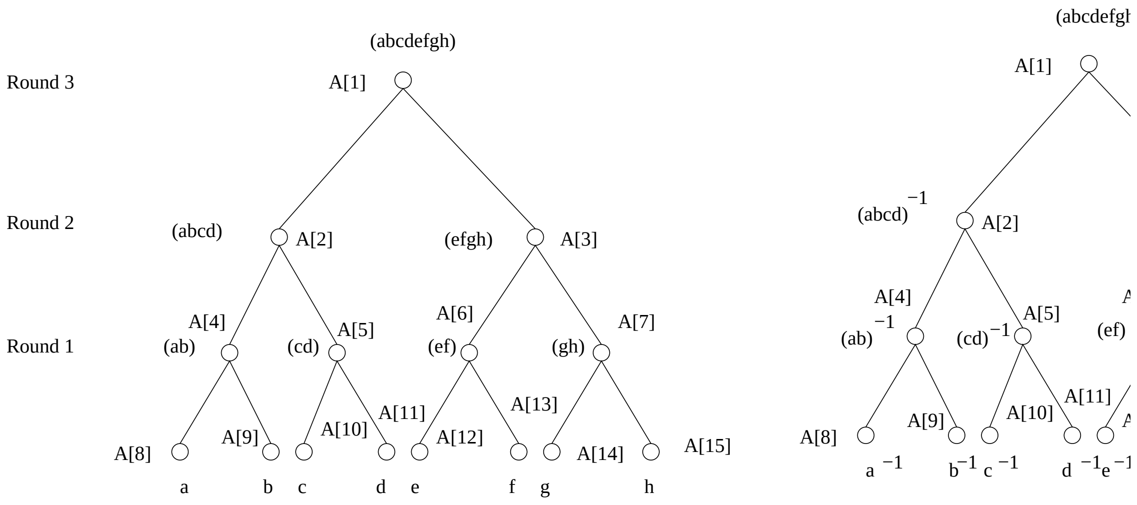

Figure 6.2: Algorithm 15 in action with $n = 8$. The figure demonstrates various parallel rounds of Algorithm 15 for inverting 8 elements $a, \cdots, h$. Round 4 is the inversion round.

Chapter 7

Pipelined Computation of the Scalar Multiplication in Elliptic Curve Cryptosystems

In this chapter, we propose a pipelining scheme for implementing Elliptic Curve Cryptosystems (ECC). The scalar multiplication is the dominant operation in ECC. It is computed by a series of point additions and doublings. The pipelining scheme is based on a key observation: to start the subsequent operation one need not wait until the current one exits. The next operation can begin while a part of the current operation is still being processed. Also, the proposed scheme can be made resistant to side-channel attacks (SCA). Our scheme compares favourably to all SCA resistant sequential and parallel methods.

7.1 Introduction

The fundamental operation in ECC is scalar multiplication, namely, given an integer d and an elliptic curve point P, the computation of dP. It is computed by a series of doubling (ECDBL) and addition (ECADD) operation of the point P, depending upon the bit sequence representing d. Several methods have been proposed in literature to compute the scalar multiplication in a secure and efficient way. For an excellent review see [62]. The performance of all these methods is dependent on the efficiency of the elliptic curve group operations: ECDBL and ECADD. These operations in affine coordinates involve inversion, which is a very costly operation, particularly over prime fields. To avoid inversion various co-ordinate systems have been proposed in literature. Here, we will use Jacobian coordinates.

In this chapter we describe a very general technique to compute the scalar multiplication.

It can be applied to any scalar multiplication method that only uses doubling and addition (or subtraction), with or without pre-computations.

The computation of scalar multiplication proceeds in a series of doublings and additions. A key observation is that these operations can be computed in a pipeline, so that the subsequent operation need not wait till the current one exits. The ECADD and ECDBL algorithms have their own set of inputs. These algorithms can be divided into parts some of which can be executed with only a part of the input. So one part of the algorithm can begin execution as soon as the corresponding part of the inputs is available to it. Thus two or more operations can be executed in a pipeline.

Here we propose a two stage pipeline. At any point of time there will be at most two operations in the pipeline in a "Producer-Consumer Relation". The one which enters the pipeline earlier will be producing outputs which will be consumed by the second operation as inputs. As soon c9algo1as the producer process exits the pipeline the subsequent EC-operation will enter the pipeline as the consumer process. The earlier consumer would be producing outputs now which will be consumed by the newer process.

Any processor capable of handling ECC must have capabilities (in hardware or software) for executing field arithmetic. It must have modules for field element addition, subtraction, multiplication and inversion. In computing the scalar multiplication using a co-ordinate system other than affine coordinates, only one field inversion is necessary. So the most important operations are addition and multiplication. In the pipelined architecture we need a multiplier and an adder for each of the pipe stages. The adder can be shared by the pipe stages as the addition operation is much cheaper in comparison to multiplication. Note that in this work, we will consider a squaring as a multiplication. However, this is not true in general.

Our method will also require slightly more memory. As two operations will be computed simultaneously in two pipe stages more memory will be required for processing. This extra memory requirement is discussed in details in Section 7.5.

Many countermeasures against SCA have been proposed in literature. Almost all of them involve some computational overhead to resist SCA. One of the latest methods proposed in [23] involves the least amount of computational overhead to get rid of simple power analysis attacks (SPA). The author divides each EC-operation into *atomic blocks* which are indistinguishable from the side-channel. So, in his model the computation of scalar multiplication is a sequence of indistinguishable atomic block. This can be combined with many standard countermeasures against differential power attacks (DPA). We use a variant of this method to resist SPA.

In this work we have used the computation time of an atomic block as a unit of time. One atomic block has one multiplication, two additions and one negation. As computation time of an addition and that of a negation is quite small in comparison to that of a multiplication, computation time of an atomic block is approximately that of a finite field multiplication. A point addition (mixed coordinates) takes 11 atomic blocks and a doubling takes 10 atomic blocks. So they can be computed in 10 and 11 units of time respectively. In our pipelining scheme an operation can be computed in 6 units of time. This leads to a significant improvement in performance (see Section 7.5).

7.2 Dividing ECADD and ECDBL into Atomic Blocks

We divide each EC-operation into atomic blocks. Following [23], each block contains one multiplication and two additions. Subtraction is treated as a negation followed by an addition. To accommodate subtractions we include one negation in each atomic block. The atomic blocks are presented in Table 7.1 and Table 7.2. Our ECADD and ECDBL algorithms are designed, keeping in mind scalar multiplication algorithm. In the binary algorithm, described in Table 2.9 of Section 2.9 whenever an addition is carried out, one input is fixed i.e. P. So we may assume that like ECDBL, algorithm ECADD has also one input P_i. Also, we can keep the point P in affine coordinates and gain efficiency by using mixed addition algorithm. Note that in Table 7.1 and Table 7.2, we have assumed that the ECADD and ECDBL always get their inputs (X_i, Y_i, Z_i) at three specific locations T_6, T_7, T_8 respectively. Also, the EC-operation write back their outputs as these are computed to these locations only. The coordinates of the point P(X, Y, 1), which is an argument to all addition operations are also stored in two specific locations $T_x = X, T_y = Y$. Also, the curve parameter a needs to be stored. These six locations are public in the sense that any operation in any of the two pipe stages can use them. One more location is required for the dummy operations. Both the operation in the pipeline will share this location. Besides while two operations being computed in the pipeline, each of them will have some locations (five each) private to them to store their intermediate variables. Thus the method requires at least 17 locations for the computation. The inputs to the operations, except for the first doubling, are not available to them from the beginning. The inputs are provided one by one as the previous EC-operation computes them.

In Table 7.2 we provide the mixed addition algorithm in atomic blocks. Mixed addition requires 11 multiplications and doublings involves 10. So, adding one dummy multiplication to the ECDBL and some additions/subtractions to ECADD/ECDBL, we can use whole of them as atomic blocks. However in that case we have to use these operations as atomic units of computation. So, one operation has to be completed before the other begins. We do not adopt this approach as our aim is to break the operations into parts such that a part of one can start execution while a part of another is still in the pipeline.

Table 7.1: DBL Algorithm in Atomic Blocks

DBL Algorithm
Input: $P_i(X_i, Y_i, Z_i)$
Input: $P_i = (X_i, Y_i, Z_i)$
Output: $2P_i = (X_{i+1}, Y_{i+1}, Z_{i+1})$

Δ_1	$R_1 = T_8 \times T_8$ (Z_i^2)	Δ_6	$R_4 = T_7 \times T_7$ (Y_i^2)
	$*$		$R_2 = R_4 + R_4$ $(2Y_i^2)$
	$*$		$*$
	$*$		$*$
Δ_2	$R_1 = R_1 \times R_1$ (Z_i^4)	Δ_7	$R_4 = T_6 \times R_2$ $(2X_iY_i^2)$
	$*$		$R_4 = R_4 + R_4$ (S)
	$*$		$R_4 = -R_4$ $(-S)$
	$*$		$R_5 = R_4 + R_4$ $(-2S)$
Δ_3	$R_1 = a \times R_1$ (aZ_i^4)	Δ_8	$R_3 = R_1 \times R_1$ (M^2)
	$*$		$T_6 = R_3 + R_5$ $(\underline{X_{i+1}})$
	$*$		$*$
	$*$		$R_4 = T_6 + R_4$ $(X_{i+1} - S)$
Δ_4	$R_2 = T_6 \times T_6$ (X_i^2)	Δ_9	$R_2 = R_2 \times R_2$ $(4Y_i^4)$
	$R_3 = R_2 + R_2(2X_i^2)$		$R_2 = R_2 + R_2(8Y_i^4)$
	$*$		$*$
	$R_2 = R_3 + R_2$ $(3X_i^2)$		$*$
Δ_5	$T_8 = T_7 \times T_8$ (Y_iZ_i)	Δ_{10}	$T_7 = R_1 \times R_4$ $(M(X_{i+1} - S))$
	$T_8 = T_8 + T_8$ $(\underline{Z_{i+1}})$		$T_7 = T_7 + R_2$ $(-Y_{i+1})$
	$*$		$T_7 = -T_7$ $(\underline{Y_{i+1}})$
	$R_1 = R_1 + R_2$ (M)		$*$

Table 7.2: ADD Algorithm in Atomic Blocks

ADD Algorithm
Input: $P = (T_x, T_y), P_i = (X_i, Y_i, Z_i)$
Output: $P + P_i = (X_{i+1}, Y_{i+1}, Z_{i+1})$.

Γ_1	$R_1 = T_8 \times T_8 \ (Z_i^2)$ $*$ $*$ $*$	Γ_7	$R_2 = R_2 \times R_4 \ (-U_1W^2)$ $R_5 = R_2 + R_2 \ (-2U_1W^2)$ $*$ $*$
Γ_2	$R_2 = T_x \times R_1 \ (U_1)$ $*$ $R_2 = -R_2 \ (-U_1)$ $*$	Γ_8	$R_1 = R_4 \times R_1 \ (W^3)$ $R_1 = R_1 + R_5 \ (W^3 - 2U_1W^2)$ $R_3 = -R_3 \ (-S_1)$ $R_5 = R_3 + T_7 \ (S_2 - S_1 = -R)$
Γ_3	$R_3 = T_y \times T_8 \ (YZ_i)$ $*$ $*$ $*$	Γ_9	$T_6 = R_5 \times R_5 \ (R^2)$ $T_6 = T_6 + R_1 \ (\underline{X_{i+1}})$ $R_2 = T_6 + R_2 \ (X_{i+1} - U_1W^2)$
Γ_4	$R_3 = R_3 \times R_1 \ (S_1)$ $R_1 = R_2 + T_6 \ (-W)$ $R_1 = -R_1 \ (W)$ $*$	Γ_{10}	$R_2 = R_5 \times R_2 \ (-R(X_{i+1} - U_1W^2))$ $*$ $*$
Γ_5	$T_8 = R_1 \times T_8 \ (\underline{Z_{i+1}})$ $*$ $*$ $*$	Γ_{11}	$T_7 = R_3 \times R_4 \ (-S_1W^2)$ $T_7 = T_7 + R_2 \ (\underline{Y_{i+1}})$ $*$ $*$
Γ_6	$R_4 = R_1 \times R_1 \ (W^2)$ $*$ $*$ $*$		

7.2.1 An Analysis of ECADD and ECDBL

Let us analyze the ECADD and ECDBL algorithms presented in the Table 7.1 and Table 7.2. To ECDBL, there are three inputs, namely, X_i, Y_i, Z_i. It computes the double of the input point. Let us look at the various atomic blocks more closely.

We make the following observations on ECDBL:

- The atomic blocks $\Delta_1, \Delta_2, \Delta_3$ can be computed with the input Z_i only.
- Input X_i is needed by ECDBL at block Δ_4 and thereafter.
- The block Δ_5 needs the input Y_i as well. But Δ_5 produces the output Z_{i+1}. So, the next EC-operation can begin after ECDBL completes Δ_5.
- The atomic block Δ_8 produces the output X_{i+1}.
- Δ_{10} produces the output Y_{i+1} and the process terminates.

We have similar observations on ECADD:

- The atomic blocks $\Gamma_1, \Gamma_2, \Gamma_3$ can be computed with the input Z_i only.
- Input X_i is needed by ECADD at block Δ_4 and thereafter.
- Γ_5 produces the output Z_{i+1}. So, the next EC-operation can begin after ECADD completes Γ_5.
- The input Y_i is not required till the atomic block Γ_8.
- Γ_9 produces the output X_{i+1} and Γ_{11} produces Y_{i+1} and the process terminates.

The most interesting part of the division is that both the operations perfectly match in a producer-consumer relation. In most situations as we will see in the next section, as soon as an output is produced by the producer process the consumer process consumes it. In some situation when the consumer process requires the input before it is produced by the producer, so the consumer process has to wait an atomic block. However, such situations will not arise much frequently, hence it does not affect the efficiency much.

Table 7.3: ECADD and ECDBL in the pipeline

	DBL-DBL		DBL-ADD		ADD-DBL	
Time	PS1	PS2	PS1	PS2	PS1	PS2
k	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k+1$	$\Delta_1^{(i)}$	-	$\Delta_1^{(i)}$	-	$\Gamma_1^{(i)}$	-
$k+2$	$\Delta_2^{(i)}$	-	$\Delta_2^{(i)}$	-	$\Gamma_2^{(i)}$	-
$k+3$	$\Delta_3^{(i)}$	-	$\Delta_3^{(i)}$	-	$\Gamma_3^{(i)}$	-
$k+4$	$\Delta_4^{(i)}$	-	$\Delta_4^{(i)}$	-	$\Gamma_4^{(i)}$	-
$k+5$	$\Delta_5^{(i)}$	-	$\Delta_5^{(i)}$	-	$\Gamma_5^{(i)}$	-
$k+6$	$\Delta_1^{(i+1)}$	$\Delta_6^{(i)}$	$\Gamma_1^{(i+1)}$	$\Delta_6^{(i)}$	$\Delta_1^{(i+1)}$	$\Gamma_6^{(i)}$
$k+7$	$\Delta_2^{(i+1)}$	$\Delta_7^{(i)}$	$\Gamma_2^{(i+1)}$	$\Delta_7^{(i)}$	$\Delta_2^{(i+1)}$	$\Gamma_7^{(i)}$
$k+8$	$\Delta_3^{(i+1)}$	$\Delta_8^{(i)}$	$\Gamma_3^{(i+1)}$	$\Delta_8^{(i)}$	$\Delta_3^{(i+1)}$	$\Gamma_8^{(i)}$
$k+9$	$\Delta_4^{(i+1)}$	$\Delta_9^{(i)}$	$\Gamma_4^{(i+1)}$	$\Delta_9^{(i)}$	$*$	$\Gamma_9^{(i)}$
$k+10$	$*$	$\Delta_{10}^{(i)}$	$\Gamma_5^{(i+1)}$	$\Delta_{10}^{(i)}$	$\Delta_4^{(i+1)}$	$\Gamma_{10}^{(i)}$
$k+11$	$\Delta_5^{(i+1)}$	$*$	$\vdots$	$\Gamma_6^{(i+1)}$	$*$	$\Gamma_{11}^{(i)}$
$k+12$	$\vdots$	$\Delta_6^{(i+1)}$	$\vdots$	$\Gamma_7^{(i+1)}$	$\Delta_5^{(i+1)}$	$*$

7.3 Pipelining the Scalar Multiplication Algorithm in ECC

In this section we describe our pipelining scheme – a two stage pipeline, each executing part of an EC-operation in parallel. In the following discussion, we assume that the EC-operation executing in pipe stage 1 gets its inputs when it needs. Later we will see, it is not always true. However, such cases will not occur very frequently.

In the computation of the scalar multiplication, an ECDBL is followed by an ECADD or an ECDBL, but an ECADD is always followed by an ECDBL. So in the proposed pipeline we always see a pattern like ECDBL(producer)-ECDBL(consumer) or ECDBL(producer)-ECADD(consumer) or ECADD(producer)-ECDBL(consumer). This is true even if the scalar is represented in NAF. Let us see how these operations play their parts in this producer-consumer relation. We show this in the Table 7.3. The atomic blocks Γ_i's belong to an ECADD and Δ_j's belong to ECDBL. Besides we have given a superscript to each of them to denote which EC-operation has entered the pipeline earlier. In the following description we will refer to pipe stage 1 and pipe stage 2 as PS1 and PS2 respectively. Also, in this discussion *our unit of time is time taken to execute one atomic block*. In the next three subsections we will discuss how ECADD and ECDBL coupled with each other behave in the pipeline.

7.3.1 DBL-DBL Scenario

Let us first consider the ECDBL-ECDBL scenario. It is presented in Columns 2 and 3 of Table 7.3.

- Let us assume that the first ECDBL (say, ECDBL$^{(i)}$) enters PS1 at time $k+1$.
- At time $k+5$, ECDBL$^{(i)}$ produces its first output (Z_{i+1}) and enters PS2 and the second doubling ECDBL$^{(i+1)}$ enters the stage PS1 of the pipeline.
- At time $k+8$, ECDBL$^{(i)}$ produces its second output. ECDBL$^{(i+1)}$ completes its 3rd atomic block $\Delta_3^{(i+1)}$ at the same time.
- At time $k+9$ ECDBL$^{(i)}$ computes $\Delta_9^{(i)}$ and ECDBL$^{(i+1)}$ computes $\Delta_4^{(i+1)}$. Note that ECDBL$^{(i+1)}$ requires its second input i.e. Y_i in this block, which is available to it. It is computed by ECDBL$^{(i)}$ in the last atomic block.
- During time $k+10$, ECDBL$^{(i)}$ computes $\Delta_{10}^{(i+1)}$ and computes its third output (Y_{i+1}). ECDBL$^{(i+1)}$ should compute $\Delta_5^{(i)}$. But it needs its third input which is being computed at this time only. Hence it waits. ECDBL$^{(i)}$ terminates at the end of time $k+10$.
- ECDBL$^{(i+1)}$ computes $\Delta_5^{(i)}$ and produces its first input and moves to PS2 in the next time unit. Although the other pipe stage is vacant now it can not be utilized as ECDBL$^{(i+1)}$ has not yet produced its first output.

Note that in this scenario, when two ECDBL's enter the pipeline one by one, two pipeline stages remain idle (one at time $k+10$ and another at time $k+11$) during the computations. We have marked them by '*' in the table. If the attacker using the side-channel information can detect this he may be able to conclude that two doubling were being computed now. To keep the adversary at bay we can compute two dummy blocks at these times. That will also implement the wait for the other process.

7.3.2 DBL-ADD Scenario

Let us consider the situation when an ECDBL is followed by an ECADD. This scenario is described in Columns 4 and 5 of Table 7.3. Unlike the previous discussion we will refer to the operations as ECADD and ECDBL only, without any superscript. Note that the ECDBL has entered the pipeline first and ECADD later. Suppose the ECDBL starts at time $k + 1$. We can see that:

- At time $k + 5$, ECDBL computes block $\Delta_5^{(i)}$. It produces its first output and then it enters PS2 at the next time unit.
- At time $k + 6$, the ECADD enters the pipeline at PS1. It uses the output of ECDBL.
- At time $k + 8$ the ECADD computes $\Gamma_3^{(i+1)}$. At the same time the ECDBL completes its block $\Delta_8^{(i)}$ and produces the output X_{i+1}.
- At time $k+9$, ECADD computes $\Gamma_4^{(i+1)}$. It needs its second input for this computation, which is produced by the ECDBL in the previous time interval.
- At time $k + 10$, the ECDBL computes its last atomic block and computed its third output. The ECADD computes $\Gamma_5^{(i+1)}$. The ECDBL exits after this time interval. The output computed by the ECDBL is required by the ECADD two time units later.

In this scenario the coupling of operations is perfect. No pipeline stages are wasted.

7.3.3 ADD-DBL Scenario

Let us consider the third scenario, i.e. an ECADD followed by an ECDBL. Suppose, an ECADD enters the pipeline at time $k + 1$, which gets its input from the previous process, which must be an ECDBL. As it is clear from the previous scenario, it runs till Γ_5 without any hitch. Let us discuss other important events during this computation process.

- During the time interval $k + 5$, the ECADD computes its first output Z_{i+1} and enters PS2 at the next time interval. The ECDBL enters PS1.
- As the ECDBL completes Δ_3 it needs the next input. At that time, ECADD would have completed Γ_8. It can produce the desired output in the next atomic block only. Hence the ECDBL has to wait for one unit of time.
- In time $k + 10$, ECADD and ECDBL process the blocks Γ_{10} and Δ_4 respectively. In the next unit of time ECDBL requires its third input, i.e. the Y-co-ordinate, which has not yet been computed by the producer process ECADD. Hence the doubling has to wait for another unit of time.

- During time $k+11$, the ECADD in its atomic block Γ_{11} produces the desired output, completes its execution and exits the pipeline.
- In the next time unit i.e $k+12$, the ECADD computes Δ_5, produces its first output paving the way for another EC-operation to enter the pipeline.

As we have seen above, in this scenario also, three pipeline stages are wasted. For the reasons mentioned earlier, these vacant slots should be filled in by computation of dummy atomic blocks.

One can easily convince oneself that these choices are optimal. The computation of Z_{i+1} requires Y_i which is only provided in the final stage of the previous doubling. Hence, a wait stage cannot be avoided.

7.4 Example

In Table 7.4, we have exhibited the computation process for a small scalar multiplier $38 = 100110$. To compute $38P$, one has to carry out EC-operations as DBL, DBL, DBL, ADD, DBL, ADD, DBL. Note that this multiplier encompasses all possibilities, i.e. DBL-DBL, DBL-ADD and ADD-DBL. The computation takes 46 units of time. In the table we have shown how the computation progresses. Each atomic block has been assigned a superscript to denote the serial number of the EC-operation to which it belongs. Also some atomic blocks are prefixed or suffixed by (X) or (Y) or (Z). A suffix indicates that at that atomic block the EC-operation outputs the corresponding value. A prefix indicates that at the specified atomic block the EC-operation consumes that input. Also, a '#' sign in the time column indicates that an EC-operation exits the pipeline at that time. A '*' in a pipe stage indicates a dummy atomic block has to be computed there. A '-' indicates no computation.

As we can check from the Table 7.4, in the pipelining scheme, the first EC-operation which is usually an ECDBL, completes in 10 time units. After that an EC-operation (be it an ECDBL or an ECADD) completes in every 6 units of time.

7.5 Implementation and Results

In this section we will discuss the issues related to the implementation of the scheme. Also, we will demonstrate the speed-up that can be achieved in an implementation.

Hardware Requirement:

Table 7.4: An Example of the Pipelining

Time	PS1	PS2	Time	PS1	PS2
1	$(Z)\Delta_1^{(1)}$	-	24	$\Delta_2^{(5)}$	$\Gamma_7^{(4)}$
2	$\Delta_2^{(1)}$	-	25	$\Delta_3^{(5)}$	$(Y)\Gamma_8^{(4)}$
3	$\Delta_3^{(1)}$	-	26	*	$\Gamma_9^{(4)}(X)$
4	$(X)\Delta_4^{(1)}$	-	27	$(X)\Delta_4^{(5)}$	$\Gamma_{10}^{(4)}$
5	$(Y)\Delta_5^{(1)}(Z)$	-	28#	*	$\Gamma_{11}^{(4)}(Y)$
6	$(Z)\Delta_1^{(2)}$	$\Delta_6^{(1)}$	29	$(Y)\Delta_5^{(5)}(Z)$	*
7	$\Delta_2^{(2)}$	$\Delta_7^{(1)}$	30	$(Z)\Gamma_1^{(6)}$	$\Delta_6^{(5)}$
8	$\Delta_3^{(2)}$	$\Delta_8^{(1)}(X)$	31	$\Gamma_2^{(6)}$	$\Delta_7^{(5)}$
9	$(X)\Delta_4^{(2)}$	$\Delta_9^{(1)}$	32	$\Gamma_3^{(6)}$	$\Delta_8^{(5)}(X)$
10#	*	$\Delta_{10}^{(1)}(Y)$	33	$(X)\Gamma_4^{(6)}$	$\Delta_9^{(5)}$
11	$(Y)\Delta_5^{(2)}(Z)$	*	34#	$\Gamma_5^{(6)}(Z)$	$\Delta_{10}^{(5)}(Y)$
12	$(Z)\Delta_1^{(3)}$	$\Delta_6^{(2)}$	35	$(Z)\Delta_1^{(7)}$	$\Gamma_6^{(6)}$
13	$\Delta_2^{(3)}$	$\Delta_7^{(2)}$	36	$\Delta_2^{(7)}$	$\Gamma_7^{(6)}$
14	$\Delta_3^{(3)}$	$\Delta_8^{(2)}(X)$	37	$\Delta_3^{(7)}$	$(Y)\Gamma_8^{(6)}$
15	$(X)\Delta_4^{(3)}$	$\Delta_9^{(2)}$	38	*	$\Gamma_9^{(6)}(X)$
16#	*	$\Delta_{10}^{(2)}(Y)$	39	$(X)\Delta_4^{(7)}$	$\Gamma_{10}^{(6)}$
17	$(Y)\Delta_5^{(3)}(Z)$	*	40#	*	$\Gamma_{11}^{(6)}(Y)$
18	$(Z)\Gamma_1^{(4)}$	$\Delta_6^{(3)}$	41	$(Y)\Delta_5^{(7)}(Z)$	*
19	$\Gamma_2^{(4)}$	$\Delta_7^{(3)}$	42	-	$\Delta_6^{(7)}$
20	$\Gamma_3^{(4)}$	$\Delta_8^{(3)}(X)$	43	-	$\Delta_7^{(7)}$
21	$(X)\Gamma_4^{(4)}$	$\Delta_9^{(3)}$	44	-	$\Delta_8^{(7)}(X)$
22#	$\Gamma_5^{(4)}(Z)$	$\Delta_{10}^{(3)}(Y)$	45	-	$\Delta_9^{(7)}$
23	$(Z)\Delta_1^{(5)}$	$\Gamma_6^{(4)}$	46#	-	$\Delta_{10}^{(7)}(Y)$

As the proposed scheme processes two EC-operation simultaneously, we will require more hardware support than is generally required for ECC. To implement the pipe stages we will require a multiplier and an adder for each of the pipe stages. As addition is a much cheaper operation than multiplication one adder can be shared between the pipe stages. We do not need separate (multiplicative) inverter for each pipe stage. In fact, we need only an inversion after completing all the doublings and additions. So one inverter would suffice. Thus, in comparison to a sequential computation we need only one additional multiplier to implement the proposed scheme.

Memory Requirement:

In general ECADD requires 7 locations and ECDBL requires 6 locations in memory in sequential execution, where one EC-operation is executed at a time. So the whole scalar multiplication can be computed using 7 locations for the doublings and additions. The proposed scheme requires 17 memory locations i.e. 10 extra memory locations. If the field elements are of 160 bits, then the additional memory requirement is 1600 bits or 200 bytes.

Synchronization:

As we have said there are seven locations where some values will be stored during whole process of computation. So, if two processes working at two stages of the pipeline wish to access these values simultaneously, conflict may arise. Particularly, if one process is trying to read and the other is trying to write the same location at the same time, then it will lead to a very serious problem. But fortunately, that situation will never occur. As may be recalled these locations store the values X, Y, Z, a and also the locations where the ECADD and ECDBL get their inputs and store their outputs (T_6, T_7, T_8) are public in nature. The locations storing a is used by ECDBL only. The locations storing X, Y, Z are used by ECADD only. So there can not be any conflict for these locations. Conflicts also do not arise for the locations T_6, T_7, T_8. As can be checked, ECADD uses T_8 in the atomic blocks Γ_1, Γ_3 and Γ_5 only. ECDBL uses T_8 in the atomic blocks Δ_1 and Δ_5 only. As can be checked from the tables these atomic blocks are never computed simultaneously. Similarly we can check for other locations.

Resistance Against SCA:

As the technique uses side-channel atomicity, it is secure against simple power analysis under the assumption (cf. [23, 28]) that dummy operations cannot be detected and that squarings and multiplications are indistinguishable from the side channel. Note that the Hamming weight of the scalar is leaked; we come back on this later. To resist DPA Joye-Tymen's curve randomization [69] can be easily adopted into the scheme. It will require two more storage locations. As the scheme uses affine representation of the point, it is does not adapt directly to Coron's point randomization [36]. However, note that after the first doubling, the output point is no more affine. Hence it can be randomized. Also this later randomization does not compromise the security because, the first EC-operation is always a doubling.

A second option is to do the preprocessing step $T_6 = T_x \times Z^2, T_7 = T_y \times Z^3, T_8 = Z$, for some randomly chosen Z. This requires 4 multiplications and the input to the first doubling is no longer affine; hence, the costs are higher than in the first proposal. Both ways there is absolutely no problem in the scheme to adapt to scalar randomizations.

Performance:

We discuss the performance of the scheme in depth. There are two multipliers, one for each of the pipe stages. The multiplications in the atomic blocks being executed in the pipe stages are computed in parallel. As said earlier, the scheme can be made resistant against DPA, using various randomization techniques. That will require some routine computations. In the discussion below we neglect these routine computations. Also, we will neglect the routine computation required at the end to convert the result from the Jacobian to affine coordinates.

To compute mP, if m has hamming weight h and length n then one has to compute $n-1$ ECDBL and h ECADD. In a sequential execution that will consume $10(n-1) + 11h$ units of time.

In the binary algorithm with the scalar multiplier m expressed in binary, $h = n/2$ on the average. An ECDBL operation requires 10 atomic blocks and an ECADD requires 11 atomic blocks to complete. So, the computation of the scalar multiplication requires $10(n-1) + 11(n/2) = 15.5n - 10$ atomic blocks. That is, one has to compute about 15.5 atomic blocks per bit on the average. If m is represented in 160 bits, i.e. $n = 160$, the scalar multiplication can be carried out by executing 2470 atomic blocks or in 2470 units of time.

In NAF representation of the multiplier, $h = n/3$ on the average. So the computation time is $10(n-1) + 11 \times n/3$ time units. That is one has to compute about $13.6n$ atomic blocks or 13.6 atomic blocks per bit of the multiplier. If m is expressed in NAF and $n = 160$, the computation requires to execute 2177 atomic blocks. That is the computation takes 2177 units of time.

The binary methods NAF or ordinary use less memory than our methods. For sake of fairness let us compare the performance of our method with with the method using w-NAF (see [159]). The method requires storing of 2^{w-2} points and $n-1$ doubling and $1/(w+1)$ additions on the average. For a scalar of 160 bits with $w = 5$, the method in a sequential execution requires to store 16 points and computes the scalar multiplication in 1893 units of

Table 7.5: Comparison of Performance for $n = 160$

Algorithm	Binary	NAF	w-NAF ($w = 4$)
Sequential	2477	2177	1893
Pipelined	1438	1278	1152

time. As we will see subsequently it is worse than our scheme applied to binary and NAF methods. If w-NAF technique is used in pipelining scheme with those extra storage, then the scalar multiplication can be computed in only 1120 units of time.

As we can check from the table, in the pipelining scheme, the first EC-operation which is usually an DBL, completes in 10 time units. In fact, as we take the base point in affine coordinates, first three blocks are not necesssary and it needs only 7 blocks. If we use Coron's randomization here 5 more blocks are required for that. After that an EC-operation (be it an DBL or an ADD) completes in every 6 units of time. Let m be represented by n bits with hamming weight h. Then the scalar multiplication will involve $h+n-1$ EC-operations ($n-1$ doublings and h additions). The first doubling will take 7 units of time and the other $n+h-2$ will be computed in 6 units of time in the pipelining scheme. So it will take $7 + 6(n + h - 2) + 5 = 6(n + h)$ units of time. For a scalar multiplier of length n bits represented in binary form, $h = n/2$ on average. Thus the pipelining scheme will require $6(n + n/2) = 9n$ units of time on the average. For $n = 160$ the proposed scheme will take 1440 units of time to compute the scalar multiplication.

If the scalar multiplier is expressed in NAF, then $h = n/3$ on the average. Hence time requirement will be $8n$ time-units. This implies, for $n = 160$ the time required is 1280. In either case it is a speed-up of around 41 percent.

Note that in both cases described above our method is better than even sequential w-NAF method. If w-NAF is used in pipelining scheme with those extra storage, then for $w = 4$, the scalar multiplication can be computed in 1152 units of time. We have summarized this discussion in the Table 7.5.

Comparison with Parallel Implementations

Parallelized computation of scalar multiplication on ECC was described for the first time by Koyama and Tsuruoka in [82]. A special hardware was used to carry out the computation in their proposal. We compare our scheme with some of the recent proposals which are claimed to be SCA resistant. The scheme proposed in [44], uses a parallelized encapsulated-add-and-double algorithm using Montgomery arithmetic. This algorithm uses two multipliers and takes $10[m]$ computations per bit of the scalar. Our algorithm as shown previously with NAF representation of the scalar takes only $8[m]$ computation per bit. The storage requirements are similar. Furthermore, we can obtain additional speed-up by allowing precomputations. In [6], the authors have proposed efficient algorithms for computing the scalar multiplication with SIMD (*Single Instruction Multiple data*). Similar and more efficient algorithms are also proposed in [64]. In [64] the authors have given two proposals. The first proposal, like our scheme, does not use precomputations and takes $1629[m]$ to compute the scalar multiplication. They have taken $[s] = 0.8[m]$ and the cost includes all routine calculation including the cost of Joye-Tymen's countermeasure for DPA. In contrast, pipelining requires only $1319[m]$ (all inclusive). Their second proposal uses precomputed points, applies signed window expansions of the scalar and is quite efficient. However, in a later work with Möller, the same authors (see [66]) remark that using a precomputed table in affine coordinates is not secure against *fixed table attacks*, a differential power attack. Even in Jacobian coordinates while using a fixed precomputed table, the values in the table should always be randomized before use.

7.6 Conclusion and Further Work

The pipelining scheme proposed in this article enhances the computation of scalar multiplication significantly if a multiplier for each pipe stage is allowed. The computation is secure against SCA. It is an interesting idea to see how this technique can be extended to higher genus hyperelliptic curves. The designing of an ECC processor based on this technique is expected to be quite efficient.

Chapter 8

SCA resistant Parallel Explicit Formula for Addition and Doubling of Divisors in the Jacobian of Hyperelliptic Curves of Genus 2

Hyperelliptic curve cryptosystems (HECC) are gradually gaining popularity due to their versatility. For general curves of low genus there is no subexponential algorithm to solve the hyperelliptic curve discrete logarithm problem (HCDLP). So, like ECC, HECC provides a high level of security with much smaller keys compared to RSA. That makes HECC the most suitable cryptosystem for resource constrained devices like smart cards and mobile devices. The advantage of HECC over ECC is that the underlying field in HECC is much smaller. For example, in HECC a genus 2 curve over an underlying field of order 2^{80} can provide as much security as an elliptic curve over an underlying field of order 2^{160}. That makes it suitable for implementation on small devices with limited computing power. Another advantage of HECC over ECC is that in HECC there are many more curves to choose for an implementation. Furthermore, one can choose special curve equations from a sufficiently large variety allowing faster doublings.

Although for suitably chosen system parameters HECC is secure against generic attacks, a straight-forward implementation is vulnerable to side-channel attacks (SCA). Since small devices like smart cards, mobile phones for which HECC is the most ideal cryptosystem, are generally used in hostile outdoor environments, implementation of HECC in those devices must be robust against such attacks.

The most important and costly operation in HECC is the scalar multiplication. Also this is the main operation involving the secret key of the user. Hence computation of scalar multiplication must be secured against SCA. Scalar multiplication is generally computed by a sequence of divisor additions (HCADD) and doublings (HCDBL). These two operations are computed by two different algorithms which differ in complexity and hence in computing time and other attributes perceivable from the side-channel.

Although cost of these group operations have gone down drastically due to extensive research in this field, the operation count reveals that the complexity of addition and doubling differ quite significantly making SCA applicable to a naive implementation. For elliptic curves, various countermeasures have been proposed in literature to resist SCA. The proposal made in [23] has been proved to be most cost effective ones in case of ECC. The computational overhead involved in this countermeasure is almost negligible introducing only a few further additions and no multiplications.

The method divides the point addition and doubling algorithms to small parts, called side-channel atomic blocks, which are indistinguishable from the side-channel. When the scalar multiplication is computed with these addition and doubling algorithms, it becomes a computation of a series of atomic blocks and the side-channel information becomes uniform.

In this chapter, we present the hyperelliptic curve divisor addition and doubling formula divided into side-channel atomic blocks. We use the formulae proposed in [84], [86] are the most efficient formulae for general curves of genus 2. The formulae proposed in [84] for even characteristic involve inversion. Inversions are the most costly field operations. These can not be put in an atomic block. and can easily be identified due to their power trace. Accordingly they do not fit in the small atomic block setting. Therefore following [23], we have made the whole operation one block each. The most efficient addition and doubling formulae for odd characteristic are proposed in [86]. *In the present work we treat multiplication and squaring in fields of odd characteristic to be of same cost as often they both use the multiplication algorithm.* Under that assumption, HCADD involves 44 multiplications and HCDBL requires 41. Hence we have divided them into 44 and 41 atomic blocks respectively.

The beauty of our algorithms is that these can be easily implemented in parallel. If the system admits two field element multipliers, each capable of carrying out a multiplication each in parallel, then our algorithms can be implemented in parallel. We state the formula for odd characteristic in a manner such that the blocks with odd index can be processed by one processor and all the blocks with even index by another. For even characteristic we have prescribed the order in which the operations can be computed in parallel. Both schemes are again resistant against SPA. To avoid DPA we propose to use by Avanzi's [9] countermeasures. Namely, one can randomize the point representation in the odd characteristic case as the coordinates are projective. In even characteristic we can use a transformation to an isomorphic curve as we do not assume any special property of the curve.

8.1 Our Methodology

First of all we mention here that the task at our hand is very much implementation dependent. We concentrate on curves of genus 2 and choose different coordinate systems for even and odd characteristic. In characteristic 2, inversions can be computed rather efficiently, hence, we stick to affine coordinates [84], especially since the inversion free systems trade the one inversion per group operation for several multiplication, e. g. the cost for an addition with inversion is $1[i] + 22[m] + 3[s]$ and that without inversion is $38[m] + 6[s]$. In odd characteristic we deal with the new coordinates [86] as they are most efficient. We assume the input to be affine, therefore we can use mixed coordinates.

In even characteristic h_2 can always be made $h_2 \in \{0, 1\}$. Hence, the multiplications with h_2 are not counted; for an actual implementation with fixed parameters one can insert the values. Furthermore, some other coefficients might be zero – in this case the respective steps are replaced by dummy operation. Note, that the use of special forms of curves (see e. g. [133]) allows to obtain faster doublings, however, we tried to stay as general as possible providing universally applicable formulae. The only restriction we make is to assume $f_4 = 0$, which can be guaranteed if the characteristic of the underlying field is not 5.

Our aim in this work is to devise parallelizable addition and doubling formula in atomic blocks. As the formulae in even characteristic involves inversion, we go for making the whole operation of addition only one block and similar one block for doubling. As a pre-processing we divided the formula into three-address codes like $p = q \; op \; r$, where op is one of the field operations $\{+, -, *, /\}$. Also we eliminated the reuse of variable names. Thus after pre-processing each explicit formula becomes a sequence of three address codes. The left hand side of each code being unique.

8.1.1 Even Characteristic

The complexities of the divisor addition and doubling for curves over fields of even characteristic are $1[1] + 21[m] + 3[s]$ and $1[1] + 22[m] + 5[s]$ besides some additions each. In binary fields the cost of inversion is seen to be 8 to 10 times costlier than multiplication. The squarings are quite cheaper than multiplication, but quite costlier than addition. Hence the atomic blocks corresponding to addition and doubling must match perfectly, i.e. they must have the same operations at the corresponding positions.

The inversions must be placed at the same locations in both addition and doubling. Let us divide each block into two parts each. We name the part before the inversion in addition to be A_1 and that following inversion to be A_2. Similar parts for doubling are named as D_1 and D_2. For the atomic blocks to be indistinguishable from the side channel, the cost of A_1 and D_1 must be equal. Also, the cost of A_2 and D_2 must be equal. Looking at the explicit formula in [84] we find that $cost(A_1) = 9[m] + 1[s]$ and $cost(D_1) = 11[m] + 2[s]$. So a natural way of attaining $cost(A_1) = cost(D_1)$ is to add 2 dummy multiplications and 1 dummy squaring in A_1.

We observe that the inversion step in doubling needs r and s_1. Hence, computation of s_0 can be postponed to a later stage. The computation of s_0 needs one multiplication which can be carried out after the inversion. By doing so we can reduce one dummy multiplication in A_1.

Now, the cost of the parts A_2 and D_2 are $13[m] + 2[s]$ and $12[m] + 3[s]$ respectively. Hence introducing one dummy squaring to A_2 and a dummy multiplication to D_2 we can make their costs equal. In Tables C.1, C.2 (Appendix C) we have shown HCADD and HCDBL algorithms for even characteristic as one atomic block each. In these tables, we have denoted dummy additions, squarings and multiplications by the notations $*[a]$, $*[s]$, $*[m]$ respectively.

In the tables there are expressions involving h_2T_k; as $h_2 \in \{0, 1\}$, this either means 0 or T_k. Additions involving the curve coefficients might turn out to be dummy additions if the respective coefficient is 0. If $h_1 = 0$, step 50 can be skipped completely, making the dummy multiplication in HCDBL redundant.

Notice that both the operations HCADD and HCDBL now have the same number and same order of field operations. Hence, they are indistinguishable from the side channel. They can be safely used in binary algorithm for computing the scalar multiplication. In Subsection 8.2, we will show how they can be computed in parallel by two multipliers without compromising security.

8.1.2 Odd Characteristic

For explicit formulae for curves over fields of odd characteristic, mixed addition involves $38[m] + 6[s]$ and doubling involves $34[m] + 7[s]$. We consider the mixed addition formula in which the one input device is in affine coordinates and the other in Lange's "new" coordinates. We assume that multiplication and squaring in fields of odd characteristics are indistinguishable from the side-channel. So we may assume divisor addition requires $44[m]$ and doubling requires $41[m]$. Following [23] we design our atomic blocks to contain one multiplication, two additions and a negation as an aid to subtraction.

We wish to split the explicit formulae into atomic blocks which will easily conform to a parallel implementation as well.

In [109] the we have proposed a general methodology to parallelize any explicit formula. The proposed methodology requires to create a precedence digraph of the multiplication operations. The vertices represent the multiplication operations. There is one arc from a vertex v_1 to v_2 if the later can be computed after the first one without any intermediate multiplication operation.

In the situation we are in now, one feels tempted to split the explicit formulae into atomic blocks first and then go for the said methodology treating one atomic block as one operation in order to create a precedence graph of atomic blocks. One important observation of our study is that such a strategy may not bear fruit. That is simply because one can split the HCADD and HCDBL into atomic operation in several ways, hence the precedence graph is not unique. Therefore we use the other strategy, i. e. we start by using the methodology of [109] to find out the multiplication operations which can be implemented in parallel. Put them in different atomic blocks and then distribute the addition and subtraction operation suitably among them. We take care such that while two atomic blocks are being processed in parallel no conflict occurs. We say a conflict has occurred under the two situations described below:

- Two processors processing two atomic blocks in parallel try to write the same location at the same time.
- Or, while one is writing one location the other processor is trying to read the same location.

We present our results for even and odd characteristics in Appendix C. For odd characteristic, we present the HCDBL algorithm split into atomic blocks in three tables, namely Table C.3, Table C.4, Table C.5 and HCADD (mixed coordinates) is presented in Table C.6, Table C.7 and Table C.7. One can observe that these explicit formula can be processed in parallel. One processor can process the the atomic blocks indexed by odd numbers while the other can process the blocks with even index. No conflict will take place.

8.1.3 Memory Requirement

Memory requirement is an important consideration for an algorithm which is likely to be used in resource constrained devices. In even characteristic, as can be seen in the tables, HCADD uses 14 registers and HCDBL uses 11. Note that in HCADD we use the registers T_1, T_2, T_3, T_4 to store the base divisor, which is a common argument to all HCADD operations invoked during the computation of scalar multiplication. Therefore, we can not use these registers to store intermediate variables. Besides those we need at most 5 registers to store non-zero curve parameters h_0, h_1, h_2, f_2, f_3. Also we need 1 register to implement dummy operations.

Thus for even characteristic 20 registers are required to compute the scalar multiplication. In odd characteristic HCADD uses 23 registers and HCDBL uses 17. Again in HCADD we do not store intermediate values in 4 registers which we reserve for the base divisor in affine coordinates. Only two curve coefficients are needed to be stored, namely f_2, f_3. We need 1 register for dummy operations. Hence, a total of 26 registers are required to implement the scalar multiplication. Note that we have used more registers to enable the algorithm to be implemented in parallel. To avoid conflicts, we do not use a register used by the atomic block Γ_i in Γ_{i+1} for odd i, even though it is free. HCADD and HCDBL designed for sequential implementation would require slightly smaller number of registers.

It is worthwhile to note that the registers are around 80 bits here, capable of holding one field element. In ECC, the registers have double size, showing that in relation we do not need much more space while achieving not only SPA resistance but also parallel executable formulae.

8.2 Parallel Implementation

A system having two multipliers and two adders and hardware for other field operations (one each) can implement our algorithm. Design of one such hyperelliptic parallel coprocessor has been recently proposed in [15]. The same architecture except for the scheduler will work nicely for our scheme, too. As addition is an inexpensive operation in comparison to multiplication, we can use only one adder to minimize the chip area. Our algorithm seems to require some more registers. It can be brought down by choosing a suitable curve. Besides, as the register size is half the size of the ones used for ECC, the memory requirement is not as high as it seems from the register count.

As mentioned earlier the formulae for odd characteristic can be easily implemented in parallel. One set of multiplier and adder can be engaged in computing the atomic blocks with odd subscript (e. g. $\Gamma_1, \Gamma_3, \cdots$ in HCADD), while the other set of multiplier and adder can be assigned the atomic blocks with even indices (e. g. $\Gamma_2, \Gamma_4, \cdots$ in HCADD).

For even characteristic we have described in Table 8.1 a novel way of executing the algorithm in parallel. In the table M_1 and M_2 are the two multipliers. We propose to compute the additions sequentially by an adder. In the Op Code column we have explicitly mentioned which multiplier will carry out which multiplication. The multiplications are referred to by the serial number assigned to them in the HCADD and HCDBL (even characteristics) algorithm presented as single atomic block each in Appendix C. As can be seen in the table, both the algorithms HCADD and HCDBL can be executed in 31 rounds of computation including 16 multiplication rounds and one inversion round. In the operation column we mention which operation is to be carried out in a specific round. Note that if any one of HCADD and HCDBL is implemented using the Table 8.1, the same pattern of computation is followed, namely, $2[a], 4[m], 1[a], 1[m], 6[a], 2[m], 2[a], 2[m], 3[a], 2[m], 2[a], 2[m], 1[i], 4[m], 1[a], 2[m],$

Table 8.1: Parallel Implementation in Even Characteristic

Rnd	Operation	Op Code	Rnd	Operation	Op Code
1	+	1, 2	17	+	36
2	*	M_1: 3, M_2 4	18	*	M_1: 34, M_2 37
3	*	M_1: 5, M_2 6	19	+	38
4	+	7	20	*	M_1: 39, M_2 43
5	*	M_1: 8, M_2 -	21	+	40, 41, 42
6	+	9,10,11,12,13,14	22	*	M_1: 44, M_2 46
7	*	M_1: 15, M_2 16	23	+	45, 47, 48, 49
8	+	17, 18	24	*	M_1: 50, M_2 -
9	*	M_1: 19, M_2 20	25	+	51, 52, 53
10	+	21, 22, 25	26	*	M_1: 54, M_2 61
11	*	M_1: 23, M_2 26	27	+	55, 56
12	+	24, 27	28	*	M_1: 57, M_2 -
13	*	M_1: 28, M_2 31	29	+	58, 59, 60, 62
14	Inversion 29				
15	*	M_1: 30, M_2 32	30	*	M_1: 63, M_2 -
16	*	M_1: 33, M_2 35	31	+	64, 65, 66

$1[a], 2[m], 3[a], 2[m], 4[a], 1[m], 3[a], 2[m], 2[a], 1[m], 4[a], 1[m], 3[a]$. Hence, the parallel computation is also SPA resistant.

8.3 Conclusion

In this chapter we have presented the divisor addition and doubling formulae for hyperelliptic curves of genus 2 in side-channel atomic blocks. The most efficient formulae presented in [84, 86] have been used as the basic formulae. An implementation using these set of formulae is resistant against simple power attacks. Any countermeasure against DPA can be incorporated to it to make it SCA resistant. The presented algorithms are easily parallelizable and a parallel implementation is also expected to be SPA resistant.

Chapter 9

New Table Look-up Methods for Faster Frobenius Map Based Scalar Multiplication Over $GF(p^n)$

We describe a new scalar multiplication algorithm for elliptic and hyperelliptic curve cryptosystems. The algorithm is obtained by combining Koblitz's idea of using Frobenius automorphism along with a very special kind of look-up table. In the case where the base point is unknown, we present an efficient algorithm to compute the look-up table online. Our algorithm applies to prime power fields $GF(p^n)$. One important subclass of such fields are Optimal Extension Fields (OEF's) which are believed to be ideal for efficient implementation of cryptographic primitives. Over prime power fields, our algorithm compares favourably to other known algorithms for scalar multiplication.

9.1 Introduction

Elliptic and hyperelliptic curves provide a rich source of cyclic groups over which the discrete logarithm problem is believed to be hard. Hence these groups are suitable for defining public key cryptosystems. The dominant operation in any such cryptosystem is the so called *scalar multiplication*, which is the operation of computing mX, where m is an integer and X is either a point of an elliptic curve or a reduced divisor in the Jacobian of a hyperelliptic curve.

The efficiency of an elliptic or hyperelliptic curve cryptosystem is crucially dependent on the speed of scalar multiplication. Not surprisingly, this has led to a tremendous research in algorithms for fast scalar multiplication. These algorithms fall naturally into two classes.

- General algorithms which work for any cyclic group.
- Algorithms which exploit the algebraic properties of elliptic and hyperelliptic curves.

One of the most important technique of the second kind is the use of endomorphisms to speed up scalar multiplication. This was first proposed by Koblitz [78] and has also been studied by later authors (for example see [26, 30, 50, 75, 83, 159, 160]). The most natural endomorphism is the Frobenius automorphism and was initially proposed by Koblitz [78]. A series of research papers have resulted in the applicability of the Frobenius map technique to elliptic and hyperelliptic curves over any finite field. See [83] for a more detailed description of the development of this technique.

Let F_q be the underlying field. The Frobenius map technique (and hence our algorithm) really applies when the q is a prime power (rather than a prime). The case $p = 2$ has been explored extensively by the researcher community. Our algorithms apply to the case $p > 2$, for example, for Optimal Extension Fields. Optimal Extension Fields (OEF's) are finite fields of the form $GF(p^m),\ p > 2$, where p and m are chosen to match the underlying hardware. OEF's, optimally utilizing the underlying hardware offer considerable advantage in software implementations of elliptic curve cryptosystems. In prime power fields, of which OEF's are special cases, our algorithm provides a substantial reduction in the number of point arithmetic (addition/doubling) operations as compared to other existing algorithms.

In this work, we concentrate on developing a new scalar multiplication algorithm based on the Frobenius map. The basic idea of the algorithm is known and has been described for both elliptic curves [75, 160, 159] and hyperelliptic curves [26, 83]. The principle innovation that we introduce is a very special kind of look-up table. Given a point X, we define a look-up table Tab_X in the following manner: The table stores 3^h points and for $(a_0, \ldots, a_{h-1}) \in \{0, \pm 1\}^h$, we define $\mathsf{Tab}_X[a_0, \ldots, a_{h-1}]$ to be the point

$$\mathsf{Tab}_X[a_0, \ldots, a_{h-1}] \quad = \quad a_0 X + a_1 \phi(X) + \cdots + a_{h-1}\phi^{h-1}(X) \qquad (9.1)$$

where ϕ is the Frobenius map. This is a simple idea but does not seem to have occurred in the literature before. A proper utilization of this idea provides a substantial reduction in the number of point arithmetic operations.We also extend the idea to the situation when $\{a_0, \ldots, a_{h-1}\} \in \{0, \pm 1, \pm 2, \ldots, \pm 2^w\}^h$ for some $w > 0$. The table can be precomputed and stored when the point X is known in advance. However, there are applications where the point X is not known in advance. We present an algorithm to compute Tab_X in such a situation. It turns out that by using a simple trick, it is possible to reduce the number of point additions needed to compute Tab_X.

The size of the look-up table is determined by the number of points to be stored. In the case, where the underlying field is represented using normal basis, the number of points required to be stored is quite small. However, if polynomial basis representation of the field elements are used, then the storage requirement increases. Thus our algorithm is most useful when the underlying field is represented using normal basis.

9.2 Preliminaries

In this chapter again, by a *point* we will mean either a point on an elliptic curve or a reduced divisor of an hyperelliptic curve. The main operation for realizing elliptic and hyperelliptic curve cryptosystems is computing mX, where m is an integer and X is a point. This operation is called *scalar multiplication*. Our focus in this chapter will be to obtain efficient algorithms for scalar multiplication foe curves over GF(p^n) using Frobenius map. In this section we will have an overview of the concepts relevant to the current chapter.

9.2.1 Prime Power Fields

For cryptographic applications, binary fields $GF(2^n)$ and prime fields $GF(p)$ were considered most attractive for software implementations. Later Optimal Extension Fields (OEF's) [11], a special class of finite fields of the form $GF(p^n)$ were proposed, where p and n were chosen suitably to exploit the underlying hardware optimally for performance gain. An OEF is a finite field of the form $GF(p^n)$, where (i) p is a pseudo-Mersenne prime and (ii) an irreducible binomial $P(x) = x^n - \omega$ exists over $GF(p)$. The prime p is generally chosen to be very close to the word size of the processor, so that each machine word can accommodate one element of the subfield $GF(p)$ and each element of the OEF $GF(p^n)$, can be accommodated in n words, with minimum wastage of memory. Also, OEF's allow efficient modular reduction for arithmetic in the extension field. The algorithms proposed in this work are suitable for the prime power fields of type $GF(p^n)$, which contains the OEF's as a subclass of it.

9.2.2 Normal Basis

Let q be a prime power. A field F_{q^n} is said to have a normal basis if it has a basis (over F_q) of the form $\{\alpha, \alpha^q, \cdots, \alpha^{q^{n-1}}\}$. Any element of the field can be represented as $x = \sum_{j=0}^{n-1} a_j \alpha^{q^j}$ or briefly as an ordered n-tuple $x = (a_0, \cdots, a_{n-1})$. In the field F_{q^n}, we have, $x^q = (\sum_{j=0}^{n-1} a_j \alpha^{q^j})^q = (\sum_{j=0}^{n-1} a_j \alpha^{q^{j+1}})$. Thus if x is represented by the tuple $(a_0, \cdots, a_{n-1})$ then x^q is represented by $(a_{n-1}, a_0, \cdots, a_{n-2})$, as $\alpha^{q^n} = 1$. With a normal basis representation of elements, x^q can be computed from x by a circular shift operation only. See [92] for more details on normal basis.

9.2.3 Frobenius Map

Let F_q be a finite field. The Frobenius map $\phi : F_{q^n} \rightarrow F_{q^n}$ is an automorphism of F_{q^n} and is defined as $\phi(x) = x^q$. The map is extended to points of an elliptic or hyperelliptic curve over F_{q^n} in the following manner: A point of an elliptic curve is represented using a pair of elements of F_{q^n}; similarly a reduced divisor of a hyperelliptic curve is represented using a tuple of elements of F_{q^n}. An application of the Frobenius map to a point is to actually apply the map individually to the field elements which represent the point. We note that ϕ^n is the identity map on F_{q^n}. If the field F_{q^n} is represented using a normal basis, then the computation of $\phi(x)$ is "for free". Further, as observed in [160, 159], in the case $q = 2$, the Frobenius map is $\phi(x) = x^2$ and hence can be computed using a field squaring which is a relatively cheap operation even if polynomial basis representation of elements is used.

9.2.4 Scalar Multiplication using Frobenius Map

In [78], Koblitz had suggested the use of Frobenius map to speed up scalar multiplication algorithm. This idea has later been developed by several authors [30, 50, 75, 118, 156, 159]. For hyperelliptic curves, it has been shown [83, 26] that the Frobenius map based method can be used over any field of finite characteristic.

Let q be a prime power, F_q be the finite field of order q and F_{q^n} an extension field of F_q. Let C be the curve of genus g to be used for the cryptosystem and we consider the F_{q^n}-rational points of C. Let ϕ be the Frobenius map from F_{q^n} to F_{q^n}. Let m be an integer, X a point (either a point of an elliptic curve or a reduced divisor of a hyperelliptic curve) and we wish to compute mX. The base-ϕ expansion of m is $\sum_{i=0}^{n-1} u_i \phi^i$, where under reasonable assumptions each u_i is an integer in the range $[-q^g, q^g]$. It is possible to obtain the base-ϕ expansion of m. Next we define some additional parameters which will be required in the rest of the chapter.

1. $A = \max\lfloor \log_2(|u_i|) \rfloor$.
2. For $i \in \{0, \ldots, n-1\}$ write $|u_i| = \sum_{j=0}^{A} u'_{i,j} 2^i$, where $u'_{i,j} \in \{0, 1\}$.
3. $u_{i,j} = \mathsf{sgn}(u_i) u'_{i,j}$, where $\mathsf{sgn}(u_i)$ is the sign of u_i.
4. For $0 \leq i \leq n-1$, define $X_0 = X$ and $X_i = \phi^i(X_0) = \phi^i(X)$.
5. Parameters h and w are respectively the column and row window sizes.
6. Parameters s and r are defined by the equation:
   $n = s \times h + r$, where r is a unique integer in the set $\{1, \ldots, h\}$.
6. Parameter $k = \lceil (A + 1)/w \rceil$.

The expression mX can be written as

$$\left.\begin{array}{rcl} mX & = & u_0X_0 + u_1X_1 + \cdots + u_{n-1}X_{n-1} \\ & = & (u_{0,0} + u_{0,1}2 + \cdots + u_{0,A}2^A)X_0 \\ & & +(u_{1,0} + u_{1,1}2 + \cdots + u_{1,A}2^A)X_1 \\ & & +\cdots \\ & & +(u_{n-1,0} + u_{n-1,1}2 + \cdots + u_{n-1,A}2^A)X_{n-1} \end{array}\right\} \tag{9.2}$$

We consider the above expression to be an $n \times (A+1)$ matrix. Let $\tau = q^n$. Then depending on the nature of the underlying field F_τ, there are several cases.

1. Case $n = 1$ and $\tau = q$ is a prime: In this case, (9.2) reduces to a single row. In this situation, the Frobenius map based technique does not really apply. Hence we will not consider this kind of fields in this work.

2. Case $n > 1$: In this situation (9.2) will have more than one rows and the Frobenius map technique can be applied. It will be convenient to divide this into two subcases.

    - Subcase $q = 2$: The field is F_{2^n} and the curves are the binary Koblitz curves. In this case each $u_i \in \{0, \pm 1\}$ and hence (9.2) is actually a single column. This is the other extreme to Case 1 above. In [160, 159], equation (9.2) is called the ϕ-adic expansion of m.

    - Subcase $q > 2$: In this situation, (9.2) has a more square shape and again our algorithm offers improvements over existing algorithms.

The following simple algorithm can be used to compute mX from (9.2) (see [26, 75, 83, 159, 160]).

Algorithm 19 The Basic Algorithm

```
Input : integer $m = \sum_{i=0}^{n-1} u_i\phi^i$ and point $X$ .
Output : $mX$.
    1. For $0 \le i \le n-1$ and $0 \le j \le A$, compute $X_i$ and $u_{i,j}$;
    2. Set $Y = \sum_{i=0}^{n-1} u_{i,A}X_i$;
    3. For $j = A-1$ down to 0
    4.        $Y = 2Y$; $Y = Y + \sum_{i=0}^{n-1} u_{i,j}X_i$
    5. return $Y$.
```

Proposition 39 *In the above algorithm, the average numbers of additions and doublings needed to compute mX are $n(A+1)/2$ and A respectively.*

9.3 Basic Table Look-Up Methods

We describe a new table look-up method to compute mX from (9.2). We observe that the right hand side of (9.2) has the structure of a matrix. Algorithm 19 performs a column by column computation. Our first observation is the fact that the number of rows in (9.2) is equal to n (the extension degree of F_{q^n} over F_q) and is independent of both m and X. Given a point X, we define a table Tab_X in the following manner: There are 3^n entries in Tab_X which are indexed by the elements of $\{0, \pm 1\}^n$. For any $(b_0, \ldots, b_{n-1}) \in \{0, \pm 1\}^n$, we define

$$\mathsf{Tab}_X[b_0, \ldots, b_{n-1}] = b_0X_0 + \cdots + b_{n-1}X_{n-1} = b_0X + b_1\phi(X) + \cdots + b_{n-1}\phi^{n-1}(X). \quad (9.3)$$

Hence the look-up table Tab_X stores 3^n points. If this table is available, then computing mX becomes quite easy and is described by the following algorithm.

Algorithm 20 (Our First Scheme)

```
Input : $m = \sum_{i=0}^{n-1} u_i\phi^i$ and point $X$ .
Output : $mX$.
    1. Set $Y = \mathsf{Tab}_X[u_{0,A}, \ldots, u_{n-1,A}]$
    2. For $j = A - 1$ down to 0
    3.      $Y = 2Y$;
    4.      $Y = Y + \mathsf{Tab}_X[u_{0,j}, \ldots, u_{n-1,j}]$;
    5. return $Y$.
```

Proposition 40 *Algorithm 20 computes mX using A additions and A doublings. The table Tab_X stores 3^n points.*

9.3.1 Using Smaller Look-up Tables

In Algorithm 20 we use a table of 3^n points, where n is the extension degree of F_{q^n} over F_q. If n is relatively small (≤ 4), then the table is of moderate size. However, if n is larger, then the required storage space may be prohibitively high. In this section, we show how to tackle this problem.

Let h ($1 \leq h \leq n$) be a small positive integer which is the column window size. Write $n = s \times h + r$, where r is a unique integer from $\{1, \ldots, h\}$. Then for $(a_0, \ldots, a_{n-1}) \in \{0, \pm 1\}^n$

we can write

$$\begin{aligned} a_0X_0 + a_1X_1 + \cdots + a_{n-1}X_{n-1} &= a_0X_0 + a_1X_1 + \cdots + a_{h-1}X_{h-1} \\ &\quad + a_hX_h + a_{h+1}X_{h+1} + \cdots + a_{2h-1}X_{2h-1} \\ &\quad \vdots \\ &\quad + a_{(s-1)h}X_{(s-1)h} + a_{(s-1)h+1}X_{(s-1)h+1} + \cdots + a_{sh-1}X_{sh-1} \\ &\quad + a_{sh}X_{sh} + a_{sh+1}X_{sh+1} + \cdots + a_{sh+r-1}X_{sh+r-1}. \end{aligned}$$

For $0 \le i \le s$, we define a set of tables $\mathsf{Tab}_X^{(i)}$ in the following manner: Define $p = (s+1)h$ and set $X_n = \cdots = X_{p-1} = 0$. Each table $\mathsf{Tab}_X^{(i)}$ stores 3^h points indexed by elements of $\{0, \pm1\}^h$. For $(b_0, \ldots, b_{h-1}) \in \{0, \pm1\}^h$, define

$$\mathsf{Tab}_X^{(i)}(b_0, \ldots, b_{h-1}) = b_0X_{hi} + b_1X_{hi+1} + \cdots + b_{h-1}X_{hi+h-1}. \tag{9.4}$$

Note that $X_n = \cdots = X_{p-1} = 0$ and hence $\mathsf{Tab}_X^{(s)}$ stores only 3^r points.

The following algorithm can now be used to compute mX.

Algorithm 21 (Using a Smaller Look-up Table)

Input *: $m = \sum_{i=0}^{n-1} u_i\phi^i$, $n = s \times h + r$ and point X.*
Output *: mX.*

```
1. Set $Y = \mathsf{Tab}_X^{(s)}[u_{sh,A}, u_{sh+1,A}, \ldots, u_{sh+r-1,A}, 0, \cdots, 0]$;
2. For $i = s-1$ down to 0 set $Y = Y + \mathsf{Tab}_X^{(i)}[u_{ih,A}, u_{ih+1,A}, \ldots, u_{(i+1)h-1,A}]$;
3. For $j = A-1$ down to 0
4.     $Y = 2Y$;
5.     $Y = Y + \mathsf{Tab}_X^{(s)}[u_{sh,j}, u_{sh+1,j}, \ldots, u_{sh+r-1,j}, 0, \cdots, 0]$;
6.     For $i = s-1$ down to 0 set $Y = Y + \mathsf{Tab}_X^{(i)}[u_{ih,j}, u_{ih+1,j}, \ldots, u_{(i+1)h-1,j}]$;
7. End for;
8. return $Y$.
```

Proposition 41 *Algorithm 21 correctly computes mX using $(s+1)A$ additions and A doublings. For $0 \le i \le s-1$, table $\mathsf{Tab}_X^{(i)}$ stores 3^h points and $\mathsf{Tab}_X^{(s)}$ stores 3^r points. Thus the total number of points stored is $s \times 3^h + 3^r$.*

The storage requirement decreases from $3^n = 3^{sh+r}$ points to $s3^h + 3^r$ points. The trade-off is an increase in the number of additions. In the situation where the field F_{q^n} is represented using a normal basis, the storage requirement can be further reduced. This is based on the following observation.

Proposition 42 *For any $(b_0, \ldots, b_{h-1}) \in \{0, \pm 1\}^h$ and $i > 0$ we have,*

$$\mathsf{Tab}_X^{(i)}[b_0, \ldots, b_{h-1}] = \phi^{hi}(\mathsf{Tab}_X^{(0)}[b_0, \ldots, b_{h-1}]).$$

Proof : We compute

$$\begin{aligned}\mathsf{Tab}_X^{(i)}[b_0, \ldots, b_{h-1}] &= b_0 X_{hi} + b_1 X_{hi+1} + \cdots + b_{h-1} X_{hi+h-1} \\ &= b_0 \phi^{hi}(X) + b_1 \phi^{hi+1}(X) + \cdots + \phi^{hi+h-1}(X) \\ &= \phi^{hi}(b_0 X + b_1 \phi(X) + \cdots + b_{h-1}\phi^{h-1}(X)) \\ &= \phi^{hi}(\mathsf{Tab}_X^{(0)}[b_0, b_1, \ldots, b_{h-1}]).\end{aligned}$$

This completes the proof. ■

Since the field is represented using a normal basis, the map ϕ can be computed simply by a circular shift (see Section 9.2.3). Thus instead of storing the $(s+1)$ tables $\mathsf{Tab}_X^{(0)}, \ldots, \mathsf{Tab}_X^{(s)}$ we simply store the table $\mathsf{Tab}_X^{(0)}$ and for $i > 0$ we use Proposition 42 to compute any entry of $\mathsf{Tab}_X^{(i)}$ as and when required. Using this idea we obtain the following improvement.

Proposition 43 *Suppose the field F_{q^n} is represented using a normal basis. Then Algorithm 21 requires to store 3^h points. The numbers of additions and doublings remain the same as Proposition 41.*

9.4 General Table Look-Up Methods

In this section, we present our general table look-up algorithm. Let w $(1 \leq w \leq A+1)$ be a positive integer which is the row window size and set $k = \lceil (A+1)/w \rceil$. We express all u_i occurring in the base-ϕ expansion of m in the base 2^w. In such an expansion we will use the elements of the set $\Omega_w = \{0, \pm 1, \pm 2, \cdots, \pm 2^{w-1}\}$ as digits. Note that since $\{0, \pm 1, \pm 2, \ldots, \pm(2^{w-1} - 1), 2^{w-1}\}$ is a complete system of residues modulo 2^w, any integer m can be represented uniquely in base 2^w using these numbers as digits. The set Ω_w has one extra digit which ensures that the set is closed under negation. Thus, if $u = \sum_{i=0}^{t} a_i 2^{wi}$ then a representation of $-m$ over Ω_w can be obtained by simply negating all the a_i's. For $0 \leq i \leq n-1$, write

$$u_i = c_{i,0} + c_{i,1} 2^w + \cdots + c_{i,k} 2^{wk}, \tag{9.5}$$

where $c_{i,j} \in \Omega_w$. Then

$$\left.\begin{array}{lcl} u_0X_0 & = & (c_{0,0} + c_{0,1}2^w + ... + c_{0,k}2^{wk})X_0 \\ u_1X_1 & = & (c_{1,0} + c_{1,1}2^w + ... + c_{1,k}2^{wk})X_1 \\ \vdots & \vdots & \vdots \\ u_{n-1}X_{n-1} & = & (c_{n-1,0} + c_{n-1,1}2^w + ... + c_{n-1,k}2^{wk})X_{n-1}. \end{array}\right\} \tag{9.6}$$

We have to compute $mX = u_0X_0 + \cdots + u_{n-1}X_{n-1}$. Let h $(1 \leq h \leq n)$ be a small integer (which is the column window size) and write $n = s \times h + r$ where r is a unique integer in the set $\{1, \ldots, h\}$. We define $(s+1)$ tables $\mathsf{Tab}_X^{(0)}, \ldots, \mathsf{Tab}_X^{(s)}$, where each $\mathsf{Tab}_X^{(i)}$ stores $(2^w+1)^h$ points. The entries of $\mathsf{Tab}_X^{(i)}$ are indexed by elements of $\Omega_w^h = \underbrace{\Omega_w \times \cdots \times \Omega_w}_{h}$. For $(a_0, \ldots, a_{h-1}) \in \Omega_w^h$ we define

$$\mathsf{Tab}_X^{(i)}[a_0, \ldots, a_{h-1}] \quad = \quad a_0X_{ih} + a_1X_{ih+1} + \cdots + a_{h-1}X_{(i+1)h-1}. \tag{9.7}$$

Let $p = (s+1)h$ and set $X_n = \cdots = X_{p-1} = 0$. Hence $\mathsf{Tab}_X^{(s)}$ stores only $(2^w+1)^r$ points. With this set of tables at our disposal we can compute mX using Algorithm 22.

Algorithm 22 (Generalized Algorithm)

```
Input : $m = \sum_{i=0}^{n-1} u_i\phi^i$ and point $X$ .
Output : $mX$.
    1. Set $Y = \mathsf{Tab}_X^{(s)}[c_{sh,k}, c_{sh+1,k}, \ldots, c_{n-1,k}, 0, \ldots, 0]$;
    2. For $i = s-1$ down to 0
    3.        $Y = Y + \mathsf{Tab}_X^{(i)}[c_{ih,k}, c_{ih+1,k}, \ldots, c_{(i+1)h-1,k}]$;
    4. For $j = k-1$ down to 0
    5.        $Y = 2^wY$;
    6.        $Y = Y + \mathsf{Tab}_X^{(s)}[c_{sh,j}, c_{sh+1,j}, \ldots, c_{n-1,j}, 0, \ldots, 0]$;
    7.        For $i = s-1$ down to 0
    8.              $Y = Y + \mathsf{Tab}_X^{(i)}[c_{ih,j}, c_{ih+1,j}, \ldots, c_{(i+1)h-1,j}]$;
    9. return $Y$.
```

Proposition 44 *Algorithm 22 correctly computes mX using $(k-1)+ks$ additions and $(k-1)w$ doublings. For $0 \leq i \leq s-1$, table $\mathsf{Tab}_X^{(i)}$ stores $(2^w+1)^h$ points and table $\mathsf{Tab}_X^{(s)}$ stores $(2^w+1)^r$ points. Thus a total of $s(2^w+1)^h + (2^w+1)^r$ points are required to be stored.*

As in Section 9.3.1, the storage requirement can be further reduced if the field F_{q^n} is represented using a normal basis. This is based on the following observation.

Proposition 45 *For any $(a_0, \ldots, a_{h-1}) \in \Omega_w^h$, and $i > 0$, we have*

$$\mathsf{Tab}_X^{(i)}[a_0, \ldots, a_{h-1}] = \phi^{hi}(\mathsf{Tab}_X^{(0)}[a_0, \ldots, a_{h-1}]).$$

Since the field is represented using a normal basis, the map ϕ is easy to compute online. Hence it is sufficient to store only $\mathsf{Tab}_X^{(0)}$ and compute the required entry of $\mathsf{Tab}_X^{(i)}$ as and when required. This gives us the following result.

Proposition 46 *If F_{q^n} is represented using a normal basis, then Algorithm 23 requires to store only $(2^w + 1)^h$ points. The numbers of additions and doublings remain the same as in Proposition 44.*

9.5 Unknown Point

The algorithms described so far use one or more look-up tables. These tables are parametrized by a point X. If the point X is known in advance (as in signature generation for ElGamal algorithms), then the tables can be precomputed and stored. However, there are applications where the point is not known in advance (for example in variants of Diffie-Hellman key agreement protocols). In such a situation, the look-up tables have to be computed online. In this section, we describe algorithms for this task.

We start by describing an algorithm to compute the tables used in Algorithm 21. For this it is sufficient to describe an algorithm to compute $\mathsf{Tab}_X^{(0)}$. The Frobenius map can be used to compute the other tables from $\mathsf{Tab}_X^{(0)}$. Let X be a point and we wish to compute the table $\mathsf{Tab}_X^{(0)}$ having 3^h entries and indexed by the elements of the set $\{0, \pm 1\}^h$. For any vector $\alpha \in \{0, \pm 1\}^l$, we define $-\alpha$ to be the vector obtained from α by negating all the components of α. The following algorithm computes $\mathsf{Tab}_X^{(0)}$.

Algorithm 23 (Computation of $\mathsf{Tab}_X^{(0)}$ for Unknown Point)

Input : X.
Output : $\mathsf{Tab}_X^{(0)}$ used in Algorithm 21.

```
1. Compute $X_0, \ldots, X_{h-1}$.
2. $\mathsf{Tab}_X^{(0)}[0, 0, \ldots, 0] = 0$; $\mathsf{Tab}_X^{(0)}[1, 0, \ldots, 0] = X$; $\mathsf{Tab}_X^{(0)}[-1, 0, \ldots, 0] = -X$;
3. For $l = 1$ to $h - 2$
4.     For $\alpha \in \{0, \pm 1\}^l$ set $\mathsf{Tab}_X^{(0)}[\alpha, 1, \ldots, 0] = X_{l+1} + \mathsf{Tab}_X[\alpha, 0, \ldots, 0]$;
```

```
5.      For $\alpha \in \{0, \pm 1\}^l$ set $\mathsf{Tab}_X^{(0)}[\alpha, -1, \ldots, 0] = -\mathsf{Tab}_X[-\alpha, 1, \ldots, 0]$;
6. End.
```

Proposition 47 *Algorithm 23 correctly computes* $\mathsf{Tab}_X^{(0)}$ *used in Algorithm 21 using* $\frac{1}{2}(3^h - 3)$ *point additions and* $(h-1)$ *Frobenius map computations. The tables* $\mathsf{Tab}_X^{(1)}, \ldots, \mathsf{Tab}_X^{(s)}$ *used in Algorithm 21 can be computed from* $\mathsf{Tab}_X^{(0)}$ *using* $sh3^h$ *Frobenius map computations.*

Proof : First we prove the correctness. Let $\beta \in \{0, \pm 1\}$. If $\beta = (0, \ldots, 0)$, then clearly Algorithm 23 computes $\mathsf{Tab}_X^{(0)}[\beta] = 0$. So assume that $\beta \neq (0, \ldots, 0)$ and write $\beta = (\alpha, b, 0, \ldots, 0)$, where $b \neq 0$ and $\alpha \in \{0, \pm 1\}^l$ for some $l \geq 0$. We show that $\mathsf{Tab}_X^{(0)}[\beta]$ is computed correctly. If $b = 1$, we have by definition $\mathsf{Tab}_X^{(0)}[\beta] = \langle \alpha, (X_0, \ldots, X_l) \rangle + X_{l+1} = \mathsf{Tab}_X^{(0)}[\alpha, 0, \ldots, 0] + X_{l+1}$, where $\langle \rangle$ denotes the usual inner product. On the other hand, if $b = -1$, then

$$\begin{aligned} \mathsf{Tab}_X^{(0)}[\beta] &= \langle \alpha, (X_0, \ldots, X_l) \rangle - X_{l+1} \\ &= -(-\langle \alpha, (X_0, \ldots, X_l) \rangle + X_{l+1}) \\ &= -(\langle -\alpha, (X_0, \ldots, X_l) \rangle + X_{l+1}) \\ &= -(\mathsf{Tab}_X^{(0)}[-\alpha, 0, \ldots, 0] + X_{l+1}) \\ &= -\mathsf{Tab}_X^{(0)}[-\alpha, 1, \ldots, 0]. \end{aligned}$$

This completes the proof of correctness. Since $X_i = \phi^i(X)$, Step 1 of Algorithm 23 requires $h-1$ applications of the Frobenius map. The number of point additions is clearly $3 + 3^2 + \cdots + 3^{h-1} = \frac{1}{2}(3^h - 3)$.

Note that for any $\beta \in \{0, \pm 1\}^h$, and $i \geq 0$, we have $\mathsf{Tab}_X^{(i+1)}[\beta] = \phi^h(\mathsf{Tab}_X^{(i)}[\beta])$. To compute $\mathsf{Tab}_X^{(i+1)}$ from $\mathsf{Tab}_X^{(i)}$ we need $h3^h$ computations of the Frobenius map. Since s tables have to be computed, a total of $sh3^h$ computations of the Frobenius map is required. ■

Note that if the field F_{q^n} is represented using normal basis, then for $i > 0$ the tables $\mathsf{Tab}_X^{(i)}$ need not be stored. Also the Frobenius map computation is essentially "for free".

Now we turn to the problem of computing the set of tables used in Algorithm 22. For $0 \leq i \leq h-1$ and $j \in \Omega_w$, we use the variable $Z_{i,j}$ to store the value of jX_i.

Algorithm 24 (Computing Other Tables)

Input : X;
Output : $\mathsf{Tab}_X^{(0)}$ used in Algorithm 22.

```
1.  For $i = 0$ to $h-1$, set $Z_{i,0} = 0$; $Z_{0,1} = X$;
2.  For $j = 2$ to $2^w - 1$
3.      $Z_{0,j} = Z_{0,j-1} + X$; $Z_{0,-j} = -Z_{0,j}$;
4.  End for;
5.  For $i = 1$ to $h-1$
6.      For $j = 1$ to $2^w - 1$
7.          $Z_{i,j} = \phi(Z_{i-1,j})$; $Z_{i,-j} = -Z_{i,j}$;
8.      End for;
9.  End For;
10. For $j \in \Omega_w$, set $\mathsf{Tab}_X^{(0)}[j, 0, \ldots, 0] = Z_{0,j}$;
11. For $l = 1$ to $h-1$
12.     For $\alpha \in \Omega_w^l$
13.         For $j = 1$ to $2^w - 1$ set $\mathsf{Tab}_X^{(0)}[\alpha, j, 0, \ldots, 0] = Z_{l+1,j} + \mathsf{Tab}_X^{(0)}[\alpha, 0, \ldots, 0]$;
14.         For $j = 1$ to $2^w - 1$ set $\mathsf{Tab}_X^{(0)}[\alpha, -j, 0, \ldots, 0] = -\mathsf{Tab}_X^{(0)}[-\alpha, j, 0, \ldots, 0]$;
15.     End for;
16. End for;
17. End.
```

The following result whose proof is similar to that of Proposition 47 states the correctness and complexity of Algorithm 24.

Proposition 48 *Algorithm 24 correctly computes $\mathsf{Tab}_X^{(0)}$ used in Algorithm 22 using*

$$\left((2^w - 2) + \frac{2^{2w} - 1}{2^w}((2^w + 1)^{h-1} - 1)\right)$$

point additions and $(h-1)(2^w - 1)$ computations of the Frobenius map. Further, the tables $\mathsf{Tab}_X^{(1)}, \ldots, \mathsf{Tab}_X^{(s)}$ used in Algorithm 22 can be computed using an additional $sh(2^w + 1)^h$ computations of the Frobenius map.

9.6 Results and Comparison

In this section, we present detailed results and also compare our algorithm with known scalar multiplication algorithms. At the outset, we would like to point out that the Frobenius map based method (and hence our algorithm) is really useful in the situation where the underlying field is a prime power field (rather than a prime field). Hence all our comparisons are to algorithms which work over prime power fields, in particular Optimal Extension Fields.

We recall the parameters of the algorithms (see Section 9.2.4): n is the field extension degree; h and w are respectively the column and row window sizes; $k = \lceil (A+1)/w \rceil$ and s, r are defined by the equation $n = s \times h + r$, where r is a unique integer from the set $\{1, \ldots, h\}$. Table 9.1 summarizes the results for scalar multiplication using Algorithm 22. Algorithm 21 is obtained from Algorithm 22 by putting $w = 1$. Further, Algorithm 20 is obtained from Algorithm 22 by putting $w = 1$ and $h = n$.

The first two columns of Table 9.1 gives the numbers of additions and doublings required. The third column gives the number of points required to be stored when normal basis is used and the fourth column gives the number of points required to be stored when standard (or polynomial) basis is used. Table 9.2 summarizes the result for Algorithm 24. The first half of Table 9.2 gives the numbers of additions and Frobenius map computations required to prepare the table $\mathsf{Tab}_X^{(0)}$ and the second half gives the additional number of Frobenius map computations required to prepare the tables $\mathsf{Tab}_X^{(1)}, \ldots, \mathsf{Tab}_X^{(s)}$. Note that Algorithm 23 can be obtained from Algorithm 24 by setting $w = 1$. Let us denote the numbers of point additions and point doublings by $\mathbf{A}$ and $\mathbf{D}$ respectively. From Table 9.1, $\mathbf{A} = (k-1) + ks$ and $\mathbf{D} = (k-1)w$. Using the fact that $k = \lceil (A+1)/w \rceil$ and $s \simeq \lfloor n/h \rfloor$ we have the $\mathbf{A} \simeq (n(A+1))/wh$ and $\mathbf{D} \simeq A$. The parameters w and h are respectively the row and column window sizes and hence wh is the size of the $w \times h$ submatrix window.

Table 9.1: Summary of Algorithm 22.

Additions	*Doublings*	*Normal basis*	*Standard basis*
$(k-1) + ks$	$(k-1)w$	$(2^w+1)^h$	$s(2^w+1)^h + (2^w+1)^r$

Table 9.2: Summary of Algorithm 24.

$\mathsf{Tab}_X^{(0)}$		$\mathsf{Tab}_X^{(1)}, \ldots, \mathsf{Tab}_X^{(s)}$
Additions	*Frobenius map*	*Frobenius map*
$(2^w - 2) + \frac{(2^{2w}-1)}{2^w}\big((2^w+1)^{h-1} - 1\big)$	$(h-1)(2^w-1)$	$sh(2^w+1)^h$

Define $t = n(A+1)$. Then the total number of bits required to represent the integer m in binary is $\simeq t$. The usual binary add-and-double algorithm requires t point doublings and on an average $(t/2)$ add point additions. A more efficient algorithm uses a non adjacent form (NAF) representation of m [159]. A window method using NAF and look-up table requires $t/(\omega+1)$ additions and t doublings while storing $2^{\omega-2}$ points, where ω is the window size (see [45]). The basic Frobenius map based algorithm (Algorithm 6 in Chapter 2) requires A doublings and $t/2$ additions. These results are summarized in Table 9.3 which clearly show the superiority of Algorithm 22 over the other algorithms. Algorithm 22 achieves the

Table 9.3: Comparison with other algorithms.

Algorithm	*additions*	*doublings*
Binary	max $(t-1)$; avg $(t/2)$	t
ω-NAF [45]	$t/(\omega+1)$	t
Algorithm 6	max $(t-1)$; avg $(t/2)$	A
Algorithm 22	$t/(wh)$	A

speed-up by using a look-up table. This look-up table can either be precomputed or can be computed online. Further, depending on the basis representation of the underlying field, the amount of storage space can vary.

In Table 9.4, we present results of storage and computational requirements under various conditions. The values in Table 9.4 clearly shows that the storage and computational requirements of the look-up tables can vary. For example, in the situation $n = 4$, $h = 2$ and $w = 1$, it is sufficient to work with total storage space for 18 points (9 if normal basis is used). Only 3 point additions and 19 Frobenius map computations are required to compute the tables. The number of additions required for scalar multiplications is approximately $(t/wh) = t/2$ and the number of doublings is approximately A. This is better than the binary method. On the other hand, for $n = 12$, $h = 2$ and $w = 4$, using normal basis representation and storage for 289 points, the number of additions in the scalar multiplication can be brought down to $t/(wh) = t/8$. A total of 269 point additions are required to compute the tables in this situation. Thus Algorithm 22 provides a wide choice of trade-offs between storage space and efficiency of scalar multiplication.

9.7 Conclusion and Further Research

We have described a new table look-up algorithm for performing Frobenius map based scalar multiplication for elliptic and hyperelliptic curve cryptosystems over prime power fields. The algorithm compares favourably with previous Frobenius map based algorithms and other scalar multiplication algorithms. Note that, we have not used NAF representation of the multiplier in our algorithm. Using NAF will further increase the efficiency of the proposed algorithm. Also, recently in [128, 129] more general techniques have been proposed for utilizing the Frobenius map. It is an interesting work to see how the algorithm proposed in this

Table 9.4: Storage and computation requirements of the look-up tables.

Parameters			*storage requirements*		*computational requirements*		
n	h	w	*normal basis*	*standard basis*	$\mathsf{Tab}_X^{(0)}$		$\mathsf{Tab}_X^{(i)}$ for $i > 0$
					Additions	*Frob*	*Frob*
4	4	1	81	81	39	3	0
		2	625	625	467	9	0
	2	1	9	18	3	1	18
		2	25	50	17	3	50
		3	81	162	69	7	162
8	4	1	81	162	39	3	324
		2	625	1250	467	9	2500
	2	1	9	36	3	1	54
		2	25	100	17	3	150
		3	81	324	69	7	486
12	4	1	81	243	39	3	648
		2	625	1875	467	9	5000
	2	1	9	54	3	1	90
		2	25	150	17	3	250
		3	81	486	69	7	810
		4	289	1734	269	15	2890

work can be modified to suit the generalization and how much of performance enhancement can be achieved. In conclusion, it can be said that our algorithm (or a version modified to suit the general scenario) is a serious contender for implementing scalar multiplication for elliptic and hyperelliptic curve cryptosystems.

Chapter 10

Open Problems and Suggestions

This area of research has many open problems whose solutions have ample practical value. The discovery of side-channel attacks has thrown an open challenge to cryptographic community. Although many attacks and countermeasures have been proposed in literature, no general security model has yet been developed. Besides, combining side-channels is another challenging avenue which has not been explored fully. In fact, our studies has also opened many new avenues to work on. They are not only interesting from a theoretical point of view, but also are of much practical significance. We list a few of them below.

- The parallelization idea described in Chapter 3 can be extended to curves of higher genus and also to Picard Curves. Explicit formula for addition and doubling of divisors over such curves are already available in literature.
- Although our register minimization technique in Chapter 4 produces minimum number of registers required for any explicit formula, its output depends upon the nature of the input file. The input file is generally a sequence of three address codes. There is a vast literature in compiler construction studies on efficient methods for converting an arithmetic formula into three address codes. Our parsing program which converted the explicit formulae into three address codes may not be the optimal one. Therefore there is still some scope for improvement.
- The parallel algorithm presented in Chapter 6 is expected to be suitable for HECC also.
- The pipelining idea (Chapter 7) can be extended to hyperelliptic curves. Also, one can develop a very efficient (hyper)elliptic curve coprocessor which utilizes this idea in hardware. Besides, the 2 stage pipelining scheme proposed in Chapter 7 can be integrated to more fancier and more efficient ideas (like point tripling, computing $2P + Q$

or Window based methods) leading to more efficient implementations. It is worthwhile investigating the performance of the scheme with pre-computations. Also, instead of using side-channel atomic blocks to thwart SPA one can investigate performance and security aspects of the scheme using using other SPA countermeasures. The scheme is proposed for curves over fields of characteristics > 3. We believe that the same idea can also be used for curves over fields of characteristics 2 and 3. Also, one can develop algorithms based on this idea to be utilized in more advanced processors having SIMD capabilities.

- The addition and doubling algorithm in side-channel atomic blocks presented in Chapter 8 are capable of being implemented in parallel by two processors. Can it be extended to 4 or more processors?
- Many efficiently computable endomorphisms have been identified which are capable of enhancing the efficiency of computation of scalar multiplication in ECC and HECC. It is an interesting idea to see how the new table look up method described in Chapter 9 can be extended to these endomorphisms.

The dissertation consists of several methods of computing the scalar multiplication. We conclude the thesis with suggestions as to which method will be most suitable under a given circumstance.

The *I/M ratio* has been reported to be between 8 and 10 over binary fields and between 30 to 50 over prime fields [45]. So over binary fields, affine arithmetic is preferable to inversion-free arithmetic for both ECC and HECC. For a sequential implementation of HECC, the algorithm presented in Chapter 5 is preferable. For a parallel implementation with 2 multipliers, the SPA resistant algorithms for addition and doubling presented in Chapter 8 has edge over others. For a parallel implementation with more than 2 processors one can go for the parallel point arithmetic presented in Chapter 3. One can use the encapsulated add and double formulae presented there or can adopt other measures to thwart simple power attacks. Adequate measures has to be incorporated to immunize the implementation against differential power attacks.

For a sequential implementation over prime fields, one can use a pair of addition and doubling formula presented in Chapter 4 as it uses minimum amount of memory. It can be incorporated to the efficient scalar multiplication algorithm presented in Chapter 5. For a parallel implementation with 2 multipliers, the SPA resistant algorithms presented in Chapter 8 can be used. When there is scope for utilizing more than 2 multipliers, inversion-free arithmetic presented in Chapter 3 is preferable. It has been shown in Chapter 3 that in a parallel implementation with more than 2 multipliers, inversion-free arithmetic is preferable to affine.

For a sequential implementation of ECC over arbitrary fields our algorithm presented in Chapter 5 is quite efficient. However it requires more memory and not suitable for small devices. For an implementation with 2 multipliers over prime fields, the pipelining scheme is the most attractive one. For a parallel implementation over binary fields or over prime fields when more than 2 multipliers are available, our algorithms presented in Chapter 6 can be used.

Bibliography

[1] A compendium of NP-optimization problems.

`http://www.nada.kth.se/~viggo/problemlist/compendium.html`

[2] L. Addleman, J. DeMarrais and M. Huang. A subexponential algorithm for discrete logarithm over the rational subgroup of the Jacobians of large genus hyperelliptic curves over finite fields. In *Algebraic Number Theory*, LNCS 877, pp 28-40, Springer-Verlag, 1994.

[3] L. Addleman and M. Huang. Primality testing and Abelian Varieties over Prime Fields. In *Lecture Notes in Coputer Science*, 1512, Springer-Verlag, 1992.

[4] G. Agnew, R. Mullin, I. Onyszchuk, and S. Vanstone. An Implementation for a Fast Public Key Cryptosystem. In *Journal of Cryptology*, Vol 3, 1991, pp 63-79.

[5] D. Agrawal, B. Archambeault, J. Rao, P. Rahotgi. The EM side-channel(s). In *CHES 2002*, LNCS 2523, pp 29-45, Springer-Verlag, 2002.

[6] K. Aoki, F. Hoshino, T. Kobayashi and H. Oguro. Elliptic Curve Arithmetic Using SIMD. In *ISC, 2001*, LNCS 2200, pp. 235-247, Springer-Verlag, 2001.

[7] Details of the project available at `http://www.arehcc.com`.

[8] S. Arno and F. S. Wheeler. Signed Digit representation of Minimal Hamming Weight, In *IEEE Trans on Computers*, 42(8), pp 1007-1009, 1993.

[9] R. M. Avanzi. Countermeasures Against Differential Power Analysis for Hyperelliptic Curve Cryptosystems. In *CHES 2003*, LNCS 2779, pages 366- 381, Springer-Verlag, 2003.

[10] R. M. Avanzi. On Multi-exponentiation in Cryptography. Tech Report 2002/154, Cryptology e-Print Archive. Available at `http://eprint.iacr.org/2002/154`

[11] D. V. Bailey and C. Paar. Optimal Extensions Fields for Fast Arithmetic in Public-key Algorithms. In *Crypto'98*, pages 472–485, LNCS 1462, 1998.

[12] D. V. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Application to Elliptic Curve Cryptography In *Journal of Cryptology*, 14: 153-176, 2001.

[13] R. Balasubramaniam and N. Koblitz. The Improbability that an Elliptic Curve has Subexponential Discrete Log Problem under the Menezes-Okamoto-Vanstone Algorithm. *Journal of Cryptology*, 11: 141-145, 1998.

[14] F. Bao, R. H. Deng, Y. Han, A. B. Jeng,A. D. Narasinghalu and T-H. Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults In *Security Protocols Workshop, 1997*, LNCS 1361, pp 115-124, Springer, 1997.

[15] G. Bertoni, L. Breveglieri, T. Wollinger and C. Paar. Finding Optimum Parallel Coprocessor Design for Genus 2 Hyperelliptic Curve Cryptosystems. Cryptology ePrint Archive, Report 2004/29, 2004.

[16] I. Biehl, B. Meyer and V. Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In Crypto'2000 LNCS 1880, pp 131-146, Springer-Verlag, 2000.

[17] O. Billet and M. Joye. The Jacobi Model of an Elliptic Curve and Side-channel Analysis In *Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, Vol 2643, 2003.

[18] I. F. Blake, G. Seroussi, N. Smart *Elliptic Curve Cryptography*, Vol 265 of London Mathematical Society, Cambridge University Press, 2000.

[19] D. Le Brigand. Decoding of Codes on Hyperelliptic Curve. *Eurocode'90*, LNCS 514, pp 126-134, Springer-Verlag, 1991.

[20] E. Brior and M. Joye. Weierstrass Elliptic Curves and Side-Channel Attacks. In *PKC 2002*, LNCS 2274, pages 335-345, 2002.

[21] M. Brown, D. Hankerson, J. Lopez and A. Menezes Software Implementation of the NIST Elliptic Curves over Prime Fields. In *Topics in Cryptology, CT-RSA, 2001*, pages 250-265, LNCS 2020, 2001.

[22] D. G. Cantor. Computing in the Jacobian of a Hyperelliptic curve. In *Mathematics of Computation*, volume 48, pages 95-101, 1987.

[23] B. C. -Mames, M. Ciet and M. Joye. *Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity*, *IEEE Trans on Computers*, 53(6):760-768, 2004.

[24] L. Charlap and D. Robbins An Elementary Introduction to Elliptic Curves, CRD Expository Report 31, Centre for Communication Research, Princeton, 1988.

[25] L. Charlap and D. Robbins An Elementary Introduction to Elliptic Curves II, CRD Expository Report 34, Centre for Communication Research, Princeton.

[26] Y. J. Choie and J. W. Lee. Speeding up the Scalar Multiplication in the Jacobian of Hyperelliptic Curves using Frobenius map. *Indocrypt 2002*, LNCS 2551, pp 285–295, Springer Verlag 2002.

[27] D. V. Chudonovski and G. V. Chudonovski. Sequences of Numbers Generated in Formular Groups and New Primality and Factorisation Tests. In *Adv. in Appl. Math.*, 7:385-434, 1987.

[28] M. Ciet. *Aspects of Fast and Secure Arithmetics for Elliptic Curve Cryptography* , Ph.D. Thesis, Louvain-la-Neuve, Belgique, 2003.

[29] M. Ciet and M. Joye. Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults. To appear in *Designs, Codes and Cryptography.*

[30] M. Ciet, T. Lange, F. Sica and J.-J. Quisquater. Improved algorithms for efficient arithmetic on elliptic curves using fast endomorphisms. In *Eurocrypt 2003*, LNCS 2656, pp 388-400, 2003.

[31] W. E. Clark and J. J. Liang. On Arithmetic Weight for a General Radix Representation of Integers. In *IEEE Trans on Information Theory*, 19: 823-826, 1973.

[32] C. Clavier and M. Joye Universal Exponentiation Algorithm - a First Step Towards Provable SPA resistance. In *CHES 2001*, LNCS 2162, pp 300-308, Springer-Verlag, 2001.

[33] H. Cohen, A. Miyaji, and T. Ono. Efficient Elliptic Curve Exponentiation Using Mixed Co-ordinates, In *Asiacrypt'98*, LNCS 1514, pp. 51-65, Springer-Verlag, 1998.

[34] D. Coppersmith. Fast Evaluation of Logarithms in Fields of Characteristic 2. In *IEEE Trans. on Information Theory*, Vol. 30, pp 587-594, 1984.

[35] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, MIT Press, Cambridge, 1997.

[36] J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES 1999*, LNCS 1717, pp 292-302, Springer-Verlag, 1999.

[37] W. Diffie and M. Hellman. New Directions in Cryptography. In IEEE Trans. on Information Theory, 22:644-654, 1976.

[38] S. Duquesne. Montgomery Scalar Multiplication for Genus 2 Curves. To appear at *ANTS 2004*.

[39] I. Duursma, P. Gaudry ans F. Morrain. Speeding up the Discrete Logarithm Computation on Curves with Isomorphisms. in *Asiacrypt'99*, LNCS 1716, pp 103-121, Springer-Verlag, 1999.

[40] T. ElGamal A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms In IEEE Trans. on Information Theory, 31:469-472, 1985

[41] L. Encinas, A. J. Menezes and J. Masque. Isomorphism Classes of Genus-2 Hyperelliptic Curves over Finite Fields. In *Applicable Algebra in Engineering, Communication and Computing*, 13 (2002), 57-65.

[42] A. Enge. *Elliptic Curves and Their Application to Cryptography: An Introduction.* Kluwer Academic Publishers, 1999.

[43] A. Enge. The Extended Euclid Algorithm on Polynomials and the Computational Efficiency of Hyperelliptic Curves, Preprint, Nov 1999.

[44] W. Fischer, C. Giraud, E. W. Knudsen, J. -P. Seifert. Parallel Scalar Multiplication on General Elliptic Curves over $\mathbf{F}_p$ hedged against Non-Differential Side-Channel Attacks, Available at IACR eprint Archive, Technical Report No 2002/007, `http://www.eprint.iacr.org`.

[45] K. Fong, D. Hankerson, J. López and A. Menezes. Field inversion and point halving revisited. Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2003.

[46] G. Frey and H.-G. Ruck. A Remark Concerning m-divisibility and Discrete Logarithm Problem in the Divisor Class Group of Curves. In *Math. of Computation*, 62: 865-874, 1994.

[47] S. Galbraith and N. P. Smart. A Cryptographic Application of Weil Descent. In *Cryptography and Coding*, LNCS 1746, pp 191-200, Springer-Verlag, 1999.

[48] M.R. Garey and D.S. Johnson. Computers and Intractibility: A Guide to the Theory of NP-completeness. W.H. Freeman, San Francisco, 1979.

[49] R. P. Gallant, R. J. Lambert and S. A. Vanstone. Improving the Parallelized Pollard's Lambda Search on Anomalous Binary Curves. *Mathematics of Computation*, 69:1600-1705, 2000.

[50] R. P. Gallant, R. J. Lambert and S. A. Vanstone. Faster Point Multiplication on Elliptic Curves using Efficient Endomorphisms. In *Crypto 2001*, LNCS 2139, pp. 190–200, 2001.

[51] J. M. G. Garcia, R. M. Garcia. Parallel Algorithm for Multiplication on Elliptic Curves. Cryptology ePrint Archive, Report 2002/179, (2002), Available at `http://eprint.iacr.org`

[52] P. Gaudry. An Algorithm for Solving the Discrete Logarithm Problem on Hyperelliptic Curves. In *Eurocrypt 2000*, LNCS 1807, pp 19-34, Springer-Verlag.

[53] P. Gaudry and R. Harley. Counting Points on Hyperelliptic Curves over Finite Fields. In *ANTS IV*, volume 1838 of LNCS; pp 297-312, Berlin, 2000, Springer-Verlag.

[54] P. Gaudry, F. Hess and N. P. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. In *J. of Cryptology*, 15(1): 19-46, 2002.

[55] D. Gordon Discreet logarithm in GF_p using number field sieve. In *SIAM Journal of Discrete Mathematics*, Vol 6, pp 124-138, 1993.

[56] D. M. Gordon. A Survey of Fast Exponentiation Methods. In *Journal of Algorithms*, 27(1): 129-146, 1998.

[57] L. Goubin A Refined Power Analysis Attack against Elliptic Curve Cryptosystems. *PKC 2003*, LNCS 2567, pp 199-211, Springer-Verlag, 2003.

[58] C. Gunther, T. Lange and A. Stein. Speeding Up the Arithmetic on Koblitz Curves of Genus Two, *Selected Areas in Cryptography, SAC 2001*, LNCS, pp. 106–117, 2001.

[59] J. Ha and S. Moon. Randomized Signed Scalar Multiplication of ECC to Resist Power Attcks. In *CHES 2002*, LNCS 2523, pp. 551–562, 2001.

[60] R. Harley. Fast Arithmetic on Genus 2 Curves. *Avaiable at http://cristal.inria.fr/ harley/hyper,2000.*

[61] D. Hankerson, J. L.Hernandez and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *CHES 2000*, LNCS 1964, pp 1-24, Springer-Verlag, 2000.

[62] D. Hankerson, A. Menezes and S. Vanstone *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004.

[63] IEEE Standard 1363-2000, *IEEE Standard Specification for Public Key Cryptography*, Computer Society, August 29, 2000.

[64] T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side-Channel Attacks Technical Report CORR 2002-03, University of Waterloo,2002. Available at http://www.cacr.math.uwaterloo.ca.

[65] T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side-Channel Attacks In *PKC'2002*, LNCS 2274, pp 280-296, Springer-Verlag 2002.

[66] T. Izu, B. Möller and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant Against Side Channel Attacks. *Indocrypt 2002*, LNCS 2551, pp 296-313, Springer-Verlag.

[67] T. Izu and T. Takagi. Fast Elliptic Curve Multiplications with SIMD operation, In *ICICS 2002*, LNCS, pp 217-230, Springer-Verlag.

[68] M. Jacobson, A. J. Menezes and A. Stein. Solving Elliptic Curve Discrete Logarithm Problems Using Weil Descent. *Journal of the Ramanujan Mathematical Society*, 16:231-260, 2001.

[69] M. Joye and C. Tymen. Protection against Differential Sttacks for Elliptic Curve Cryptography. *CHES 2001*, LNCS 2162, pp 377-390, Springer-Verlag, 2001.

[70] M. Joye and J. Quisquater. Hessian Elliptic Curves and Side-Channel Attacks. *CHES 2001*, LNCS 2162, pp 402-410, Springer-Verlag, 2001.

[71] M. Katagi, I. Kitamura, T. Akishita, T. Takagi. A Timing Attack on Hyperelliptic Curve Cryptosystems. Technical Report 2003/203, IACR ePrint Archive, Available at `http://www.eprint.iacr.org/2003/203`

[72] K. Kedlaya. Counting Points on Hyperelliptic Curves Using Monski-Washnitzer Cohomology. In *J. Ramanujan Math. Soc.*, 16(2001), pp 323-338, 2001.

[73] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, 726–737, 1990.

[74] E. Knudsen. Elliptic Scalar Multiplication Using Point Halving. *Asiacrypt 1999*, LNCS 1716, pp 135-149, Springer-Verlag, 1999.

[75] T. Kobayashi, H. Morita, K. Kobayashi and F. Hoshino. Fast elliptic curve algorithm combining Frobenius map and table reference to adapt to higher characteristic. *Eurocrypt 1999*, LNCS 1592, pp 176–189, Springer-Verlag, 1999.

[76] N. Koblitz. Elliptic Curve Cryptosystems. In *Mathematics of Computations*, 48:203-209, 1987.

[77] N. Koblitz. Hyperelliptic Cryptosystems. In *Journal of Cryptology*, 1: pages 139–150, 1989.

[78] N. Koblitz. CM Curves with Good Cryptographic Properties. *Advances in Cryptology - Crypo'91*, LNCS 576, pp. 279–287,Springer Verlag 1992.

[79] N. Koblitz. *Algebraic Aspects of Cryptology*, Algorithms and Computation in Mathematics. Springer Verlag, 1998.

[80] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems, *Crypto'96*, LNCS 1109, pp. 104-113, Springer-Verlag, 1996.

[81] P. Kocher, J. Jaffe and B, Jun. Differential Power Analysis. In *CRYPTO'99*, LNCS 1666, pp. 388-397, Springer-Verlag, 1999.

[82] K. Koyama and Y. Tsuruoka. Speeding up Elliptic Curve Cryptosystems Using a Signed Binary Windows Method, In *Crypto'92*, LNCS 740, pp 345-357, Springer-Verlag, 1992.

[83] T. Lange. *Efficient Arithmetic on Hyperelliptic Curves.* PhD thesis, Universität Gesamthochsschule Essen, 2001.

[84] T. Lange. Efficient Arithmetic on Genus 2 Curves over Finite Fields via Explicit Formulae. Cryptology ePrint Archive, Report 2002/121, 2002. `http://eprint.iacr.org/`.

[85] T. Lange. Inversion-free Arithmetic on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/147, 2002. `http://eprint.iacr.org/`.

[86] T. Lange. Weighted Co-ordinates on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/153, 2002. `http://eprint.iacr.org/`.

[87] T. Lange. Formulae for Arithmetic on Genus 2 Hyperelliptic Curves `http://www.itsc.ruhr-uni-bochum.de/tanja/preprints.html`, 2003.

[88] T. Lange. Montgomery Addition for Genus Two Curves. To appear at *ANTS 2004*.

[89] T. Lange and P. K. Mishra. *SCA resistant Parallel Explicit Formula for Addition and Doubling of Divisors in the Jacobian of Hyperelliptic Curves of Genus 2.* Preprint, 2004.

[90] H. W. Lenstra, J. Pila and C. Pomerance. A Hyperelliptic Smoothness Test I. *Philosophical Transactions of the Royal Society of London A*, 345, pp 397-408, 1993.

[91] P. -Y. Liardet and N. P. Smart. Preventing SPA/DPA in ECC Systems Using Jacobi Form. In *CHES 2001*, LNCS 2162, pp 391-401, Springer-Verlag, 2001.

[92] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, revised edition, 1994.

[93] P. Lockhart On the Discriminant of a Hyperelliptic Curve. In *Trans. Amer. Math. Soc.*, 342(1994), No. 2, pp 729-752.

[94] J. Lopez and R. Dahab. Fast Multiplication on Elliptic Curve over $GF(2^m)$ without Precomputations, In *CHES 99*, LNCS 1717, pp 316-327, Springer-Verlag, 1999.

[95] J. Lopez and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^m)$, In *SAC 99*, LNCS 1556, pp 201-212, Springer-Verlag, 1999.

[96] K. Matsuo, J. Chao and S. Tsujii. Fast Genus Two Hyperelliptic Curve Cryptosystems. In *ISEC2001, IEICE*,2001.

[97] M. Maurer, A. J. Menezes and E. Teske. Analysis of the GHS Weil Descent Attack on the ECDLP over Characteristic two Finite Fields of Composite Degree. *LMS Journal of Computation and Mathematics*, 5:127-174, 2002.

[98] K. McCurley. A Key Distribution System Equivalent to factoring. In *Journal of Cryptology*, Vol 1, 1988, pp 95-105.

[99] A. J. Menezes. *Application of Finite Fields,* Kluwer Academic Press, 1993.

[100] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems,* Kluwer Academic Press, 1993.

[101] A. J. Menezes and M. Qu. Analysis of Weil Descent Attack of Gaudry, Hess and Smart. In *CT-RSA*, LNCS 2020, pp 308-318, Springer-Verlag, 2001.

[102] A. Menezes, Y. Wu and R. Zuccherato. An Elementary Introduction to Hyperelliptic Curves. Technical Report CORR 96-19, University of Waterloo(1996), Canada. Available at http://www.cacr.math.uwaterloo.ca.

[103] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[104] A. J. Menezes, T. Okamoto and S. A. Vanstone. Reducing Elliptic Curve Logarithm to Logarithm in a Finite Field, In *Proceedings of 23rd ACM Symposium on Theory of Computing (STOC 91)*, pp 80-89, 1991.

[105] A. J. Menezes, E. Teske and A. Weng. Weak Fields for ECC. Cryptology ePrint Archive, Technical Report 2003/128. Available at `http://www.eprint.iacr.org/2003/128/`, June 2003.

[106] V. S. Miller. *Use of Elliptic Curves in Cryptography,* In *Crypto'85*, LNCS 218, pp. 417-426, Springer-Verlag, 1985.

[107] P. K. Mishra. Pipelined Computation of Scalar Multiplication in Elliptic Curve Cryptosystems. To appear in *CHES 2004*.

[108] P. K. Mishra, P. Pal and P. Sarkar. Towards Minimizing Memory Requirement for Implementation of Hyperelliptic Curve Cryptosystems. Preprint, 2004.

[109] P. K. Mishra and P. Sarkar. Parallelizing Explicit Formula for Arithmetic in the Jacobian of Hyperelliptic Curves (Extended Abstract) In *Asiacrypt 2003*, LNCS , pp, Springer-Verlag, 2003. Full version available at Cryptology ePrint Archive, Report 2003/180, 2003. `http://eprint.iacr.org/`

[110] P. K. Mishra and P. Sarkar *Inversion of Several Field Elements: A New Parallel Algorithm* Cryptology ePrint Archive, Report 2003/264, 2003. `http://eprint.iacr.org/`

[111] P. K. Mishra and P. Sarkar *Application of Montgomery's Trick to Scalar Multiplication for Elliptic and Hyperelliptic Curves Using a Fixed Base Point* In PKC 2004, LNCS , pp , Springer-Verlag, 2004.

[112] Y. Miyamoto, H. Doi, K. Matsuo, J. Chao and S. Tsujii. A Fast Addition Algorithm for Genus 2 Hyperelliptic Curves. In *Proc of SCIS2002, IEICE, Japan*, pp 497-502,2002, in Japanese.

[113] B. Möller. Securing Elliptic Curve Point Multiplication against Side-Channel Attacks. In *Proc. of ISC 2001*, pages 324-334, 2001.

[114] B. Möller. Parallelizable Elliptic Curve Point Multiplication Method with Resistance against Side-Channel Attacks. In *ISC 2002*, LNCS 2433, pp. 402–413, Springer-Verlag, 2002.

[115] B. Möller. Personal Communication.

[116] P. Montgomery. Speeding the Pollard and Elliptic Curve Methods for Factorisation. In *Math. Comp.*, vol 48, pp 243-264, 1987.

[117] F. Morain and J. Olivos. Speeding up the Computation on an Elliptic Curve Using Addition-Subtraction Chains. In *Inform. Theor. Appl.*, 24: 531-543, 1990.

[118] V. Müller. Fast Multiplication on Elliptic Curves over Small Fields of Characteristic Two. *Journal of Cryptology*, 11(4):219–234, 1998.

[119] D. Mumford. *Tata Lectures on Theta II*, Birkhäuser, 1984.

[120] K. Nagao. Improving Group Law Algorithms for Jacobians of Hyperelliptic Curves. *ANTS IV*, LNCS 1838, Berlin 2000, Springer-Verlag.

[121] National Institute of Standard and Technology, Recommended Elliptic Curves for Federal Government Use, Appendix to FIPS 186-2, 2000. Available at `http://csrc.nist.gov/publication/fips/fips 186-2/fips 186-2.pdf`

[122] K. Okeya, K. Miyazaki and K. Sakurai. A fast Scalar Multiplication Method with Randomized Projective Coordinates on a Montgomery Form Elliptic Curve Secure against Side-Channel Attacks. In *ICISC 2001*, LNCS 2288, pp 428-439, Springer-Verlag 2001.

[123] K. Okeya and D.-G. Han. Side Channel Attack on Ha-Moon's Countermeasure of Randomized Signed Scalar Multiplier. In *Indocrypt 2003*, LNCS 2904, pp 334-348, Springer-Verlag 2000.

[124] K. Okeya and K. Sakurai. Power Analysis Breaks Elliptic Curve Cryptosystem even Secure against Timing Attack. In *Indocrypt 2000*, LNCS 1977, pp 178-190, Springer-Verlag 2000.

[125] K. Okeya and K. Sakurai. Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y-coordinate on a Montgomery form Elliptic Curve. In *CHES 2001*, LNCS 2162, pp 126-141, Springer-Verlag 2001.

[126] E. Oswald. On Side-Channel Attacks and Application of Algorithmic Countermeasures. *Ph.D. Thesis*, Graz University of Technology, Austria, 2003.

[127] E. Oswald and M. Aigner. *Randomised Addition-Subtraction Chains as a Countermeasure against Power Attcks.* In *CHES 2001*, LNCS 2162, pp 39-50, Springer-Verlag, 2001.

[128] T. Park, E. Kim, K. Park and M. Lee. A General Expansion Method Using Efficient Endomorphism. In *ICISC, 2003*, LNCS, Springer-Verlag 2003.

[129] T. Park, K. Park and M. Lee. Efficient Scalar Multiplication in Hyperelliptic Curves using a new Frobenius Expansion. In *ICISC, 2003*, LNCS, Springer-Verlag 2003.

[130] S. Paulus and A. Stein. Comparing Real and Imaginary Quadratic Arithmetics of Divisor Class Groups of Hyperelliptic Curves. In *Algorithmic Number Theory ANTS-III*, LNCS 1423, pp 576-591, 1998.

[131] J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves. Cryptology ePrint Archive, Report 2003/26, 2003. `http://eprint.iacr.org/`.

[132] J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. Low Cost Security: Explicit Formulae for Genus 4 Hyperelliptic Curves . Cryptology ePrint Archive, Report 2003/97, 2003. `http://eprint.iacr.org/` .

[133] J. Pelzl and T. Wollinger and C. Paar. High Performance Arithmetic for Hyperelliptic Curve Cryptosystems of Genus Two. Cryptology ePrint Archive, Report 2003/097, 2003. `http://eprint.iacr.org/`.

[134] S. Pohlig and M. Hellman. An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance, *IEEE Transactions on Information Theory*, 24: 106-110, 1978.

[135] J. Pollard. Monte Carlo Methods for Index Computation (mod p). In *Mathematics of Computation*, 32:918-924, 1978.

[136] G. W. Rietwiesner. Binary Arithmetic, In *Advances in Computers*, 1: 231-308, 1960.

[137] H. Rück. On the Discrete Logarithm in the Divisor Class Group of Curves, *Mathematics of Computation*, 68:805-806, 1999.

[138] Y. Sakai and K. Sakurai. Timing Attack against Implementation of a Parallel Algorithm for Modular Exponentiation. In *ACNS 2003*, LNCS 2846, pp.319-330, 2004.

[139] P. Sarkar, P. K. Mishra and R. Barua *A Parallel Algorithm for Computing Simultaneous Inversions with Application to Elliptic Curve Scalar Multiplication (Extended Abstract)* To appear in Proceedings of IEEE Mid-West Conference in System and Circuit, 2003.

[140] P. Sarkar, P. K. Mishra and R. Barua. *New Table Look-up Methods for Faster Frobenius Map Based Scalar Multiplication Over* $GF(p^n)$ To appear in ACNS 2004.

[141] T. Satoh and K. Araki Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves, *Commentarii Mathematici Universitatis Sancti Pauli*, 47:81-92, 1998.

[142] W. Schindler A Combined Timing and Power Attacks. In *PKC 2002*, LNCS 2274, pp. 263-279, Springer-Verlag, 2002.

[143] O. Schirokauer. Discrete Logarithms and Local Units. *Philosophical Transactions of the Royal Society of London A,* Vol 345, pp 409-423, 1993.

[144] C. Schnorr. Efficient Signature Generation by Smart Cards. In *Journal of Cryptology*, Vol 3, 1991, pp 161-174.

[145] R. Schoof. Nonsingular Plane Cubic Curves over Finite Fields, in *Journal of Combinatorial Theory*, A 46, 1987.

[146] R. Schroeppel. Elliptic curve point halving wins big. *Proceedings of 2nd Midwest Arithmetical Geometry in Cryptography Workshop.* Urbana, Illinois, November 2000.

[147] R. Sethi. Complete register allocation problems. *SIAM Journal of Computing*, 4, 226–248, 1975.

[148] M. Seysen. DPA-Gegenmaβnahmen bei einer ECDSA-Implementierung auf Chipkarten. Presented at DPA Workshop, Bonn (BSI), ECC Brainpool, 2001.

[149] Standards for Efficient Cryptography Group/Certicom Research, SEC 1: Elliptic Curve Cryptography, version 1.0, 2000. Available at `http://www.secg.org`.

[150] Standards for Efficient Cryptography Group/Certicom Research, SEC 2: Recommended Elliptic Curve Cryptography Domain Parameters, version 1.0, 2000. Available at `http://www.secg.org`.

[151] H. Shacham and D. Boneh. Improving SSL Handshake Performance via Batching. In *CT-RSA*, LNCS 2020, Springer-Varlag, 2001.

[152] D. Shanks. A Theory of Factorisation and Genera. In *Proc. Symp. Pure Math.*, 20:415-440, 1971.

[153] F. Sica, M. Ciet and J.-J. Quisquater. Analysis of the Gallant-Lambert-Vanstone method based on efficient endomorphisms: elliptic and hyperelliptic curves. *SAC 2002*, LNCS, Springer-Verlg, 2002.

[154] J. Silverman. *The Arithmetic of Elliptic Curves*, Springer-Verlag, New York, 1986.

[155] J. Silverman. *Advanced Topics in the Arithmetic of Elliptic Curves.* Springer-Verlag, New York, 1994.

[156] N. P. Smart. Elliptic curve cryptosystems over small fields of odd characteristic. *Journal of Cryptology*, 12(2):141-151, 1999.

[157] N. P. Smart. The Discrete Logarithm Problem on Elliptic curves of Trace one. *Journal of Cryptology*, 12:193-196, 1999.

[158] N. P. Smart. Hessian Form of an Elliptic curve. *CHES 2001*, LNCS 2162, pp 118-125, Springer-Verlag, 2001.

[159] J. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19:195–249, 2000.

[160] J. Solinas. An improved Algorithm on a Family of Elliptic Curves. In *Advances in Cryptology - Crypto'97*, LNCS 1294, pp.357–371, Springer-Verlag.

[161] J. Solinas. Low Weight Binary Representation for Pair of Integers. Technical Report CORR 2001-41, CACR. Available at `www.cacr.math.uwaterloo.ca/ techreports/ 2001/corr2001-41.ps`

[162] A. M. Spallek. Kurven vom Geschletch 2 und irhe Anwendung in Public-Key-Kryptosystemen. PhD Thesis, Universitat Gesamthochschule, Essen, 1994.

[163] M. Takahashi. Improving Harley Algorithms for Jacobians of Genus 2 Hyperelliptic Curves. In *Proc of SCIS 2002*, ICICE, Japan, 2002, in Japanese.

[164] P. Van Oorschot and M. Weiner. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12:1-28, 1999.

[165] C. D. Walter and S. Thompson. Distinguishing Exponent Digits by Observing Modular Subtractions. In *Progress in Cryptology, CT-RSA, 2001*, LNCS 2020, pp 192-207, Springer-Verlag, 2001.

[166] L. C. Washington *Elliptic Curves, Number Theory and Cryptography.* CRC Press, 2003.

[167] M. Weiner and R. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems, In *SAC'98*, LNCS 1556, 190-200, Springer-Verlag, 1999.

[168] P. Wright. *Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer*, Viking Press, 1987.

[169] S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Trans. on Computers*, vol. 49, no. 9, pages 967–970, Sept. 2000.

[170] S.-M. Yen, S. Kim, S. Lim, and S. Moon, A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In: *ICICS 2001*, LNCS 2288, pages 414–427, 2002. Springer-Verlag 2002.

Appendix A

Details of Parallel Versions of Explicit Formula

The organisation of this section is as follows.

- Parallel version of the explicit formula for addition using inversion free arithmetic of [85] is presented in Section A.1.
- Parallel version of the explicit formula for doubling using inversion free arithmetic of [85] is presented in Section A.2.
- Parallel version of the explicit formula for addition using affine coordinates ([84]) is presented in Section A.3.
- Parallel version of the explicit formula for doubling using affine coordinates ([84]) is presented in Section A.4.

A.1 Addition Using Inversion Free Arithmetic

Algorithm
Input: Divisors $D_1 = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$ and $D_2 = [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$.
Output: Divisor $D_1 + D_2 = [U_1', U_0', V_1', V_0', Z']$
Initial buffer: $U_{11}, U_{10}, V_{11}, V_{10}, Z_1, U_{21}, U_{20}, V_{21}, V_{20}, Z_2$.

<u>**Round 1**</u>
AM01. $Z = Z_1 Z_2$; **AM02.** $\widetilde{U}_{21} = Z_1 U_{21}$; **AM03.** $\widetilde{U}_{20} = Z_1 U_{20}$; **AM04.** $\widetilde{V}_{21} = Z_1 V_{21}$;
AM05. $\widetilde{V}_{20} = Z_1 U_{20}$; **AM06.** $p_1 = U_{11} Z_2$; **AM07.** $p_2 = U_{10} Z_2$; **AM08.** $p_3 = V_{11} Z_2$.

Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_1, p_2, p_3.$
AA01. $p_4 = p_1 - \widetilde{U}_{21}$; **AA02.** $p_5 = \widetilde{U}_{20} - p_2$; **AA03.** $p_6 = p_3 - \widetilde{V}_{21}$; **AA04.** $p_7 = Z_1 + U_{11}$.
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_3, p_4, p_5, p_6, p_{17}, p_7, Z.$

<u>**Round 2**</u>
AM09. $p_8 = U_{11}p_4$; **AM10.** $p_9 = Z_1p_5$; **AM11.** $p_{10} = Z_1p_4$; **AM12.** $p_{11} = p_4^2$;
AM13. $p_{12} = p_4p_6$; **AM14.** $p_{13} = h_1Z$; **AM15.** $p_{14} = f_4Z$; **AM16.** $p_{15} = V_{10}Z_2$
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_{15}, p_3, p_4, p_5, p_{17}, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}.$
AA05. $p_{16} = p_{15} - \widetilde{V}_{20}$; **AA06.** $p_{17} = p_{16} + p_6$; **AA07.** $p_{18} = p_8 + p_9$;
AA08. $p_{19} = p_{18} + p_{10}$; **AA09.** $p_{20} = p_4 + \widetilde{U}_{21}$;
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_{15}, p_3, p_4, p_{17}, p_7, p_{12}, p_{13}, p_{14}, p_{18}, p_{19}, p_{20}$

<u>**Round 3**</u>
AM17. $p_{21} = p_5p_{18}$; **AM18.** $p_{22} = p_{11}U_{10}$; **AM19.** $p_{23} = p_{19}p_{17}$; **AM20.** $p_{24} = p_{18}p_{16}$
AM21. $p_{25} = p_{12}p_7$; **AM22.** $p_{26} = p_{12}U_{10}$;
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_{15}, p_3, p_4, p_{13}, p_{14}, p_{20}, p_{21}, p_{22}, p_{23}, p_{24}, p_{25}, p_{26}$
AA10. $r = p_{21} + p_{22}$; **AA11.** $s_1 = p_{23} - p_{24} - p_{25}$; **AA12.** $s_0 = p_{24} - p_{26}$;
AA13. $p_{27} = \widetilde{U}_{21} + \widetilde{U}_{20}$; **AA14.** $p_{28} = p_{13} + 2\widetilde{V}_{21}$;
AA15. $p_{29} = p_4 + 2\widetilde{U}_{21} - p_{14}$;
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, r, s_1, s_0, p_{15}, p_3, p_4, p_{20}, p_{27}, p_{28}, p_{29}$

<u>**Round 4**</u>
AM23. $R = Zr$; **AM24.** $s_0 = s_0Z$; **AM25.** $s_3 = s_1Z$; **AM26.** $S = s_0s_1$;
AM26. $p_{30} = s_1p_4$; **AM27.** $p_{31} = rp_{29}$; **AM28.** $p_{32} = s_1p_{28}$ **AM29.** $t = s_1p_{20}$
Buffer: $\widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, r, s_1, s_0, R, s_3, S, t, p_{15}, p_3, p_4, p_{27}, p_{30}, p_{31}, p_{32}, p_{27}$
AA16. $p_{33} = s_0 - t$, **AA17.** $p_{34} = t - 2s_0$
Buffer: $\widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, r, s_1, s_0, R, s_3, S, p_{15}, p_3, p_4, p_{27}, p_{30}, p_{31}, p_{32}, p_{33}, p_{34}$

<u>**Round 5**</u>
AM30. $S_3 = s_3^2$; **AM31.** $\widetilde{R} = Rs_3$; **AM32.** $\widetilde{S} = s_3s_1$; **AM33.** $\widetilde{\widetilde{S}} = s_0s_1$;
AM34. $l_0 = S\widetilde{U}_{20}$; **AM35.** $p_{35} = h_2p_{33}$; **AM36.** $p_{36} = s_0^2$; **AM37.** $p_{37} = R^2$;
Buffer: $\widetilde{U}_{21}, \widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{S}, S, p_{15}, p_3, p_{27}, p_{30}, p_{31}, p_{32}, p_{34}, p_{35}, p_{36}, p_{37}$
AA18. $p_{38} = \widetilde{S} + S$; **AA19.** $p_{39} = p_{35} + p_{32}$;
Buffer: $\widetilde{U}_{21}, \widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{S}, p_{15}, p_3, p_{27}, p_{30}, p_{31}, p_{34}, p_{36}, p_{38}, p_{39}$

<u>**Round 6**</u>
AM38. $\widetilde{\widetilde{R}} = \widetilde{R}\widetilde{S}$; **AM39.** $l_2 = \widetilde{S}\widetilde{U}_{21}$; **AM40.** $p_{40} = p_{38}p_{27}$; **AM41.** $p_{41} = p_{30}p_{34}$;

AM42. $p_{42} = p_3\widetilde{S}$; **AM43.** $p_{43} = Rp_{39}$; **AM44.** $p_{44} = h_2\widetilde{R}$; **AM45.** $p_{45} = p_{15}\widetilde{R}$;

Buffer: $\widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{\widetilde{R}}, p_{31}, p_{36}, p_{37}, p_{40}, p_{41}, p_{42}, p_{43}, p_{44}$

AA20. $l_1 = p_{40} - l_2 - l_0$; **AA21.** $l_2 = l_2 + \widetilde{\widetilde{S}}$;

AA22. $U_0' = p_{36} + p_{41} + p_{42} + p_{43} + p_{31}$; **AA23.** $U_1' = 2\widetilde{\widetilde{S}} - p_{45} + p_{44} - p_{37}$;

AA24. $l_2 = l_2 - U_1'$; **AA25.** $p_{46} = U_0' - l_1$;

Buffer: $U_0', U_1', \widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{\widetilde{R}}, p_{46}$

<u>Round 7</u>

AM46. $p_{47} = U_0'l_2$; **AM47.** $p_{48} = S_3l_0$; **AM48.** $p_{49} = U_1'l_2$; **AM49.** $p_{50} = S_3p_{46}$;

AM50. $Z' = \widetilde{R}S_3$; **AM51.** $U_0' = \widetilde{R}U_0'$; **AM52.** $U_1' = \widetilde{R}U_1'$;

Buffer state: $U_0', U_1', \widetilde{V}_{21}, \widetilde{V}_{20}, \widetilde{\widetilde{R}}, p_{47}, p_{48}, p_{49}, p_{50}, Z'$

AA26. $p_{51} = p_{47} - p_{48}$; **AA27.** $p_{52} = p_{49} + p_{50}$;

Buffer: $U_0', U_1', \widetilde{V}_{21}, \widetilde{V}_{20}, \widetilde{\widetilde{R}}, p_{51}, p_{52}, Z'$

<u>Round 8</u>

AM53. $p_{53} = \widetilde{\widetilde{R}}\widetilde{V}_{20}$; **AM54.** $p_{54} = \widetilde{\widetilde{R}}\widetilde{V}_{21}$; **AM55.** $p_{55} = h_0Z'$; **AM56.** $p_{56} = h_1Z'$;

AM57. $p_{57} = h_2U_0'$; **AM58.** $p_{58} = h_2U_1'$;

Buffer state: $U_0', U_1', p_{51}, p_{52}, p_{53}, p_{54}, p_{55}, p_{55}, p_{56}, p_{57}, p_{58}, Z'$

AA28. $p_{59} = p_{51} - p_{53} - p_{55}$; **AA29.** $p_{60} = p_{52} - p_{54} - p_{56}$;

AA30. $V_0' = p_{57} + p_{59}$; **AA31.** $V_1' = p_{58} + p_{60}$;

Buffer state: $U_0', U_1', V_0', V_1', Z'$

A.2 Doubling Using Inversion Free Arithmetic

Algorithm

Input: Divisors $D_1 = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$.

Output: Divisor $2D_1 = [U_1'', U_0'', V_1'', V_0'', Z'']$.

Initial Buffer: $U_{11}, U_{10}, V_{11}, V_{10}, Z_1$.

<u>Round 1</u>

DM01. $q_0 = Z_1^2$; **DM02.** $q_1 = h_1Z_1$; **DM03.** $q_2 = h_2U_{11}$; **DM04.** $q_3 = h_0Z_1$;

DM05. $q_4 = h_2U_{10}$; **DM06.** $q_5 = f_4U_{11}$; **DM07.** $q_6 = h_2V_{11}$; **DM08.** $q_7 = f_2Z_1$;

DM09. $q_8 = V_{11}h_1$; **DM10.** $q_9 = V_{10}h_2$; **DM11.** $q_{10} = f_4U_{10}$;

Buffer: $q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}$

DA01. $\widetilde{V}_1 = q_1 + 2V_{11} - q_2$; **DA02.** $\widetilde{V}_0 = q_3 + 2V_{10} - q_4$;

DA03. $q_{11} = 2U_{10}$; **DA04.** $inv_1 = -\widetilde{V}_1$; **DA05.** $q_{12} = q_7 - q_8 - q_9 - 2q_{10}$;

DA06. $q_{13} = 2q_{11} + q_{10} + q_6$; **DA07.** $q_{14} = q_{11} + 2q_7 + q_6$;
Buffer: $inv_1, \widetilde{V}_1, \widetilde{V}_0, q_0, q_{14}, q_{11}q_{12}, q_{13}$

Round 2
DM12. $q_{15} = V_{11}^2$; **DM13.** $q_{16} = U_{11}^2$; **DM14.** $q_{17} = \widetilde{V}_0 Z_1$; **DM15.** $q_{18} = U_{11}\widetilde{V}_1$;
DM16. $q_{19} = \widetilde{V}_1^2$; **DM17.** $q_{20} = f_3 q_0$; **DM18.** $q_{21} = q_{12}Z_1$; **DM19.** $q_{22} = q_{13}Z_1$;
DM20. $q_{23} = q_{14}Z_1$; **DM21.** $q_{24} = h_2 U_{11}$; **DM22.** $q_{25} = h_1 Z_1$;
Buffer:$inv_1, \widetilde{V}_1, \widetilde{V}_0, q_0, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20}, q_{21}, q_{22}, q_{23}, q_{24}, q_{25}$
DA08. $q_{26} = q_{17}q_{18}$; **DA09.** $q_{27} = q_{20} + q_{16}$; **DA10.** $q_{28} = q_{22} - q_{27}$;
DA11. $k_1 = 2q_{16} + q_{27} - q_{23}$; **DA12.** $q_{29} = q_{21} - q_{15}$;**DA13.** $q_{30} = 2V_{10} - q_{24} + q_{25}$;
Buffer:$inv_1, \widetilde{V}_0, k_1, q_0, q_{19}, q_{26}, q_{27}, q_{28}, q_{29}, q_{30}$

Round 3
DM23. $q_{31} = \widetilde{V}_0 q_{26}$; **DM24.** $q_{32} = q_{19}U_{10}$; **DM25.** $q_{33} = U_{11}q_{28}$; **DM26.** $q_{34} = Z_1 q_{29}$;
DM27. $q_{35} = k_1 inv_1$; **DM28.** $q_{36} = f_4 Z_1$; **DM29.** $q_{37} = Z_1 U_{10}$;
Buffer:$inv_1, k_1, q_0, q_{26}, q_{31}, q_{32}, q_{37}, q_{30}, q_{33}, q_{34}, q_{35}, q_{36}$
DA14. $r = q_{31} + q_{32}$; **DA15.** $k_0 = q_{33} + q_{34}$;**DA16.** $q_{38} = k_0 + k_1$; **DA17.** $q_{39} = inv_1 + q_{26}$;
DA18. $q_{40} = 1 + U_{11}$; **DA19.** $q_{41} = 2U_{11} - q_{36}$;
Buffer:$q_0, r, k_0, q_{26}, q_{37}, q_{30}, q_{35}, q_{36}, q_{38}, q_{39}, q_{40}, q_{41}$

Round 4
DM30. $R = q_0 r$; **DM31.** $q_{42} = q_{38}q_{39}$; **DM32.** $q_{43} = q_{35}q_{40}$; **DM33.** $q_{44} = q_{35}q_{37}$;
DM34. $q_{45} = k_0 q_{26}$; **DM35.** $q_{46} = r q_{41}$;
Buffer:$R, q_{30}, q_{45}, q_{36}, q_{42}, q_{43}, q_{44}, q_{46}$
DA20. $s_3 = q_{42} - q_{45} - q_{43}$; **DA21.** $s_0 = q_{45} - q_{44}$;
Buffer: $R, s_0, s_3, q_{30}, q_{46}$

Round 5
DM36. $q_{47} = R^2$; **DM37.** $q_{48} = s_0 s_3$; **DM38.** $s_1 = s_3 Z_1$; **DM39.** $S_0 = s_0^2$;
DM40. $t = h_2 s_0$; **DM41.** $q_{49} = q_{30}s_3$; **DM42.** $q_{50} = h_2 R$; **DM43.** $q_{51} = Z_1 q_{46}$;
Buffer: $S_0, t, s_1, q_{47}, q_{48}, q_{49}, q_{51}, q_{50}$
Addition phase
No addition required at this step.
Buffer: Same as above.

Round 6
DM44. $\widetilde{R} = R s_1$; **DM45.** $S_1 = s_1^2$; **DM46.** $q_{52} = s_1 s_3$; **DM47.** $S = q_{48}Z_1$;
DM48. $l_0 = U_{10}q_{48}$; **DM49.** $q_{53} = R q_{49}$; **DM50.** $q_{54} = q_{50}s_1$;
Buffer: $\widetilde{R}, S_1, S, S_0, t, l_0, q_{47}, q_{48}, q_{52}, q_{53}, q_{51}, q_{54}$

DA22. $q_{55} = U_{11} + U_{10}$; **DA23.** $q_{56} = q_{48} + q_{52}$;
DA24. $U_0'' = S_0 + q_{53} + t + q_{51}$; **DA25.** $U_1'' = 2S + q_{54} - q_{47}$;
Buffer:$U_0'', U_1'', l_0, S_1, \widetilde{R}, q_{55}, q_{52}, q_{56}$

Round 7
DM51. $\widetilde{\widetilde{R}} = \widetilde{R}q_{52}$; **DM52.** $q_{57} = q_{56}q_{55}$; **DM53.** $q_{58} = S_1 l_0$; **DM54.** $Z'' = S_1\widetilde{R}$;
DM55. $q_{59} = \widetilde{R}U_1''$ **DM56.** $q_{60} = \widetilde{R}U_0''$; **DM57.** $l_2 = U_{11}s_1$;
Buffer: $U_0'', U_1'', Z'', \widetilde{\widetilde{R}}, S_1, l_0, l_1, l_2, q_{57}, q_{58}, q_{59}, q_{60}$
DA26. $l_1 = q_{57} - l_2 - l_0$; **DA27.** $l_2 = l_2 + S - U_1''$;
DA28. $q_{61} = U_0'' - l_1$;
Buffer:$U_0'', U_1'', Z'', \widetilde{\widetilde{R}}, S_1, l_2, q_{58}, q_{59}, q_{60}, q_{61}$

Round 8
DM58. $q_{62} = U_0''l_2$; **DM59.** $q_{63} = U_1''l_2$; **DM60.** $q_{64} = S_1 q_{61}$; **DM61.** $q_{65} = h_2 q_{60}$;
DM62. $q_{66} = \widetilde{\widetilde{R}}V_{10}$; **DM63.** $q_{67} = h_0 Z''$; **DM64.** $q_{68} = h_2 q_{59}$; **DM65.** $q_{69} = \widetilde{\widetilde{R}}V_{11}$;
DM66. $q_{70} = h_1 Z''$;
Buffer: $Z'', q_{58}, q_{59}, q_{60}, q_{62}, q_{63}, q_{64}, q_{65}, q_{66}, q_{67}, q_{68}, q_{69}, q_{70}$
DA29. $q_{71} = q_{62} + q_{58}$; **DA30.** $q_{72} = q_{63} + q_{64}$;
DA31. $U_0'' = q_{60}$; **DA32.** $U_1'' = q_{59}$; **DA33.** $V_0'' = q_{71} + q_{65} - q_{66} - q_{67}$;
DA34. $V_1'' = q_{72} + q_{68} - q_{69} - q_{70}$;
Buffer:$U_0'', U_1'', Z'', V_0'', V_1''$

A.3 Addition Using Affine Coordinates

Algorithm
Input: Divisors $D_1 = [u_{11}, u_{10}, v_{11}, v_{10}]$ and $D_2 = [u_{21}, u_{20}, v_{21}, v_{20}]$.
Output: Divisor $D_1 + D_2 = [u_1', u_0', v_1', v_0']$
Initial buffer: $u_{11}, u_{10}, v_{11}, v_{10}, u_{21}, u_{20}, v_{21}, v_{20}$.

Round 1
AA01. $inv_1 = u_{11} - u_{21}$; **AA02.** $q_1 = u_{20} - u_{10}$;
AA03. $q_2 = v_{10} - v_{20}$; **AA04.** $q_3 = v_{11} - v_{21}$;
Buffer: $inv_1 = inv_1, q_1, q_2, q_3$.
AM01. $q_4 = u_{11}inv_1$; **AM02.** $q_5 = inv_1^2$; **AM03.** $q_6 = inv_1 q_3$;
Buffer: $inv_1, q_1, q_4, q_5, q_2, q_3, q_6$

Round 2

AA05. $q_7 = q_1 + q_4$; **AA06.** $q_8 = q_2 + q_3$; **AA07.** $q_9 = inv_1 + q_7$;**AA08.** $q_{10} = 1 + u_{11}$;
Buffer: $inv_1, q_1, q_5, q_7, q_2, q_6, q_8, q_9, q_{10}$.
AM04. $q_{11} = u_{10}q_5$ **AM05.** $q_{12} = q_1q_7$; **AM06.** $q_{13} = q_8q_9$; **AM07.** $q_{14} = q_{10}q_6$;
AM08. $q_{15} = u_{10}q_6$; **AM09.** $q_{16} = q_7q_9$;
Buffer: $inv_1, q_1, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}, q_{16}$

Round 3

AA09. $r = q_{11} + q_{12}$; **AA10.** $s_1^{'} = q_{13} - q_{16} - q_{14}$; **AA11.** $s_0^{'} = q_{16} - q_{15}$;
Buffer: $s_0^{'}, s_1^{'}, r, inv_1, q_{12}$
AM10. $q_{17} = rs_1^{'}$; **AM11.** $q_{18} = s_1^{'2}$; **AM12.** $q_{19} = r^2$; **AM13.** $q_{20} = rs_0^{'}$;
Buffer:$r, inv_1, q_{12}, q_{17}, q_{18}, q_{19}, q_{20}$

Inversion Round

AM14. $q_{21} = q_{17}^{-1}$;
Buffer:$r, inv_1, q_{12}, q_{21}, q_{18}, q_{19}, q_{20}$

Round 4

AM15. $q_{22} = rq_{21}$; **AM16.** $q_{23} = q_{21}q_{18}$; **AM17.** $q_{24} = q_{21}q_{19}$; **AM18.** $s_0^{''} = q_{21}q_{20}$;
Buffer: $s_0^{''}, inv_1, q_{12}, q_{22}, q_{23}, q_{24},$
AA12. $l_2^{'} = u_{21} + s_0^{''}$, **AA13.** $q_{25} = s_0^{''} - u_{11}$ **AA14.** $q_{26} = s_0^{''} - inv_1 + h_2q_{24}$
AA15. $q_{27} = h + 2v_{21}$
Buffer:$s_0^{''}, l_2^{'}, inv_1, q_{12}, q_{22}, q_{23}, q_{24}, q_{25}, q_{26}, q_{27}$

Round 5

AM19. $q_{28} = q_{24}^2$; **AM20.** $q_{29} = u_{21}s_0^{''}$; **AM21.** $l_0^{'} = u_{20}s_0^{''}$; **AM22.** $q_{30} = q_{25}q_{26}$;
AM23. $q_{31} = q_{27}q_{24}$;
Buffer: $s_0^{''}, l_0^{'}, l_2^{'}, inv_1, q_{12}, q_{22}, q_{23}, q_{24}, q_{28}, q_{29}, q_{30}, q_{31}$
AA16. $l_1^{'} = q_{29} + u_{20}$; **AA17.** $q_{32} = 2u_{21} + p_1 - f_4$;
AA18. $u_1^{'} = 2s_0^{''} - inv_1 + h_2q_{24} - q_{28}$ **AA19.** $q_{33} = l_2^{'} - u_1^{'}$
Buffer: $l_0^{'}, l_1^{'}, u_1^{'}, q_{12}, q_{22}, q_{23}, q_{28}, q_{30}, q_{31}, q_{32}, q_{33}$

Round 6

AM24. $q_{34} = q_{28}q_{32}$; **AM25.** $q_{35} = u_1^{'}q_{33}$; **AM26.** $q_{36} = q_{33}q_{23}$; **AM27.** $q_{37} = l_0^{'}q_{23}$;
Buffer: $l_0^{'}, l_1^{'}, u_1^{'}, q_{12}, q_{22}, q_{23}, q_{30}, q_{31}, q_{34}, q_{33}, q_{35}, q_{36}, q_{37}$
AA20. $u_0^{'} = q_{30} - u_{10} + l_1^{'} + q_{31} + q_{34}$; **AA21.** $q_{38} = q_{35} + u_0^{'} - l_0^{'}$;
Buffer: $u_1^{'}, u_0^{'}, q_{12}, q_{22}, q_{23}, q_{34}, q_{33}, q_{38}, q_{36}, q_{37}$

Round 7

AM28. $q_{39} = q_{23}q_{38}$; **AM29.** $p_{41} = u_0^{'}q_{33}$; **AM30.** $q_{40} = u_0^{'}q_{36}$;
Buffer state: $u_1^{'}, u_0^{'}, q_{12}, q_{22}, q_{39}, p_{41}, q_{37}, q_{40}$
AA22. $q_{41} = q_{40} - q_{37}$; **AA23.** $v_1^{'} = q_{39} - v_{21} - h_1 + h_2u_1^{'}$;
AA24. $v_0^{'} = q_{41} - v_{20} - h_0 + h_2u_0^{'}$;
Buffer: $u_0^{'}, u_1^{'}, v_0^{'}, v_1^{'}$

A.4 Doubling with Affine Coordinates

Doubling Algorithm with Inversion
Input: Divisors $D_1 = [u_{11}, u_{10}, v_{11}, v_{10}]$.
Output: Divisor $2D_1 = [U_1^{''}, U_0^{''}, V_1^{''}, V_0^{''}]$.
Initial Buffer: $u_{11}, u_{10}, v_{11}, v_{10}$.

Round 1
DA01. $\widetilde{v}_1 = h_1 + 2v_{11} - h_2u_{11}$;
DA02. $\widetilde{v}_0 = h_0 + 2v_{10} - h_2u_{10}$;
DA03. $inv_1^{'} = \widetilde{v}_1$; **DA04.** $p_1 = 2u_{10}$;
Buffer: $\widetilde{v}_1, \widetilde{v}_0, inv_1^{'}, p_1$
DM01. $p_2 = v_{11}^2$; **DM02.** $p_3 = u_{11}^2$; **DM03.** $p_4 = \widetilde{v}_1^2$; **DM04.** $p_5 = u_{11}\widetilde{v}_1$;
DM05. $p_6 = v_{11}h_1$;
Buffer: $\widetilde{v}_1, \widetilde{v}_0, inv_1^{'}, p_2, p_3, p_4, p_5, p_1, p_6$

Round 2
DA05. $inv_0^{'} = \widetilde{v}_1 - p_5$;**DA06.** $p_7 = f_3 + p_3$; **DA07.** $k_1^{'} = 2(p_3 - f_4u_{11} + p_7 - p_1 - v_{11}h_2$;
DA08. $p_8 = 2p_1 - p_7 + f_4u_{11} + v_{11}h_2$;
Buffer:$\widetilde{v}_0, inv_1^{'}, inv_0^{'}, k_1^{'}, p_2, p_3, p_4, p_6, p_8$
DM06. $p_9 = \widetilde{v}_0 inv_0^{'}$; **DM07.** $p_{10} = q_3u_{10}$; **DM08.** $p_{11} = u_{11}p_8$; **DM09.** $p_{12} = k_1^{'}inv_1^{'}$;
Buffer:$inv_1^{'}, inv_0^{'}, k_1^{'}, p_4, p_9, p_{10}, p_6, p_{11}, p_{12}$

Round 3
DA09. $R = p_9 + p_{10}$; **DA10.** $k_0^{'} = p_{11} + f_2 - p_2 - 2f_4u_{10} - p_6 - v_{10}h_2$;
DA11. $p_{13} = k_0^{'} + k_1^{'}$; **DA12.** $p_{14} = inv_0^{'} + inv_1^{'}$; **DA13** $p_{15} = 1 + u_{11}$;
Buffer:$inv_0^{'}, k_0^{'}, R, p_4, p_{12}, p_{13}, p_{14}, p_{15}$
DM10. $p_{16} = inv_0^{'}k_0^{'}$; **DM11.** $p_{17} = p_{13}p_{14}$; **DM12.** $p_{18} = p_{12}p_{15}$; **DM13.** $p_{19} = u_{10}p_{12}$;
Buffer:$R, p_4, p_{16}, p_{17}, p_{18}, p_{19}$

Round 4

DA14. $S_0^{'} = p_{16} - p_{19}$;**DA15.** $S_1^{'} = p_{17} - p_{16} - p_{18}$;

Buffer:$R, S_1^{'}, S_0^{'}, p_4$

DM14. $p_{20} = RS_1^{'}$; **DM15.** $p_{21} = {S_1^{'}}^2$; **DM16.** $p_{22} = R^2$ **DM17.** $p_{23} = RS_0^{'}$

Buffer: $S_1^{'}, S_0^{'}, p_4, p_{20}, p_{21}, p_{22}, p_{23}$

Inversion Round

DM18. $p_{24} = p_{20}^{-1}$;

Buffer: $S_1^{'}, S_0^{'}, R, , p_4, p_{24}, p_{21}, p_{22}, p_{23}$

Round 5

DM19. $p_{25} = Rp_{24}$; **DM20.** $p_{26} = p_{24}p_{21}$; **DM21.** $p_{27} = p_{24}p_{22}$; **DM22.** $S_0^{''} = p_{24}p_{23}$;

Buffer: $S_1^{'}, S_0^{'}, S_0^{''}, p_4, p_{26}, p_{27}$

Addition phase

DA16. $l_2^{'} = u_{11} + S_0^{''}$; **DA17.** $p_{28} = h_2(S_0^{''} - u_{11}) + 2v_{11} + h_1$;

Buffer: $S_1^{'}, S_0^{'}, S_0^{''}, l_2^{'}, p_4, p_{26}, p_{27}, p_{28}$.

Round 6

DM23. $p_{29} = p_{27}^2$; **DM24.** $p_{30} = u_{11}S_0^{''}$; **DM25.** $l_0^{'} = u_{10}S_0^{''}$; **DM26.** $p_{31} = p_{27}p_{28}$;

Buffer: $S_1^{'}, S_0^{'}, S_0^{''}, l_2^{'}, l_0^{'}, p_4, p_{26}, p_{27}, p_{29}, p_{30}, p_{31}$

DA18. $U_1^{''} = 2S_0^{''} + p_{27}h_2 - p_{29}$; **DA19.** $p_{32} = l_2^{'} - U_1^{''}$;

Buffer:$S_1^{'}, S_0^{'}, S_0^{''}, l_0^{'}, U_1^{''}, p_4, p_{26}, p_{29}, p_{30}, p_{31}, p_{32}$

Round 7

DM27. $p_{33} = p_{29}(2u_{11} - f_4)$; **DM28.** $p_{34} = U_1^{''}p_{32}$;

DM29. $p_{35} = {S_0^{''}}^2$ **DM30.** $p_{36} = p_{26}p_{32}$;

Buffer: $S_1^{'}, S_0^{'}, l_0^{'}, U_1^{''}, p_4, p_{26}, p_{30}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}$

DA20. $U_0^{''} = p_{35} + p_{33} + p_{31}$; **DA21.** $q_{36} = p_{34} + U_0^{''} - l_1^{'}$;

Buffer:$S_1^{'}, S_0^{'}, , l_0^{'}, U_1^{''}, U_0^{''}, p_4, p_{26}, p_{30}, p_{32}, q_{36}, p_{36}$

Round 8

DM31. $p_{37} = U_0^{''}p_{32}$; **DM32.** $p_{38} = p_{26}q_{36}$; **DM33.** $p_{39} = l_0^{'}p_{26}$; **DM34.** $p_{40} = U_0^{''}p_{36}$;

Buffer: $S_1^{'}, S_0^{'}, U_1^{''}, U_0^{''}, p_4, p_{30}, p_{38}, p_{36}, p_{39}, p_{40}$

DA22. $V_1^{''} = p_{38} - v_{11} - h_1 + U_1^{''}h_2$; **DA23.** $p_{41} = p_{40} - p_{39}$;

DA24. $V_0^{''} = p_{41} - v_{10} - h_0 + h_2U_0^{''}$;

Buffer:$U_0^{''}, U_1^{''}, V_0^{''}, V_1^{''}$

A.5 x-coordinate only encapsulated add and double for ECC

Algorithm
Input: x and z co-ordinates of points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$.
Output: x and z coordinates of $P_1 + P_2$, X_3 and Z_3 and those of $2P_1$, X_4 and Z_4
z-coordinates of $P_1 + P_2 = Z_3$ and of $2P_1 = Z_4$
Initial buffer: $X_1, X_2, Z_1, Z_2,$.

<u>**Round 1**</u>
AM01. $T_1 = X_1X_2$; **AM02.** $T_2 = Z_1Z_2$; **AM03.** $T_3 = X_1Z_2$; **AM04.** $T_4 = X_2Z_1$;
Buffer: T_1, T_2, T_3, T_4.
AA01. $T_5 = T_3 + T_4$; **AA02.** $T_{14} = T_3 - T_4$;
Buffer: $T_1, T_2, T_3, T_5, T_{14}$.

<u>**Round 2**</u>
AM05. $T_6 = aT_2$; **AM06.** $T_{10} = T_2^2$; **AM07.** $T_{15} = T_{14}^2$; **AM08.** $T_{21} = T_3^2$;
Buffer: $T_1, T_2, T_3, T_5, T_6, T_{10}, T_{15}, T_{21}$.
AA03. $T_7 = T_1 + T_6$;
Buffer: $T_2, T_3, T_5, T_6, T_7, T_{10}, T_{15}, T_{21}$

<u>**Round 3**</u>
AM09. $T_8 = T_5T_7$; **AM10.** $T_{11} = bT_{10}$; **AM11.** $T_{17} = X_3'T_{15}$; **AM12.** $Z_3 = Z_3'T_{15}$
AM13. $T_{22} = T_6T_2$;
Buffer: $T_2, T_3, T_8, T_{21}, T_{11}, T_{17}, T_{22}, Z_3$
AA04. $T_9 = 2T_8$; **AA05.** $T_{12} = 2T_{11}$; **AA06.** $T_{12} = 2T_{12}$; **AA07.** $T_{13} = T_9 + T_{12}$;
AA08. $T_{23} = T_{21} - T_{22}$; **AA09.** $T_{28} = T_{21} + T_{22}$;
Buffer:$T_2, T_3, T_{17}, Z_3, T_{13}, T_{23}, T_{28}$

<u>**Round 4**</u>
AM14. $T_{16} = Z_3'T_{13}$; **AM15.** $T_{24} = T_{23}^2$; **AM16.** $T_{25} = T_{11}T_2$; **AM17.** $T_{29} = T_3T_{28}$;
Buffer: $T_2, T_3, T_{17}, Z_3, T_{16}, T_{24}, T_{25}, T_{29}$
AA10. $X_3 = T_{16} - T_{17}$; **AA11.** $T_{30} = T_{29} + T_{25}$
Buffer:$T_2, T_3, X_3, Z_3, T_{24}, T_{25}, T_{30}$

<u>**Round 5**</u>
AM18. $T_{26} = T_{25}T_3$; **AM19.** $T_{31} = T_2T_{30}$;
Buffer: $X_3, Z_3, T_{24}, T_{26}, T_{31}$
AA12. $T_{27} = 2T_{26}$; **AA13.** $T_{27} = 2T_{27}$; **AA14.** $T_{27} = 2T_{27}$; **AA15.** $X_4 = T_{24} - T_{27}$;

AA16. $Z_4 = 2T_{31}$;
Buffer: X_3, Z_3, X_4, Z_4

Appendix B

HECC Formulae in ERSF with Minumum Register Usage

We present the HECC explicit formuae in explicit register specified format in this appendix. In Appendix B.1 we present the formuae when some registers are not reused (for example, the registers containing the coefficients of the base divisors in addition formulae). In the formulae presented in the next section all registers have been reused.

B.1 HECC Addition Formulae in ERSF with Selective Register Reuse

Algorithm HCADD$^{\mathcal{N}o}$ of [86]
Curve Constants Used: $None$
Input Variables: $U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}$,
$U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2$

1. $R_1 := U11$ 5. $R_5 := Z11$ 9. $R_9 := U20$ 13. $R_{13} := Z22$	2. $R_2 := U10$ 6. $R_6 := Z12$ 10. $R_{10} := V21$ 14. $R_{14} := z21$	3. $R_3 := V11$ 7. $R_7 := z11$ 11. $R_{11} := V20$	4. $R_4 := V10$ 8. $R_8 := U21$ 12. $R_{12} := Z21$
15. $R_{15} := R_1 * R_{14}$ 19. $R_{18} := R_{12} * R_{13}$ 23. $R_{19} := R_7 + R_1$ 27. $R_{14} := R_{14} - R_{11}$ 31. $R_{20} := R_{16} * R_7$ 35. $R_{19} := R_{17} * R_{19}$ 39. $R_{14} := R_{20} * R_{14}$ 43. $R_{18} := R_{21} * R_{18}$ 47. $R_{13} := R_6 * R_{13}$ 51. $R_{12} := R_5 * R_{12}$ 55. $R_{20} := R_{14} * R_{19}$ 59. $R_{10} := R_{18} * R_{10}$ 63. $R_{16} := R_{16} * R_{14}$ 67. $R_{22} := R_{15} + R_8$ 71. $R_{18} := R_{15} * R_{18}$ 75. $R_9 := R_{22} * R_9$ 79. $R_{22} := R_{20} * R_{20}$ 83. $R_{18} := R_{19} + R_{18}$ 87. $R_9 := R_9 - R_{18}$ 91. $R_{13} := R_{15} * R_{14}$ 95. $R_{15} := R_{14} * R_{13}$ 99. $R_8 := R_{10} + R_8$ 103. $R_9 := R_{22} * R_9$	16. $R_{16} := R_2 * R_{14}$ 20. $R_{14} := R_{14} * R_{18}$ 24. $R_{10} := R_{10} * R_{17}$ 28. $R_{18} := R_{14} + R_{17}$ 32. $R_8 := R_8 * R_7$ 36. $R_{17} := R_2 * R_{17}$ 40. $R_{17} := R_{14} - R_{17}$ 44. $R_{14} := R_{18} - R_{14}$ 48. $R_{19} := R_{15} * R_{15}$ 52. $R_{19} := R_{12} * R_{12}$ 56. $R_{19} := R_{17} * R_{19}$ 60. $R_{11} := R_{18} * R_{11}$ 64. $R_{21} := R_{12} * R_{13}$ 68. $R_{18} := R_{18} * R_{22}$ 72. $R_{22} := R_{17} + R_{14}$ 76. $R_9 := R_9 - R_{17}$ 80. $R_{11} := R_{22} * R_{11}$ 84. $R_{16} := R_{18} + R_{16}$ 88. $R_{17} := R_{18} + R_{17}$ 92. $R_{13} := R_{23} - R_{13}$ 96. $R_9 := R_9 + R_{10}$ 100. $R_{10} := R_{14} * R_8$ 104. $R_9 := R_{15} - R_9$	17. $R_{17} := R_5 * R_6$ 21. $R_{18} := R_3 * R_{14}$ 25. $R_{11} := R_{11} * R_{17}$ 29. $R_9 := R_9 * R_7$ 33. $R_{15} := R_{15} - R_8$ 37. $R_{21} := R_1 * R_{15}$ 41. $R_{21} := R_7 * R_{15}$ 45. $R_{14} := R_{14} - R_{19}$ 49. $R_{19} := R_{19} * R_2$ 53. $R_{13} := R_{13} * R_{19}$ 57. $R_{18} := R_{18} * R_{20}$ 61. $R_{18} := R_{14} * R_{14}$ 65. $R_{12} := R_{12} * R_{12}$ 69. $R_{22} := R_{17} + R_{17}$ 73. $R_{17} := R_{17} * R_9$ 77. $R_{11} := R_{17} + R_{11}$ 81. $R_{23} := R_{17} + R_{17}$ 85. $R_{18} := R_{14} * R_8$ 89. $R_8 := R_8 + R_{15}$ 93. $R_{13} := R_{13} - R_{12}$ 97. $R_{10} := R_{10} + R_{10}$ 101. $R_{10} := R_{10} - R_{11}$	18. $R_{17} := R_7 * R_{17}$ 22. $R_{14} := R_4 * R_{14}$ 26. $R_{17} := R_{18} - R_{10}$ 30. $R_{16} := R_9 - R_{16}$ 34. $R_{17} := R_{15} * R_{17}$ 38. $R_{20} := R_{21} + R_{20}$ 42. $R_{21} := R_{20} + R_{21}$ 46. $R_{18} := R_{16} * R_{20}$ 50. $R_{18} := R_{18} + R_{19}$ 54. $R_{13} := R_{13} * R_{18}$ 58. $R_{17} := R_{17} * R_{20}$ 62. $R_{14} := R_{14} * R_{20}$ 66. $R_{13} := R_{13} * R_{13}$ 70. $R_{18} := R_{18} - R_{22}$ 74. $R_9 := R_9 + R_8$ 78. $R_{17} := R_{20} * R_{19}$ 82. $R_{19} := R_{19} * R_{19}$ 86. $R_8 := R_8 + R_8$ 90. $R_8 := R_8 * R_{13}$ 94. $R_{14} := R_{17} - R_{13}$ 98. $R_{10} := R_{16} + R_{10}$ 102. $R_9 := R_9 - R_8$
$Up_1 := R_{13}$ $Zp_1 := R_{20}$	$Up_0 := R_8$ $Zp_2 := R_{21}$	$Vp_1 := R_9$ $zp_1 := R_{22}$	$Vp_0 := R_{10}$ $zp_2 := R_{12}$

Number of registers used $= 23$

Algorithm mHCADD$^{\mathcal{N}o}$ of [86]
Curve Constants Used: $None$
Input Variables: $U_{10}, U_{11}, V_{10}, V_{11}, U_{20}, U_{21}, V_{20}, V_{21}, Z_{21}, Z_{22}, z_{21}, z_{22}$
Output Variables: $Up_0, Up_1, Vp_0, Vp_1, Zp_1, Zp_2, zp_1, zp_2$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := U_{21}$	6. $R_6 := U_{20}$	7. $R_7 := V_{21}$	8. $R_8 := V_{20}$
9. $R_9 := Z_{21}$	10. $R_{10} := Z_{22}$	11. $R_{11} := z_{21}$	12. $R_{12} := z_{22}$
13. $R_{10} := R_9 * R_{10}$	14. $R_{13} := R_{11} * R_{10}$	15. $R_{14} := R_1 * R_{11}$	16. $R_{14} := R_{14} - R_5$
17. $R_{15} := R_2 * R_{11}$	18. $R_{15} := R_6 - R_{15}$	19. $R_{16} := R_1 * R_{14}$	20. $R_{16} := R_{16} + R_{15}$
21. $R_{15} := R_{15} * R_{16}$	22. $R_{17} := R_{14} * R_{14}$	23. $R_{17} := R_{17} * R_2$	24. $R_{15} := R_{15} + R_{17}$
25. $R_{10} := R_{15} * R_{10}$	26. $R_{17} := R_{10} * R_9$	27. $R_{10} := R_{10} * R_{10}$	28. $R_{18} := R_{17} * R_{17}$
29. $R_4 := R_4 * R_{13}$	30. $R_4 := R_4 - R_8$	31. $R_3 := R_3 * R_{13}$	32. $R_3 := R_3 - R_7$
33. $R_{13} := R_{16} + R_{14}$	34. $R_{16} := R_{16} * R_4$	35. $R_4 := R_4 + R_3$	36. $R_4 := R_{13} * R_4$
37. $R_3 := R_{14} * R_3$	38. $R_4 := R_4 - R_{16}$	39. $R_{13} := 1 + R_1$	40. $R_{13} := R_3 * R_{13}$
41. $R_3 := R_2 * R_3$	42. $R_3 := R_{16} - R_3$	43. $R_4 := R_4 - R_{13}$	44. $R_{13} := R_{15} * R_4$
45. $R_{15} := R_3 * R_{11}$	46. $R_9 := R_4 * R_9$	47. $R_{16} := R_9 * R_9$	48. $R_{19} := R_3 * R_4$
49. $R_8 := R_{13} * R_8$	50. $R_7 := R_{13} * R_7$	51. $R_{11} := R_{19} * R_{11}$	52. $R_{13} := R_4 * R_4$
53. $R_{20} := R_{13} + R_{19}$	54. $R_{19} := R_{19} * R_6$	55. $R_8 := R_{19} + R_8$	56. $R_8 := R_{16} * R_8$
57. $R_1 := R_1 * R_4$	58. $R_1 := R_3 - R_1$	59. $R_3 := R_{14} * R_4$	60. $R_3 := R_{15} - R_3$
61. $R_1 := R_1 * R_3$	62. $R_3 := R_{11} + R_{11}$	63. $R_4 := R_{13} * R_5$	64. $R_{11} := R_4 + R_{11}$
65. $R_{13} := R_{14} * R_{13}$	66. $R_3 := R_3 - R_{13}$	67. $R_3 := R_3 - R_{18}$	68. $R_{11} := R_{11} - R_3$
69. $R_{13} := R_{11} * R_3$	70. $R_6 := R_5 + R_6$	71. $R_6 := R_{20} * R_6$	72. $R_6 := R_6 - R_{19}$
73. $R_4 := R_6 - R_4$	74. $R_1 := R_1 + R_4$	75. $R_4 := R_4 + R_7$	76. $R_6 := R_7 + R_7$
77. $R_2 := R_2 * R_{16}$	78. $R_1 := R_1 - R_2$	79. $R_1 := R_1 + R_6$	80. $R_2 := R_5 + R_5$
81. $R_2 := R_2 + R_{14}$	82. $R_2 := R_2 * R_{10}$	83. $R_1 := R_1 + R_2$	84. $R_2 := R_{11} * R_1$
85. $R_2 := R_2 - R_8$	86. $R_4 := R_4 - R_1$	87. $R_4 := R_{16} * R_4$	88. $R_4 := R_{13} - R_4$
$Up_1 := R_3$	$Up_0 := R_1$	$Vp_1 := R_4$	$Vp_0 := R_2$
$Zp_1 := R_9$	$Zp_2 := R_{17}$	$zp_1 := R_{16}$	$zp_2 := R_{18}$

Number of registers used $= 20$

Algorithm HCDBL$^{\mathcal{N}o}$ of [86]
Curve Constants Used: f_3, f_2
Input Variables: $U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2$

1. $R_1 := U1$	2. $R_2 := U0$	3. $R_3 := V1$	4. $R_4 := V0$
5. $R_5 := Z1$	6. $R_6 := Z2$	7. $R_7 := z1$	8. $R_8 := z2$
9. $R_9 := R_4 * R_7$	10. $R_{10} := R_3 * R_3$	11. $R_{11} := R_2 * R_7$	12. $R_{12} := R_1 * R_1$
13. $R_{13} := R_{12} - R_{11}$	14. $R_{14} := R_1 * R_3$	15. $R_9 := R_9 - R_{14}$	16. $R_9 := R_4 * R_9$
17. $R_{14} := R_{10} * R_2$	18. $R_9 := R_{14} + R_9$	19. $R_6 := R_6 * R_9$	20. $R_6 := R_6 * R_7$
21. $R_5 := R_6 * R_5$	22. $R_5 := R_5 + R_5$	23. $R_{14} := R_5 * R_5$	24. $R_6 := R_6 * R_6$
25. $R_6 := R_6 + R_6$	26. $R_6 := R_6 * R_1$	27. $R_{15} := R_7 * R_7$	28. $R_{16} := f_3 * R_{15}$
29. $R_{16} := R_{16} + R_{12}$	30. $R_{12} := R_{12} - R_{11}$	31. $R_{12} := R_{13} + R_{12}$	32. $R_{12} := R_{12} + R_{16}$
33. $R_{12} := R_8 * R_{12}$	34. $R_{13} := R_{15} * R_7$	35. $R_{13} := R_{13} * f_2$	36. $R_{15} := R_{11} + R_{11}$
37. $R_{15} := R_{15} + R_{11}$	38. $R_{15} := R_{15} + R_{11}$	39. $R_{15} := R_{15} - R_{16}$	40. $R_{15} := R_1 * R_{15}$
41. $R_{13} := R_{15} + R_{13}$	42. $R_8 := R_8 * R_{13}$	43. $R_8 := R_8 - R_{10}$	44. $R_{10} := R_{16} + R_3$
45. $R_{13} := R_8 * R_{16}$	46. $R_3 := R_{12} * R_3$	47. $R_8 := R_8 + R_{12}$	48. $R_8 := R_{10} * R_8$
49. $R_8 := R_8 - R_{13}$	50. $R_{10} := R_3 * R_{11}$	51. $R_{10} := R_{13} - R_{10}$	52. $R_{11} := 1 + R_1$
53. $R_3 := R_3 * R_{11}$	54. $R_3 := R_8 - R_3$	55. $R_7 := R_3 * R_7$	56. $R_8 := R_9 * R_7$
57. $R_3 := R_8 * R_3$	58. $R_4 := R_8 * R_4$	59. $R_4 := R_4 + R_4$	60. $R_6 := R_3 + R_6$
61. $R_3 := R_3 + R_3$	62. $R_8 := R_{10} * R_{10}$	63. $R_8 := R_8 + R_6$	64. $R_8 := R_8 + R_6$
65. $R_8 := R_8 + R_6$	66. $R_6 := R_8 + R_6$	67. $R_8 := R_{10} * R_7$	68. $R_9 := R_{10} * R_3$
69. $R_3 := R_7 * R_3$	70. $R_{10} := R_7 * R_7$	71. $R_{11} := R_9 + R_3$	72. $R_3 := R_3 * R_1$
73. $R_9 := R_9 * R_2$	74. $R_1 := R_2 + R_1$	75. $R_1 := R_{11} * R_1$	76. $R_1 := R_1 - R_9$
77. $R_2 := R_9 + R_4$	78. $R_1 := R_1 - R_3$	79. $R_1 := R_1 + R_3$	80. $R_3 := R_3 + R_8$
81. $R_1 := R_1 - R_6$	82. $R_1 := R_{10} * R_1$	83. $R_2 := R_{10} * R_2$	84. $R_4 := R_8 + R_8$
85. $R_4 := R_4 - R_{14}$	86. $R_3 := R_3 - R_4$	87. $R_8 := R_3 * R_6$	88. $R_2 := R_8 - R_2$
89. $R_3 := R_3 * R_4$	90. $R_1 := R_3 - R_1$		
$Up1 := R_4$	$Up0 := R_6$	$Vp1 := R_1$	$Vp0 := R_2$
$Zp1 := R_7$	$Zp2 := R_5$	$zp1 := R_{10}$	$zp2 := R_{14}$

Number of registers used $= 16$

Algorithm HCADD$^{\mathcal{N}e}_{h_2 \neq 0}$
Curve Constants Used: h_2, h_1, h_0
Input Variables: $U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}, z_{12}, z_{13}, z_{14}$,
$U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := Z_{11}$	6. $R_6 := Z_{12}$	7. $R_7 := z_{11}$	8. $R_8 := z_{12}$
9. $R_9 := z_{13}$	10. $R_{10} := z_{14}$	11. $R_{11} := U_{21}$	12. $R_{12} := U_{20}$
13. $R_{13} := V_{21}$	14. $R_{14} := V_{20}$	15. $R_{15} := Z_{21}$	16. $R_{16} := Z_{22}$
17. $R_{17} := z_{21}$	18. $R_{18} := z_{22}$	19. $R_{19} := z_{23}$	20. $R_{20} := z_{24}$
21. $R_{21} := R_3 * R_{20}$	22. $R_{20} := R_4 * R_{20}$	23. $R_{22} := R_2 * R_{17}$	24. $R_{23} := R_1 * R_{17}$
25. $R_{24} := R_7 + R_1$	26. $R_{13} := R_{13} * R_{10}$	27. $R_{21} := R_{21} + R_{13}$	28. $R_{14} := R_{14} * R_{10}$
29. $R_{20} := R_{20} + R_{14}$	30. $R_{25} := R_{20} + R_{21}$	31. $R_{12} := R_{12} * R_7$	32. $R_{22} := R_{22} + R_{12}$
33. $R_{26} := R_{22} * R_7$	34. $R_{11} := R_{11} * R_7$	35. $R_{23} := R_{23} + R_{11}$	36. $R_{21} := R_{23} * R_{21}$
37. $R_{24} := R_{21} * R_{24}$	38. $R_{21} := R_{21} * R_2$	39. $R_{27} := R_1 * R_{23}$	40. $R_{26} := R_{27} + R_{26}$
41. $R_{20} := R_{26} * R_{20}$	42. $R_{21} := R_{20} + R_{21}$	43. $R_{27} := R_{23} * R_7$	44. $R_{27} := R_{26} + R_{27}$
45. $R_{25} := R_{27} * R_{25}$	46. $R_{20} := R_{25} + R_{20}$	47. $R_{20} := R_{20} + R_{24}$	48. $R_{24} := R_{22} * R_{26}$
49. $R_{25} := R_{12} + R_{11}$	50. $R_{19} := R_9 * R_{19}$	51. $R_{26} := R_{23} * R_{23}$	52. $R_{26} := R_{26} * R_2$
53. $R_{24} := R_{24} + R_{26}$	54. $R_{19} := R_{24} * R_{19}$	55. $R_{17} := R_7 * R_{17}$	56. $R_{26} := R_{20} * R_{17}$
57. $R_{24} := R_{24} * R_{26}$	58. $R_{13} := R_{24} * R_{13}$	59. $R_{14} := R_{24} * R_{14}$	60. $R_{24} := R_{21} * R_{17}$
61. $R_{21} := R_{21} * R_{26}$	62. $R_{12} := R_{21} * R_{12}$	63. $R_{14} := R_{12} + R_{14}$	64. $R_{17} := R_{19} * R_{17}$
65. $R_{19} := R_{23} * R_{19}$	66. $R_{27} := R_{23} + R_{11}$	67. $R_{27} := R_{23} * R_{27}$	68. $R_{23} := R_{23} * R_{20}$
69. $R_{20} := R_{20} * R_{26}$	70. $R_{11} := R_{20} * R_{11}$	71. $R_{21} := R_{21} + R_{20}$	72. $R_{21} := R_{21} * R_{25}$
73. $R_{12} := R_{21} + R_{12}$	74. $R_{20} := R_{22} * R_{20}$	75. $R_{12} := R_{12} + R_{11}$	76. $R_{12} := R_{12} + R_{13}$
77. $R_{13} := R_{24} * R_{24}$	78. $R_{21} := R_{27} * R_{23}$	79. $R_{13} := R_{13} + R_{21}$	80. $R_{13} := R_{13} + R_{20}$
81. $R_{20} := R_{24} + R_{27}$	82. $R_{20} := h_2 * R_{20}$	83. $R_{21} := R_{26} * R_{24}$	84. $R_{11} := R_{11} + R_{21}$
85. $R_{21} := h_1 * R_{26}$	86. $R_{20} := R_{20} + R_{21}$	87. $R_{19} := R_{20} + R_{19}$	88. $R_{19} := R_{17} * R_{19}$
89. $R_{13} := R_{13} + R_{19}$	90. $R_{12} := R_{12} + R_{13}$	91. $R_{19} := R_{23} * R_{26}$	92. $R_{20} := R_{26} * R_{26}$
93. $R_{12} := R_{20} * R_{12}$	94. $R_{14} := R_{20} * R_{14}$	95. $R_{21} := R_{26} * R_{17}$	96. $R_{22} := R_{17} * R_{17}$
97. $R_{24} := R_{20} * R_{21}$	98. $R_{25} := h_2 * R_{21}$	99. $R_{27} := h_2 * R_{21}$	100. $R_{11} := R_{11} + R_{27}$
101. $R_{19} := R_{19} + R_{25}$	102. $R_{19} := R_{19} + R_{22}$	103. $R_{11} := R_{11} + R_{23}$	104. $R_{25} := R_{11} * R_{13}$
105. $R_{14} := R_{25} + R_{14}$	106. $R_{11} := R_{11} * R_{23}$	107. $R_{11} := R_{11} + R_{12}$	108. $R_{12} := h_1 * R_{24}$
109. $R_{11} := R_{11} + R_{12}$	110. $R_{12} := h_0 * R_{24}$	111. $R_{12} := R_{14} + R_{12}$	
$U_p1 := R_{19}$	$Up_0 := R_{13}$	$Vp_1 := R_{11}$	$Vp0 := R_{12}$
$Zp1 := R_{26}$	$Zp2 := R_{17}$	$zp1 := R_{20}$	$zp2 := R_{22}$
$zp3 := R_{21}$	$zp4 := R_{24}$		

Number of registers used $= 27$

Algorithm mHCADD$^{\mathcal{N}e}_{h_2 \neq 0}$
Curve Constants Used: h_2, h_1, h_0.
Input Variables: $U_{11}, U_{10}, V_{11}, V_{10}, U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := U_{21}$	6. $R_6 := U_{20}$	7. $R_7 := V_{21}$	8. $R_8 := V_{20}$
9. $R_9 := Z_{21}$	10. $R_{10} := Z_{22}$	11. $R_{11} := z_{21}$	12. $R_{12} := z_{22}$
13. $R_{13} := z_{23}$	14. $R_{14} := z_{24}$		
15. $R_{15} := R_1 * R_{11}$	16. $R_{15} := R_{15} + R_5$	17. $R_{11} := R_2 * R_{11}$	18. $R_{11} := R_6 + R_{11}$
19. $R_{16} := R_1 * R_{15}$	20. $R_{16} := R_{16} + R_{11}$	21. $R_{17} := R_{11} * R_{16}$	22. $R_{18} := R_{15} * R_{15}$
23. $R_{18} := R_{18} * R_2$	24. $R_{17} := R_{17} + R_{18}$	25. $R_{13} := R_{17} * R_{13}$	26. $R_{18} := R_4 * R_{14}$
27. $R_{18} := R_{18} + R_8$	28. $R_{14} := R_3 * R_{14}$	29. $R_{14} := R_{14} + R_7$	30. $R_{19} := R_{16} + R_{15}$
31. $R_{16} := R_{16} * R_{18}$	32. $R_{18} := R_{18} + R_{14}$	33. $R_{18} := R_{19} * R_{18}$	34. $R_{14} := R_{15} * R_{14}$
35. $R_{18} := R_{18} - R_{16}$	36. $R_{19} := 1 + R_1$	37. $R_{19} := R_{14} * R_{19}$	38. $R_{14} := R_2 * R_{14}$
39. $R_{14} := R_{16} + R_{14}$	40. $R_{16} := R_{18} - R_{19}$	41. $R_{17} := R_{17} * R_{16}$	42. $R_{18} := R_{14} * R_9$
43. $R_{14} := R_{14} * R_{16}$	44. $R_8 := R_{17} * R_8$	45. $R_7 := R_{17} * R_7$	46. $R_{17} := R_{16} * R_{16}$
47. $R_{16} := R_{16} * R_9$	48. $R_{11} := R_{17} * R_{11}$	49. $R_{19} := R_{17} + R_{14}$	50. $R_{14} := R_{14} * R_6$
51. $R_8 := R_{14} + R_8$	52. $R_{20} := R_{17} * R_5$	53. $R_{17} := R_{15} * R_{17}$	54. $R_9 := R_{13} * R_9$
55. $R_{13} := R_{13} * R_{13}$	56. $R_{13} := R_{15} * R_{13}$	57. $R_5 := R_5 + R_6$	58. $R_5 := R_{19} * R_5$
59. $R_5 := R_5 - R_{14}$	60. $R_5 := R_5 - R_{20}$	61. $R_5 := R_5 + R_7$	62. $R_6 := h_2 * R_9$
63. $R_6 := R_{18} + R_6$	64. $R_6 := R_{18} * R_6$	65. $R_7 := R_{16} * R_{18}$	66. $R_7 := R_{20} + R_7$
67. $R_{14} := R_{16} * R_{16}$	68. $R_8 := R_{14} * R_8$	69. $R_{15} := R_{16} * R_9$	70. $R_{18} := R_9 * R_9$
71. $R_{19} := R_{17} + R_{18}$	72. $R_{20} := h_2 * R_{15}$	73. $R_{20} := R_{20} + R_{17}$	74. $R_{20} := R_1 * R_{20}$
75. $R_6 := R_6 + R_{20}$	76. $R_6 := R_6 + R_{11}$	77. $R_{11} := h_1 * R_{15}$	78. $R_6 := R_6 + R_{11}$
79. $R_6 := R_6 + R_{13}$	80. $R_5 := R_5 + R_6$	81. $R_5 := R_{14} * R_5$	82. $R_{11} := R_{14} * R_{15}$
83. $R_{13} := h_2 * R_{15}$	84. $R_{13} := R_{19} + R_{13}$	85. $R_{19} := h_2 * R_{15}$	86. $R_7 := R_7 + R_{19}$
87. $R_7 := R_7 + R_{17}$	88. $R_{19} := R_7 * R_6$	89. $R_8 := R_{19} + R_8$	90. $R_7 := R_7 * R_{17}$
91. $R_5 := R_7 + R_5$	92. $R_7 := h_1 * R_{11}$	93. $R_5 := R_5 + R_7$	94. $R_7 := h_0 * R_{11}$
95. $R_7 := R_8 + R_7$			
$Up_1 := R_{13}$	$Up_0 := R_6$	$Vp_1 := R_5$	$Vp_0 := R_7$
$Zp_1 := R_{16}$	$Zp_2 := R_9$	$zp_1 := R_{14}$	$zp_2 := R_{18}$
$zp_3 := R_{15}$	$zp_4 := R_{11}$		

Number of registers used $= 20$

Algorithm HCDBL$^{\mathcal{N}e}_{h_2 \neq 0}$
Curve Constants Used: h_2, h_1, h_0, f_3, f_2.
Input Variables: $U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_1$	2. $R_2 := U_0$	3. $R_3 := V_1$	4. $R_4 := V_0$
5. $R_5 := Z_1$	6. $R_6 := Z_2$	7. $R_7 := z_1$	8. $R_8 := z_2$
9. $R_9 := z_3$	10. $R_{10} := z_4$		
11. $R_{11} := h_1 * R_7$	12. $R_{12} := h_1 * h_1$	13. $R_{13} := h_2 * h_2$	14. $R_{14} := h_1 * R_1$
15. $R_{15} := R_3 * h_1$	16. $R_{16} := R_7 * R_7$	17. $R_{12} := R_{12} * R_{16}$	18. $R_{16} := f_3 * R_{16}$
19. $R_{17} := h_2 * R_2$	20. $R_{14} := R_{14} + R_{17}$	21. $R_{17} := h_0 * R_7$	22. $R_{14} := R_{14} + R_{17}$
23. $R_{14} := R_7 * R_{14}$	24. $R_{17} := R_4 * h_2$	25. $R_{15} := R_{15} + R_{17}$	26. $R_{17} := f_2 * R_{10}$
27. $R_{15} := R_{15} + R_{17}$	28. $R_{15} := R_{10} * R_{15}$	29. $R_{17} := h_2 * R_1$	30. $R_{11} := R_{11} + R_{17}$
31. $R_{17} := R_1 * R_1$	32. $R_{13} := R_{13} * R_{17}$	33. $R_{12} := R_{12} + R_{13}$	34. $R_{12} := R_{12} * R_2$
35. $R_{13} := R_{16} + R_{17}$	36. $R_{16} := h_2 * R_{17}$	37. $R_{14} := R_{14} + R_{16}$	38. $R_8 := R_{13} * R_8$
39. $R_{16} := R_3 * h_2$	40. $R_{16} := R_{16} * R_9$	41. $R_8 := R_8 + R_{16}$	42. $R_{16} := h_0 * R_2$
43. $R_{14} := R_{16} * R_{14}$	44. $R_{12} := R_{12} + R_{14}$	45. $R_9 := R_9 * R_{12}$	46. $R_{10} := R_9 * R_{10}$
47. $R_{12} := R_3 * R_3$	48. $R_{14} := R_1 * R_8$	49. $R_{12} := R_{14} + R_{12}$	50. $R_{12} := R_{12} + R_{15}$
51. $R_{14} := R_{13} + R_{11}$	52. $R_{13} := R_{12} * R_{13}$	53. $R_{11} := R_8 * R_{11}$	54. $R_8 := R_{12} + R_8$
55. $R_8 := R_{14} * R_8$	56. $R_8 := R_8 + R_{13}$	57. $R_{12} := h_2 * R_1$	58. $R_{14} := 1 + R_1$
59. $R_{14} := R_{11} * R_{14}$	60. $R_{11} := R_2 * R_{11}$	61. $R_8 := R_8 + R_{14}$	62. $R_{11} := R_{11} * R_7$
63. $R_{11} := R_{13} + R_{11}$	64. $R_{13} := h_2 * R_{11}$	65. $R_{14} := h_1 * R_7$	66. $R_{12} := R_{12} + R_{14}$
67. $R_{12} := R_8 * R_{12}$	68. $R_{12} := R_{13} + R_{12}$	69. $R_{12} := R_{10} * R_{12}$	70. $R_7 := R_8 * R_7$
71. $R_9 := R_9 * R_7$	72. $R_3 := R_9 * R_3$	73. $R_4 := R_9 * R_4$	74. $R_9 := R_{11} * R_{11}$
75. $R_9 := R_9 + R_{12}$	76. $R_{12} := R_{11} * R_7$	77. $R_{11} := R_{11} * R_8$	78. $R_8 := R_7 * R_8$
79. $R_{13} := R_8 + R_{11}$	80. $R_8 := R_8 * R_1$	81. $R_{11} := R_{11} * R_2$	82. $R_{12} := R_8 + R_{12}$
83. $R_4 := R_{11} + R_4$	84. $R_1 := R_1 + R_2$	85. $R_1 := R_{13} * R_1$	86. $R_1 := R_1 - R_{11}$
87. $R_1 := R_1 - R_8$	88. $R_1 := R_1 + R_3$	89. $R_1 := R_1 + R_9$	90. $R_2 := R_7 * R_7$
91. $R_1 := R_2 * R_1$	92. $R_3 := R_2 * R_4$	93. $R_4 := R_7 * R_{10}$	94. $R_8 := R_{10} * R_{10}$
95. $R_{11} := R_2 * R_4$	96. $R_{13} := h_2 * R_4$	97. $R_{14} := h_2 * R_4$	98. $R_{14} := R_8 + R_{14}$
99. $R_{12} := R_{12} + R_{13}$	100. $R_{12} := R_{12} + R_{14}$	101. $R_{13} := R_{12} * R_9$	102. $R_3 := R_{13} - R_3$
103. $R_{12} := R_{12} * R_{14}$	104. $R_1 := R_{12} - R_1$	105. $R_{12} := R_{11} * h_1$	106. $R_1 := R_1 + R_{12}$
107. $R_{12} := R_{11} * h_0$	108. $R_3 := R_3 + R_{12}$		
$Up_1 := R_{14}$	$Up_0 := R_9$	$Vp_1 := R_1$	$Vp_0 := R_3$
$Zp_1 := R_7$	$Zp_2 := R_{10}$	$zp_1 := R_2$	$zp_2 := R_8$
$zp_3 := R_4$	$zp_4 := R_{11}$		

Number of registers used $= 17$

Algorithm HCADD$^{\mathcal{N}e}_{h_2=0}$
Curve Constants Used: h_0.
Input Variables: $U_{21}, U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}, z_{12}, z_{13}, z_{14},$
$U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{21}$	2. $R_2 := U_{11}$	3. $R_3 := U_{10}$	4. $R_4 := V_{11}$
5. $R_5 := V_{10}$	6. $R_6 := Z_{11}$	7. $R_7 := Z_{12}$	8. $R_8 := z_{11}$
9. $R_9 := z_{12}$	10. $R_{10} := z_{13}$	11. $R_{11} := z_{14}$	12. $R_{12} := U_{20}$
13. $R_{13} := V_{21}$	14. $R_{14} := V_{20}$	15. $R_{15} := Z_{21}$	16. $R_{16} := Z_{22}$
17. $R_{17} := z_{21}$	18. $R_{18} := z_{22}$	19. $R_{19} := z_{23}$	20. $R_{20} := z_{24}$
21. $R_{21} := R_4 * R_{20}$	22. $R_{20} := R_5 * R_{20}$	23. $R_{22} := R_3 * R_{17}$	24. $R_{23} := R_2 * R_{17}$
25. $R_{24} := R_8 + R_2$	26. $R_{13} := R_{13} * R_{11}$	27. $R_{21} := R_{21} + R_{13}$	28. $R_{11} := R_{14} * R_{11}$
29. $R_{14} := R_{20} + R_{11}$	30. $R_{20} := R_{14} + R_{21}$	31. $R_{12} := R_{12} * R_8$	32. $R_{22} := R_{22} + R_{12}$
33. $R_{25} := R_{22} * R_8$	34. $R_{26} := R_1 * R_8$	35. $R_{23} := R_{23} + R_{26}$	36. $R_{21} := R_{23} * R_{21}$
37. $R_{24} := R_{21} * R_{24}$	38. $R_{21} := R_{21} * R_3$	39. $R_{27} := R_2 * R_{23}$	40. $R_{25} := R_{27} + R_{25}$
41. $R_{14} := R_{25} * R_{14}$	42. $R_{21} := R_{14} + R_{21}$	43. $R_{27} := R_{23} * R_8$	44. $R_{27} := R_{25} + R_{27}$
45. $R_{20} := R_{27} * R_{20}$	46. $R_{14} := R_{20} + R_{14}$	47. $R_{14} := R_{14} + R_{24}$	48. $R_{20} := R_{22} * R_{25}$
49. $R_{24} := R_{23} + R_{26}$	50. $R_{19} := R_{10} * R_{19}$	51. $R_{25} := R_{23} * R_{23}$	52. $R_{25} := R_{25} * R_3$
53. $R_{20} := R_{20} + R_{25}$	54. $R_{19} := R_{20} * R_{19}$	55. $R_{25} := R_{14} * R_{14}$	56. $R_{24} := R_{25} * R_{24}$
57. $R_{17} := R_8 * R_{17}$	58. $R_{25} := R_{14} * R_{17}$	59. $R_{20} := R_{20} * R_{25}$	60. $R_{14} := R_{14} * R_{25}$
61. $R_{22} := R_{22} * R_{14}$	62. $R_{13} := R_{20} * R_{13}$	63. $R_{11} := R_{20} * R_{11}$	64. $R_{20} := R_{19} * R_{19}$
65. $R_{20} := R_{20} * R_{17}$	66. $R_{20} := R_{24} + R_{20}$	67. $R_{20} := R_{23} * R_{20}$	68. $R_{23} := R_{23} * R_{14}$
69. $R_{19} := R_{19} * R_{17}$	70. $R_{17} := R_{21} * R_{17}$	71. $R_{21} := R_{21} * R_{25}$	72. $R_{24} := R_{17} * R_{25}$
73. $R_{27} := R_{21} + R_{14}$	74. $R_{14} := R_{14} * R_{26}$	75. $R_{21} := R_{21} * R_{12}$	76. $R_{12} := R_{12} + R_{26}$
77. $R_{12} := R_{27} * R_{12}$	78. $R_{12} := R_{12} + R_{14}$	79. $R_{14} := R_{14} + R_{24}$	80. $R_{12} := R_{12} + R_{21}$
81. $R_{12} := R_{12} + R_{13}$	82. $R_{11} := R_{21} + R_{11}$	83. $R_{13} := R_{17} * R_{17}$	84. $R_{13} := R_{13} + R_{20}$
85. $R_{13} := R_{13} + R_{22}$	86. $R_{17} := R_{25} * R_{25}$	87. $R_{11} := R_{17} * R_{11}$	88. $R_{20} := R_{25} * R_{19}$
89. $R_{21} := R_{19} * R_{19}$	90. $R_{22} := R_{23} + R_{21}$	91. $R_{14} := R_{14} + R_{22}$	92. $R_{23} := R_{14} * R_{22}$
93. $R_{24} := 1 * R_{20}$	94. $R_{13} := R_{13} + R_{24}$	95. $R_{14} := R_{14} * R_{13}$	96. $R_{11} := R_{14} + R_{11}$
97. $R_{12} := R_{12} + R_{13}$	98. $R_{12} := R_{17} * R_{12}$	99. $R_{14} := R_{17} * R_{20}$	100. $R_{12} := R_{23} + R_{12}$
101. $R_{23} := R_{14} * 1$	102. $R_{12} := R_{12} + R_{23}$	103. $R_{23} := R_{14} * h_0$	104. $R_{11} := R_{11} + R_{23}$
$Up_1 := R_{22}$	$Up_0 := R_{13}$	$Vp_1 := R_{12}$	$Vp_0 := R_{11}$
$Zp_1 := R_{25}$	$Zp_2 := R_{19}$	$zp_1 := R_{17}$	$zp_2 := R_{21}$
$zp_3 := R_{20}$	$zp_4 := R_{14}$		

Number of registers used $= 27$

Algorithm mHCADD$^{\mathcal{N}e}_{h_2=0}$
Curve Constants Used: h_1, h_0.
Input Variables: $U_{21}, U_{11}, U_{10}, V_{11}, V_{10}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{21}$	2. $R_2 := U_{11}$	3. $R_3 := U_{10}$	4. $R_4 := V_{11}$
5. $R_5 := V_{10}$	6. $R_6 := U_{20}$	7. $R_7 := V_{21}$	8. $R_8 := V_{20}$
9. $R_9 := Z_{21}$	10. $R_{10} := Z_{22}$	11. $R_{11} := z_{21}$	12. $R_{12} := z_{22}$
13. $R_{13} := z_{23}$	14. $R_{14} := z_{24}$		
15. $R_{15} := R_2 * R_{11}$	16. $R_{15} := R_{15} + R_1$	17. $R_{11} := R_3 * R_{11}$	18. $R_{11} := R_6 + R_{11}$
19. $R_{16} := R_2 * R_{15}$	20. $R_{16} := R_{16} + R_{11}$	21. $R_{17} := R_{11} * R_{16}$	22. $R_{18} := R_{15} * R_{15}$
23. $R_{18} := R_{18} * R_3$	24. $R_{17} := R_{17} + R_{18}$	25. $R_{13} := R_{17} * R_{13}$	26. $R_{18} := R_{13} * R_{13}$
27. $R_{13} := R_{13} * R_9$	28. $R_5 := R_5 * R_{14}$	29. $R_5 := R_5 + R_8$	30. $R_{14} := R_4 * R_{14}$
31. $R_{14} := R_{14} + R_7$	32. $R_{19} := R_{16} + R_{15}$	33. $R_{16} := R_{16} * R_5$	34. $R_5 := R_5 + R_{14}$
35. $R_5 := R_{19} * R_5$	36. $R_{14} := R_{15} * R_{14}$	37. $R_5 := R_5 - R_{16}$	38. $R_{19} := 1 + R_2$
39. $R_{19} := R_{14} * R_{19}$	40. $R_{14} := R_3 * R_{14}$	41. $R_{14} := R_{16} + R_{14}$	42. $R_5 := R_5 - R_{19}$
43. $R_{16} := R_{17} * R_5$	44. $R_{17} := R_{14} * R_9$	45. $R_{14} := R_{14} * R_5$	46. $R_8 := R_{16} * R_8$
47. $R_7 := R_{16} * R_7$	48. $R_{16} := R_{17} * R_{17}$	49. $R_{19} := R_5 * R_5$	50. $R_5 := R_5 * R_9$
51. $R_9 := R_{17} * R_5$	52. $R_{11} := R_{11} * R_{19}$	53. $R_{17} := R_{19} + R_{14}$	54. $R_{14} := R_{14} * R_6$
55. $R_8 := R_{14} + R_8$	56. $R_{20} := R_{13} * R_{13}$	57. $R_{21} := R_5 * R_{13}$	58. $R_{22} := R_5 * R_5$
59. $R_6 := R_1 + R_6$	60. $R_6 := R_{17} * R_6$	61. $R_6 := R_6 - R_{14}$	62. $R_{14} := R_{19} * R_1$
63. $R_6 := R_6 - R_{14}$	64. $R_6 := R_6 + R_7$	65. $R_7 := R_{14} + R_9$	66. $R_9 := R_{19} * R_2$
67. $R_9 := R_9 + R_{18}$	68. $R_9 := R_{15} * R_9$	69. $R_{14} := R_{15} * R_{19}$	70. $R_{14} := R_{14} + R_{20}$
71. $R_9 := R_{16} + R_9$	72. $R_9 := R_9 + R_{11}$	73. $R_7 := R_7 + R_{14}$	74. $R_{11} := R_7 * R_{14}$
75. $R_{15} := h_1 * R_{21}$	76. $R_9 := R_9 + R_{15}$	77. $R_7 := R_7 * R_9$	78. $R_{15} := h_1 * R_{21}$
79. $R_6 := R_6 + R_{15}$	80. $R_6 := R_6 + R_9$	81. $R_6 := R_{22} * R_6$	82. $R_6 := R_{11} + R_6$
83. $R_{11} := h_0 * R_{21}$	84. $R_8 := R_8 + R_{11}$	85. $R_8 := R_{22} * R_8$	86. $R_{11} := R_{22} * R_{21}$
87. $R_7 := R_7 + R_8$			
$Up_1 := R_{14}$	$Up_0 := R_9$	$Vp_1 := R_6$	$Vp_0 := R_7$
$Zp_1 := R_5$	$Zp_2 := R_{13}$	$zp_1 := R_{22}$	$zp_2 := R_{20}$
$zp_3 := R_{21}$	$zp_4 := R_{11}$		

Number of registers used $= 22$

Algorithm HCDBL$^{\mathcal{N}e}_{h_2=0}$ in [86]
Curve Constants: h_1, h_0, f_3, f_2.
Input Variables: $U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_1$	2. $R_2 := U_0$	3. $R_3 := V_1$	4. $R_4 := V_0$
5. $R_5 := Z_1$	6. $R_6 := Z_2$	7. $R_7 := z_1$	8. $R_8 := z_2$
9. $R_9 := z_3$	10. $R_{10} := z_4$		
11. $R_{11} := h_1 * R_2$	12. $R_{12} := h_1 * R_1$	13. $R_{13} := f_2 * R_{10}$	14. $R_{14} := R_1 * h_0$
15. $R_{11} := R_{11} + R_{14}$	16. $R_{11} := h_1 * R_{11}$	17. $R_{14} := h_0 * h_0$	18. $R_{14} := R_{14} * R_7$
19. $R_{11} := R_{11} + R_{14}$	20. $R_{11} := R_{11} * R_{10}$	21. $R_{14} := h_0 * R_7$	22. $R_{12} := R_{12} + R_{14}$
23. $R_{14} := R_3 * R_3$	24. $R_{15} := R_1 * R_1$	25. $R_{16} := R_7 * R_7$	26. $R_{16} := f_3 * R_{16}$
27. $R_{15} := R_{16} + R_{15}$	28. $R_8 := R_8 * R_{15}$	29. $R_{15} := 1 + R_1$	30. $R_{16} := R_1 * R_8$
31. $R_{14} := R_{16} + R_{14}$	32. $R_{16} := R_3 * h_1$	33. $R_{13} := R_{13} + R_{16}$	34. $R_{13} := R_{10} * R_{13}$
35. $R_{13} := R_{14} + R_{13}$	36. $R_{14} := R_{12} + h_1$	37. $R_{12} := R_{13} * R_{12}$	38. $R_{13} := R_{13} + R_8$
39. $R_{13} := R_{14} * R_{13}$	40. $R_8 := R_8 * h_1$	41. $R_{13} := R_{13} + R_{12}$	42. $R_{14} := R_{15} * R_8$
43. $R_8 := R_2 * R_8$	44. $R_8 := R_8 * R_7$	45. $R_8 := R_{12} + R_8$	46. $R_{12} := R_{13} + R_{14}$
47. $R_{13} := R_8 * R_8$	48. $R_{14} := R_8 * R_{12}$	49. $R_{15} := R_{14} * R_2$	50. $R_2 := R_1 + R_2$
51. $R_7 := R_{12} * R_7$	52. $R_8 := R_8 * R_7$	53. $R_{12} := R_7 * R_{12}$	54. $R_{14} := R_{12} + R_{14}$
55. $R_2 := R_{14} * R_2$	56. $R_1 := R_{12} * R_1$	57. $R_2 := R_2 - R_{15}$	58. $R_2 := R_2 - R_1$
59. $R_1 := R_1 + R_8$	60. $R_8 := R_{11} * R_7$	61. $R_{10} := R_{11} * R_{10}$	62. $R_3 := R_8 * R_3$
63. $R_4 := R_8 * R_4$	64. $R_4 := R_{15} + R_4$	65. $R_2 := R_2 + R_3$	66. $R_3 := R_7 * R_4$
67. $R_4 := R_7 * R_7$	68. $R_8 := R_7 * R_{10}$	69. $R_{11} := R_{10} * R_{10}$	70. $R_1 := R_1 - R_{11}$
71. $R_{12} := R_1 * R_{11}$	72. $R_{14} := h_1 * R_8$	73. $R_{13} := R_{13} + R_{14}$	74. $R_1 := R_1 * R_{13}$
75. $R_1 := R_1 + R_3$	76. $R_2 := R_2 + R_{13}$	77. $R_2 := R_4 * R_2$	78. $R_2 := R_{12} + R_2$
79. $R_3 := R_4 * R_8$	80. $R_{12} := R_8 * h_1$	81. $R_2 := R_2 + R_{12}$	82. $R_{12} := R_8 * h_0$
83. $R_1 := R_1 + R_{12}$			
$Up_1 := R_{11}$	$Up_0 := R_{13}$	$Vp_1 := R_2$	$Vp_0 := R_1$
$Zp_1 := R_7$	$Zp_2 := R_{10}$	$zp1 := R_4$	$zp_2 := R_{11}$
$zp_3 := R_8$	$zp_4 := R_3$		

Number of registers used $= 16$

Algorithm HCADD$^{\mathcal{A}}$ of [84]
Curve Constants Used: h_2, h_1, h_0, f_4.
Input Variables: $u_{10}, u_{11}, v_{10}, v_{11}, u_{20}, u_{21}, v_{20}, v_{21}$
Output Variables: up_0, up_1, vp_0, vp_1

1. $R_1 := u_{10}$	2. $R_2 := u_{11}$	3. $R_3 := v_{10}$	4. $R_4 := v_{11}$
5. $R_5 := u_{20}$	6. $R_6 := u_{21}$	7. $R_7 := v_{20}$	8. $R_8 := v_{21}$
9. $R_9 := R_5 - R_1$	10. $R_{10} := R_3 - R_7$	11. $R_{11} := R_4 - R_8$	12. $R_{12} := R_{10} + R_{11}$
13. $R_{13} := R_2 - R_6$	14. $R_{14} := R_2 * R_{13}$	15. $R_{14} := R_{14} + R_9$	16. $R_9 := R_9 * R_{14}$
17. $R_{10} := R_{14} * R_{10}$	18. $R_{15} := R_{13} * R_{13}$	19. $R_{15} := R_{15} * R_1$	20. $R_9 := R_9 + R_{15}$
21. $R_{11} := R_{13} * R_{11}$	22. $R_{13} := R_{14} + R_{13}$	23. $R_{12} := R_{13} * R_{12}$	24. $R_{12} := R_{12} - R_{10}$
25. $R_{13} := 1 + R_2$	26. $R_{13} := R_{11} * R_{13}$	27. $R_{11} := R_1 * R_{11}$	28. $R_{10} := R_{10} - R_{11}$
29. $R_{11} := R_{12} - R_{13}$	30. $R_{12} := R_9 * R_{11}$	31. $R_{11} := R_{11} * R_{11}$	32. $R_{12} := 1/R_{12}$
33. $R_{11} := R_{11} * R_{12}$	34. $R_{12} := R_9 * R_{12}$	35. $R_9 := R_9 * R_{12}$	36. $R_{10} := R_{10} * R_{12}$
37. $R_{12} := R_{10} - R_{13}$	38. $R_{13} := R_{10} - R_2$	39. $R_{14} := h2 * R_9$	40. $R_{12} := R_{12} + R_{14}$
41. $R_{12} := R_{13} * R_{12}$	42. $R_{12} := R_{12} - R_1$	43. $R_{13} := R_6 + R_6$	44. $R_{13} := R_{13} + R_{13}$
45. $R_{13} := R_{13} - f4$	46. $R_{14} := R_{10} + R_{10}$	47. $R_{13} := R_{14} - R_{13}$	48. $R_{14} := h2 * R_9$
49. $R_{13} := R_{13} + R_{14}$	50. $R_{14} := R_9 * R_9$	51. $R_{13} := R_{13} * R_{14}$	52. $R_{13} := R_{13} - R_{14}$
53. $R_{14} := R_5 * R_{10}$	54. $R_{15} := R_8 + R_8$	55. $R_{15} := h1 + R_{15}$	56. $R_9 := R_{15} * R_9$
57. $R_{15} := R_6 + R_{10}$	58. $R_6 := R_6 * R_{10}$	59. $R_5 := R_6 + R_5$	60. $R_6 := R_{12} + R_5$
61. $R_6 := R_6 + R_9$	62. $R_6 := R_6 + R_{13}$	63. $R_9 := R_{15} - R_{13}$	64. $R_{10} := R_{13} * R_9$
65. $R_{12} := h2 * R_{13}$	66. $R_{10} := R_{10} + R_6$	67. $R_5 := R_{10} - R_5$	68. $R_5 := R_5 * R_{11}$
69. $R_5 := R_5 - R_8$	70. $R_5 := R_5 - h1$	71. $R_5 := R_5 + R_{12}$	72. $R_8 := R_6 * R_9$
73. $R_9 := h2 * R_6$	74. $R_8 := R_8 - R_{14}$	75. $R_8 := R_8 * R_{11}$	76. $R_7 := R_8 - R_7$
77. $R_7 := R_7 - h0$	78. $R_7 := R_7 + R_9$		
$up_0 := R_6$	$up_1 := R_{13}$	$vp_0 := R_7$	$vp_1 := R_5$

Number of registers used $= 15$

Algorithm HCDBL$^{\mathcal{A}}$ of [84]
Curve Constants: $h_2, h_1, h_0, f4, f_3, f_2$
Input Variables: u_1, u_0, v_1, v_0
Output Variables: up_0, up_1, vp_0, vp_1

1. $R_1 := u_1$	2. $R_2 := u_0$	3. $R_3 := v_1$	4. $R_4 := v_0$
5. $R_5 := f4 * R_1$	6. $R_6 := 2 * R_3$	7. $R_6 := h_1 + R_6$	8. $R_7 := h_2 * R_1$
9. $R_6 := R_6 - R_7$	10. $R_7 := 2 * R_4$	11. $R_7 := h_0 + R_7$	12. $R_8 := h_2 * R_2$
13. $R_7 := R_7 - R_8$	14. $R_8 := R_3 * R_3$	15. $R_9 := R_1 * R_1$	16. $R_5 := R_9 - R_5$
17. $R_9 := f_3 + R_9$	18. $R_5 := 2 * R_5$	19. $R_5 := R_5 + R_9$	20. $R_{10} := 2 * R_2$
21. $R_5 := R_5 - R_{10}$	22. $R_{10} := 2 * R_{10}$	23. $R_9 := R_{10} - R_9$	24. $R_{10} := R_3 * h_2$
25. $R_5 := R_5 - R_{10}$	26. $R_{10} := f4 * R_1$	27. $R_9 := R_9 + R_{10}$	28. $R_{10} := R_3 * h_2$
29. $R_9 := R_9 + R_{10}$	30. $R_9 := R_1 * R_9$	31. $R_9 := R_9 + f_2$	32. $R_8 := R_9 - R_8$
33. $R_9 := R_1 * R_6$	34. $R_{10} := R_7 - R_9$	35. $R_{10} := R_7 * R_{10}$	36. $R_7 := R_7 - R_9$
37. $R_9 := R_6 * R_6$	38. $R_9 := R_2 * R_9$	39. $R_9 := R_9 + R_{10}$	40. $R_{10} := 2 * f4$
41. $R_{10} := R_{10} * R_2$	42. $R_8 := R_8 - R_{10}$	43. $R_{10} := R_3 * h_1$	44. $R_8 := R_8 - R_{10}$
45. $R_{10} := R_4 * h_2$	46. $R_8 := R_8 - R_{10}$	47. $R_{10} := R_7 + R_6$	48. $R_7 := R_8 * R_7$
49. $R_6 := R_5 * R_6$	50. $R_5 := R_8 + R_5$	51. $R_5 := R_{10} * R_5$	52. $R_5 := R_5 - R_7$
53. $R_8 := 1 + R_1$	54. $R_8 := R_6 * R_8$	55. $R_5 := R_5 - R_8$	56. $R_6 := R_2 * R_6$
57. $R_6 := R_7 - R_6$	58. $R_7 := R_9 * R_5$	59. $R_5 := R_5 * R_5$	60. $R_7 := 1/R_7$
61. $R_5 := R_5 * R_7$	62. $R_7 := R_9 * R_7$	63. $R_8 := R_9 * R_7$	64. $R_6 := R_6 * R_7$
65. $R_7 := 2 * R_6$	66. $R_9 := R_8 * R_8$	67. $R_{10} := R_8 * h_2$	68. $R_7 := R_7 + R_{10}$
69. $R_7 := R_7 - R_9$	70. $R_{10} := R_6 - R_1$	71. $R_{10} := h_2 * R_{10}$	72. $R_{11} := 2 * R_3$
73. $R_{10} := R_{10} + R_{11}$	74. $R_{10} := R_{10} + h_1$	75. $R_8 := R_8 * R_{10}$	76. $R_{10} := R_1 + R_6$
77. $R_{10} := R_{10} - R_7$	78. $R_{11} := R_6 * R_6$	79. $R_8 := R_{11} + R_8$	80. $R_{11} := 2 * R_1$
81. $R_{11} := R_{11} - f4$	82. $R_9 := R_9 * R_{11}$	83. $R_8 := R_8 + R_9$	84. $R_1 := R_1 * R_6$
85. $R_1 := R_1 + R_2$	86. $R_2 := R_2 * R_6$	87. $R_6 := R_7 * R_{10}$	88. $R_9 := R_7 * h_2$
89. $R_6 := R_6 + R_8$	90. $R_1 := R_6 - R_1$	91. $R_1 := R_1 * R_5$	92. $R_1 := R_1 - R_3$
93. $R_1 := R_1 - h_1$	94. $R_1 := R_1 + R_9$	95. $R_3 := R_8 * R_{10}$	96. $R_6 := h_2 * R_8$
97. $R_2 := R_3 - R_2$	98. $R_2 := R_2 * R_5$	99. $R_2 := R_2 - R_4$	100. $R_2 := R_2 - h_0$
101. $R_2 := R_2 + R_6$			
$up_0 := R_8$	$up_1 := R_7$	$vp_0 := R_2$	$vp_1 := R_1$

Number of registers used $= 11$

Algorithm HCADD$^{\mathcal{A}}$ of [133]
Input Variables: $u_{10}, u_{11}, v_{10}, v_{11}, u_{20}, u_{21}, v_{20}, v_{21}$
Output Variables: up_0, up_1, vp_0, vp_1

1. $R_1 := u10$	2. $R_2 := u11$	3. $R_3 := v10$	4. $R_4 := v11$
5. $R_5 := u20$	6. $R_6 := u21$	7. $R_7 := v20$	8. $R_8 := v21$
9. $R_9 := R_2 + R_6$	10. $R_{10} := R_5 - R_1$	11. $R_{11} := R_3 - R_7$	12. $R_{12} := R_4 - R_8$
13. $R_{13} := R_{11} + R_{12}$	14. $R_{14} := R_2 - R_6$	15. $R_{12} := R_{14} * R_{12}$	16. $R_{15} := R_2 * R_{14}$
17. $R_{15} := R_{15} + R_{10}$	18. $R_{10} := R_{10} * R_{15}$	19. $R_{11} := R_{15} * R_{11}$	20. $R_{15} := R_{15} + R_{14}$
21. $R_{13} := R_{15} * R_{13}$	22. $R_{13} := R_{13} - R_{11}$	23. $R_{14} := R_{14} * R_{14}$	24. $R_{14} := R_{14} * R_1$
25. $R_{10} := R_{10} + R_{14}$	26. $R_{14} := 1 + R_2$	27. $R_{14} := R_{12} * R_{14}$	28. $R_{12} := R_1 * R_{12}$
29. $R_{11} := R_{11} - R_{12}$	30. $R_{12} := R_{13} - R_{14}$	31. $R_{13} := R_{10} * R_{12}$	32. $R_{12} := R_{12} * R_{12}$
33. $R_{13} := 1/R_{13}$	34. $R_{12} := R_{12} * R_{13}$	35. $R_{13} := R_{10} * R_{13}$	36. $R_{10} := R_{10} * R_{13}$
37. $R_{11} := R_{11} * R_{13}$	38. $R_{13} := R_6 + R_{11}$	39. $R_{14} := R_{11} - R_2$	40. $R_{15} := R_{13} - R_2$
41. $R_{14} := R_{14} * R_{15}$	42. $R_{14} := R_{14} - R_1$	43. $R_6 := R_6 * R_{11}$	44. $R_6 := R_6 + R_5$
45. $R_{14} := R_{14} + R_6$	46. $R_{14} := R_{14} + R_{10}$	47. $R_{10} := R_{10} * R_{10}$	48. $R_9 := R_{10} * R_9$
49. $R_9 := R_{14} + R_9$	50. $R_5 := R_5 * R_{11}$	51. $R_{11} := R_{11} + R_{13}$	52. $R_{11} := R_{11} - R_2$
53. $R_{10} := R_{11} - R_{10}$	54. $R_{11} := R_{13} - R_{10}$	55. $R_{13} := R_{10} * R_{11}$	56. $R_{13} := R_{13} + R_9$
57. $R_{11} := R_9 * R_{11}$	58. $R_5 := R_{11} - R_5$	59. $R_6 := R_{13} - R_6$	60. $R_6 := R_6 * R_{12}$
61. $R_5 := R_5 * R_{12}$	62. $R_6 := R_6 - R_8$	63. $R_5 := R_5 - R_7$	64. $R_6 := R_6 - 1$
$up_0 := R_9$	$up_1 := R_{10}$	$vp_0 := R_5$	$vp_1 := R_6$

Number of registers used $= 15$

Algorithm HCDBL$^{\mathcal{A}}$ of [133]
Input Variables: u_0, u_1, v_0, v_1
Output Variables: up_0, up_1, vp_0, vp_1

1. $R_1 := u0$	2. $R_2 := u1$	3. $R_3 := v0$	4. $R_4 := v1$
5. $R_5 := R_4 * R_4$	6. $R_6 := R_1 * R_1$	7. $R_7 := R_1 + R_2$	8. $R_8 := R_2 * R_2$
9. $R_2 := R_2 * R_8$	10. $R_5 := R_2 + R_5$	11. $R_5 := R_5 + R_4$	12. $R_1 := R_1 * R_5$
13. $R_9 := R_5 + R_8$	14. $R_7 := R_7 * R_9$	15. $R_7 := R_7 + R_2$	16. $R_7 := R_7 + R_1$
17. $R_1 := 1/R_1$	18. $R_6 := R_6 * R_1$	19. $R_1 := R_5 * R_1$	20. $R_9 := R_8 * R_1$
21. $R_2 := R_2 + R_9$	22. $R_2 := R_2 + R_5$	23. $R_5 := R_6 + R_8$	24. $R_1 := R_1 + R_5$
25. $R_9 := R_6 * R_6$	26. $R_{10} := R_9 * R_8$	27. $R_{10} := R_{10} * R_8$	28. $R_8 := R_{10} + R_8$
29. $R_8 := R_8 + R_6$	30. $R_5 := R_5 * R_8$	31. $R_{10} := R_8 + R_9$	32. $R_1 := R_1 * R_{10}$
33. $R_1 := R_1 + R_6$	34. $R_1 := R_1 + R_5$	35. $R_5 := R_5 + R_7$	36. $R_1 := R_1 + 1$
37. $R_3 := R_5 + R_3$	38. $R_1 := R_1 + R_2$	39. $R_1 := R_1 + R_4$	
$up_0 := R_8$	$up_1 := R_9$	$vp_0 := R_3$	$vp_1 := R_1$

Number of registers used $= 10$

B.2 HECC Addition Formulae in ERSF with Full Register Reuse

Algorithm HCADD$^{\mathcal{N}o}$ of [86]
Curve Constants Used: $None$
Input Variables: $U11, U10, V11, V10, Z11, Z12, z11, U21, U20, V21, V20, Z21, Z22, z21$
Output Variables: $Up1, Up0, Vp1, Vp0, Zp1, Zp2, zp1, zp2$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := Z_{11}$	6. $R_6 := Z_{12}$	7. $R_7 := z_{11}$	8. $R_8 := U_{21}$
9. $R_9 := U_{20}$	10. $R_{10} := V_{21}$	11. $R_{11} := V_{20}$	12. $R_{12} := Z_{21}$
13. $R_{13} := Z_{22}$	14. $R_{14} := z_{21}$		
15. $R_{15} := R_1 * R_{14}$	16. $R_{16} := R_2 * R_{14}$	17. $R_{17} := R_5 * R_6$	18. $R_{17} := R_7 * R_{17}$
19. $R_{18} := R_{12} * R_{13}$	20. $R_{14} := R_{14} * R_{18}$	21. $R_3 := R_3 * R_{14}$	22. $R_4 := R_4 * R_{14}$
23. $R_{14} := R_7 + R_1$	24. $R_6 := R_6 * R_{13}$	25. $R_5 := R_5 * R_{12}$	26. $R_{12} := R_5 * R_5$
27. $R_{13} := R_5 * R_5$	28. $R_6 := R_6 * R_{13}$	29. $R_{11} := R_{11} * R_{17}$	30. $R_{10} := R_{10} * R_{17}$
31. $R_3 := R_3 - R_{10}$	32. $R_4 := R_4 - R_{11}$	33. $R_{17} := R_4 + R_3$	34. $R_9 := R_9 * R_7$
35. $R_{16} := R_9 - R_{16}$	36. $R_{18} := R_{16} * R_7$	37. $R_8 := R_8 * R_7$	38. $R_{15} := R_{15} - R_8$
39. $R_1 := R_1 * R_{15}$	40. $R_1 := R_1 + R_{18}$	41. $R_{18} := R_{16} * R_1$	42. $R_4 := R_1 * R_4$
43. $R_{19} := R_{15} * R_{15}$	44. $R_{19} := R_{19} * R_2$	45. $R_{18} := R_{18} + R_{19}$	46. $R_6 := R_6 * R_{18}$
47. $R_5 := R_5 * R_6$	48. $R_6 := R_6 * R_6$	49. $R_3 := R_{15} * R_3$	50. $R_7 := R_7 * R_{15}$
51. $R_1 := R_1 + R_7$	52. $R_1 := R_1 * R_{17}$	53. $R_1 := R_1 - R_4$	54. $R_7 := R_3 * R_{14}$
55. $R_2 := R_2 * R_3$	56. $R_2 := R_4 - R_2$	57. $R_1 := R_1 - R_7$	58. $R_3 := R_1 * R_1$
59. $R_4 := R_2 * R_{13}$	60. $R_7 := R_1 * R_{13}$	61. $R_{13} := R_{18} * R_7$	62. $R_2 := R_2 * R_7$
63. $R_1 := R_1 * R_7$	64. $R_{10} := R_{13} * R_{10}$	65. $R_{11} := R_{13} * R_{11}$	66. $R_{13} := R_{16} * R_1$
67. $R_{14} := R_2 + R_1$	68. $R_{15} := R_7 * R_4$	69. $R_{16} := R_7 * R_7$	70. $R_{17} := R_{15} + R_{15}$
71. $R_4 := R_4 * R_4$	72. $R_{18} := R_1 * R_8$	73. $R_{15} := R_{18} + R_{15}$	74. $R_1 := R_{15} * R_1$
75. $R_1 := R_{17} - R_1$	76. $R_1 := R_1 - R_{12}$	77. $R_{15} := R_{15} - R_1$	78. $R_{17} := R_{15} * R_1$
79. $R_{19} := R_2 * R_9$	80. $R_2 := R_2 + R_2$	81. $R_9 := R_9 + R_8$	82. $R_9 := R_{14} * R_9$
83. $R_9 := R_9 - R_{19}$	84. $R_{11} := R_{19} + R_{11}$	85. $R_9 := R_9 - R_{18}$	86. $R_9 := R_9 + R_{10}$
87. $R_{10} := R_{10} + R_{10}$	88. $R_{11} := R_{16} * R_{11}$	89. $R_{14} := R_{15} + R_8$	90. $R_8 := R_8 + R_8$
91. $R_3 := R_3 * R_{14}$	92. $R_2 := R_3 - R_2$	93. $R_2 := R_{15} * R_2$	94. $R_3 := R_8 + R_{15}$
95. $R_3 := R_3 * R_6$	96. $R_2 := R_4 + R_2$	97. $R_2 := R_2 + R_{13}$	98. $R_2 := R_2 + R_{10}$
99. $R_2 := R_2 + R_3$	100. $R_3 := R_{15} * R_2$	101. $R_3 := R_3 - R_{11}$	102. $R_4 := R_9 - R_2$
103. $R_4 := R_{16} * R_4$	104. $R_4 := R_{17} - R_4$		
$Up_1 := R_1$	$Up_0 := R_2$	$Vp_1 := R_4$	$Vp_0 := R_3$
$Zp_1 := R_7$	$Zp_2 := R_5$	$zp_1 := R_{16}$	$zp_2 := R_{12}$

Number of registers used $= 19$

Algorithm mHCADD$^{\mathcal{N}}o$ of [86]
Curve Constants Used: $None$
Input Variables: $U_{10}, U_{11}, V_{10}, V_{11}, U_{20}, U_{21}, V_{20}, V_{21}, Z_{21}, Z_{22}, z_{21}, z_{22}$
Output Variables: $Up_0, Up_1, Vp_0, Vp_1, Zp_1, Zp_2, zp_1, zp_2$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := U_{21}$	6. $R_6 := U_{20}$	7. $R_7 := V_{21}$	8. $R_8 := V_{20}$
9. $R_9 := Z_{21}$	10. $R_{10} := Z_{22}$	11. $R_{11} := z_{21}$	12. $R_{12} := z_{22}$
13. $R_{10} := R_9 * R_{10}$	14. $R_{13} := R_{11} * R_{10}$	15. $R_{14} := R_1 * R_{11}$	16. $R_{14} := R_{14} - R_5$
17. $R_{15} := R_2 * R_{11}$	18. $R_{15} := R_6 - R_{15}$	19. $R_{16} := R_1 * R_{14}$	20. $R_{16} := R_{16} + R_{15}$
21. $R_{15} := R_{15} * R_{16}$	22. $R_{17} := R_{14} * R_{14}$	23. $R_{17} := R_{17} * R_2$	24. $R_{15} := R_{15} + R_{17}$
25. $R_{10} := R_{15} * R_{10}$	26. $R_{17} := R_{10} * R_9$	27. $R_{10} := R_{10} * R_{10}$	28. $R_{18} := R_{17} * R_{17}$
29. $R_4 := R_4 * R_{13}$	30. $R_4 := R_4 - R_8$	31. $R_3 := R_3 * R_{13}$	32. $R_3 := R_3 - R_7$
33. $R_{13} := R_{16} + R_{14}$	34. $R_{16} := R_{16} * R_4$	35. $R_{14} := R_{14} * R_3$	36. $R_3 := R_4 + R_3$
37. $R_3 := R_{13} * R_3$	38. $R_3 := R_3 - R_{16}$	39. $R_4 := 1 + R_1$	40. $R_4 := R_{14} * R_4$
41. $R_{13} := R_2 * R_{14}$	42. $R_{13} := R_{16} - R_{13}$	43. $R_3 := R_3 - R_4$	44. $R_4 := R_{15} * R_3$
45. $R_{14} := R_{13} * R_{11}$	46. $R_9 := R_3 * R_9$	47. $R_{15} := R_9 * R_9$	48. $R_{16} := R_{13} * R_3$
49. $R_8 := R_4 * R_8$	50. $R_4 := R_4 * R_7$	51. $R_7 := R_{16} * R_{11}$	52. $R_{11} := R_3 * R_3$
53. $R_{19} := R_{11} + R_{16}$	54. $R_{16} := R_{16} * R_6$	55. $R_8 := R_{16} + R_8$	56. $R_8 := R_{15} * R_8$
57. $R_1 := R_1 * R_3$	58. $R_1 := R_{13} - R_1$	59. $R_3 := R_{14} * R_3$	60. $R_3 := R_{14} - R_3$
61. $R_1 := R_1 * R_3$	62. $R_3 := R_7 + R_7$	63. $R_{13} := R_{11} * R_5$	64. $R_7 := R_{13} + R_7$
65. $R_{11} := R_{14} * R_{11}$	66. $R_3 := R_3 - R_{11}$	67. $R_3 := R_3 - R_{18}$	68. $R_7 := R_7 - R_3$
69. $R_{11} := R_7 * R_3$	70. $R_6 := R_5 + R_6$	71. $R_6 := R_{19} * R_6$	72. $R_6 := R_6 - R_{16}$
73. $R_6 := R_6 - R_{13}$	74. $R_1 := R_1 + R_6$	75. $R_6 := R_6 + R_4$	76. $R_4 := R_4 + R_4$
77. $R_2 := R_2 * R_{15}$	78. $R_1 := R_1 - R_2$	79. $R_1 := R_1 + R_4$	80. $R_2 := R_5 + R_5$
81. $R_2 := R_2 + R_{14}$	82. $R_2 := R_2 * R_{10}$	83. $R_1 := R_1 + R_2$	84. $R_2 := R_7 * R_1$
85. $R_2 := R_2 - R_8$	86. $R_4 := R_6 - R_1$	87. $R_4 := R_{15} * R_4$	88. $R_4 := R_{11} - R_4$
$Up_1 := R_3$	$Up_0 := R_1$	$Vp_1 := R_4$	$Vp_0 := R_2$
$Zp_1 := R_9$	$Zp_2 := R_{17}$	$zp_1 := R_{15}$	$zp_2 := R_{18}$

Number of registers used $= 19$

Algorithm HCADD$^{\mathcal{N}e}_{h_2 \neq 0}$ of [86]
Curve Constants Used: h_2, h_1, h_0
Input Variables: $U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}, z_{12}, z_{13}, z_{14}, U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := Z_{11}$	6. $R_6 := Z_{12}$	7. $R_7 := z_{11}$	8. $R_8 := z_{12}$
9. $R_9 := z_{13}$	10. $R_{10} := z_{14}$	11. $R_{11} := U_{21}$	12. $R_{12} := U_{20}$
13. $R_{13} := V_{21}$	14. $R_{14} := V_{20}$	15. $R_{15} := Z_{21}$	16. $R_{16} := Z_{22}$
17. $R_{17} := z_{21}$	18. $R_{18} := z_{22}$	19. $R_{19} := z_{23}$	20. $R_{20} := z_{24}$
21. $R_3 := R_3 * R_{20}$	22. $R_4 := R_4 * R_{20}$	23. $R_{20} := R_2 * R_{17}$	24. $R_{21} := R_1 * R_{17}$
25. $R_9 := R_9 * R_{19}$	26. $R_{13} := R_{13} * R_{10}$	27. $R_3 := R_3 + R_{13}$	28. $R_{10} := R_{14} * R_{10}$
29. $R_4 := R_4 + R_{10}$	30. $R_{14} := R_4 + R_3$	31. $R_{12} := R_{12} * R_7$	32. $R_{19} := R_{20} + R_{12}$
33. $R_{20} := R_{19} * R_7$	34. $R_{11} := R_{11} * R_7$	35. $R_{21} := R_{21} + R_{11}$	36. $R_{22} := R_1 * R_{21}$
37. $R_{20} := R_{22} + R_{20}$	38. $R_{22} := R_{19} * R_{20}$	39. $R_4 := R_{20} * R_4$	40. $R_{23} := R_{21} * R_{21}$
41. $R_{23} := R_{23} * R_2$	42. $R_{22} := R_{22} + R_{23}$	43. $R_9 := R_{22} * R_9$	44. $R_3 := R_{21} * R_3$
45. $R_{21} := R_{21} * R_7$	46. $R_{20} := R_{20} + R_{21}$	47. $R_{14} := R_{20} * R_{14}$	48. $R_{14} := R_{14} + R_4$
49. $R_{20} := R_{21} * R_9$	50. $R_1 := R_7 + R_1$	51. $R_1 := R_3 * R_1$	52. $R_1 := R_{14} + R_1$
53. $R_2 := R_3 * R_2$	54. $R_2 := R_4 + R_2$	55. $R_3 := R_{12} + R_{11}$	56. $R_4 := R_7 * R_{17}$
57. $R_7 := R_9 * R_4$	58. $R_9 := R_2 * R_4$	59. $R_4 := R_1 * R_4$	60. $R_{14} := R_{22} * R_4$
61. $R_2 := R_2 * R_4$	62. $R_{12} := R_2 * R_{12}$	63. $R_{13} := R_{14} * R_{13}$	64. $R_{10} := R_{14} * R_{10}$
65. $R_{10} := R_{12} + R_{10}$	66. $R_{14} := R_9 * R_9$	67. $R_{17} := R_{21} + R_{11}$	68. $R_{17} := R_{21} * R_{17}$
69. $R_{21} := R_{21} * R_1$	70. $R_1 := R_1 * R_4$	71. $R_{11} := R_1 * R_{11}$	72. $R_2 := R_2 + R_1$
73. $R_1 := R_{19} * R_1$	74. $R_2 := R_2 * R_3$	75. $R_2 := R_2 + R_{12}$	76. $R_2 := R_2 + R_{11}$
77. $R_2 := R_2 + R_{13}$	78. $R_3 := R_{17} * R_{21}$	79. $R_3 := R_{14} + R_3$	80. $R_1 := R_3 + R_1$
81. $R_3 := R_9 + R_{17}$	82. $R_3 := h_2 * R_3$	83. $R_9 := R_4 * R_9$	84. $R_9 := R_{11} + R_9$
85. $R_{11} := h_1 * R_4$	86. $R_3 := R_3 + R_{11}$	87. $R_3 := R_3 + R_{20}$	88. $R_3 := R_7 * R_3$
89. $R_1 := R_1 + R_3$	90. $R_2 := R_2 + R_1$	91. $R_3 := R_{21} * R_4$	92. $R_{11} := R_4 * R_4$
93. $R_2 := R_{11} * R_2$	94. $R_{10} := R_{11} * R_{10}$	95. $R_{12} := R_4 * R_7$	96. $R_{13} := R_7 * R_7$
97. $R_{14} := R_{11} * R_{12}$	98. $R_{17} := h_2 * R_{12}$	99. $R_{19} := h_2 * R_{12}$	100. $R_9 := R_9 + R_{19}$
101. $R_3 := R_3 + R_{17}$	102. $R_3 := R_3 + R_{13}$	103. $R_9 := R_9 + R_{21}$	104. $R_{17} := R_9 * R_1$
105. $R_{10} := R_{17} + R_{10}$	106. $R_9 := R_9 * R_{21}$	107. $R_2 := R_9 + R_2$	108. $R_9 := h_1 * R_{14}$
109. $R_2 := R_2 + R_9$	110. $R_9 := h_0 * R_{14}$	111. $R_9 := R_{10} + R_9$	
$Up_1 := R_3$	$Up_0 := R_1$	$Vp_1 := R_2$	$Vp_0 := R_9$
$Zp_1 := R_4$	$Zp_2 := R_7$	$zp_1 := R_{11}$	$zp_2 := R_{13}$
$zp_3 := R_{12}$	$zp_4 := R_{14}$		

Number of registers used $= 23$

Algorithm mHCADD$^{\mathcal{N}e}_{h_2 \neq 0}$ of [86]
Curve Constants Used: h_2, h_1, h_0.
Input Variables: $U_{11}, U_{10}, V_{11}, V_{10}, U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := U_{21}$	6. $R_6 := U_{20}$	7. $R_7 := V_{21}$	8. $R_8 := V_{20}$
9. $R_9 := Z_{21}$	10. $R_{10} := Z_{22}$	11. $R_{11} := z_{21}$	12. $R_{12} := z_{22}$
13. $R_{13} := z_{23}$	14. $R_{14} := z_{24}$		
15. $R_{15} := R_1 * R_{11}$	16. $R_{15} := R_{15} + R_5$	17. $R_{11} := R_2 * R_{11}$	18. $R_{11} := R_6 + R_{11}$
19. $R_{16} := R_1 * R_{15}$	20. $R_{16} := R_{16} + R_{11}$	21. $R_{17} := R_{11} * R_{16}$	22. $R_{18} := R_{15} * R_{15}$
23. $R_{18} := R_{18} * R_2$	24. $R_{17} := R_{17} + R_{18}$	25. $R_{13} := R_{17} * R_{13}$	26. $R_4 := R_4 * R_{14}$
27. $R_4 := R_4 + R_8$	28. $R_3 := R_3 * R_{14}$	29. $R_3 := R_3 + R_7$	30. $R_{14} := R_{16} + R_{15}$
31. $R_{16} := R_{16} * R_4$	32. $R_{15} := R_{15} * R_3$	33. $R_3 := R_4 + R_3$	34. $R_3 := R_{14} * R_3$
35. $R_3 := R_3 - R_{16}$	36. $R_4 := 1 + R_1$	37. $R_4 := R_{15} * R_4$	38. $R_2 := R_2 * R_{15}$
39. $R_2 := R_{16} + R_2$	40. $R_3 := R_3 - R_4$	41. $R_4 := R_{17} * R_3$	42. $R_{14} := R_2 * R_9$
43. $R_2 := R_2 * R_3$	44. $R_8 := R_4 * R_8$	45. $R_4 := R_4 * R_7$	46. $R_7 := R_3 * R_3$
47. $R_3 := R_3 * R_9$	48. $R_{11} := R_7 * R_{11}$	49. $R_{15} := R_7 + R_2$	50. $R_2 := R_2 * R_6$
51. $R_8 := R_2 + R_8$	52. $R_{16} := R_3 * R_{14}$	53. $R_{17} := R_{13} * R_{13}$	54. $R_9 := R_{13} * R_9$
55. $R_{13} := R_{15} * R_{17}$	56. $R_{15} := R_{15} * R_7$	57. $R_7 := R_7 * R_5$	58. $R_{16} := R_7 + R_{16}$
59. $R_5 := R_5 + R_6$	60. $R_5 := R_{15} * R_5$	61. $R_2 := R_5 - R_2$	62. $R_2 := R_2 - R_7$
63. $R_2 := R_2 + R_4$	64. $R_4 := h_2 * R_9$	65. $R_4 := R_{14} + R_4$	66. $R_4 := R_{14} * R_4$
67. $R_5 := R_3 * R_3$	68. $R_6 := R_5 * R_8$	69. $R_7 := R_3 * R_9$	70. $R_8 := R_9 * R_9$
71. $R_{14} := R_{15} + R_8$	72. $R_{15} := h_2 * R_7$	73. $R_{15} := R_{15} + R_{15}$	74. $R_1 := R_1 * R_{15}$
75. $R_1 := R_4 + R_1$	76. $R_1 := R_1 + R_{11}$	77. $R_4 := h_1 * R_7$	78. $R_1 := R_1 + R_4$
79. $R_1 := R_1 + R_{13}$	80. $R_2 := R_2 + R_1$	81. $R_2 := R_5 * R_2$	82. $R_4 := R_5 * R_7$
83. $R_{11} := h_2 * R_7$	84. $R_{11} := R_{14} + R_{11}$	85. $R_{13} := h_2 * R_7$	86. $R_{13} := R_{16} + R_{13}$
87. $R_{13} := R_{13} + R_{15}$	88. $R_{14} := R_{13} * R_1$	89. $R_6 := R_{14} + R_6$	90. $R_{13} := R_{13} * R_{15}$
91. $R_2 := R_{13} + R_2$	92. $R_{13} := h_1 * R_4$	93. $R_2 := R_2 + R_{13}$	94. $R_{13} := h_0 * R_4$
95. $R_6 := R_6 + R_{13}$			
$Up_1 := R_{11}$	$Up_0 := R_1$	$Vp_1 := R_2$	$Vp_0 := R_6$
$Zp_1 := R_3$	$Zp_2 := R_9$	$zp_1 := R_5$	$zp_2 := R_8$
$zp_3 := R_7$	$zp_4 := R_4$		

Number of registers used $= 18$

Algorithm HCADD$^{\mathcal{N}e}_{h_2=0}$ of [86]
Curve Constants Used: h_0, h_1.
Input Variables: $U_{21}, U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}, z_{12}, z_{13}, z_{14}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{11}$	2. $R_2 := U_{10}$	3. $R_3 := V_{11}$	4. $R_4 := V_{10}$
5. $R_5 := Z_{11}$	6. $R_6 := Z_{12}$	7. $R_7 := z_{11}$	8. $R_8 := z_{12}$
9. $R_9 := z_{13}$	10. $R_{10} := z_{14}$	11. $R_{11} := U_{21}$	12. $R_{12} := U_{20}$
13. $R_{13} := V_{21}$	14. $R_{14} := V_{20}$	15. $R_{15} := Z_{21}$	16. $R_{16} := Z_{22}$
17. $R_{17} := z_{21}$	18. $R_{18} := z_{22}$	19. $R_{19} := z_{23}$	20. $R_{20} := z_{24}$
21. $R_3 := R_3 * R_{20}$	22. $R_4 := R_4 * R_{20}$	23. $R_{20} := R_2 * R_{17}$	24. $R_{21} := R_1 * R_{17}$
25. $R_9 := R_9 * R_{19}$	26. $R_{13} := R_{13} * R_{10}$	27. $R_3 := R_3 + R_{13}$	28. $R_{10} := R_{14} * R_{10}$
29. $R_4 := R_4 + R_{10}$	30. $R_{14} := R_4 + R_3$	31. $R_{12} := R_{12} * R_7$	32. $R_{19} := R_{20} + R_{12}$
33. $R_{20} := R_{19} * R_7$	34. $R_{11} := R_{11} * R_7$	35. $R_{21} := R_{21} + R_{11}$	36. $R_{22} := R_1 * R_{21}$
37. $R_{20} := R_{22} + R_{20}$	38. $R_{22} := R_{19} * R_{20}$	39. $R_4 := R_{20} * R_4$	40. $R_{23} := R_{21} * R_{21}$
41. $R_{23} := R_{23} * R_2$	42. $R_{22} := R_{22} + R_{23}$	43. $R_9 := R_{22} * R_9$	44. $R_3 := R_{21} * R_3$
45. $R_{21} := R_{21} * R_7$	46. $R_{20} := R_{20} + R_{21}$	47. $R_{14} := R_{20} * R_{14}$	48. $R_{14} := R_{14} + R_4$
49. $R_{20} := R_9 * R_9$	50. $R_1 := R_7 + R_1$	51. $R_1 := R_3 * R_1$	52. $R_1 := R_{14} + R_1$
53. $R_2 := R_3 * R_2$	54. $R_2 := R_4 + R_2$	55. $R_3 := R_{21} + R_{11}$	56. $R_4 := R_1 * R_1$
57. $R_3 := R_4 * R_3$	58. $R_4 := R_7 * R_{17}$	59. $R_7 := R_{20} * R_4$	60. $R_3 := R_3 + R_7$
61. $R_3 := R_{21} * R_3$	62. $R_7 := R_9 * R_4$	63. $R_9 := R_2 * R_4$	64. $R_4 := R_1 * R_4$
65. $R_{14} := R_{22} * R_4$	66. $R_1 := R_1 * R_4$	67. $R_2 := R_2 * R_4$	68. $R_{17} := R_{19} * R_1$
69. $R_{19} := R_{21} * R_1$	70. $R_{13} := R_{14} * R_{13}$	71. $R_{10} := R_{14} * R_{10}$	72. $R_{14} := R_9 * R_4$
73. $R_{20} := R_2 + R_1$	74. $R_1 := R_1 * R_{11}$	75. $R_2 := R_2 * R_{12}$	76. $R_{11} := R_{12} + R_{11}$
77. $R_{11} := R_{20} * R_{11}$	78. $R_{11} := R_{11} + R_1$	79. $R_1 := R_1 + R_{14}$	80. $R_{11} := R_{11} + R_2$
81. $R_{11} := R_{11} + R_{13}$	82. $R_2 := R_2 + R_{10}$	83. $R_9 := R_9 * R_9$	84. $R_3 := R_9 + R_3$
85. $R_3 := R_3 + R_{17}$	86. $R_9 := R_4 * R_4$	87. $R_2 := R_9 * R_2$	88. $R_{10} := R_4 * R_7$
89. $R_{12} := R_7 * R_7$	90. $R_{13} := R_{19} + R_{12}$	91. $R_1 := R_1 + R_{13}$	92. $R_{14} := R_1 * R_{13}$
93. $R_{17} := h_1 * R_{10}$	94. $R_3 := R_3 + R_{17}$	95. $R_1 := R_1 * R_3$	96. $R_1 := R_1 + R_2$
97. $R_2 := R_{11} + R_3$	98. $R_2 := R_9 * R_2$	99. $R_{11} := R_9 * R_{10}$	100. $R_2 := R_{14} + R_2$
101. $R_{14} := R_{11} * h_1$	102. $R_2 := R_2 + R_{14}$	103. $R_{14} := R_{11} * h_0$	104. $R_1 := R_1 + R_{14}$
$Up_1 := R_{13}$	$Up_0 := R_3$	$Vp_1 := R_2$	$Vp_0 := R_1$
$Zp_1 := R_4$	$Zp_2 := R_7$	$zp_1 := R_9$	$zp_2 := R_{12}$
$zp_3 := R_{10}$	$zp_4 := R_{11}$		

Number of registers used $= 23$

Algorithm mHCADD$^{\mathcal{N}e}_{h_2=0}$ of [86]
Curve Constants Used: h_1, h_0.
Input Variables: $U_{21}, U_{11}, U_{10}, V_{11}, V_{10}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}$
Output Variables: $Up_1, Up_0, Vp_1, Vp_0, Zp_1, Zp_2, zp_1, zp_2, zp_3, zp_4$

1. $R_1 := U_{21}$	2. $R_2 := U_{20}$	3. $R_3 := V_{21}$	4. $R_4 := V_{20}$
5. $R_5 := Z_{21}$	6. $R_6 := Z_{22}$	7. $R_7 := z_{21}$	8. $R_8 := z_{22}$
9. $R_9 := z_{23}$	10. $R_{10} := z_{24}$		
11. $R_{11} := U11 * R_7$	12. $R_{11} := R_{11} + R_1$	13. $R_7 := U10 * R_7$	14. $R_7 := R_2 + R_7$
15. $R_{12} := U11 * R_{11}$	16. $R_{12} := R_{12} + R_7$	17. $R_{13} := R_7 * R_{12}$	18. $R_{14} := V10 * R_{10}$
19. $R_{14} := R_{14} + R_4$	20. $R_{10} := V11 * R_{10}$	21. $R_{10} := R_{10} + R_3$	22. $R_{15} := R_{12} + R_{11}$
23. $R_{12} := R_{12} * R_{14}$	24. $R_{11} := R_{11} * R_{10}$	25. $R_{10} := R_{14} + R_{10}$	26. $R_{10} := R_{15} * R_{10}$
27. $R_{10} := R_{10} - R_{12}$	28. $R_{14} := R_{11} * R_{11}$	29. $R_{14} := R_{14} * U10$	30. $R_{13} := R_{13} + R_{14}$
31. $R_9 := R_{13} * R_9$	32. $R_{14} := R_9 * R_9$	33. $R_9 := R_9 * R_5$	34. $R_{15} := 1 + U11$
35. $R_{15} := R_{11} * R_{15}$	36. $R_{11} := U10 * R_{11}$	37. $R_{11} := R_{12} + R_{11}$	38. $R_{10} := R_{10} - R_{15}$
39. $R_{12} := R_{13} * R_{10}$	40. $R_{13} := R_{11} * R_5$	41. $R_{11} := R_{11} * R_{10}$	42. $R_{15} := R_{10} * R_{10}$
43. $R_5 := R_{10} * R_5$	44. $R_7 := R_7 * R_{15}$	45. $R_{10} := R_{11} * R_{15}$	46. $R_{16} := R_9 * R_9$
47. $R_{10} := R_{10} + R_{16}$	48. $R_{17} := R_5 * R_9$	49. $R_{18} := R_{15} * U11$	50. $R_{14} := R_{18} + R_{14}$
51. $R_{11} := R_{11} * R_{14}$	52. $R_{14} := R_{13} * R_5$	53. $R_{18} := R_5 * R_5$	54. $R_{19} := R_1 + R_2$
55. $R_{13} := R_{13} * R_{13}$	56. $R_{11} := R_{13} + R_{11}$	57. $R_7 := R_{11} + R_7$	58. $R_4 := R_{12} * R_4$
59. $R_3 := R_{12} * R_3$	60. $R_1 := R_{15} * R_1$	61. $R_{11} := R_{15} + R_{11}$	62. $R_{11} := R_{11} * R_{19}$
63. $R_2 := R_{11} * R_2$	64. $R_{11} := R_{11} - R_2$	65. $R_2 := R_2 + R_4$	66. $R_4 := R_{11} - R_1$
67. $R_3 := R_4 + R_3$	68. $R_1 := R_1 + R_{14}$	69. $R_1 := R_1 + R_{10}$	70. $R_4 := R_1 * R_{10}$
71. $R_{11} := h_1 * R_{17}$	72. $R_7 := R_7 + R_{11}$	73. $R_1 := R_1 * R_7$	74. $R_{11} := h_1 * R_{17}$
75. $R_3 := R_3 + R_{11}$	76. $R_3 := R_3 + R_7$	77. $R_3 := R_{18} * R_3$	78. $R_3 := R_4 + R_3$
79. $R_4 := h_0 * R_{17}$	80. $R_2 := R_2 + R_4$	81. $R_2 := R_{18} * R_2$	82. $R_4 := R_{18} * R_{17}$
83. $R_1 := R_1 + R_2$			
$Up_1 := R_{10}$	$Up_0 := R_7$	$Vp_1 := R_3$	$Vp_0 := R_1$
$Zp_1 := R_5$	$Zp_2 := R_9$	$zp_1 := R_{18}$	$zp_2 := R_{16}$
$zp_3 := R_{17}$	$zp_4 := R_4$		

Number of registers used $= 19$

Algorithm HCADD$^{\mathcal{A}}$ of [84]
Curve Constants Used: h_2, h_1, h_0, f_4.
Input Variables: $u_{10}, u_{11}, v_{10}, v_{11}, u_{20}, u_{21}, v_{20}, v_{21}$
Output Variables: up_0, up_1, vp_0, vp_1

1. $R_1 := u_{10}$	2. $R_2 := u_{11}$	3. $R_3 := v_{10}$	4. $R_4 := v_{11}$
5. $R_5 := u_{20}$	6. $R_6 := u_{21}$	7. $R_7 := v_{20}$	8. $R_8 := v_{21}$
9. $R_9 := R_5 - R_1$	10. $R_3 := R_3 - R_7$	11. $R_4 := R_4 - R_8$	12. $R_{10} := R_3 + R_4$
13. $R_{11} := R_2 - R_6$	14. $R_{12} := R_2 * R_{11}$	15. $R_{12} := R_{12} + R_9$	16. $R_9 := R_9 * R_{12}$
17. $R_3 := R_{12} * R_3$	18. $R_{13} := R_{11} * R_{11}$	19. $R_{13} := R_{13} * R_1$	20. $R_9 := R_9 + R_{13}$
21. $R_4 := R_{11} * R_4$	22. $R_{11} := R_{12} + R_{11}$	23. $R_{10} := R_{11} * R_{10}$	24. $R_{10} := R_{10} - R_3$
25. $R_{11} := 1 + R_2$	26. $R_{11} := R_4 * R_{11}$	27. $R_4 := R_1 * R_4$	28. $R_3 := R_3 - R_4$
29. $R_4 := R_{10} - R_{11}$	30. $R_{10} := R_9 * R_4$	31. $R_4 := R_4 * R_4$	32. $R_{10} := 1/R_{10}$
33. $R_4 := R_4 * R_{10}$	34. $R_{10} := R_9 * R_{10}$	35. $R_9 := R_9 * R_{10}$	36. $R_3 := R_3 * R_{10}$
37. $R_{10} := R_3 - R_{11}$	38. $R_2 := R_3 - R_2$	39. $R_{11} := h_2 * R_9$	40. $R_{10} := R_{10} + R_{11}$
41. $R_2 := R_2 * R_{10}$	42. $R_1 := R_2 - R_1$	43. $R_2 := 2 * R_6$	44. $R_2 := R_2 + R_{11}$
45. $R_2 := R_2 - f_4$	46. $R_{10} := 2 * R_3$	47. $R_{10} := R_{10} - R_{11}$	48. $R_{11} := h_2 * R_9$
49. $R_{10} := R_{10} + R_{11}$	50. $R_{11} := R_9 * R_9$	51. $R_2 := R_2 * R_{11}$	52. $R_{10} := R_{10} - R_{11}$
53. $R_{11} := R_5 * R_3$	54. $R_{12} := 2 * R_8$	55. $R_{12} := h_1 + R_{12}$	56. $R_9 := R_{12} * R_9$
57. $R_{12} := R_6 + R_3$	58. $R_3 := R_6 * R_3$	59. $R_3 := R_3 + R_5$	60. $R_1 := R_1 + R_3$
61. $R_1 := R_1 + R_9$	62. $R_1 := R_1 + R_2$	63. $R_2 := R_{12} - R_{10}$	64. $R_5 := R_{10} * R_2$
65. $R_6 := h_2 * R_{10}$	66. $R_5 := R_5 + R_1$	67. $R_3 := R_5 - R_3$	68. $R_3 := R_3 * R_4$
69. $R_3 := R_3 - R_8$	70. $R_3 := R_3 - h_1$	71. $R_3 := R_3 + R_6$	72. $R_2 := R_1 * R_2$
73. $R_5 := h_2 * R_1$	74. $R_2 := R_2 - R_{11}$	75. $R_2 := R_2 * R_4$	76. $R_2 := R_2 - R_7$
77. $R_2 := R_2 - h_0$	78. $R_2 := R_2 + R_5$		
$up_0 := R_1$	$up_1 := R_{10}$	$vp_0 := R_2$	$vp_1 := R_3$

Number of registers used $= 13$

Algorithm HCADD$^{\mathcal{A}}of$ [133]
Input Variables: $u_{10}, u_{11}, v_{10}, v_{11}, u_{20}, u_{21}, v_{20}, v_{21}$
Output Variables: up_0, up_1, vp_0, vp_1

1. $R_1 := u_{10}$	2. $R_2 := u_{11}$	3. $R_3 := v_{10}$	4. $R_4 := v_{11}$
5. $R_5 := u_{20}$	6. $R_6 := u_{21}$	7. $R_7 := v_{20}$	8. $R_8 := v_{21}$
9. $R_9 := R_2 + R_6$	10. $R_{10} := 1 + R_2$	11. $R_4 := R_4 - R_8$	12. $R_3 := R_3 - R_7$
13. $R_{11} := R_3 + R_4$	14. $R_{12} := R_5 - R_1$	15. $R_{13} := R_2 - R_6$	16. $R_4 := R_{13} * R_4$
17. $R_{10} := R_4 * R_{10}$	18. $R_4 := R_1 * R_4$	19. $R_{14} := R_2 * R_{13}$	20. $R_{14} := R_{14} + R_{12}$
21. $R_{12} := R_{12} * R_{14}$	22. $R_3 := R_{14} * R_3$	23. $R_{14} := R_{14} + R_{13}$	24. $R_{11} := R_{14} * R_{11}$
25. $R_{11} := R_{11} - R_3$	26. $R_{10} := R_{11} - R_{10}$	27. $R_3 := R_3 - R_4$	28. $R_4 := R_{13} * R_{13}$
29. $R_4 := R_4 * R_1$	30. $R_4 := R_{12} + R_4$	31. $R_{11} := R_4 * R_{10}$	32. $R_{10} := R_{10} * R_{10}$
33. $R_{11} := 1/R_{11}$	34. $R_{10} := R_{10} * R_{11}$	35. $R_{11} := R_4 * R_{11}$	36. $R_4 := R_4 * R_{11}$
37. $R_3 := R_3 * R_{11}$	38. $R_{11} := R_4 * R_4$	39. $R_9 := R_{11} * R_9$	40. $R_{12} := R_6 + R_3$
41. $R_{13} := R_3 - R_2$	42. $R_6 := R_6 * R_3$	43. $R_6 := R_6 + R_5$	44. $R_5 := R_5 * R_3$
45. $R_3 := R_3 + R_{12}$	46. $R_3 := R_3 - R_2$	47. $R_3 := R_3 - R_{11}$	48. $R_2 := R_{12} - R_2$
49. $R_2 := R_{13} * R_2$	50. $R_1 := R_2 - R_1$	51. $R_1 := R_1 + R_6$	52. $R_1 := R_1 + R_4$
53. $R_1 := R_1 + R_9$	54. $R_2 := R_{12} - R_3$	55. $R_4 := R_3 * R_2$	56. $R_4 := R_4 + R_1$
57. $R_2 := R_1 * R_2$	58. $R_2 := R_2 - R_5$	59. $R_4 := R_4 - R_6$	60. $R_4 := R_4 * R_{10}$
61. $R_2 := R_2 * R_{10}$	62. $R_4 := R_4 - R_8$	63. $R_2 := R_2 - R_7$	64. $R_4 := R_4 - 1$
$up_0 := R_1$	$up_1 := R_3$	$vp_0 := R_2$	$vp1 := R_4$

Number of registers used $= 14$

Appendix C

HECC Addition and Doubling Formulae in Atomic Blocks

Table C.1: HCDBL and HCADD Algorithms as One Atomic Block

	Algorithm HCADD	**Algorithm HCDBL**
Input:	$D_1 = (u_{10}, u_{11}, v_{10}, v_{11})$ $D_2 = (u_{20}, u_{21}, v_{20}, v_{21})$	$D = (u_0, u_1, v_0, v_1)$
Output:	$D_1 + D_2 = (u'_0, u'_1, v'_0, v'_1)$	$2D = (u'_0, u'_1, v'_0, v'_1)$
Init:	$T_1 \leftarrow u_{10}, T_2 \leftarrow u_{11}$ $T_3 \leftarrow v_{10}, T_4 \leftarrow v_{11}, T_5 \leftarrow u_{20}$ $T_6 \leftarrow u_{21}, T_7 \leftarrow v_{20}, T_8 \leftarrow v_{21}$	$T_1 \leftarrow u_0, T_2 \leftarrow u_1$ $T_3 \leftarrow v_0, T_4 \leftarrow v_1$
1	$T_9 = T_2 + h_2T_6\ (inv_1)$	$T_5 = h_1 + T_2\ (\tilde{v}_1 = inv_1)$
2	$T_{10} = T_1 + T_5\ (z_2)$	$T_7 = h_0 + T_1\ (\tilde{v}_0)$
3	$T_{11} = T_9 * T_9$	$T_6 = T_2 * T_2\ (w_1)$
4	$*[s]$	$T_8 = T_5 * T_5\ (w_2)$
5	$T_{13} = T_2 * T_9$	$T_9 = T_2 * T_5\ (w_3)$
6	$T_{11} = T_1 * T_{11}$	$T_8 = T_1 * T_8$
7	$T_{12} = T_{10} + T_{13}\ (inv_0)$	$T_9 = T_7 + T_9\ (inv_0)$
8	$T_{13} = T_{10} * T_{12}$	$T_7 = T_7 * T_9$
9	$T_{10} = T_{11} + T_{13}\ (r)$	$T_7 = T_8 + T_7\ (r)$
10	$T_3 = T_3 + T_7\ (w_0)$	$T_8 = f_3 + T_6\ (w_3)$
11	$T_4 = T_4 + T_8\ (w_1)$	$T_{10} = T_8 + h_2T_4\ (k'_1)$
12	$T_{11} = T_9 + T_{12}$	$T_8 = h_2T_4 + T_8$
13	$T_{13} = T_3 + T_4$	$T_8 = f_2 + h_2T_3$
14	$T_{14} = 1 + T_2$	$T_{11} = h_1 + T_4$
15	$T_3 = T_3 * T_{12}\ (w_2)$	$T_8 = T_2 * T_8$
16	$T_4 = T_4 * T_9\ (w_3)$	$T_{11} = T_4 * T_{11}$
17	$*[a]$	$T_8 = T_8 + T_{11}$
18	$*[a]$	$T_8 = T_6 + T_8\ (k'_0)$
19	$T_{12} = T_{11} * T_{13}$	$T_{11} = T_8 * T_9\ (w_0)$
20	$T_{13} = T_4 * T_{14}$	$T_{12} = T_{10} * T_5\ (w_1)$
21	$T_{12} = T_3 + T_{12}$	$T_5 = T_5 + T_9$
22	$T_{12} = T_{12} + T_{13}\ (s'_1)$	$T_8 = T_8 + T_{10}$
23	$T_4 = T_1 * T_4$	$T_5 = T_5 * T_8$
24	$T_4 = T_3 + T_4\ (s'_0)$	$T_5 = T_5 + T_{11}$
25	$*[a]$	$T_8 = 1 + T_2$
26	$T_3 = T_{10} * T_{12}$	$T_8 = T_{12} * T_8$
27	$*[a]$	$T_5 = T_5 + T_8\ (s'_1)$
28	$*[m]$	$T_8 = T_7 * T_5$
29	$T_3 = 1/T_3\ (w_1)$	$T_8 = 1/T_8\ (w_1)$
30	$T_{13} = T_3 * T_{10}\ (w_2)$	$T_9 = T_7 * T_8\ (w_2)$

Table C.2: HCDBL and HCADD Algorithms as One Atomic Block (Contd..)

	Algorithm HCADD	**Algorithm HCDBL**
31	$T_{12} = T_{12} * T_{12}$	$T_5 = T_5 * T_5$
32	$T_3 = T_3 * T_{12}$ (w_3)	$T_5 = T_5 * T_8$ (w_3)
33	$T_{10} = T_{10} * T_{13}$ (w_4)	$T_7 = T_7 * T_9$ (w_4)
34	$T_{12} = T_{10} * T_{10}$ (w_5)	$T_8 = T_7 * T_7$ (w_5)
35	$T_4 = T_4 * T_{13}$ (s''_0)	$T_6 = T_1 * T_{12}$
36	$T_{13} = T_4 + T_6$ (l'_2)	$T_6 = T_{11} + T_6$ (s'_0)
37	$T_{14} = T_4 * T_6$	$T_6 = T_6 * T_9$ (s''_0)
38	$T_{14} = T_{14} + T_5$ (l'_1)	$T_9 = T_2 + T_6$ (l'_2)
39	$T_{11} = T_4 * T_5$ (l'_0)	$T_{10} = T_2 * T_6$
40	$T_2 = T_2 + T_4$	$T_{10} = T_{10} + T_1$ (l'_1)
41	$T_4 = T_4 + T_9$	$T_2 = T_6 + T_2$
42	$T_4 = T_4 + h_2 T_{10}$	$T_2 = h_2 T_2 + h_1$
43	*[s]	$T_{11} = T_6 * T_6$
44	$T_4 = T_2 * T_4$	$T_6 = T_1 * T_6$ (l'_0)
45	$T_2 = T_9 + T_{10}$	$*[a]$
46	$T_9 = T_9 * T_{12}$	$T_2 = T_7 * T_2$
47	$T_4 = T_1 + T_4$	$T_1 = T_{11} + T_2$ (u'_0)
48	$T_4 = T_4 + T_{14}$	$T_2 = h_2 T_7 + T_8$ (u'_1)
49	$T_2 = T_2 + T_{12}$ (u'_1)	$T_7 = T_9 + T_6$ (w_1)
50	$T_{10} = h_1 * T_{10}$	$*[m]$
51	$T_4 = T_4 + T_{10}$	*[a]
52	$T_1 = T_4 + T_9$ (u'_0)	*[a]
53	$T_{10} = T_2 + T_{13}$ (w_1)	*[a]
54	$T_9 = T_2 * T_{10}$	$T_8 = T_2 * T_7$
55	$T_9 = T_1 + T_9$	$T_8 = T_8 + T_1$
56	$T_9 = T_9 + T_{14}$ (w_2)	$T_8 = T_8 + T_{10}$ (w_2)
57	$T_9 = T_3 * T_9$	$T_8 = T_8 * T_5$
58	$T_9 = T_8 + T_9$	$T_8 = T_8 + T_4$
59	$T_9 = h_1 + T_9$	$T_8 = T_8 + h_1$
60	$T_4 = h_2 T_2 + T_9$ (v'_1)	$T_4 = T_8 + h_2 T_2$ (v'_1)
61	$T_9 = T_1 * T_{10}$	$T_9 = T_1 * T_7$
62	$T_9 = T_9 + T_{11}$ (w_2)	$T_7 = T_6 + T_9$ (w_2)
63	$T_9 = T_3 * T_9$	$T_5 = T_7 * T_5$
64	$T_9 = T_7 + T_9$	$T_5 = T_5 + T_3$
65	$T_9 = h_0 + T_9$	$T_7 = T_5 + h_0$
66	$T_3 = h_2 T_1 + T_9$ (v'_0)	$T_3 = T_7 + h_2 T_1$ (v'_1)
	$u'_0 \leftarrow T_1, u'_1 \leftarrow T_2, v'_0 \leftarrow T_3, v'_1 \leftarrow T_4$	$u'_0 \leftarrow T_1, u'_1 \leftarrow T_2, v'_0 \leftarrow T_3, v'_1 \leftarrow T_4$

Table C.3: HCDBL Algorithm in Atomic Blocks

Algorithm HCDBL
Input: $D = (U_0, U_1, V_0, V_1, Z_1, Z_2, z_1, z_2)$
Output: $2D = (U_0', U_1', V_0', V_1', Z_1', Z_2', z_1', z_2')$
Init: $T_1 = U_0, T_2 = U_1, T_3 = V_0, T_4 = V_1, T_5 = Z_1, T_6 = Z_2, T_7 = z_1, T_8 = z_2$

Γ_1	$T_{10} = T_1 * T_7\ (\tilde{U}_0)$	Γ_2	$T_9 = T_4 * T_4\ (w_0)$
	*		*
	*		*
	*		*
Γ_3	$T_{11} = T_2 * T_2\ (w_1)$	Γ_4	$T_{12} = T_3 * T_7$
	*		*
	*		*
	*		*
Γ_5	$T_{13} = T_2 * T_4\ (U_1V_1)$	Γ_6	$T_{14} = T_9 * T_1\ (w_0U_0)$
	*		*
	$T_{13} = -T_{13}$		$T_{10} = -T_{10}$
	$T_{13} = T_{12} + T_{13}\ (inv_0)$		$T_{15} = T_{11} - T_{10}$
Γ_7	$T_{12} = T_7 * T_7$	Γ_8	$T_{16} = T_3 * T_{13}$
	*		$T_{14} = T_{14} + T_{16}\ (r)$
	*		$T_{10} = -T_{10}$
	*		$T_{17} = T_{15} + T_{15}$
Γ_9		Γ_{10}	$T_{12} = T_{12} * T_7$
	$T_{15} = T_{15} + T_{11}\ (f_3z_3 + w_1)$		$T_{16} = T_{10} + T_{10}$
	*		*
	$T_{17} = T_{17} + T_{15}$		$T_{16} = T_{16} + T_{16}$
Γ_{11}	$T_6 = T_6 * T_{14}$	Γ_{12}	$T_{17} = T_8 * T_{17}$
	*		
	*		$T_{15} = -T_{15}$
	*		$T_{16} = T_{16} + T_{15}$
Γ_{13}	$T_{12} = T_{12} * f_2$	Γ_{14}	$T_{16} = T_2 * T_{16}$
	*		*
	$T_4 = -T_4\ (inv_1)$		
	*		*
Γ_{15}	$T_6 = T_6 * T_7$	Γ_{16}	$T_{11} = T_{17} * T_4\ (w_1)$
	*		$T_{12} = T_{16} + T_{12}$
	*		$T_{15} = -T_{15}$
	*		*

Table C.4: HCDBL Algorithm in Atomic Blocks

Algorithm HCDBL(Contd..)

Γ_{17}	$T_{12} = T_8 * T_{12}$ $T_{16} = 1 + T_2$ $(1 + U1)$ $T_9 = -T_9$ $T_{12} = T_{12} + T_9$ (k_0)	Γ_{18}	$T_5 = T_6 * T_5$ $T_5 = T_5 + T_5$ $*$ $*$
Γ_{19}	$T_6 = T_6 * T_6$ $*$ $*$ $*$	Γ_{20}	$T_{16} = T_{11} * T_{16}$ $*$ $*$ $*$
Γ_{21}	$T_{15} = T_{12} * T_{13}$ $(k_0 inv_0)$ $T_{13} = T_{13} + T_4$ $*$ $T_{12} = T_{12} + T_{17}$	Γ_{22}	$T_{10} = T_{11} * T_{10}$ $*$ $*$ $*$
Γ_{23}	$T_6 = T_6 * T_2$ $T_6 = T_6 + T_6$ $T_{10} = -T_{10}$ $T_{10} = T_{15} + T_{10}$ (s_0)	Γ_{24}	$T_{12} = T_{13} * T_{12}$ $T_{12} = T_{12} - T_{15}$ $T_{16} = -T_{16}$ $T_{12} = T_{12}T_{12} = T_{12} + T_{16}$ (s_1) (s_1)
Γ_{25}	$T_7 = T_{12} * T_7$ (Z_1') $*$ $*$ $*$	Γ_{26}	$T_9 = T_{10} * T_{10}$ (S_0) $*$ $T_{10} = -T_{10}$ $*$
Γ_{27}	$T_8 = T_7 * T_7$ (z_1') $*$ $*$ $*$	Γ_{28}	$T_{17} = T_{10} * T_7$ (S) $*$ $*$ $*$
Γ_{29}	$T_{14} = T_{14} * T_7$ (R) $*$ $T_4 = -T_4$ $*$	Γ_{30}	$T_{11} = T_5 * T_5$ (z_2') $T_{13} = T_{17} + T_{17}$ $(2S)$ $T_{11} = -T_{11}$ $T_{13} = T_{13} - T_{11}$ (U_1')
Γ_{31}	$T_4 = T_{14} * T_4$ $(R * V_1)$ $T_6 = T_4 + T_6$ $*$ $T_6 = T_6 + T_6$	Γ_{32}	$T_{10} = T_{10} * T_{12}$ (s_0) $*$ $T_{11} = -T_{11}$ $*$

Table C.5: HCDBL Algorithm in Atomic Blocks

Algorithm HCDBL(Contd..)

Γ_{33}	$T_{12} = T_7 * T_{12}$ (s_1)	Γ_{34}	$T_{10} = T_{10} * T_1$ (l_0)
	$T_{15} = T_{10} + T_{12}$		$T_6 = T_6 + T_6$
	$*$		$*$
	$T_{16} = T_1 + T_2$		$T_1 = T_9 + T_6$ (U'_0)
Γ_{35}	$T_{12} = T_{12} * T_2$ (l_2)	Γ_{36}	$T_{14} = T_{14} * T_3$ (RV_0)
	$T_9 = T_{12} + T_{10}$		$T_3 = T_{10} + T_{14}$
	$T_1 = -T_1$		$*$
	$T_{12} = T_{12} + T_{17}$ (l_2)		$T_3 = T_3 + T_{14}$
Γ_{37}	$T_{15} = T_{15} * T_{16}$	Γ_{38}	$T_3 = T_8 * T_3$
	$T_{15} = T_{15} + T_9$ (l_1)		$T_4 = T_4 + T_4$
	$T_{13} = -T_{13}$		$T_{13} = -T_{13}$
	$T_{12} = T_{12} + T_{13}$ (l_2)		$*$
Γ_{39}	$T_{16} = T_{12} * T_1$ (w_0)	Γ_{40}	$T_{14} = T_{12} * T_{13}$ (w_1)
	$*$		$T_{15} = T_{15} + T_1$
	$T_3 = -T_3$		$T_1 = -T_1$
	$T_3 = T_{16} + T_3$ (V'_0)		$T_4 = T_{15} + T_4$
Γ_{41}	$T_4 = T_8 * T_4$		
	$*$		
	$T_4 = -T_4$		
	$T_4 = T_{14} - T_4$		

Table C.6: HCADD Algorithm in Atomic Blocks

Algorithm HCADD
Input: $D_1 = (U_{10}, U_{11}, V_{10}, V_{11}, 1, 1, 1, 1)$, $D_2 = (U_{20}, U_{21}, V_{20}, V_{21}, Z_{21}, Z_{22}, z_{21}, z_{22})$
Output: $D_1 + D_2 = (U'_0, U'_1, V'_0, V'_1, Z'_1, Z'_2, z'_1, z'_2)$
Initialisation: $T_1 = U_{10}, T_2 = U_{11}, T_3 = V_{10}, T_4 = V_{11}, T_5 = U_{20}, T_6 = U_{21}$,
$T_7 = V_{20}, T_8 = V_{21}, T_9 = Z_{21}, T_{10} = Z_{22}, T_{11} = z_{21}, T_{12} = z_{22}$

Γ_1	$T_{13} = T_9 * T_{10}$ (z_{23})	Γ_2	$T_{14} = T_2 * T_{11}$ $(U_{11}z_{21})$
	$*$		$*$
	$*$		$T_6 = -T_6$
	$*$		$T_{14} = T_{14} + T_6$ (inv_1)
Γ_3	$T_{15} = T_1 * T_{11}$ $(U_{10}z_{21})$	Γ_4	$T_{16} = T_9 * T_9$ (Z_1)
	$*$		$*$
	$T_{15} = -T_{15}$		$*$
	$T_{15} = T_5 + T_{15}$ (y_2)		$*$
Γ_5	$T_{12} = T_{11} * T_{13}$ $(z_{21}z_{23})$	Γ_6	$T_{17} = T_2 * T_{14}$
	$*$		$T_{17} = T_{17} + T_{15}$ (inv_0)
	$T_{15} = -T_{15}$		$*$
	$*$		$*$
Γ_7	$T_{13} = T_{15} * T_{17}$	Γ_8	$T_{11} = T_{14} * T_{14}$ $(inv1^2)$
	$*$		$*$
	$*$		$T_6 = -T_6$
	$*$		$*$
Γ_9	$T_{18} = T_3 * T_{12}$	Γ_{10}	$T_{19} = T_4 * T_{12}$
	$*$		$*$
	$T_7 = -T_7$		$T_8 = -T_8$
	$T_{18} = T_{18} + T_7$ (w_0)		$T_{19} = T_{19} + T_8$ (w_1)
Γ_{11}	$T_{10} = T_{10} * T_{16}$ $(\tilde{Z}_2)$	Γ_{12}	$T_{18} = T_{17} * T_{18}$ (w_2)
	$*$		$T_{17} = T_{17} + T_{14}$
	$T_7 = -T_7$		$T_8 = -T_8$
	$*$		$T_{12} = T_{18} + T_{19}$
Γ_{13}	$T_{19} = T_{14} * T_{19}$ (w_3)	Γ_{14}	$T_{17} = T_{17} * T_{12}$
	$*$		$*$
	$*$		$T_{18} = -T_{18}$
	$*$		$T_{17} = T_{17} + T_{18}$
Γ_{15}	$T_{11} = T_{11} * T_1$	Γ_{16}	$T_{12} = T_{19} * T_2$
	$T_{11} = T_{13} + T_{11}$ (r)		$*$
	$*$		$T_{12} = -T_{12}$
	$*$		$T_{12} = T_{17} + T_{12}$ (s_1)

Table C.7: HCADD Algorithm in Atomic Blocks

Algorithm HCADD(Contd..)

Γ_{17}	$T_{10} = T_{10} * T_{11}$ $*$ $T_{12} = -T_{12}$ $*$	Γ_{18}	$T_{13} = T_1 * T_{19}$ $*$ $T_{13} = -T_{13}$ $T_{18} = T_{18} + T_{13}$ (s_0)
Γ_{19}	$T_{13} = T_9 * T_{10}$ (Z'_2) $*$ $*$ $*$	Γ_{20}	$T_{17} = T_{10} * T_{10}$ $(t3nZ2)$ $T_{19} = T_6 + T_6$ $*$ $T_{19} = T_{19} + T_{14}$
Γ_{21}	$T_{10} = T_{12} * T_{16}$ (Z'_1) $*$ $*$ $*$	Γ_{22}	$T_{19} = T_{17} * T_{19}$ $*$ $*$ $*$
Γ_{23}	$T_{17} = T_{12} * T_{12}$ (S_1) $*$ $*$ $*$	Γ_{24}	$T_{16} = T_{18} * T_{16}$ (S_0) $*$ $*$ $*$
Γ_{25}	$T_{20} = T_{10} * T_{16}$ (S) $*$ $*$ $*$	Γ_{26}	$T_9 = T_{16} * T_{16}$ (S_0) $*$
Γ_{27}	$T_{18} = T_{18} * T_{10}$ (s_0) $*$ $*$ $*$	Γ_{28}	$T_{12} = T_{12} * T_{10}$ (s_1) $*$
Γ_{29}	 $T_{21} = T_{16} + T_{20}$ (l_2) $*$ $T_{20} = T_{20} + T_{20}$ $(2S)$	Γ_{30}	$T_{11} = T_{11} * T_{10}$ (R) $*$ $*$ $*$
Γ_{31}	$T_{21} = T_{18} * T_5$ (l_0) $T_5 = T_6 + T_5$ $T_{21} = -T_{21}$ $*$	Γ_{32}	$T_{22} = T_{11} * T_8$ $(V'1)$ $T_8 = T_{18} + T_{12}$ $T_{18} = -T_{18}$ $T_{23} = T_{14} + T_6$

Table C.8: HCADD Algorithm in Atomic Blocks

Algorithm HCADD(Contd..)

Γ_{33}	$T_8 = T_8 * T_5$	Γ_{34}	$T_{17} = T_{17} * T_{23}$
	$T_8 = T_8 + T_{21}$		$T_{17} = T_{17} + T_{18}$
	$T_{16} = -T_{16}$		$*$
	$T_8 = T_8 + T_{16}$		$T_{17} = T_{17} + T_{18}$
Γ_{35}	$T_{17} = T_{14} * T_{17}$	Γ_{36}	$T_{15} = T_{15} * T_{12}$
	$T_{17} = T_9 + T_{17}$		$T_{15} = T_{15} + T_{22}$
	$*$		$T_{14} = -T_{14}$
	$*$		$T_{15} = T_{15} + T_{22}$
Γ_{37}	$T_9 = T_{13} * T_{13}$ (z'_2)	Γ_{38}	$T_6 = T_{10} * T_{10}$ (z'_1)
	$*$		$*$
	$T_9 = -T_9$		
	$T_{20} = T_{20} + T_9$		
Γ_{39}	$T_{14} = T_{14} * T_{12}$	Γ_{40}	$T_{11} = T_{11} * T_7$
	$T_{14} = T_{14} + T_{20}$		$T_{15} = T_{15} + T_{19}$
	$T_{14} = -T_{14}$		$T_9 = -T_9$
	$T_{21} = T_{21} + T_{14}$		$T_{15} = T_{17} + T_{15}$
Γ_{41}	$T_{12} = T_{21} * T_{15}$ (w_0)	Γ_{42}	$T_7 = T_{21} * T_{14}$ (w_1)
	$T_{11} = T_{21} + T_{11}$		$T_8 = T_8 + T_{22}$
	$*$		$T_{15} = -T_{15}$
	$*$		$T_8 = T_8 + T_{15}$
Γ_{43}	$T_8 = T_6 * T_8$	Γ_{44}	$T_{11} = T_6 * T_{11}$
	$*$		$*$
	$T_8 = -T_8$		$T_{11} = -T_{11}$
	$T_7 = T_7 + T_8$		$T_{12} = T_{12} + T_{11}$

$U'_0 \leftarrow T_{15}, U'_1 \leftarrow T_{14}, V'_0 \leftarrow T_{12}, V'_1 \leftarrow T_7,$
$Z'_1 \leftarrow T_{10}, Z'_2 \leftarrow T_{13}, z'_1 \leftarrow T_6, z'_2 \leftarrow T_9.$