Efficient Algorithms for Multi-Scalar Multiplications

Diploma Thesis

supervised by Prof. Tsuyoshi Takagi
Future University - Hakodate
School of Systems Information Science

presented by Erik Dahmen
Department of Mathematics
Technical University of Darmstadt

November 2005

Acknowledgment

I want to thank Prof. Tsuyoshi Takagi for investing such a great deal of time to introduce me to the very interesting and challenging topic of efficient multi-scalar multiplication. I also want to thank Dr. Katsuyuki Okeya who taught me many interesting things while supervising me during my internship at Hitachi SDL. Further, I want to thank Daniel Schepers and Katja Schmidt-Samoa for proofreading this thesis and Hans-Otto Dahmen, Kai Endres, Andreas Müller and Roland Walter for pointing out typos. Finally, I want to note that this thesis has been financially supported by Hitachi SDL in terms of the cooperation with the Technical University of Darmstadt.

Abstract

Since the internet was made accessible for the public, it is used more and more for the exchange of confidential data. Over the past few years, the number of electronic frauds increased and nowadays poses a serious threat. It is therefore extremely important to have methods to secure transactions and communications made via the internet, or in general in any electronic environment.

The science that made security in electronic environments its business is called cryptography. Nowadays, there exist several methods, so-called cryptosystems, which enable users to secure their communications. Due to their tamper resistance and mobility, cryptosystems are often implemented on smart cards. However, since smart cards have only the size of a credit card, their computational power and memory is very limited. It is therefore crucial to compute the operations required by a cryptosystem as efficient as possible.

The basic mathematic operation in cryptosystems are scalar multiplications and more general, sums of scalar multiplications, so-called multi-scalar multiplications. This thesis analyzes several methods to compute a multi-scalar multiplication in an efficient way. Here efficient means not only as fast as possible, but also using as little memory as possible. In detail, there exist several basic algorithms to compute a multi-scalar multiplication. The runtime of those algorithms can be decreased if special representations of the scalars are deployed. The emphasis of this thesis is on such representations.

This thesis is organized as follows: Chapter 1 introduces the basic concept of cryptography and smart cards. Chapter 2 discusses elliptic curves and their application to cryptography. Chapter 3 introduces the basics about integer representations. Chapter 4 reviews several basic algorithms to compute a multi-scalar multiplication and Chapters 5 and 6 introduce special representations of the scalars to speed up those algorithms. In Chapter 7, the author compares those representations and in Chapter 8, he estimates the total computational costs for computing a multi-scalar multiplication. Finally, Chapter 9 states the authors conclusion.

Contents

1 Introduction **1**
- 1.1 Encryption Schemes . . . 1
  - 1.1.1 Symmetric Schemes . . . 2
  - 1.1.2 Asymmetric Schemes . . . 2
  - 1.1.3 Hybrid Schemes . . . 2
- 1.2 Digital Signatures . . . 3
- 1.3 Certification Authorities . . . 3
- 1.4 Embedded Security . . . 4

2 Elliptic Curves **6**
- 2.1 Defining an Additive Group . . . 6
- 2.2 Coordinate Systems and Addition Formulas . . . 8
  - 2.2.1 Affine Coordinates . . . 8
  - 2.2.2 Jacobian Coordinates . . . 10
  - 2.2.3 Modified Jacobian Coordinates . . . 11
  - 2.2.4 Mixed Coordinates . . . 12
- 2.3 Elliptic Curves in Cryptography . . . 13
- 2.4 Elliptic Curve Cryptosystems . . . 14
  - 2.4.1 Diffie-Hellman Key Exchange . . . 14
  - 2.4.2 ElGamal Cryptosystem . . . 14
  - 2.4.3 Elliptic Curve Digital Signature Algorithm . . . 15

3 Representations of Integers **17**
- 3.1 The Binary Representation . . . 17
- 3.2 General Base-2 Representations . . . 17
- 3.3 The Weight of a Representation . . . 18

4 Multi-Scalar Multiplication Algorithms **20**
- 4.1 Binary Methods . . . 20
  - 4.1.1 Right-to-Left Binary Method . . . 20
  - 4.1.2 Left-to-Right Binary Method . . . 22
  - 4.1.3 Left-to-Right vs. Right-to-Left . . . 24
- 4.2 Interleave Method . . . 25

4.3 Shamir Method 27
4.4 Elliptic Curves and Precomputation 28
4.5 Lim-Lee Combing 29

5 Low-Weight Representations **31**
5.1 The width-w Non Adjacent Form 31
5.2 The Joint Sparse Form 35

6 Left-to-Right producible Low-Weight Representations **42**
6.1 The Mutual Opposite Form 42
6.2 The width-w Mutual Opposite Form 44
6.3 The Left-to-Right Joint Sparse Form 48

7 Computing a Multi-Scalar Multiplication **53**
7.1 Speeding up the Interleave Method 53
7.2 Speeding up the Shamir Method 55
7.3 Comparison 57

8 Field Operations **61**
8.1 Evaluation Stage 62
8.2 Precomputation Stage 64
8.3 Total Costs 66

9 Conclusion **69**
9.1 Outlook and Further Research 70

Bibliography **71**

List of Algorithms

1 ECDSA Signature Generation 16
2 ECDSA Signature Verification 16

3 Decimal to Binary 18

4 Right-To-Left Binary Method 21
5 General Right-To-Left Binary Method 22
6 Left-to-Right Binary Method 23
7 General Left-to-Right Binary Method 24
8 Interleave Method 26
9 Shamir Method 27

10 Decimal to wNAF 32

11 Binary to MOF 43
12 MOF to wMOF 46
13 MOF to ltrJSF 50

List of Figures

1.1 Basic layout of a smart card 5

2.1 Elliptic curve point addition and doubling 7

7.1 ECADD operations required by the Interleave method 59
7.2 Points to precompute for the Interleave method 59
7.3 ECADD operations required by the Shamir method 60
7.4 Points to precompute for the Shamir method 60

List of Tables

5.1 Example values of $\mathcal{AHD}(w\text{NAF})$ 34
5.2 Example values of $\mathcal{AJHD}_k(\text{JSF})$ 41

6.1 Example values of $\mathcal{AJHD}_k(\text{ltrJSF})$ 52

7.1 Costs for the Interleave method 57
7.2 Costs for the Shamir method 57

8.1 Coordinate systems for the evaluation stage 63
8.2 Field multiplications for the evaluation stage 64
8.3 Coordinate systems for the precomputation stage 66
8.4 Field multiplications for the precomputation stage 66
8.5 Total number of field multiplications 67

List of Abbreviations

AHD	Average Hamming density.
AJHD	Average joint Hamming density.
DLP	Discrete Logarithm Problem.
ECADD	Elliptic Curve Point Addition.
ECDBL	Elliptic Curve Point Doubling.
ECDLP	Elliptic Curve Discrete Logarithm Problem.
HD	Hamming density.
HW	Hamming weight.
JHD	Joint Hamming density.
JHW	Joint Hamming weight.
JSF	Joint Sparse Form.
ltrJSF	Left-to-Right Joint Sparse Form.
MOF	Mutual Opposite Form.
wNAF	Width-w Non Adjacent Form.
wMOF	Width-w Mutual Opposite Form.

List of Symbols

k	Number of scalars.
n	Bit length of a scalar.
d	Scalar, i.e. a positive integer.
$d[i]$	The i-th bit of the scalar d, $i = 1, \ldots, n$.
d_j	The j-th scalar, $j = 1, \ldots, k$.
$d_j[i]$	The i-th bit of the j-th scalar, $j = 1, \ldots, k$, $i = 1, \ldots, n$.
$\bar{x}$	$-x$, where x is an integer.
$\mathcal{D}$	Digit set.
$\lvert\mathcal{D}\rvert$	The order of the digit set.
$\mathcal{X}$	Class of $\mathcal{D}$-representations.
M	Field multiplication.
S	Field squaring.
I	Field inversion.
$\mathbb{F}_p$	Prime field.
$E(\mathbb{F}_p)$	Additive group of points on an elliptic curve.
$\mathcal{A}$	Affine coordinates.
$\mathcal{J}$	Jacobian coordinates.
$\mathcal{J}^m$	Modified Jacobian coordinates.

1 Introduction

In this modern, computer dominated society of ours, the necessity for electronic security cannot be denied. Every day, confidential information is sent via email, credit cards are used for electronic payment and contracts are made without the counterparties actually coming into face-to-face contact. Such procedures bear a high risk, because in an unsecured environment users cannot detect if the content of a message was read or changed by an unauthorized person during transmission. Also, there is no way to verify the identity of the conversational partner. To make those security issues more transparent, the following four major security targets have been defined.

Confidentiality Only the designated receivers of a message must be able to read its content.

Integrity The receivers of a message must be able to decide whether the content of the message has been changed during transmission or not.

Authenticity The receivers of a message must be able to verify the identity of the sender of the message.

Non-repudiation The receivers of a message must be able to prove the identity of the sender to a third person.

The science, that deals with the achievement of those security targets is called *Cryptography* and in the following, the standard cryptographic approaches to achieve confidentiality, integrity, authenticity and non-repudiation, so-called *cryptographic schemes* or *cryptosystems* are described.

1.1 Encryption Schemes

The purpose of *encryption schemes* is to cover confidentiality. As the name suggests, this is achieved by encrypting the message. This is done by an *encryption function* $\mathcal{E}$. The reverse process, the decryption, is done by a *decryption function* $\mathcal{D}$. Besides the message m, the encryption function requires the input of an *encryption key* e. It returns the encrypted message, the *ciphertext* c. The ciphertext and a *decryption key* d are the input for the decryption function

which returns the original message, the *plaintext*. The respective formulas are given as

$$\mathcal{E}_e(m) = c \qquad \mathcal{D}_d(c) = m$$

In cryptography, there are three different approaches to encrypt messages.

1.1.1 Symmetric Schemes

In *symmetric schemes* the encryption and decryption keys are the same or can easily be calculated from each other. For that reason, it is often spoken of just one *secret key* which can encrypt as well as decrypt messages and therefore must be kept secret.

While the encryption and decryption with symmetric schemes is very fast, there is a major drawback, namely the key-exchange between communicating parties. Any two persons who wish to communicate must share a distinct secret key which has to be exchanged in a secure way. First of all, the number of keys to exchange is immense and second, it is not obvious how the keys can easily be exchanged in a secure way.

1.1.2 Asymmetric Schemes

The main property of *asymmetric schemes* is, that the decryption key cannot easily be derived from the encryption key. The security of asymmetric schemes is usually based on a complex mathematical problem which means, that if someone is able to solve the underlying problem, he is also able to compute the decryption key from the encryption key and can therefore break the scheme.

The benefit of asymmetric schemes is, that since the decryption key cannot be recovered easily from the encryption key, the encryption key can be made public. For that reason asymmetric schemes are also referred to as *public-key* schemes and the encryption and decryption keys are also called *public key* and *private key*, respectively.

With public keys, the key-exchange is no problem anymore. If someone wants to send an encrypted message, he just has to access a public server to obtain the recipient's public key. However, there is also a drawback. Since such schemes are based on complex mathematical problems, the involved operations are very costly and for that reason the encryption and decryption processes are very slow.

1.1.3 Hybrid Schemes

Hybrid schemes are a mixture of symmetric and asymmetric schemes and aim for using their respective advantages, namely the speed of symmetric schemes and the simple key-exchange of asymmetric schemes. In the first step, a secret

key, sometimes also called *session key*, is generated and used to encrypt the data with a symmetric scheme. Then the session key, which is usually very small, is encrypted using an asymmetric scheme and the recipients public key. Both the encrypted data and the encrypted session key are sent to the receiver who at first decrypts the session key using his private key and then decrypts the data using the recovered session key.

1.2 Digital Signatures

The remaining three security targets integrity, authenticity and non-repudiation are achieved by the use of *digital signatures*. Signature schemes work similar to asymmetric schemes, namely they are based on a complex mathematical problem and they use private and public keys. Also, there are functions $\mathcal{S}$ and $\mathcal{V}$ for generating and verifying signatures, respectively. The input of the *signature generation function* $\mathcal{S}$ is the message to sign m and the private key of the signer d. The output is the *signature* s of the message. The input of the *verification function* $\mathcal{V}$ is the message, the signature and the public key of the signer e. This function returns `true` if the signature is valid and `false` otherwise. The formulas are

$$\mathcal{S}_d(m) = s \qquad \mathcal{V}_e(m, s) \in \{true, false\}$$

The integrity of the message is guaranteed if the signature is approved valid, because the verification function compares the message and the signature to come to a conclusion. If either the signature or the message have been altered during transmission, this comparison fails.

Authenticity and non-repudiation are also guaranteed if the signature is approved valid. This is because the verification function returns `true` only if the signature was generated using the private key associated with the public key used for verification.

1.3 Certification Authorities

With the techniques of encryption schemes and digital signatures one might think that all problems concerning the four security targets are solved. But there is another point of concern, the way the public keys are obtained. In both techniques described above it is assumed that the public key used for message encryption and signature verification indeed belongs to the person one thinks it does. This is not guaranteed immediately.

Suppose the malicious person Oscar is able to place his public key on a public server under Alice's name. If Bob wants to send an encrypted message to Alice, he accesses the server to download the key which he thinks belongs to Alice. If

Bob uses this key to encrypt a message, Oscar will be able to decrypt it and can get access to confidential data.

This example shows, that it is very important to be able to verify the authenticity of the public keys. One way to solve this problem is to use a *certification authority* (CA). At first, all relevant data of a user like name, email address and public key is stored in a so-called certificate. The main purpose of a CA is to verify the contents of such a certificate and to sign it using the CA's private key. Now it is only necessary that the public key of the CA is obtained securely. This can be achieved by including it into operating systems or by publishing it in press.

Now if Bob downloads Alice's certificate he can easily check if the public key is authentic by verifying the name in the certificate and the signature made by the CA. If Oscar placed a false certificate on the server, Bob would detect it immediately.

1.4 Embedded Security

According to the last sections, the most sensitive information in cryptographic schemes are the private keys used for signing and decrypting. It is therefore of utmost importance that those keys are stored securely. The obvious way to store them, for example on local hard drives, floppy disks, CDs or USB sticks is problematic. If some cryptographic operation requires the secret key, it has to be transfered into the computers memory. If this computer is infected with a virus or trojan, the security of the private key is endangered.

A more sophisticated approach is to store the private keys on *smart cards*. Smart cards are credit card sized computers and capable of performing some basic operations. Usually a smart card is equipped with the following components:

CPU The *central processing unit* controls the other elements of the smart card and performs the cryptographic operations. CPUs are available with 8, 16 and 32 bits and usually operate at a frequency of about 5 MHz.

ROM The *read only memory* is non-volatile memory, meaning that information stored on it is not lost if the power is switched off. The ROM can be written only once and is used to store the operating system of the smart card. Currently smart cards are equipped with about 100 kBytes of ROM.

EEPROM The *electronically erasable programmable read only memory* is also non-volatile, but contrary to the ROM it can be erased and rewritten about 100,000 times. The private keys and cryptographic parameters are stored here. The size of the EEPROM is currently about 32 kBytes.

RAM The *random access memory* is volatile memory, meaning its content is lost after the power is switched off. The RAM holds temporary information required by the calculations performed and the operating system. The current smart card technology offers up to 4 kBytes of RAM.

AU The *arithmetic unit* is the cryptographic co-processor. It implements basic operations like addition, multiplication and modular exponentiation. Not every smart card is equipped with an AU because they are quite expensive.

I/O Ports In accordance with the ISO 7816-2 standard, smart cards have eight *input/output* connectors which provide the power supply for the smart card and are used for the data transfer between the card and the reader.

The values in the above description were taken from the URLs [Ren05, Phi05]. Figure 1.1 shows an example for the basic layout of a smart card.

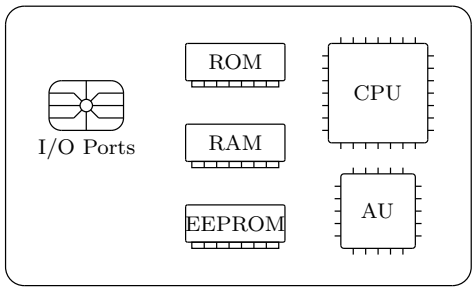


Figure 1.1: Basic layout of a smart card

The great advantage of smart cards is, that they are capable of performing operations on their own. Hence, all calculations involving the private key can be performed on the smart card and the private key never has to leave its secure environment. Another advantage of smart cards is, that they are independent of other hardware, meaning that they can be used with any card reader. Therefore, smart cards can be used for various tasks apart from signing and decrypting messages, for example to control access to buildings.

The drawback of smart cards is, that they offer only little computational power and memory as shown in the above description. It is therefore extremely important that the operations performed by a smart card can be computed efficiently, i.e. by using as little memory and CPU as possible. The first step is to consider which operations actually have to be computed by a smart card. This is done in the next chapter, where cryptographic schemes which are suitable for implementation on smart cards are introduced. In the subsequent chapters, this thesis will turn its attention to how the operations involved in those cryptographic schemes can be computed efficiently.

2 Elliptic Curves

The main purpose of this chapter is to introduce three cryptographic schemes which are suitable for implementation on smart cards. Since all those schemes are based on the additive group of points on an *elliptic curve*, at first the basic concept of elliptic curves over prime fields is explained.

Definition 2.1. *Let $\mathbb{F}_p$ denote a prime field, where p is a prime number. A prime field consists of the integers*

$$\mathbb{F}_p = \{0, 1, \ldots, p-1\}$$

and all arithmetic operations are computed modulo p. Those operations are field multiplications (M), field squarings (S) and field inversions (I).

In this thesis, the ratio between inversions and multiplications I/M is set to $I = 30M$ and the ratio between squarings and multiplications S/M is set to $S = 0.8M$, as it is customary nowadays. Therefore, inversions are very costly compared to multiplications and squarings and should be avoided.

The implicit equation of an elliptic curve over a prime field is given as

$$E : y^2 = x^3 + ax + b \tag{2.1}$$

where a, b belong to the prime field $\mathbb{F}_p$ and $p > 3$. A further condition on a and b is, that the so-called discriminant $\Delta = 4a^3 + 27b^2$ is non-zero. This ensures, that the partial derivatives in x and y never vanish simultaneously and the curve is therefore smooth.

2.1 Defining an Additive Group

The points on an elliptic curve can be used to define an additive abelian group with a geometrical group operation as stated in [Kob99]. The group elements are

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p | y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

where $\mathcal{O}$ is the so-called *point of infinity* which serves as neutral group element. Hence

$$P + \mathcal{O} = \mathcal{O} + P = P$$

is defined for any point $P = (x, y) \in E(\mathbb{F}_p)$.

The rules for the inverse of a point and the addition of two points can be derived from the following definition.

Definition 2.2. *Let L be a line which intersects the elliptic curve in the three points $P = (x_P, y_P), Q = (x_Q, y_Q), R = (x_R, y_R) \in E(\mathbb{F}_p)$. Then*

$$P + Q + R = \mathcal{O}$$

holds.

If $x_P \neq x_R \neq x_Q$ holds, there are three cases to examine:

i) $x_P = x_Q, y_P = -y_Q$: In other words, Q is the reflection of P across the x-axis and L is the horizontal line through P and Q. Here, the third point of intersection R is the point of infinity $\mathcal{O}$. Hence $P + Q = \mathcal{O}$ holds and therefore Q is the *inverse* of P, which is denoted by $-P$.

ii) $x_P \neq x_Q$: In this case, the third point of intersection R is a distinct point on the elliptic curve. Therefore, the formula for *adding* the points P and Q is given as $P + Q = -R$.

iii) $x_P = x_Q, y_Q = y_P$: In this case, L is the tangent on E in $P = Q$. Therefore, the formula for a point *doubling* is given as $2P = -R$.

Figure 2.1 illustrates the elliptic curve point addition and doubling which from now on are denoted by ECADD and ECDBL, respectively.

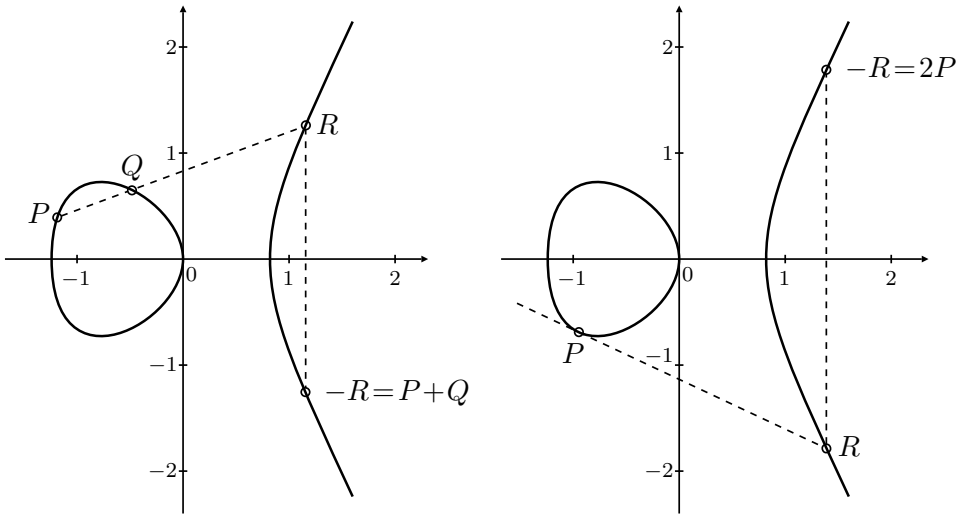


Figure 2.1: Elliptic curve point addition and doubling

The commutativity ($P + Q = Q + P$) of $E(\mathbb{F}_p)$ follows directly from the construction of the group operation. The associativity ($P + (Q + R) = (P + Q) + R$) can be verified using the following fact from projective geometry:

Proposition 2.3. *Let L_1, L_2, L_3 be three lines that intersect a cubic curve in nine points $P_1, \ldots, P_9$ (counting multiplicity) and let L_1', L_2', L_3' be three lines that intersect the cubic curve in nine points $Q_1, \ldots, Q_9$. If $P_i = Q_i$ for $i = 1, \ldots, 8$, then also $P_9 = Q_9$.*

The six lines are set as follows

L_1 : the line through P, Q and $-(P+Q)$

L_2 : the line through $R, -R$ and $\mathcal{O}$

L_3 : the line through $-P, -(Q+R)$ and $S = P+(Q+R)$

L_1' : the line through Q, R and $-(Q+R)$

L_2' : the line through $P, -P$ and $\mathcal{O}$

L_3' : the line through $-(P+Q), -R$ and $S' = (P+Q)+R$

Now the lines L_1, L_2, L_3 and L_1', L_2', L_3' have eight points of intersection in common, namely $P, -P, Q, R, -R, -(P+Q), -(Q+R)$ and $\mathcal{O}$. One can therefore conclude that $S = S'$ which proves the associativity.

Finally, the scalar multiplication dP, where d is a positive integer and $P \in E(\mathbb{F}_p)$ is defined as

$$dP = \underbrace{P + \ldots + P}_{d \text{ times}}$$

In the case where $d < 0$, $-P$ is added to itself $|d|$ times.

2.2 Coordinate Systems and Addition Formulas

The next step is to derive explicit formulas for point additions (ECADD) and point doublings (ECDBL). This section explains those formulas in different coordinate systems and also compares the number of field operations required.

2.2.1 Affine Coordinates

The most straight forward coordinates to use are *affine coordinates* ($\mathcal{A}$). Here, the formulas for point additions and point doublings can be derived using the geometrical structure of the group operation introduced in Section 2.1. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two distinct points lying on the curve, with $Q \neq -P$. The target is to calculate $P + Q = (x_3, y_3)$. The equation of the line L which intersects P and Q is given as

$$L : y = \lambda x + \gamma, \tag{2.2}$$

where

$$\lambda = \frac{(y_2 - y_1)}{(x_2 - x_1)}, \qquad \gamma = y_1 - \lambda x_1$$

The third point where L intersects the curve is $R = (\tilde{x}, \tilde{y})$. Since $P + Q = -R$, $(x_3, y_3) = (\tilde{x}, -\tilde{y})$ holds and inserting this into (2.2) yields a formula for the y-coordinate of $P + Q$.

$$\begin{aligned} & \tilde{y} &=& \lambda\tilde{x} + \gamma \\ \iff & y_3 &=& -\lambda x_3 - \gamma \\ & &=& -\lambda x_3 - y_1 + \lambda x_1 \\ & &=& \lambda(x_1 - x_3) - y_1 \end{aligned}$$

The x-coordinate of $P + Q$ is obtained by inserting (2.2) into the equation of the elliptic curve. This yields

$$\begin{aligned} & (\lambda x + \gamma)^2 &=& x^3 + ax + b \\ \iff & 0 &=& x^3 - \lambda^2 x^2 + (a - 2\lambda\gamma)x - \gamma^2 + b \end{aligned}$$

This equation can be solved by using the fact, that the sum of the roots of a monic polynomial is equal to minus the coefficient of the variable of the second highest power. The three roots are x_1, x_2, x_3 and the coefficient is $-\lambda^2$. Therefore $x_1 + x_2 + x_3 = \lambda^2$ holds and since two of those roots are given by the x-coordinates of the points P and Q, x_3 can be calculated. Hence, the formula for a point addition (ECADD) in affine coordinates is:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \\ \lambda &= \tfrac{(y_2 - y_1)}{(x_2 - x_1)} \end{aligned} \tag{2.3}$$

Next, the case $P = Q$ has to be examined to obtain the formula for a point doubling. The only difference to the former case is that λ is now given as the derivative

$$\lambda = \frac{dy}{dx} = \frac{3x_1^2 + a}{2y_1}$$

in $P = (x_1, y_1)$, because the line L is now the tangent on the curve in P. The formula for a point doubling (ECDBL) can be derived by using the same arguments as above and is given as

$$\begin{aligned} x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \\ \lambda &= \tfrac{(3x_1 + a)}{(2y_1)} \end{aligned} \tag{2.4}$$

Note, that $x_1 = x_2$ holds in that case. The computational costs for both operations are

$$\begin{aligned} \text{ECADD}_{\mathcal{A}} &= 2M + S + I \\ \text{ECDBL}_{\mathcal{A}} &= 2M + 2S + I \end{aligned}$$

The drawback of affine coordinates is, that the required field inversion is very costly compared to multiplications and squarings. To avoid inversions, alternative coordinate systems such as Projective, Jacobian, Chudnovsky Jacobian or modified Jacobian coordinates can be used. In this thesis, only those coordinate systems which are of most interest for a multi-scalar multiplication are reviewed. A full review of all coordinate systems can be found in [CMO98].

2.2.2 Jacobian Coordinates

Points in *Jacobian coordinates* ($\mathcal{J}$) are represented as a triple $P = (X, Y, Z)$ and the transformation between affine and Jacobian coordinates is:

$$\begin{aligned} T_{\mathcal{A}\to\mathcal{J}} &: (x, y) \mapsto (x, y, 1) \\ T_{\mathcal{J}\to\mathcal{A}} &: (X, Y, Z) \mapsto (X/Z^2, Y/Z^3) \end{aligned}$$

By applying the replacements $x = X/Z^2$ and $y = Y/Z^3$ to (2.1), the equation of the elliptic curve becomes:

$$E_{\mathcal{J}} : Y^2 = X^3 + aXZ^4 + bZ^6 \tag{2.5}$$

The formula for a point addition (ECADD) is obtained by applying the replacements to (2.3). Let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $P + Q = (X_3, Y_3, Z_3)$. Further, let $U_1 = X_1Z_2^2,\ U_2 = X_2Z_1^2,\ S_1 = Y_1Z_2^3,\ S_2 = Y_2Z_1^3,\ r = S_2 - S_1,\ H = U_2 - U_1$.

$$\begin{aligned} x_3 &= \frac{\left(\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3}\right)^2}{\left(\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2}\right)^2} - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2} \\ &= \frac{\left(Y_2Z_1^3 - Y_1Z_2^3\right)^2}{\left(X_2Z_1^2 - X_1Z_2^2\right)^2 Z_1^2Z_2^2} - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2} \\ &= \frac{r^2}{H^2Z_1^2Z_2^2} - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2} \\ &= \frac{r^2 - X_1Z_2^2H^2 - X_2Z_1^2H^2}{(Z_1Z_2H)^2} \\ &= \frac{r^2 - 2X_1Z_2^2H^2 - H^3}{(Z_1Z_2H)^2} \\ &= \frac{r^2 - 2U_1H^2 - H^3}{(Z_1Z_2H)^2} = \frac{X_3}{Z_3^2} \end{aligned}$$

$$\begin{aligned}
y_3 &= \frac{\left(\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3}\right)}{\left(\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2}\right)} \left(\frac{X_1}{Z_1^2} - \frac{X_3}{Z_3^2}\right) - \frac{Y_1}{Z_1^3} \\
&= \frac{Y_2 Z_1^3 - Y_1 Z_2^3}{(X_2 Z_1^2 - X_1 Z_2^2)\, Z_1 Z_2} \left(\frac{X_1}{Z_1^2} - \frac{X_3}{(Z_1 Z_2 H)^2}\right) - \frac{Y_1}{Z_1^3} \\
&= \frac{r}{H Z_1 Z_2} \left(\frac{X_1 (Z_2 H)^2 - X_3}{(Z_1 Z_2 H)^2}\right) - \frac{Y_1}{Z_1^3} \\
&= \frac{r\,(X_1 Z_2^2 H^2 - X_3) - Y_1 Z_2^3 H^3}{(Z_1 Z_2 H)^3} \\
&= \frac{r\,(U_1 H^2 - X_3) - S_1 H^3}{(Z_1 Z_2 H)^3} = \frac{Y_3}{Z_3^3}
\end{aligned}$$

In total, this yields

$$\begin{aligned}
X_3 &= r^2 - 2U_1 H^2 - H^3 \\
Y_3 &= r\left(U_1 H^2 - X_3\right) - S_1 H^3 \\
Z_3 &= Z_1 Z_2 H
\end{aligned}$$

The formula for a point doubling (ECDBL), where $P = (X_1, Y_1, Z_1)$ and $2P = (X_3, Y_3, Z_3)$ is obtained by applying the same replacements to (2.4) and given as

$$\begin{aligned}
X_3 &= T \\
Y_3 &= -8Y_1^4 + M(S - T) \\
Z_3 &= 2Y_1 Z_1,
\end{aligned}$$

where $S = 4X_1 Y_1^2,\ M = 3X_1^2 + aZ_1^4,\ T = -2S + M^2$.

The costs for point additions and doublings in Jacobian coordinates are

$$\begin{aligned}
\text{ECADD}_{\mathcal{J}} &= 12M + 4S \\
\text{ECDBL}_{\mathcal{J}} &= 4M + 6S
\end{aligned}$$

and no inversion is required anymore, since the Z-coordinate is used for the denominator.

2.2.3 Modified Jacobian Coordinates

The purpose of *modified Jacobian coordinates* ($\mathcal{J}^m$) is to provide faster point doublings while neglecting the speed of point additions. This is achieved by representing the Jacobian coordinates internally as the quadruple (X, Y, Z, aZ^4), where a is the first parameter of the elliptic curve.

The formula for a point addition (ECADD), where $P = (X_1, Y_1, Z_1, aZ_1^4)$, $Q = (X_2, Y_2, Z_2, aZ_2^4)$ and $P + Q = (X_3, Y_3, Z_3, aZ_3^4)$ is given as

$$\begin{aligned} X_3 &= r^2 - 2U_1H^2 - H^3 \\ Y_3 &= r\left(U_1H^2 - X_3\right) - S_1H^3 \\ Z_3 &= Z_1Z_2H \\ aZ_3^4 &= aZ_3^4, \end{aligned}$$

where $U_1 = X_1Z_2^2,\ U_2 = X_2Z_1^2,\ S_1 = Y_1Z_2^3,\ S_2 = Y_2Z_1^3,\ r = S_2 - S_1,\ H = U_2 - U_1$.

The formula for a point doubling (ECDBL), where $P = (X_1, Y_1, Z_1, aZ_1^4)$ and $2P = (X_3, Y_3, Z_3, aZ_3^4)$ is given as

$$\begin{aligned} X_3 &= T \\ Y_3 &= -U + M(S - T) \\ Z_3 &= 2Y_1Z_1 \\ aZ_3^4 &= 2U(aZ_1^4), \end{aligned}$$

where $S = 4X_1Y_1^2,\ U = 8Y_1^4,\ M = 3X_1^2 + (aZ_1^4),\ T = -2S + M^2$.

The costs for point additions and doublings in modified Jacobian coordinates are

$$\begin{aligned} \text{ECADD}_{\mathcal{J}^m} &= 13M + 6S \\ \text{ECDBL}_{\mathcal{J}^m} &= 4M + 4S \end{aligned}$$

and one can see that in the case of an ECDBL operation, two squarings are saved compared to Jacobian coordinates.

2.2.4 Mixed Coordinates

It is also possible to mix different coordinate systems to further speed up point additions and doublings. The notation $\mathcal{C}_1 + \mathcal{C}_2 \rightarrow \mathcal{C}_3$ means, that for a point addition one point is given in $\mathcal{C}_1$ coordinates, the other in $\mathcal{C}_2$ coordinates and the result is obtained in $\mathcal{C}_3$ coordinates. For a point doubling, $2\mathcal{C}_1 \rightarrow \mathcal{C}_2$ means that the input is given in $\mathcal{C}_1$ coordinates and the result is obtained in $\mathcal{C}_2$ coordinates. The costs for the three most interesting mixed coordinate systems for multi-scalar multiplications are

$$\begin{aligned} \text{ECADD}_{\mathcal{J}+\mathcal{A}\rightarrow\mathcal{J}^m} &= 9M + 5S \\ \text{ECADD}_{\mathcal{J}^m+\mathcal{A}\rightarrow\mathcal{J}^m} &= 9M + 5S \\ \text{ECDBL}_{2\mathcal{J}^m\rightarrow\mathcal{J}} &= 3M + 4S \end{aligned}$$

because they provide the fastest point addition and doubling of all coordinate systems reviewed here.

2.3 Elliptic Curves in Cryptography

Until the late eighties, cryptosystems were mainly using the multiplicative group $(\mathbb{F}_p)^*$. To use the additive group of points on an elliptic curve for cryptographic purposes was independently proposed by Koblitz [Kob87] and Miller [Mil86]. Their idea was that cryptosystems, which exploit that the *discrete logarithm problem* (DLP) is a complex mathematical problem, can also be adjusted to work with elliptic curves.

Definition 2.4. *Let x and y be elements of the multiplicative group $(\mathbb{F}_p)^*$ such that $y = x^d \bmod p$ holds for some secret integer d. The* **discrete logarithm problem** *(DLP) is to compute d whilst knowing only x and y.*

The DLP for elliptic curves has to be slightly modified, because the group is additive. Let P and Q be elements of $E(\mathbb{F}_p)$ such that $P = dQ$ holds for some secret integer d. The **elliptic curve discrete logarithm problem** *(ECDLP) is to compute d whilst knowing only P and Q.*

In the group $(\mathbb{F}_p)^*$ exist sub-exponential algorithms to solve the DLP, e.g. the Index-Calculus Algorithm [Odl84]. That is the reason why the secret keys nowadays have to be at least 1024-bits to guarantee security. Then again, there exists no sub-exponential algorithm to solve the ECDLP in the group of points on an "well chosen" elliptic curve. This means, that if the parameters defining the curve are chosen careless, there are also sub-exponential algorithms to solve the ECDLP [MOV93].

Recommended elliptic curves for cryptographic purposes can be found in [NIST01]. The best methods to solve the ECDLP on those curves are Shanks Babystep-Giantstep-Algorithm [Sha69] and Pollards-ρ-Algorithm [Pol78]. Both algorithms use only the fact that they are working in a group, without exploiting a special group structure. In total, both algorithms require $O(\sqrt{ord(Q)})$ operations to solve the ECDLP and are therefore exponential. Since nowadays 2^{80} operations are assumed to be computationally infeasible, it is sufficient to choose the parameters of the elliptic curve p, a, b and the base point Q such, that its order is a 160-bit number. Therefore it is also sufficient to choose the secret keys to be 160-bits to guarantee security. According to [Mil86] it is extremely unlikely that a sub-exponential algorithm will ever work on general elliptic curves. Hence, parameters with a small bit length should also suffice in the future.

When used in conjunction with smart cards, elliptic curves have two great advantages. At first, the required memory to store the secret keys is reduced by a factor of 6.4, compared to 1024-bit keys. Second, the required field multiplications can be computed much faster, since the runtime of a multiplication depends quadratic on the input length of the multipliers. In other words, a multiplication with 160-bit numbers is about 41 times faster than a multiplication with 1024-bit numbers. The result is a significant saving of operations.

2.4 Elliptic Curve Cryptosystems

This section reviews three cryptographic schemes which exploit that the ECDLP is hard to solve. Although they were all originally designed for the group $(\mathbb{F}_p)^*$ and therefore exploit that the DLP is a complex mathematical problem, they can be adjusted to work with elliptic curves.

Throughout this section, it is assumed that the elliptic curves used in the cryptosystems are chosen in accordance with [NIST01] and that the order of the public points as well as the chosen parameters are 160-bit numbers.

2.4.1 Diffie-Hellman Key Exchange

The *Diffie-Hellman key exchange* protocol was proposed by Diffie and Hellman in [DH76]. Its purpose is to allow Alice and Bob to agree on a secret key over an insecure channel, like the internet.

At first a public point $Q \in E(\mathbb{F}_p)$ is required. The remainder of the protocol works as follows:

1. Alice and Bob each choose a random scalar $k_a, k_b \in \mathbb{F}_p$, respectively.
2. Alice calculates $Q_a = k_a Q$ and sends Q_a to Bob. Bob calculates $Q_b = k_b Q$ and sends Q_b to Alice.
3. Alice computes $k_a Q_b$ and Bob computes $k_b Q_a$.

Now both possess the secret point $Q_{ab} = k_a k_b Q$ and the desired secret key can be chosen as the x-coordinate of the point Q_{ab}.

The *Diffie-Hellman problem* is to compute Q_{ab} by using only the three points Q_a, Q_b and Q which are transmitted over the insecure channel. If an eavesdropper Eve can solve the ECDLP, she can extract k_a from Q_a and retrieve Q_{ab}. However, it is unknown if Eve can solve the ECDLP if she can recover Q_{ab} from Q_a, Q_b and Q, i.e. it is not known if the ECDLP and the Diffie-Hellman problem are equivalent.

This protocol can be used to establish a secure tunnel between two parties, e.g. the SSH protocol uses this technique to exchange the secret key required for a symmetric scheme.

2.4.2 ElGamal Cryptosystem

The *ElGamal cryptosystem* is an extension of the Diffie-Hellman key exchange protocol and its purpose is to encrypt and decrypt messages. It was originally proposed by ElGamal in [ElG85].

Suppose that Bob wants to send a message $M \in E(\mathbb{F}_p)$ to Alice. At first, Alice has to generate a public and a private key. Alice chooses a random scalar

$k_a \in \mathbb{F}_p$ and calculates $Q_a = k_a Q$, where Q is again a public point. The pair (Q, Q_a) is Alice's public key and k_a is her private key.

Prior to sending the message, Bob has to obtain Alice's public key in a secure way. To encrypt the message he performs the following steps:

1. Generate a random integer $r \in \mathbb{F}_p$.
2. Compute $P_1 = rQ$.
3. Compute $P_2 = M + rQ_a$.

Then Bob sends P_1 and P_2 to Alice who computes

$$\begin{aligned} P_2 - k_a P_1 &= M + rQ_a - k_a(rQ) \\ &= M + rQ_a - rQ_a \\ &= M \end{aligned}$$

and can therefore decrypt the message M.

If Eve is able to solve the ECDLP, she can decrypt the message M by retrieving the private key k_a from the public key Q_a. But again the other direction is unknown.

The difference to the Diffie-Hellman protocol is, that Alice has to generate her key pair only once and not anew every time someone wants to send her an encrypted message. Hence, this scheme is more convenient for the exchange of email.

2.4.3 Elliptic Curve Digital Signature Algorithm

The *elliptic curve digital signature algorithm* (ECDSA) [JM99, Van92] is the elliptic curve analogue of the digital signature algorithm (DSA), which is the digital signature standard used by the U.S. government [NIST01]. The ECDSA is also based on the ECDLP and it requires the following public system parameters.

E	an elliptic curve defined over $\mathbb{F}_p$
q	the largest prime factor of the order of $E(\mathbb{F}_p)$
P	a point in $E(\mathbb{F}_p)$ with order q

Also, a *one-way* function H is required.

Definition 2.5. *Given a function $H : X \to Y$ and $y \in Y$ such that $H(x) = y$ holds for some $x \in X$. H is called a* **one way** *function, if the problem to find an $\tilde{x} \in X$, such that $H(\tilde{x}) = y$ holds, is computationally infeasible.*

Suppose Alice wants to sign a message m she is about to send to Bob. At first, she has to generate a key pair. Alice randomly chooses her private key a within the range $1 < a < q - 1$ and then calculates her public key as $Q = aP$. Next, she signs the message m using Algorithm 1. Finally, Alice sends the message m and its signature (r, s) to Bob.

Algorithm 1 ECDSA Signature Generation

Require: Message m.
Ensure: Signature (s, r) of m.
1: choose $k \in \{1, \ldots, q-1\}$ randomly
2: $R = (x, y) \leftarrow\!\shortmid kP$
3: $r \leftarrow\!\shortmid x \mod q$
4: **if** $r = 0$ **then** goto step 1
5: $s \leftarrow\!\shortmid k^{-1}(H(m) + ar) \mod q$
6: **if** $s = 0$ **then** goto step 1
7: **return** (r, s)

At first, Bob has to obtain Alice's public key. Then, he verifies the signature of the message using Algorithm 2.

Algorithm 2 ECDSA Signature Verification

Require: Message m, signature (s, r).
Ensure: *true*, if the signature is valid, $false$, otherwise.
1: **if** $(r, s) \notin \{1, 2, \ldots, q-1\}^2$ **return** $false$
2: $u \leftarrow\!\shortmid H(m)s^{-1} \mod q$
3: $v \leftarrow\!\shortmid rs^{-1} \mod q$
4: $R = (x, y) \leftarrow\!\shortmid uP + vQ$
5: **if** $x = r \mod q$ **return** *true*
6: **return** $false$

The correctness of Algorithm 2 can be verified by using that if Alice indeed generated the signature, $s \equiv k^{-1}(H(m) + ar) \bmod q$ and therefore $k \equiv s^{-1}(H(m) + ar) \mod q$ holds. Thus, step 4 can be rearranged to

$$\begin{aligned} uP + vQ &= uP + vaP \\ &= (u + va)P \\ &= \left(H(m)s^{-1} + rs^{-1}a\right) P \\ &= \left(s^{-1}\left(H\left(m\right) + ar\right)\right) P \\ &= kP \end{aligned}$$

and therefore $x = r \bmod q$ holds.

3 Representations of Integers

Most of the operations required by the above explained cryptosystems involve scalars, which are positive integral numbers, i.e. integers. The purpose of this chapter is to give a brief introduction into how those integers can be represented. Apart from the well known decimal representation, several other ways to represent an integer exist. The emphasis of this chapter is on so-called *base-2 representations*, where the integer is represented by the sum of multiple powers of two.

3.1 The Binary Representation

The simplest base-2 representation is the uniquely determined *binary representation*.

Definition 3.1. *The vector* $(d[n-1], \ldots, d[0])$ *is called the* **binary representation** *of the integer* d*, if*

$$d = \sum_{i=0}^{n-1} d[i] \cdot 2^i$$

and $d[i] \in \{0, 1\}, \forall i = 0, \ldots, n-1$.

The length of this representation, the so-called *bit length* n is calculated as $n = \lfloor \log_2 d \rfloor + 1$. The $d[i]$ are called *bits*, which is short for *binary digits*. Algorithm 3 represents one way to generate the binary representation from the decimal representation.

Example 3.2. *The vector* $(1, 0, 1, 1, 0, 1)$ *is the binary representation of* 45 *with bit length* 6*, since* $2^5 + 2^3 + 2^2 + 1 = 45$ *and* $\lfloor \log_2(45) \rfloor + 1 = 6$.

3.2 General Base-2 Representations

Apart from the binary representation, another more general approach for base-2 representations exists. The main idea is to permit other digits than 0 and 1 in the representation. The set of valid digits is called the *digit set* and denoted by $\mathcal{D}$. The number of elements in the digit set, i.e. its order is denoted by $|\mathcal{D}|$.

Algorithm 3 Decimal to Binary

Require: Integer d in its decimal representation.
Ensure: Binary representation $(d[n-1], \ldots, d[0])$ of d.

```
1: n ← 0
2: while d ≠ 0 do
3:     if d mod 2 = 1 then
4:         d[n] ← 1
5:     else
6:         d[n] ← 0
7:     end if
8:     d ← ⌊d/2⌋
9:     n ← n + 1
10: end while
11: return (d[n−1], ..., d[0]), where n = ⌊log_2 d⌋ + 1
```

Definition 3.3. *The vector* $(d[n-1], \ldots, d[0])$ *a called a* $\mathcal{D}$**-representation** *of the integer* d*, if*

$$d = \sum_{i=0}^{n-1} d[i] \cdot 2^i$$

and $d[i] \in \mathcal{D}, \forall i = 0, \ldots, n-1$.

If $\mathcal{D} = \{0, \pm 1\}$ holds, the representation is also called a *signed binary representation*. More general, if $\mathcal{D} = \{0, \pm 1, \ldots, \pm x\}$ holds, the representation is also called a *signed representation*.

In general, $\mathcal{D}$-representations loose the property of uniqueness. For example $(1, 0, 1, 1, 1, \bar{1})$ and $(1, 1, 0, \bar{1}, 0, 1)$ are both signed binary representations of 45 with bit length 6, where $\bar{1} = -1$.

In the following, classes of $\mathcal{D}$-representations, which can be generated by applying a certain algorithm to another representation will be discussed. In general these classes are denoted by $\mathcal{X}$. An example for such a class is the binary representation which can be generated from the decimal representation by applying Algorithm 3.

3.3 The Weight of a Representation

For comparison purposes, it is necessary to measure the quality of $\mathcal{D}$-representations. This can be done by using the weight of either one $\mathcal{D}$-representation separately, or several $\mathcal{D}$-representation at once. Let $\mathcal{X}$ be a class of $\mathcal{D}$-representations generated by a certain algorithm.

Definition 3.4. *Let* $r = (d[n-1], \ldots, d[0])$ *be a* $\mathcal{D}$*-representation with bit length* n*. The* **Hamming weight** *(HW) of* r *is the number of non-zero digits in*

r and denoted by $\mathcal{HW}(r)$. The **Hamming density** *(HD) of r is given as $\mathcal{HD}(r) := \mathcal{HW}(r)/n$. The* **average Hamming density** *(AHD) of a class of $\mathcal{D}$-representations $\mathcal{X}$ is the expected Hamming density of a randomly chosen $\mathcal{D}$-representation in $\mathcal{X}$ with bit length $n \to \infty$ and denoted by $\mathcal{AHD}(\mathcal{X})$.*

Definition 3.5. *Let $r_1 = (d_1[n-1], \ldots, d_1[0]), \ldots, r_k = (d_k[n-1], \ldots, d_k[0])$ be k $\mathcal{D}$-representations with bit length n. The* **joint Hamming weight** *(JHW) of $r_1, \ldots, r_k$ is the number of non-zero columns, i.e. columns with at least one entry different from zero in the matrix*

$$\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_k \end{pmatrix} = \begin{pmatrix} d_1[n-1] & \ldots & d_1[0] \\ d_2[n-1] & \ldots & d_2[0] \\ \vdots & & \vdots \\ d_k[n-1] & \ldots & d_k[0] \end{pmatrix}$$

and denoted by $\mathcal{JHW}(r_1, \ldots, r_k)$. If some $\mathcal{D}$-representations are less than n bits, zeros are padded to the left as required. The **joint Hamming density** *(JHD) of $r_1, \ldots, r_k$ is given as $\mathcal{JHD}(r_1, \ldots, r_k) := \mathcal{JHW}(r_1, \ldots, r_k)/n$. The* **average joint Hamming density** *(AJHD) of a class of $\mathcal{D}$-representations $\mathcal{X}$ is the expected joint Hamming density of k randomly chosen $\mathcal{D}$-representations in $\mathcal{X}$ with bit length $n \to \infty$ and denoted by $\mathcal{AJHD}_k(\mathcal{X})$.*

Example 3.6. *Consider the two binary representations*

$$\begin{array}{rcll} r_1 & = & (1,0,1,1,0,1,0,1,1,0,1,0) & (= 2906) \\ r_2 & = & (1,0,1,0,1,0,0,1,1) & (= 339) \end{array}$$

with bit lengths 12 and 9, respectively. Using the above definitions shows that

$$\begin{array}{rclrclrcl} \mathcal{HW}(r_1) & = & 7 & \mathcal{HW}(r_2) & = & 5 & \mathcal{JHW}(r_1, r_2) & = & 8 \\ \mathcal{HD}(r_1) & = & 7/12 & \mathcal{HD}(r_2) & = & 5/9 & \mathcal{JHD}(r_1, r_2) & = & 8/12 \end{array}$$

Lemma 3.7. *In the case of the binary representation, the digits 0 and 1 appear each with a probability of $1/2$. Therefore*

$$\mathcal{AHD}(\text{binary}) = \frac{1}{2}$$

holds. If considering the binary representations of k integers, the probability for a zero column is given as $1/2^k$ and therefore

$$\mathcal{AJHD}_k(\text{binary}) = 1 - \frac{1}{2^k}$$

holds.

4 Multi-Scalar Multiplication Algorithms

According to Chapter 2, the basic operation in elliptic curve cryptosystems is a scalar multiplication dP and more general, a sum of scalar multiplications

$$\sum_{j=1}^{k} d_j P_j,$$

where d_j are the scalars and P_j are points on an elliptic curve, $j = 1, \ldots, k$. Such a sum of scalar multiplications is called a *multi-scalar multiplication*.

In fact, those multi-scalar multiplications are the most time consuming operations and since an implementation on devices with little computational power and memory is desired, they have to be computed efficiently.

This chapter at first introduces two algorithms that can be used to compute a scalar multiplication. After comparing those two algorithms, two extensions of the superior algorithm are introduced which are specifically designed for multi-scalar multiplications.

4.1 Binary Methods

Binary methods are methods for the efficient computation of a scalar multiplication dP. As the name suggests, they were originally designed to use the binary representation of the scalar. However, as it will turn out they can also be adjusted to work with $\mathcal{D}$-representations of the scalar. There exist two different binary methods, one that parses the scalar starting at the least significant bit, i.e. *right-to-left*, and one that parses the scalar starting at the most significant bit, i.e. *left-to-right*.

4.1.1 Right-to-Left Binary Method

The task is to compute a scalar multiplication dP, where d is an n-bit scalar and P is a point on an elliptic curve. If the binary representation of the scalar d is considered, it is possible to write

$$\begin{aligned} dP &= (d[n-1]2^{n-1} + d[n-2]2^{n-2} + \ldots + d[1]2 + d[0])\, P \\ &= d[n-1]2^{n-1}P + d[n-2]2^{n-2}P + \ldots + d[1]2P + d[0]P \end{aligned} \tag{4.1}$$

This equation is evaluated starting at the least significant bit $d[0]$, i.e. right-to-left. In the i-th iteration, 2^iP is added to the intermediate result, if the current bit $d[i]$ is 1. This method is represented in Algorithm 4, where the register X stores the result and Q_1 stores the point 2^iP in the i-th iteration.

Algorithm 4 Right-To-Left Binary Method

Require: Point $P \in E(\mathbb{F}_p)$, n-bit scalar d in its binary representation.
Ensure: Scalar multiplication dP

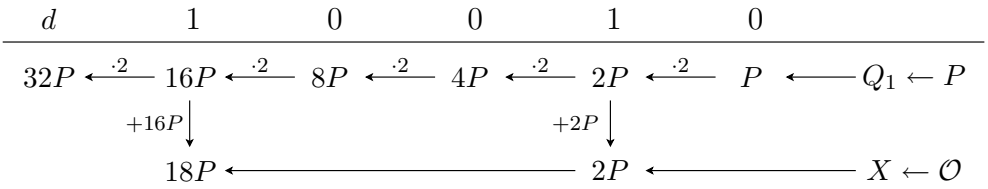
1: $X \leftarrow \mathcal{O}$
2: $Q_1 \leftarrow P$
3: **for** $i = 0$ to $n - 1$ **do**
4: **if** $d[i] = 1$ **then**
5: $X \leftarrow \text{ECADD}(X, Q_1)$
6: **end if**
7: $Q_1 \leftarrow \text{ECDBL}(Q_1)$
8: **end for**
9: **return** X

Algorithm 4 performs an ECADD operation each time the current digit $d[i]$ is 1, hence with probability 1/2. An ECDBL operation is performed in each iteration. Therefore, the right-to-left binary method on average requires

$$n \text{ ECDBL} + n \cdot \frac{1}{2} \text{ ECADD}$$

operations to compute a scalar multiplication dP, where the scalar is represented in the binary representation.

Example 4.1. *Let $d = 18$ with binary representation $(1, 0, 0, 1, 0)$. The following figure shows the sequence of ECADD and ECDBL operations performed by the right-to-left binary method to compute the scalar multiplication dP.*



Algorithm 4 can also be adjusted to work with $\mathcal{D}$-representations. The difference is, that instead of adding only 2^iP to the result, $t \cdot 2^iP, t \in \mathcal{D}$ has to be added, depending on the current digit $d[i]$. The adjusted version is shown in Algorithm 5.

Algorithm 5 General Right-To-Left Binary Method

Require: Point $P \in E(\mathbb{F}_p)$, n-bit scalar d in a $\mathcal{D}$-representation $\mathcal{X}$.
Ensure: Scalar multiplication dP

```
1: X ← O
2: Q_t ← tP, ∀t ∈ D \ {0}
3: for i = 0 to n − 1 do
4:     if d[i] ≠ 0 then
5:         X ← ECADD(X, Q_{d[i]})
6:     end if
7:     Q_t ← ECDBL(Q_t), ∀t ∈ D \ {0}
8: end for
9: return X
```

The first step is to compute all points which might have to be added to the result (line 2). During runtime, the algorithm performs an ECADD operation each time the current digit $d[i]$ is non-zero, hence with probability $\mathcal{AHD}(\mathcal{X})$. Since the point $t \cdot 2^i P, t \in \mathcal{D}$ has to be added in the i-th iteration, all the $|\mathcal{D}| - 1$ points which were computed in line 2, have to be doubled in each iteration. On average, the general right-to-left binary method requires

$$n \cdot (|\mathcal{D}| - 1) \text{ ECDBL} + n \cdot \mathcal{AHD}(\mathcal{X}) \text{ ECADD}$$

operations to compute a scalar multiplication dP, where the scalar is represented in the $\mathcal{D}$-representation $\mathcal{X}$. Further, the precomputation of $|\mathcal{D}| - 2$ points is required, which are all points of the form $tP, t \in \mathcal{D} \setminus \{0, 1\}$.

Note, that additional ECADD and ECDBL operations are required for the precomputation.

The right-to-left binary method can also be used to compute a multi-scalar multiplication. This is done by computing each scalar multiplication separately and adding the results together, which requires another $(k - 1)$ ECADD operations. In the case of k scalars, the general right-to-left binary method on average requires

$$n \cdot k \cdot (|\mathcal{D}| - 1) \text{ ECDBL} + (n \cdot k \cdot \mathcal{AHD}(\mathcal{X}) + (k - 1)) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, where the scalars are represented in a $\mathcal{D}$-representation $\mathcal{X}$. Further, the precomputation of $k \cdot (|\mathcal{D}| - 2)$ points is required.

4.1.2 Left-to-Right Binary Method

Another method to compute a scalar multiplication dP is the *left-to-right binary method*. The basic idea is to successively factor out 2 in equation (4.1), which yields

$$\begin{aligned} dP &= d[n-1]2^{n-1}P + d[n-2]2^{n-2}P + \ldots + d[1]2P + d[0]P \\ &= 2\left(d[n-1]2^{n-2}P + d[n-2]2^{n-3}P + \ldots + d[1]P\right) + d[0]P \\ &\vdots \\ &= 2\left(2\left(\ldots 2\left(d[n-1]2P + d[n-2]P\right) + \ldots\right) + d[1]P\right) + d[0]P \\ &= 2\left(2\left(\ldots 2\left(2\left(d[n-1]P\right) + d[n-2]P\right) + \ldots\right) + d[1]P\right) + d[0]P \end{aligned} \tag{4.2}$$

Now it is possible to start the evaluation at the most significant bit $d[n-1]$, i.e. left-to-right. In the i-th iteration, the intermediate result X is doubled and if the current bit $d[i]$ is 1, P is added as shown in Algorithm 6.

Algorithm 6 Left-to-Right Binary Method

Require: Point $P \in E(\mathbb{F}_p)$, n-bit scalar d in its binary representation.
Ensure: Scalar multiplication dP

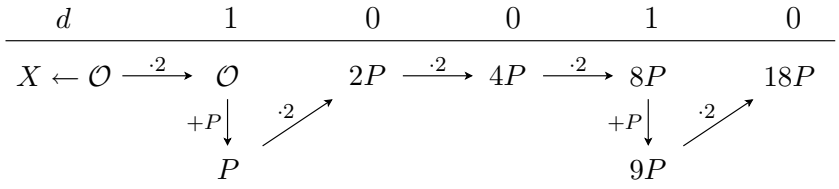
```
1: X ← O
2: for i = n − 1 down to 0 do
3:     X ← ECDBL(X)
4:     if d[i] = 1 then
5:         X ← ECADD(X, P)
6:     end if
7: end for
8: return X
```

Algorithm 6 performs an ECADD operation each time the current digit $d[i]$ is 1, hence with probability 1/2. An ECDBL operation is performed in each iteration. Therefore, the left-to-right binary method on average requires

$$n \text{ ECDBL} + n \cdot \frac{1}{2} \text{ ECADD}$$

operations to compute a scalar multiplication dP, where the scalar is represented in the binary representation.

Example 4.2. *Let $d = 18$ with binary representation $(1, 0, 0, 1, 0)$. The following figure shows the sequence of ECADD and ECDBL operations performed by the left-to-right binary method to compute the scalar multiplication dP.*



The left-to-right binary method can also be adjusted to work with $\mathcal{D}$-representations. Here, the addition with P is replaced by an addition with $t \cdot P, t \in \mathcal{D}$ and all those points have to be precomputed. The adjusted version is shown in Algorithm 7.

Algorithm 7 General Left-to-Right Binary Method

Require: Point $P \in E(\mathbb{F}_p)$, n-bit scalar d in a $\mathcal{D}$-representation $\mathcal{X}$.
Ensure: Scalar multiplication dP

```
1: X ← O
2: Q_t ← tP, ∀t ∈ D \ {0}
3: for i = n − 1 down to 0 do
4:     X ← ECDBL(X)
5:     if d[i] ≠ 0 then
6:         X ← ECADD(X, Q_{d[i]})
7:     end if
8: end for
9: return X
```

The first step is to compute all points which might have to be added to the result (line 2). During runtime, Algorithm 7 performs an ECADD operation each time the current digit $d[i]$ is non-zero, hence with probability $\mathcal{AHD}(\mathcal{X})$. Also, one ECDBL operation is performed in each iteration to double the intermediate result. On average, the general left-to-right binary method requires

$$n \cdot \text{ ECDBL} + n \cdot \mathcal{AHD}(\mathcal{X}) \text{ ECADD}$$

operations to compute a scalar multiplication dP, where the scalar is represented in the $\mathcal{D}$-representation $\mathcal{X}$. Further, the precomputation of $|\mathcal{D}| - 2$ points is required, which are all points of the form $tP, t \in \mathcal{D} \setminus \{0, 1\}$.

Also in this case, additional ECADD and ECDBL operations are required for the precomputation.

To compute a multi-scalar multiplication, each scalar multiplication is performed separately and the results are summed up, which requires additional $(k - 1)$ ECADD operations. In the case of k scalars, the general left-to-right binary method on average requires

$$n \cdot k \text{ ECDBL} + (n \cdot k \cdot \mathcal{AHD}(\mathcal{X}) + (k - 1)) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, where the scalars are represented in a $\mathcal{D}$-representation $\mathcal{X}$. Further, the precomputation of $k \cdot (|\mathcal{D}| - 2)$ points is required.

4.1.3 Left-to-Right vs. Right-to-Left

In this section, the right-to-left binary method and the left-to-right binary method are compared regarding their efficiency.

While the basic versions of both methods (Algorithm 4 and 6) require the same amount of ECADD and ECDBL operations, the right-to-left binary method requires one additional register to store $2^i P$.

In the case of the general methods (Algorithm 5 and 7), the difference is more drastic. The left-to-right binary method requires only one ECDBL operation in each iteration, while the right-to-left binary method requires one ECDBL operation for each precomputed point in each iteration. This means, that the right-to-left binary method requires $(|\mathcal{D}| - 1)$ times more ECDBL operations than its left-to-right counterpart.

Another advantage of the left-to-right binary method is, that the precomputed points for the ECADD step remain fixed during the whole runtime. It is therefore possible to represent those points in affine coordinates and use mixed coordinates for the ECADD step as introduced in Section 2.2.4. Since the only suitable coordinate systems for the right-to-left binary method are Jacobian or modified Jacobian coordinates, the ECADD step of the left-to-right binary method can be executed 15% to 27% faster.

Summarizing, the left-to-right binary method is more efficient than the right-to-left binary method, regardless of which representation of the scalars is used. For that reason, the right-to-left binary method will not be investigated any further. Instead, the next two sections introduce two enhancements of the left-to-right binary method which are specifically designed for multi-scalar multiplications.

4.2 Interleave Method

The first enhancement of the left-to-right binary method is the *Interleave method* proposed by Möller in [Möl01]. It aims for reducing the number of ECDBL operations required for a multi-scalar multiplication, by performing them simultaneously. Suppose $d_1 P_1 + d_2 P_2$ is to be computed. By using equation (4.2) it is possible to write

$$\begin{aligned}
& 2\,(2\,(\ldots 2\,(2\,(d_1[n-1]P_1) + d_1[n-2]P_1) + \ldots) + d_1[1]P_1) + d_1[0]P_1 \\
+\;& 2\,(2\,(\ldots 2\,(2\,(d_2[n-1]P_2) + d_2[n-2]P_2) + \ldots) + d_2[1]P_2) + d_2[0]P_2 \\
=\;& 2\,(2\,(\ldots 2\,(2\,(d_1[n-1]P_1 + d_2[n-1]P_2) + d_1[n-2]P_1 \\
& \qquad + d_2[n-2]P_2) + \ldots) + d_1[1]P_1 + d_2[1]P_2) + d_1[0]P_1 + d_2[0]P_2
\end{aligned}$$

In other words, first the intermediate result is doubled and then $d_1[i]P_1$ and $d_2[i]P_2$ are added if the respective digit is different from zero. Of course, this strategy can also be extended to an arbitrary number of points and scalars as shown in Algorithm 8. Also, this algorithm is adjusted already to work with $\mathcal{D}$-representations.

Algorithm 8 Interleave Method

Require: Points $P_j \in E(\mathbb{F}_p)$, n-bit scalars d_j in a $\mathcal{D}$-representation $\mathcal{X}$, $j = 1, \ldots, k$.
Ensure: Multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$

```
1: X ← O
2: for i = n − 1 down to 0 do
3:     X ← ECDBL(X)
4:     for j = 1 to k do
5:         if d_j[i] ≠ 0 then
6:             X ← ECADD(X, d_j[i]P_j)
7:         end if
8:     end for
9: end for
10: return X
```

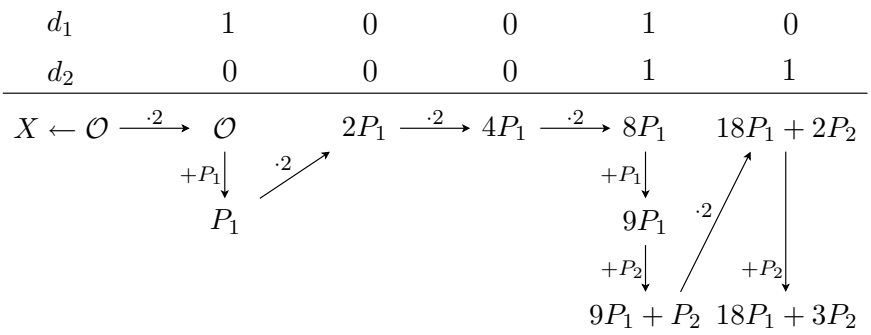
Since it would be far to costly to compute the points $tP_j, t \in \mathcal{D} \setminus \{0, 1\}, j = 1, \ldots, k$ required in line 6 anew every time they occur, they are precomputed and stored.

Algorithm 8 requires only one ECDBL operation in each iteration, which reduces the total number of ECDBL operations required by a factor k, compared to the left-to-right binary method. Also, the $(k-1)$ ECADD operations to add the final results are saved. In total, the Interleave method on average requires

$$n \text{ ECDBL} + n \cdot k \cdot \mathcal{AHD}(\mathcal{X}) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, where the scalars are represented in a $\mathcal{D}$-representation $\mathcal{X}$. Further, the precomputation of $k \cdot (|\mathcal{D}| - 2)$ points is required, which are all points of the form $tP_j, t \in \mathcal{D} \setminus \{0, 1\}, j = 1, \ldots, k$.

Example 4.3. *Let $d_1 = 18, d_2 = 3$ with binary representations $(1, 0, 0, 1, 0)$ and $(0, 0, 0, 1, 1)$, respectively. The following figure shows the sequence of ECADD and ECDBL operations performed by the Interleave method to compute the multi-scalar multiplication $d_1P_1 + d_2P_2$.*



In general, the Interleave method requires

$$n \text{ ECDBL} + n \text{ ECADD}$$

operations to compute a multi-scalar multiplication $d_1P_1 + d_2P_2$, where the n-bit scalars d_1, d_2 are represented in their binary representation. In this case, no points have to be precomputed.

4.3 Shamir Method

In this section, a further improvement of the left-to-right binary method is introduced. Although this method was proposed by ElGamal in [ElG85], it is known by the name *Shamir method* since it was described with a reference to Shamir.

The idea is to perform not only the ECDBL operations simultaneously, but also the ECADD operations. In other words, the second for-loop of the interleave method is computed by using only one ECADD operation. Algorithm 9 shows the Shamir method adjusted already for $\mathcal{D}$-representations and an arbitrary number of scalars.

Algorithm 9 Shamir Method

Require: Points $P_j \in E(\mathbb{F}_p)$, n-bit scalars d_j in a $\mathcal{D}$-representation $\mathcal{X}$, $j = 1, \ldots, k$.
Ensure: Multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$

```
1: X ← O
2: for i = n − 1 down to 0 do
3:     X ← ECDBL(X)
4:     if (d_1[i], ..., d_k[i]) ≠ (0, ..., 0) then
5:         X ← ECADD(X, d_1[i]P_1 + ... + d_k[i]P_k)
6:     end if
7: end for
8: return X
```

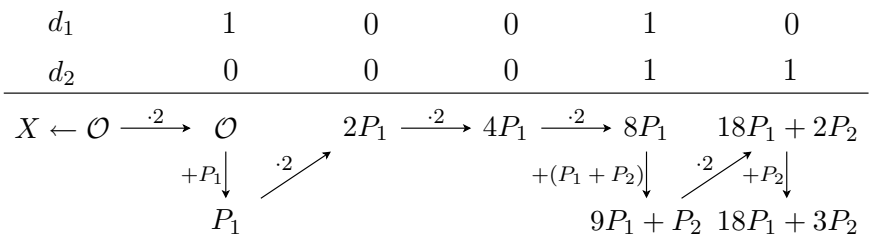
Since Algorithm 9 performs only one ECADD operation in each iteration (line 5), all points of the form $t_1P_1 + \ldots + t_kP_k$, where $t_1, \ldots, t_k \in \mathcal{D}^k$, such that at least two of the t_j are different from zero must be precomputed and stored.

The Shamir method performs an ECADD operation each time the current column $(d_1[i], \ldots, d_k[i])$ is non-zero. An ECDBL operation is performed in each iteration. Therefore, the Shamir method on average requires

$$n \text{ ECDBL} + n \cdot \mathcal{AJHD}_k(\mathcal{X}) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, where the scalars are represented in a $\mathcal{D}$-representation $\mathcal{X}$. Further, the precomputation of $|\mathcal{D}|^k - 1 - k$ points is required, which are all points of the form $t_1P_1 + \ldots + t_kP_k$, where $t_1, \ldots, t_k \in \mathcal{D}^k$, such that at least two of the t_j are different from zero.

Example 4.4. *Let $d_1 = 18, d_2 = 3$ with binary representations $(1,0,0,1,0)$ and $(0,0,0,1,1)$, respectively. The following figure shows the sequence of ECADD and ECDBL operations performed by the Shamir method to compute the multi-scalar multiplication $d_1P_1 + d_2P_2$.*



In general, the Shamir method requires

$$n \text{ ECDBL} + n \cdot \frac{3}{4} \text{ ECADD}$$

operations to compute a multi-scalar multiplication $d_1P_1 + d_2P_2$, where the n-bit scalars d_1, d_2 are represented in their binary representation. Also, the single point $P_1 + P_2$ has to be precomputed.

This shows, that compared to the Interleave method, $1/4$ of the ECADD operations can be saved at the cost of one precomputed point.

4.4 Elliptic Curves and Precomputation

So far, two algorithms specifically designed for multi-scalar multiplications were introduced. As it turned out, the precomputation of several points is required by both algorithms, depending on the $\mathcal{D}$-representation used for the scalars. This section considers how the number of points to precompute can be reduced in the case of elliptic curves.

Contrary to prime fields where inversions are very costly, inversions of points on an elliptic curve can be computed virtually for free. As explained in Section 2.1, the inverse of a point $P = (x, y) \in E(\mathbb{F}_p)$ is given as $-P = (x, -y)$. This means that the inverse of P is obtained just by changing the sign of the y-coordinate. This fact can be exploited to reduce the number of points to precompute drastically if the scalars are represented in a signed representation [MO90].

Suppose $d_1P_1 + d_2P_2$ is to be computed, where the scalars are represented in a signed binary representation, i.e. $\mathcal{D} = \{0, \pm 1\}$. Usually, the Interleave method would require the precomputation of the two points $-P_1$ and $-P_2$. Both those points are inverses of known points (P_1, P_2) and can therefore be obtained at negligible costs. For that reason the points are not precomputed and stored,

but computed anew by an *on-the-fly* inversion each time they are required by the Interleave method.

To compute $d_1P_1+d_2P_2$ with the Shamir method, usually the precomputation of the six points $-P_1, -P_2, P_1+P_2, -P_1-P_2, P_1-P_2, -P_1+P_2$ would be required. Here it is sufficient to precompute the two points $P_1 + P_2$ and $P_1 - P_2$. Then, the remaining four points $(-P_1, -P_2, -P_1 - P_2, -P_1 + P_2)$ are inverses of known points and can therefore be obtained by an on-the-fly inversion during runtime.

This shows, that in the case of two scalars given in a signed binary representation, the number of points to precompute can be reduced from two to zero in the case of the Interleave method and from six to two in the case of the Shamir method.

In the general case of k scalars, given in a signed representation, the number of points to precompute for the Interleave method can be reduced from

$$k \cdot (|\mathcal{D}| - 1) - k \qquad \text{to} \qquad k \cdot \frac{(|\mathcal{D}| - 1)}{2} - k$$

and the number of points to precompute for the Shamir method can be reduced from

$$|\mathcal{D}|^k - 1 - k \qquad \text{to} \qquad \frac{|\mathcal{D}|^k - 1}{2} - k.$$

In total this means, that when using signed representations of the scalars, the number of points to precompute can be reduced by more than 50%. This not only reduces the memory required to store those points, but also the number of ECADD and ECDBL operations which are required to compute those points. Yet another advantage of elliptic curves in conjunction with resource constrained devices.

4.5 Lim-Lee Combing

By examining the operations required by the cryptosystems introduced in Section 2.4, it turns out that the majority of the scalar multiplications involve only one scalar. To legitimate the generalization to an arbitrary number of scalars, this section introduces an efficient method to convert a single scalar multiplication dP to a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$. This method was proposed by Lim and Lee in [LL94] and is therefore often referred to as *Lim-Lee combing*.

The Lim-Lee combing method is based on the observation that an n-bit scalar d can be divided into two parts d_1, d_2 of bit length $n/2$ such that $d = 2^{n/2}d_1 + d_2$ holds. Then, a scalar multiplication dP can be written as $d_1P_1 + d_2P_2$, where $P_1 = 2^{n/2}P$ and $P_2 = P$.

In general, the scalar d is divided into k parts $d_1, \ldots, d_k$ each with bit length $a = n/k$. If n is no multiple of k, the scalar is padded with zeros to the left as

required. Then, dP can be written as

$$\begin{aligned} dP &= d_1 2^{a(k-1)}P + d_2 2^{a(k-2)}P + \ldots + d_{k-1}2^a P + d_k P \\ &= d_1 P_1 + d_2 P_2 + \ldots + d_{k-1}P_{k-1} + d_k P_k \\ &= \sum_{j=1}^{k} d_j P_j \end{aligned}$$

and an algorithm for multi-scalar multiplication like the Interleave method or the Shamir method can be applied.

Before that, the k points $P_j = 2^{a(k-j)}, j = 1, \ldots, k$ have to be precomputed which requires $n - n/k$ ECDBL operations. However, this does not increase the total costs for the multi-scalar multiplication considerably, since the bit length of each of the scalars $d_1, \ldots, d_k$ is only n/k. Therefore, both the Interleave method and the Shamir method require only n/k ECDBL operations to compute the multi-scalar multiplication. Only in the case where the bit length of the original scalar d is not a multiple of k, $k \cdot \lceil n/k \rceil - n$ extra ECDBL operations are required.

5 Low-Weight Representations

In the last chapter, two efficient methods to compute a multi-scalar multiplication were introduced. On the one hand, it turned out that the required number of ECDBL operations is always n, independent of the number of scalars involved and the $\mathcal{D}$-representation used. On the other hand, the required number of ECADD operations depends on the AHD or the AJHD of the $\mathcal{D}$-representation used for the scalars. It is therefore possible to save ECADD operations by deploying the scalars in $\mathcal{D}$-representations with a low AHD or AJHD. It also turned out, that the number of points which have to be precomputed can be more than halved, if the scalars are represented in a signed representation.

In this section, two $\mathcal{D}$-representations are introduced. One that provides a low AHD and can therefore speed up the Interleave method and one that provides a low AJHD and can therefore speed up the Shamir method. In addition, both presented $\mathcal{D}$-representations are signed representations.

In general, the process of generating a $\mathcal{D}$-representation, i.e. rewriting the scalar is called *recoding*. One way to recode scalars are so-called *window methods*. Here, the binary representation of the scalar is divided into several parts, the windows. Then, each window is recoded independently of the other windows (see [Gor98] for an overview). A more sophisticated approach are *sliding window methods*. Here, the positions and the lengths of the windows are not fixed, but depend on the structure of the scalar. Both methods reviewed in this chapter are sliding window methods.

5.1 The width-w Non Adjacent Form

The *width-w Non Adjacent Form* (wNAF) is a $\mathcal{D}$-representation, that provides a low AHD and can therefore speed up the Interleave method. It was independently proposed by Solinas in [Sol00] and by Blake, Seroussi and Smart in [BSS99]. The basic idea of the wNAF was first proposed by Miyaji, Ono and Cohen in [MOC97]. The definition of the wNAF as stated in [Sol00] is reviewed in Definition 5.1. Throughout this section, $w \geq 2$ is assumed.

Definition 5.1. *The vector* $(\delta[n], \delta[n-1], ..., \delta[0])$ *is called a* w**NAF**, *if and only if the following three properties hold.*

wNAF-1 *The most significant non-zero bit is positive.*

wNAF-2 *Among any w consecutive digits, at most one is non-zero.*

wNAF-3 *Each non-zero digit is odd and less than 2^{w-1} in absolute value.*

According to this definition, the digit set used by the wNAF is given as $\mathcal{D}_w = \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$ and consists of odd numbers only. In [MS04a] Muir and Stinson proved the following two properties of the wNAF.

1. Each scalar has a unique wNAF.
2. The wNAF of a scalar is at most one bit longer than its binary representation.

Algorithm 10 describes the generation of the wNAF from the decimal representation as stated [Sol00]. This algorithm uses the *signed modulo* operation.

Definition 5.2. *The* **signed modulo** *operation a* mods *b is defined as a* mod *b and $-b/2 \leq a < b/2$, the residue with smallest absolute value.*

Algorithm 10 Decimal to wNAF

Require: Width w, an n-bit scalar d in its decimal representation.
Ensure: The wNAF $(\delta[n], \delta[n-1], \ldots, \delta[0])$ of d.

```
1: i ← 0
2: while d ≥ 1 do
3:     if d is even then
4:         δ[i] ← 0
5:     else
6:         δ[i] ← d mods 2^w
7:         d ← d − δ[i]
8:     end if
9:     d ← d/2
10:    i ← i + 1
11: end while
12: return (δ[n], δ[n−1], ..., δ[0]).
```

Note, that d is odd each time a signed modulo operation is performed. Therefore, the absolute value of the result is always odd and $\leq 2^{w-1} - 1$, i.e. an element of the digit set.

Remark 5.3. *For $w = 2$, Algorithm 10 produces the same output as Reitwiesner's famous algorithm to generate the Non Adjacent Form (NAF) [Rei60]. In other words: 2NAF equals NAF.*

The wNAF can also be generated from the binary representation of the scalar d as shown in Example 5.5. Here, the scalar is scanned bitwise, starting at the least significant bit. If the current bit $d[i]$ is zero, $\delta[i]$ is set to zero and the scan

continues. If $d[i] = 1$ holds, the algorithm examines a window of w bits, namely $(d[i+w-1], \ldots, d[i])$. $\delta[i]$ is set to the decimal value of the current window mods 2^w and $(\delta[i+w-1, \ldots, \delta[i+1])$ is set to $(0, \ldots, 0)$. If $\delta[i] < 0$ holds, 1 is added to the remaining binary representation of the scalar, i.e. $(d[n-1], \ldots, d[i+w])$ is set to $(d[n-1], \ldots, d[i+w]) + 1$. This is justified by the following lemma.

Lemma 5.4. *Let* $(d[w-1], \ldots, d[0])$ *be the binary representation of an odd scalar* $d < 2^w$. *Further let* $r \equiv d \bmod 2^w$. *There are two cases:*

$(r > 0) : (\underbrace{0, \ldots, 0}_{w-1}, r)$ *has the same decimal value as* $(d[w-1], \ldots, d[0])$.

$(r < 0) : (1, \underbrace{0, \ldots, 0}_{w-1}, r)$ *has the same decimal value as* $(d[w-1], \ldots, d[0])$.

Proof. At first, d odd implies that r is also odd, since 2^w is even.

$(r > 0)$: This implies that $r = d$, since $d < 2^w$. Therefore $r = \sum_{i=0}^{w-1} d[i] \cdot 2^i = d$.

$(r < 0)$: This implies that $r = d - 2^w \Leftrightarrow r + 2^w = d$, since $d < 2^w$. Therefore $2^w + r = \sum_{i=0}^{w-1} d[i] \cdot 2^i = d$.

□

This construction also clarifies that the resulting representation satisfies the property wNAF-2. The 1's which have to be added to the remaining binary representation are called *carry bits*, because they carry out of the current window. Those carry bits are the reason why the wNAF of a scalar can be one bit longer than its binary representation, namely if a carry bit appears in the last window $(d[n-1], \ldots, d[n-w])$.

Example 5.5. *Let* $d = 619$ *with binary representation* $(1, 0, 0, 1, 1, 0, 1, 0, 1, 1)$ *and* $w = 3$. *The* 3*NAF of* d *is generated as follows*

	1	0	0	1	1	0	1	0	1	1
	1	0	0	1	1	0	1	0	0	3
				$\boxed{1}$	0	0	$\bar{3}$			
	1	0	1	0						
	1	0	1	0						
$\boxed{1}$	0	0	$\bar{3}$							
1	0	0	$\bar{3}$	0	0	0	$\bar{3}$	0	0	3

Hence, the 3*NAF of* 619 *is given as* $(1, 0, 0, \bar{3}, 0, 0, 0, \bar{3}, 0, 0, 3)$.

Theorem 5.6. *The AHD of the wNAF is*

$$\mathcal{AHD}(w\text{NAF}) = \frac{1}{w+1}$$

Proof. Recall that the AHD is the average density of non-zero digits of a randomly chosen wNAF with bit length $n \to \infty$. This density is given as the average number of non-zero digits divided by the average number of digits written out by the algorithm to generate the wNAF. Two cases exist:

$d[i] = 0$: In this case only one digit is written out, which is zero.

$d[i] = 1$: In this case w digits are written out, one non-zero and $w - 1$ zero.

The cases mentioned above appear each with a probability of $1/2$. Those probabilities are also correct after a carry bit was generated, since this also happens with probability $1/2$. Therefore, the AHD is given as

$$\mathcal{AHD}(w\text{NAF}) = \frac{\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1}{\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot w} = \frac{1}{w+1}$$

□

Table 5.1 shows some example values of the AHD of the wNAF for different values of w.

w	$\mathcal{AHD}(w\text{NAF})$	
2	$1/3$	≈ 0.3333
3	$1/4$	$= 0.2500$
4	$1/5$	$= 0.2000$
5	$1/6$	≈ 0.1667
6	$1/7$	≈ 0.1429

Table 5.1: Example values of $\mathcal{AHD}(w\text{NAF})$

In [MS04a], Muir and Stinson also proved that the HW of a scalar given in its wNAF is *minimal* for any choice of w. This implies, that the AHD of the wNAF is minimal amongst all $\mathcal{D}$-representations which use the digit set $\mathcal{D}_w = \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$. Therefore, no other $\mathcal{D}$-representation using this digit set provides fewer non-zero digits than the wNAF.

5.2 The Joint Sparse Form

The *Joint Sparse Form* (JSF) is a $\mathcal{D}$-representation, that provides a low AJHD and can therefore speed up the Shamir method. It was originally proposed by Solinas in [Sol01] to recode two scalars. In [Pro03], Proos generalized the JSF to an arbitrary number of scalars. Here, the goal is to generate zero columns in the matrix

$$\begin{pmatrix} d_1[n-1] & \dots & d_1[0] \\ d_2[n-1] & \dots & d_2[0] \\ \vdots & & \vdots \\ d_k[n-1] & \dots & d_k[0] \end{pmatrix}$$

which consists of the binary representations of the scalars $d_1, \dots, d_k$. The definition of the JSF as stated in [Pro03] is reviewed in Definition 5.7.

Definition 5.7. *The matrix*

$$\begin{pmatrix} \delta_1[n] & \delta_1[n-1] & \dots & \delta_1[0] \\ \delta_2[n] & \delta_2[n-1] & \dots & \delta_2[0] \\ \vdots & \vdots & & \vdots \\ \delta_k[n] & \delta_k[n-1] & \dots & \delta_k[0] \end{pmatrix}$$

where $\delta_j[i] \in \mathcal{D} = \{0, \pm 1\}$ for $i = 0, \dots, n, j = 1, \dots, k$ is called a **JSF***, if and only if the following three properties hold.*

JSF-1 *For each non-zero column i, there exists a row $(\delta_j[n], \delta_j[n-1], \dots, \delta_j[0])$ such that*

1. *$d_j[i] \neq 0$*
2. *Either $i = 0$ or there exists an $b < i$ such that $\delta_j[i-1] = \delta_j[i-2] = \dots = \delta_j[b] = 0$ and either $b = 0$ or the $(b-1)$-st column is a zero column.*

JSF-2 *No consecutive bits are $1\bar{1}$ or $\bar{1}1$.*

JSF-3 *If there exists a row $(\delta_j[n], \delta_j[n-1], \dots, \delta_j[0])$ and integers i, b such that $b < i$, $\delta_j[i+1] \neq \delta_j[i]$ and $\delta_j[i] = \delta_j[i-1] = \dots \delta_j[b] \neq 0$ then the $(i+1)$-st column is a zero column.*

According to this definition, the JSF uses the digit set $\mathcal{D} = \{0, \pm 1\}$ and is therefore a signed binary representation. Further, the JSF has the following properties as proven in [Pro03].

1. Each k scalars have a unique JSF.
2. The JSF of k scalars is at most one bit longer than the binary representation of the largest scalar.

Efficient algorithms to generate the JSF can be found in [Pro03] and [HKPR04]. However, those algorithms are long and complicated and therefore not copied here. Instead, the basic idea of generating a JSF is explained with the following two lemmata.

Lemma 5.8. *Let $(0, d[n-1], \ldots, d[1], 1)$ be the binary representation of an odd scalar $d < 2^n$. There exists a signed binary representation $(\delta[n], \delta[n-1], \ldots, \delta[0])$ of d such that $\delta[n-1] = 0$.*

Proof. If $d[n-1] = 0$ nothing has to be done, i.e.

$$(\delta[n], \delta[n-1], \ldots, \delta[0]) \leftarrow (0, d[n-1], \ldots, d[1], 1).$$

If $d[n-1] = 1$ set

$$(\delta[n], \delta[n-1], \ldots, \delta[1], \delta[0]) \leftarrow (1, d[n-1] - 1, \ldots, d[1] - 1, \bar{1}).$$

This yields a signed binary representation of d, since

$$\begin{aligned}
\sum_{i=0}^{n} \delta[i] \cdot 2^i &= -1 + \sum_{i=1}^{n-1} (d[i] - 1) \cdot 2^i + 2^n \\
&= -1 + \sum_{i=1}^{n-1} d[i] \cdot 2^i - \sum_{i=1}^{n-1} 2^i + 2^n \\
&= -1 + \sum_{i=1}^{n-1} d[i] \cdot 2^i - (2^n - 2) + 2^n \\
&= 1 + \sum_{i=1}^{n-1} d[i] \cdot 2^i
\end{aligned}$$

and $d[0] = 1$. □

The following lemma generalizes Lemma 5.8 and shows how it can be applied successively to a scalar.

Lemma 5.9. *Let $(\delta[n], \delta[n-1], \ldots, \delta[0])$ be a signed binary representation of the scalar $d < 2^n$, such that for some integer a, $\delta[i] \in \{0, 1\}$ for $i = a, \ldots, n$.*

Then for all $b \in \{a+1, \ldots, n-1\}$ a similar transformation as in Lemma 5.8 can be used to form a signed binary representation $(\delta[n]', \delta[n-1]', \ldots, \delta[0]')$ of d, such that

1. *$\delta[i]' = \delta[i]$ for $i = 0, \ldots, a-1$*
2. *$\delta[b]' = 0$*
3. *$\delta[i]' \in \{0, 1\}$ for $i = b+1, \ldots, n$*

if either $\delta[b] = 0$ *or there exists an* $w \in \{a, \dots b-1\}$ *such that* $\delta[w] = 1$. *Such a transformation will be referred to as an* **elementary transformation**.

Proof. If $\delta[b] = 0$ nothing has to be done, i.e.

$$(\delta[n]', \dots, \delta[0]') \leftarrow (\delta[n], \dots, \delta[0]).$$

If $\delta[b] = 1$, choose an $w \in \{a, \dots b-1\}$ such that $\delta[w] = 1$ holds and set

$$\begin{aligned}
(\delta[n]', \dots, \delta[b+1]') &\leftarrow (\delta[n], \dots, \delta[b+1]) + 1 \\
(\delta[b]', \dots, \delta[w+1]', \delta[w]') &\leftarrow (\delta[b] - 1, \dots, \delta[w+1] - 1, \bar{1}) \\
(\delta[w-1]', \dots, \delta[0]') &\leftarrow (\delta[w-1], \dots, \delta[0])
\end{aligned}$$

This yields a signed binary representation of d, since

$$\begin{aligned}
\sum_{i=0}^{n} \delta[i]' \cdot 2^i &= \sum_{i=0}^{w-1} \delta[i]' \cdot 2^i + \sum_{i=w}^{b} \delta[i]' \cdot 2^i + \sum_{i=b+1}^{n} \delta[i]' \cdot 2^i \\
&= \sum_{i=0}^{w-1} \delta[i] \cdot 2^i - 2^w + \sum_{i=w+1}^{b-1} (\delta[i] - 1) \cdot 2^i + \sum_{i=b+1}^{n} \delta[i] \cdot 2^i + 2^{b+1} \\
&= \sum_{i=0}^{w-1} \delta[i] \cdot 2^i - 2^w + \sum_{i=w+1}^{b-1} \delta[i] \cdot 2^i \\
&\qquad - \sum_{i=w+1}^{b-1} 2^i + \sum_{i=b+1}^{n} \delta[i] \cdot 2^i + 2^{b+1} \\
&= \sum_{i=0}^{w-1} \delta[i] \cdot 2^i - 2^w + \sum_{i=w+1}^{b-1} \delta[i] \cdot 2^i \\
&\qquad - (2^b - 2^{w+1}) + \sum_{i=b+1}^{n} \delta[i] \cdot 2^i + 2^{b+1} \\
&= \sum_{i=0}^{w-1} \delta[i] \cdot 2^i + 2^w + \sum_{i=w+1}^{b-1} \delta[i] \cdot 2^i + 2^b + \sum_{i=b+1}^{n} \delta[i] \cdot 2^i + 2^{b+1} \\
&= \sum_{i=0}^{n} \delta[i] \cdot 2^i
\end{aligned}$$

and $d[b] = 1 = d[w]$. □

Example 5.10. *Let* $(0,1,0,1,1,0,1,1,0,1,1)$ *be the binary representation of* $d = 731$ *padded with one zero to the left. The following table shows how elementary transformations can be applied starting at the least significant bit. The entry* $d[b]$ *is marked bold.*

$(d[10],\ldots,d[0])$	a	b	w
$(0,1,0,1,1,0,1,\mathbf{1},0,1,1)$	0	3	0
$(0,1,0,1,\mathbf{1},1,0,0,\bar{1},0,\bar{1})$	3	6	5
$(0,\mathbf{1},1,0,0,\bar{1},0,0,\bar{1},0,\bar{1})$	6	9	8
$(1,0,\bar{1},0,0,\bar{1},0,0,\bar{1},0,\bar{1})$			

Here, the choices for b *and* w *are made arbitrarily. For example, it is also possible to apply the first elementary transformation with* $b = 3$ *and* $w = 1$*, which yields* $(0,1,0,1,1,1,0,0,\bar{1},\bar{1},1)$.

The above lemmata and the example also clarify, why the JSF can be at most one bit longer than the original binary representation. If $(\delta[n]',\ldots,\delta[b+1]')$ is set to $(\delta[n],\ldots,\delta[b+1])+1$, this addition can carry over the most significant bit of the binary representation $(d[n-1])$ and $d[n]$ becomes 1.

The algorithm to generate the JSF works as follows. Consider the matrix

$$\begin{pmatrix} 0 & d_1[n-1] & \ldots & d_1[0] \\ 0 & d_2[n-1] & \ldots & d_2[0] \\ \vdots & \vdots & & \vdots \\ 0 & d_k[n-1] & \ldots & d_k[0] \end{pmatrix}$$

which consists of the binary representations of the scalars and one zero column to the left. The columns of this matrix are denoted by $C_n, C_{n-1},\ldots,C_0$. Further, an index a is required which denotes the current column. At first, a is set to 0.

The algorithm starts scanning, beginning at the a-th column, until it finds the smallest block of columns $C_r,\ldots,C_a$, such that an elementary transformation with $b = r$ can be applied to all rows $(d_j[r],\ldots,d_j[a])$, $j = 1,\ldots,k$. Such blocks are called *convertible blocks*. In other words, the algorithm searches for the smallest r which satisfies

1. $r \geq a$
2. For each $j = 1,\ldots,k$, either $d_j[r] = 0$ or there exists an $w \in \{a,\ldots,r-1\}$, such that $d_j[w] = 1$.

Then, an elementary transformation is applied to each row and C_r becomes a zero column. Next, the algorithm sets $a \leftarrow r+1$ and continues the scan.

After the algorithm terminates, the matrix satisfies the property JSF-1 [Pro03]. In a second stage, the algorithm applies certain replacements to ensure that the scalars also satisfy the properties JSF-2 and JSF-3, which guarantee the uniqueness. The replacements are

1. Replace $(\bar{1}, 1)$ with $(0, \bar{1})$ and $(1, \bar{1})$ with $(0, 1)$ whenever possible.

2. If there exists a row $(\delta_j[n], \delta_j[n-1], \ldots, \delta_j[0])$ and integers i, b such that $b < i$, $\delta_j[i+1] = 0$, the $i+1$-th column is no zero column and $\delta_j[i] = \delta_j[i-1] = \ldots \delta_j[b] = 1$ or $\delta_j[i] = \delta_j[i-1] = \ldots \delta_j[b] = \bar{1}$ then replace $(d_j[i+1], \ldots, d_j[b])$ with $(1, 0, \ldots, 0, \bar{1})$ or $(\bar{1}, 0, \ldots, 0, 1)$, respectively.

Also, both those replacements are applied starting at the least significant bit.

Example 5.11. *This example shows how the algorithm generates the JSF of the four scalars* $d_1 = 2716, d_2 = 801, d_3 = 3742$ *and* $d_4 = 3395$. *The convertible blocks are marked bold.*

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \mathbf{0} & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \mathbf{0} & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & \mathbf{0} & 0 & \bar{1} & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \mathbf{0} & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & \mathbf{0} & \mathbf{0} & \mathbf{1} & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} & 0 & 0 & \bar{1} & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & \mathbf{0} & \mathbf{1} & \mathbf{0} & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & \mathbf{1} & \mathbf{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & \mathbf{1} & \mathbf{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 1 & 0 \\ 0 & 1 & 1 & \mathbf{0} & \mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \mathbf{1} & \mathbf{1} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & \mathbf{0} & \mathbf{1} & 0 & \bar{1} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & \mathbf{0} & \mathbf{0} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 1 & 0 \\ 0 & \mathbf{1} & \mathbf{1} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

The next step is to apply the replacements to ensure JSF-2 and JSF-3. The entries to replace are marked bold.

$$\begin{pmatrix} 1 & 0 & \bar{1} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \mathbf{\bar{1}} & \mathbf{1} & 0 \\ 1 & 0 & \bar{1} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} \end{pmatrix}$$

After applying those replacements, the JSF of the scalars $d_1, \ldots, d_4$ is given as

$$\begin{pmatrix} 1 & 0 & \bar{1} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & 0 & \bar{1} & 0 \\ 1 & 0 & \bar{1} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \bar{1} \end{pmatrix}$$

The original algorithm proposed by Proos in [Pro03] can generate the JSF from any signed binary representation of the scalars. The difference is, that his algorithm is using a generalized elementary transformation, which can also be applied to signed binary representations. However, since the scalars are typically given in their binary representation, the above explained method is sufficient.

The next step is to compute the AJHD of the JSF as stated in [Pro03].

Theorem 5.12. *The AJHD of the JSF is*

$$\mathcal{AJHD}_k(\text{JSF}) = 1 - \frac{1}{c_k},$$

where c_k is given by the recursive formula

$$c_k = \frac{1}{2^k}\left(3 + \sum_{j=1}^{k-1} \binom{k}{j}(c_j + 1)\right)$$

and $c_1 = 1.5$.

Proof. The first step is to find an upper bound of the AJHD. Recall that a block of columns $C_b, \ldots, C_a$ is convertible, if for each row $j = 1, \ldots, k$, there exists a $w_j \in \{a, \ldots, b-1\}$ such that $d_j[w_j] = 1$. In the worst case, all those entries are in different columns, i.e. w.l.o.g. $w_1 = a, w_2 = a+1, \ldots, w_k = a+k-1$. Then the $(a+k)$-th column can be transformed to a zero column. This means, that the algorithm has to scan at most $k+1$ columns to find a convertible block, i.e. to generate one zero column and therefore $1/(k+1)$ is an upper bound for the AJHD.

The next step is to estimate the expected number of columns which have to be scanned in order to generate one zero column. Let c_k denote this number. If $k = 1$, the a-th column is 0 or 1 each with probability 1/2. If $d_1[a] = 0$ one column has to be scanned. If $d_1[a] = 1$, an elementary transformation can be used to insure that the $a+1$-th column becomes zero and therefore two columns have to be scanned. Hence, $c_1 = 1/2 \cdot 1 + 1/2 \cdot 2 = 3/2$.

Now assume that $c_1, c_2, \ldots, c_{k-1}$ are known and consider c_k. When a column is scanned, exactly j of the k entries are 1 with probability $\binom{k}{j}/2^k$. If all entries are 0, only one column has to be scanned. If all entries are 1, the next column can be transformed into a zero column and two columns have to be scanned. If

j entries are are 1, the expected number of additional columns to scan is c_{k-j} and $c_{k-j}+1$ in total. Therefore

$$\begin{aligned} c_k &= \frac{1}{2^k}\cdot 1 + \frac{\binom{k}{1}}{2^k}(c_{k-1}+1) + \ldots + \frac{\binom{k}{k-1}}{2^k}(c_1+1) + \frac{1}{2^k}\cdot 2 \\ &= \frac{1}{2^k}\left(3 + \sum_{j=1}^{k-1}\binom{k}{j}(c_j+1)\right) \end{aligned}$$

The quotient $1/c_k$ gives the average density of zero columns which are generated and therefore

$$\mathcal{AJHD}_k(\text{JSF}) = 1 - \frac{1}{c_k}$$

□

Table 5.2 shows some example values of the AJHD of the JSF for different values of k.

k	$\mathcal{AJHD}_k(\text{JSF})$	
1	1/3	≈ 0.3333
2	1/2	$= 0.5000$
3	23/39	≈ 0.5897
4	115/179	≈ 0.6424
5	4279/6327	≈ 0.6763
6	152821/218357	≈ 0.6998

Table 5.2: Example values of $\mathcal{AJHD}_k(\text{JSF})$

According to this table, the JSF of one scalar has the same AHD as the 2NAF. In fact, the output of both algorithms is exactly the same.

In [Pro03], Proos also proved that the JHW of k scalars given in their JSF is minimal. This implies, that the AJHD of the JSF is minimal amongst all $\mathcal{D}$-representations which use the digit set $\mathcal{D} = \{0, \pm 1\}$. Therefore, no other $\mathcal{D}$-representation using this digit set provides fewer non-zero columns than the JSF.

6 Left-to-Right producible Low-Weight Representations

In the last chapter, two $\mathcal{D}$-representations which minimize the AHD of one scalar and the AJHD of k scalars were introduced. One might therefore think that the research in this direction is concluded. But there still remains an issue, namely the direction in which the scalars are recoded. Both the wNAF and the JSF can only be generated starting at the least significant bit, i.e. right-to-left. This is due to the carry bit, which is generated by both algorithms.

The problem with right-to-left recoding is, that the algorithms to compute the multi-scalar multiplication start the evaluation at the most significant bit, i.e. left-to-right (see Section 4.1.3). If the scalars are to be represented in the wNAF or the JSF, the recoding has to be done in a separate step, prior to the evaluation. Hence, it is necessary to store the whole recoded scalars, which requires memory of the order of magnitude of n bits for each scalar.

Recently, a solution for this problem was found. The idea is to recode the scalars starting at the most significant bit, i.e. left-to-right. Then it becomes possible to merge the recoding of the scalars and the evaluation of the multi-scalar multiplication, by recoding only small portions of the scalars at once. Instead of storing the whole recoded scalars, only the recoded portions must be stored, which leads to a significant saving of memory.

The $\mathcal{D}$-representations introduced in this chapter are left-to-right duals of the $\mathcal{D}$-representations introduced in the last chapter. This means, that they use the same digit set and provide the same, minimal AHD and AJHD as the wNAF and the JSF, respectively. The key to left-to-right producible low-weight representations is a special signed binary representation introduced in the next section.

6.1 The Mutual Opposite Form

The *Mutual Opposite Form* (MOF) was proposed by Okeya, Schmidt-Samoa, Spahn and Takagi in [OSST04]. It is also known by the name *Alternating Greedy Expansion* as proposed by Grabner, Heuberger, Prodinger and Thuswaldner in [GHPT03]. The definition of the MOF as stated in [OSST04] is reviewed in Definition 6.1.

Definition 6.1. *The vector* $(\mu[n], \mu[n-1], ..., \mu[0])$*, where* $\mu[i] \in \mathcal{D} = \{0, \pm 1\}$ *for* $i = 0, \ldots, n$ *is called a* **MOF***, if and only if the following two properties hold.*

MOF-1 *The signs of adjacent non-zero digits (without considering zeros) are opposite.*

MOF-2 *The most significant non-zero digit is 1 and the least significant non-zero digit is* -1*.*

According to this definition, the MOF uses the digit set $\mathcal{D} = \{0, \pm 1\}$ and is therefore a signed binary representation. Also, the two following properties of the MOF were proven in [OSST04].

1. Each scalar has a unique MOF.
2. The MOF of a scalar is at most one bit longer than its binary representation.

Interestingly, the MOF of a scalar d can be generated by the *bitwise subtraction* $2d \ominus d$, as proven in [OSST04]. Algorithm 11 represents the left-to-right generation of the MOF from the binary representation.

Algorithm 11 Binary to MOF

Require: An n-bit scalar d in its binary representation $(d[n-1], \ldots, d[0])$.
Ensure: The MOF $(\mu[n], \mu[n-1], \ldots, \mu[0])$ of d.
1: $\mu[n] \leftarrow d[n-1]$
2: **for** $i = n-1$ down to 1 **do**
3: $\quad \mu[i] \leftarrow d[i-1] - d[i]$
4: **end for**
5: $\mu[0] \leftarrow -d[0]$
6: **return** $(\mu[n], \mu[n-1], ..., \mu[0])$.

In order to generate one MOF digit, Algorithm 11 requires only two consecutive binary digits. Therefore each MOF digit can be generated independently from other digits, which is clarified by the following example. In particular, the MOF can also be generated from right-to-left.

Example 6.2. *Let* $d = 3749$*. The MOF of* d *is generated from the binary representation of* d *as follows*

	1	1	1	0	1	0	1	0	0	1	0	1	0	2d
$\ominus$		1	1	1	0	1	0	1	0	0	1	0	1	d
	1	0	0	$\bar{1}$	1	$\bar{1}$	1	$\bar{1}$	0	1	$\bar{1}$	1	$\bar{1}$	

Hence, the MOF of $d = 3749$ *is given as* $(1, 0, 0, \bar{1}, 1, \bar{1}, 1, \bar{1}, 0, 1, \bar{1}, 1, \bar{1})$.

Theorem 6.3. *The AHD of the MOF is*

$$\mathcal{AHD}(\mathrm{MOF}) = \frac{1}{2}$$

Proof. The i-th MOF digit $\mu[i]$ is computed as $d[i-1] - d[i]$. Without considering the most and least significant bit,

$$\begin{aligned} \mu[i] = 0 \quad &\Leftrightarrow \quad (d[i], d[i-1]) = (1,1) \vee \\ &\qquad\quad (d[i], d[i-1]) = (0,0) \\ \mu[i] \neq 0 \quad &\Leftrightarrow \quad (d[i], d[i-1]) = (0,1) \vee \\ &\qquad\quad (d[i], d[i-1]) = (1,0) \end{aligned}$$

holds each with a probability of $1/2$. From this, it immediately follows that $\mathcal{AHD}(\mathrm{MOF}) = 1/2$. □

Remark 6.4. *The MOF equals the recoding performed by the classical Booth algorithm [Boo51] to speed up the binary multiplication $A \cdot B$. The Booth algorithm successively scans two consecutive bits of the multiplier A from right-to-left. Depending on these bits, one of the following operations is performed:*

$(a[i], a[i-1]) = (1,1)$: *No operation.*

$(a[i], a[i-1]) = (0,0)$: *No operation.*

$(a[i], a[i-1]) = (0,1)$: *Add B to the intermediate result.*

$(a[i], a[i-1]) = (1,0)$: *Subtract B from the intermediate result.*

where $a[-1]$ is defined as 0.

6.2 The width-w Mutual Opposite Form

The *width-w Mutual Opposite Form* (wMOF) is a left-to-right dual of the wNAF and can therefore be used to speed up the Interleave method. It was introduced by Okeya, Schmidt-Samoa, Spahn and Takagi in [OSST04]. Similar representations were proposed by Muir and Stinson in [MS04b] and by Avanzi in [Ava04]. The definition of the wMOF as stated in [OSST04] is reviewed in Definition 6.5. Throughout this section, $w \geq 2$ is assumed.

Definition 6.5. *The vector $(\delta[n], \delta[n-1], \ldots, \delta[0])$ is called a w***MOF** *if and only if the following three properties hold:*

wMOF-1 *The most significant non-zero digit is positive.*

wMOF-2 *All but the least significant non-zero digit x are adjoint by $w-1$ zeros as follows:*

- *In the case of $2^{k-1} < |x| < 2^k$ for an integer $2 \leq k \leq w-1$, the pattern equals $\underbrace{0\ldots0}_{k} x \underbrace{0\ldots0}_{w-k-1}$,*
- *In the case of $|x| = 1$, either the pattern equals $x\underbrace{0\ldots0}_{w-1}$ and the next lower non-zero digit has the opposite sign of x or the pattern equals $0x\underbrace{0\ldots0}_{w-2}$ and the next lower non-zero digit has the same sign as x.*

If x is the least significant non-zero digit, it is possible that the number of right-hand adjacent zeros is smaller than stated above. In addition, it is not possible that the last non-zero digit is a 1 following any non-zero digit.

wMOF-3 *Each non-zero digit is odd and less than 2^{w-1} in absolute value.*

According to this definition, the wMOF uses the same digit set as the wNAF, namely all odd digits in $\mathcal{D}_w = \{0, \pm1, \pm3, \ldots, \pm2^{w-1} - 1\}$. It is also shown in [OSST04] that

1. Each scalar has a unique wMOF.
2. The wMOF of a scalar is at most one bit longer than its binary representation.

Algorithm 12 represents the algorithm to generate the wMOF $(\delta[n], \ldots, \delta[0])$ of a scalar d from its MOF $(\mu[n], \ldots, \mu[0])$, as stated in [OSST04]. The algorithm scans the MOF bitwise, starting at the most-significant bit. If the current MOF digit $\mu[i]$ is zero, $\delta[i]$ is set to zero and the scan continues. If $\mu[i]$ is different from zero, a window consisting of w consecutive MOF digits is examined, namely $(\mu[i], \ldots, \mu[i-w+1])$. Let $\mu[l]$ be the *least significant non-zero entry* in this window, i.e. $\mu[l] \neq 0$ and $\mu[b] = 0$ for $b = i-w+1, \ldots, l-1$. In this case, $\delta[l]$ is set to the decimal value of the window and the remaining $w-1$ digits are set to zero. Note, that the window is recoded without producing any carry over.

The above explained transformation is always possible, because the largest decimal value that can be represented using w consecutive MOF digits is $2^{w-1} - 1$, which corresponds to the MOF $(1, 0, \ldots, 0, \bar{1})$. Therefore, the wMOF digit $\delta[l]$ can be set to the decimal value of the current window without risking to exceed the digit set $\mathcal{D}_w$. This is the MOF's great advantage over the binary representation, where the largest decimal value that can be represented using w bits is $2^w - 1$, namely $(1, 1, \ldots, 1)$. When using the binary representation, the

case where the decimal value is larger that $2^{w-1}-1$ has to be treated separately, which causes a carry over to the left, see Section 5.1.

To summarize, the fact that the decimal value of w consecutive MOF digits is at most $2^{w-1}-1$ is the key feature that allows the wMOF to be generated from left-to-right.

The algorithm to generate the wMOF as stated in [OSST04] doesn't compute the decimal value of the current window directly. Instead it utilizes a table T_w where the conversions for all possible windows of length w, where the most significant digit is non-zero, are stored. For example, the conversion table T_4 looks as follows:

$$
\begin{array}{llll}
1000 \rightarrow 1000 & 1\bar{1}10 \rightarrow 0030 & 1\bar{1}01 \rightarrow 0005 & 100\bar{1} \rightarrow 0007 \\
\bar{1}000 \rightarrow \bar{1}000 & \bar{1}1\bar{1}0 \rightarrow 00\bar{3}0 & \bar{1}10\bar{1} \rightarrow 000\bar{5} & \bar{1}001 \rightarrow 000\bar{7} \\
1\bar{1}00 \rightarrow 0100 & 10\bar{1}0 \rightarrow 0030 & 1\bar{1}1\bar{1} \rightarrow 0005 & 10\bar{1}1 \rightarrow 0007 \\
\bar{1}100 \rightarrow 0\bar{1}00 & \bar{1}010 \rightarrow 00\bar{3}0 & \bar{1}1\bar{1}1 \rightarrow 000\bar{5} & \bar{1}01\bar{1} \rightarrow 000\bar{7}
\end{array}
$$

This table also clarifies that the representation produced by Algorithm 12 satisfies the property wMOF-2. In addition to the table T_w, Algorithm 12 also requires the tables $T_2, \ldots T_{w-1}$ to convert the last couple of bits correctly. Efficient methods to store and access this table were also proposed in [OSST04].

Algorithm 12 MOF to wMOF

Require: Width w, an n-bit scalar d in its MOF $(\mu[n], \mu[n-1], \ldots, \mu[0])$.
Ensure: The wMOF $(\delta[n], \delta[n-1], \ldots, \delta[0])$ of d.

```
1: i ← n
2: while i ≥ w − 1 do
3:     if d[i] = 0 then
4:         δ[i] ← 0
5:         i ← i − 1
6:     else
7:         (δ[i], ..., δ[i−w+1]) ← T_w(μ[i], ..., μ[i−w+1])
8:         i ← i − w
9:     end if
10: end while
11: if i ≥ 0 then
12:     (δ[i], ..., δ[0]) ← T_{i+1}(μ[i], ..., μ[0])
13: end if
14: return (δ[n], δ[n−1], ..., δ[0]).
```

The purpose of the second if-clause (line 11) is to convert the last couple of digits correctly. Leading zeros in the window are also skipped in this case. Since each MOF digit can be generated independently from any other MOF digit, the wMOF can also be generated directly from the binary representation of the scalar d, by generating the MOF digits on-the-fly as required [OSST04].

Example 6.6. *Let $d = 619$ and $w = 3$. The MOF of d is computed using Algorithm 11 and given as $(1, \bar{1}, 0, 1, 0, \bar{1}, 1, \bar{1}, 1, 0, \bar{1})$. Algorithm 12 performs the following steps to generate the 3MOF of d.*

$$
\begin{array}{ccccccccccc}
\underbrace{1 \quad \bar{1} \quad 0} & & & 1 & 0 & \bar{1} & 1 & \bar{1} & 1 & 0 & \bar{1} \\
0 & 1 & 0 & \underbrace{1 \quad 0 \quad \bar{1}} & & & 1 & \bar{1} & 1 & 0 & \bar{1} \\
 & & & 0 & 0 & 3 & \underbrace{1 \quad \bar{1} \quad 1} & & & 0 & \bar{1} \\
 & & & & & & 0 & 0 & 3 & \underbrace{0 \quad \bar{1}} & \\
 & & & & & & & & & 0 & \bar{1} \\
\hline
0 & 1 & 0 & 0 & 0 & 3 & 0 & 0 & 3 & 0 & \bar{1}
\end{array}
$$

Hence, the 3MOF of 619 is given as $(0, 1, 0, 0, 0, 3, 0, 0, 3, 0, \bar{1})$.

Remark 6.7. *Algorithm 12 can also be applied to the MOF of the scalar from right-to-left. In this case, the result exactly equals the output of Algorithm 10. Therefore, the MOF can also be used for a carry free generation of the wNAF [OSST04].*

Theorem 6.8. *The AHD of the wMOF is*

$$\mathcal{AHD}(w\text{MOF}) = \frac{1}{w+1}$$

Proof. The AHD is the average density of non-zero digits of a randomly chosen wMOF with bit length $n \to \infty$. This density is given as the average number of non-zero digits divided by the average number of digits written out by Algorithm 12. Two cases exist:

$\mu[i] = 0$: In this case only one digit is written out, which is zero.

$\mu[i] \neq 0$: In this case w digits are written out, one non-zero and $w - 1$ zero.

Since $\mathcal{AHD}(\text{MOF}) = 1/2$, both cases appear each with a probability of $1/2$. Therefore, the AHD of the wMOF is given as

$$\mathcal{AHD}(w\text{MOF}) = \frac{\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1}{\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot w} = \frac{1}{w+1}$$

□

Since the AHD of the wMOF is the same as the AHD of the wNAF, Table 5.1 is also valid for the wMOF.

In [Ava04], Avanzi proved that the HW of a scalar given in its wMOF is minimal for any choice of w. This implies, that the AHD of the wMOF is minimal amongst all $\mathcal{D}$-representations which use the digit set $\mathcal{D}_w = \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$. Therefore, no other $\mathcal{D}$-representation using this digit set provides fewer non-zero digits than the wMOF.

6.3 The Left-to-Right Joint Sparse Form

The *Left-to-Right Joint Sparse Form* (ltrJSF) is a left-to-right dual of the JSF and can therefore be used to speed up the Shamir method. It was independently proposed by Okeya, Takagi and the author of this thesis in [DOT05a] and by Heuberger, Katti, Prodinger and Ruan in [HKPR04]. Here, the goal is to generate zero columns in the matrix

$$\begin{pmatrix} \mu_1[n] & \mu_1[n-1] & \dots & \mu_1[0] \\ \mu_2[n] & \mu_2[n-1] & \dots & \mu_2[0] \\ \vdots & \vdots & & \vdots \\ \mu_k[n] & \mu_k[n-1] & \dots & \mu_k[0] \end{pmatrix}$$

which consists of the MOF's of the scalars. The ltrJSF is a signed binary representation, i.e. it uses the digit set $\mathcal{D} = \{0, \pm 1\}$ and its generation is based on the following lemma.

Lemma 6.9. *Let $(\mu[n], \mu[n-1], \dots, \mu[0])$ be the MOF of a non-zero scalar $d < 2^n$. There exists a signed binary representation $(\delta[n], \delta[n-1], \dots, \delta[0])$ of d, such that $\delta[b] = 0$ for each $b \in \{0, \dots, n\} \setminus \{l\}$, where l is the index of the least significant non-zero digit in the MOF of d, i.e. $\mu[l] \neq 0$ and $\mu[i] = 0$, for $i = 0, \dots, l-1$.*

Proof. If $\mu[b] = 0$ nothing has to be done, i.e.

$$(\delta[n], \dots, \delta[0]) \leftarrow (\mu[n], \dots, \mu[0]).$$

If $\mu[b] \neq 0$, choose an $w \in \{l, \dots, b-1\}$, such that $(\mu[b], \mu[b-1], \dots, \mu[w+1], \mu[w])$ is equal to $(1, 0, \dots, 0, \bar{1})$ or $(\bar{1}, 0, \dots, 0, 1)$. Such a w always exists because of the first MOF property (MOF-1). Then set

$$\begin{aligned} (\delta[n], \dots, \delta[b+1]) &\leftarrow (\mu[n], \dots, \mu[b+1]) \\ (\delta[b], \delta[b-1], \dots, \delta[w]) &\leftarrow (0, \overline{\mu[b]}, \dots, \overline{\mu[b]}) \\ (\delta[w-1], \dots, \delta[0]) &\leftarrow (\mu[w-1], \dots, \mu[0]) \end{aligned}$$

In other words, one of the replacements

$$\begin{aligned} (0, 1, \dots, 1, 1) &\leftarrow (1, 0, \dots, 0, \bar{1}) \\ (0, \bar{1}, \dots, \bar{1}, \bar{1}) &\leftarrow (\bar{1}, 0, \dots, 0, 1) \end{aligned}$$

is applied to $(\mu[b], \dots, \mu[w])$. Since

$$2^n - 1 = 2^{n-1} + 2^{n-2} + \dots + 2 + 1$$

those replacements are valid and because no other digits are changed by the transformation, $(\delta[n], \dots, \delta[0])$ is indeed a signed binary representation of d. □

Lemma 6.9 also shows, that the generation of the ltrJSF doesn't generate any carry bit. Hence, the transformation explained above can be applied successively from left-to-right without further considerations.

Example 6.10. *Let* $(1, \bar{1}, 1, 0, \bar{1}, 1, 0, \bar{1}, 1, 0, \bar{1})$ *be the MOF of* $d = 731$*. Here,* $l = 0$ *holds and the following table shows how the transformation explained in Lemma 6.9 can be applied successively. The entry* $d[b]$ *is marked bold.*

$(\delta[10], \ldots, \delta[0])$	b	w
$(\mathbf{1}, \bar{1}, 1, 0, \bar{1}, 1, 0, \bar{1}, 1, 0, \bar{1})$	10	9
$(0, 1, 1, 0, \mathbf{\bar{1}}, 1, 0, \bar{1}, 1, 0, \bar{1})$	6	5
$(0, 1, 1, 0, 0, \bar{1}, 0, \bar{1}, \mathbf{1}, 0, \bar{1})$	2	0
$(0, 1, 1, 0, 0, \bar{1}, 0, \bar{1}, 0, 1, 1)$		

Here, the values for b *were chosen arbitrarily. For example, it is also possible to use* $b = 8$ *in the second step, which yields* $(0, 1, 0, 1, 1, 1, 0, \bar{1}, 1, 0, \bar{1})$*.*

The algorithm to generate the ltrJSF of k scalars works as follows. Consider the matrix

$$\begin{pmatrix} \mu_1[n] & \mu_1[n-1] & \ldots & \mu_1[0] \\ \mu_2[n] & \mu_2[n-1] & \ldots & \mu_2[0] \\ \vdots & \vdots & & \vdots \\ \mu_k[n] & \mu_k[n-1] & \ldots & \mu_k[0] \end{pmatrix}$$

which consists of the MOF's of the scalars. The columns of this matrix are denoted by $C_n, C_{n-1}, \ldots, C_0$. Further, an index a is required which denotes the current column. At first, a is set to n.

The algorithm starts scanning at the a-th column, until it finds the smallest block of columns $C_a, \ldots, C_r$ such that a conversion as explained in lemma 6.9 can be applied to all rows $(\mu_j[a], \ldots, \mu_j[r])$, $j = 1, \ldots, k$ for an $b \in \{r, \ldots, a\}$. Such blocks are called *convertible blocks*. In other words, the algorithm searches for the largest r which satisfies

1. $r \leq a$
2. There exists an $b \in \{r, \ldots, a\}$ such that for $j = 1, \ldots k$ either
   a) $\mu_j[b] = 0$ or
   b) there exists an $w_j \in \{l_j, \ldots, b-1\}$, such that $\mu_j[w_j] = -\mu_j[b]$ and $\mu_j[i] = 0$, for $i = w_j + 1, \ldots, b-1$.

Here, $l_j \in \{r, \ldots a\}$ is again the index of the least significant non-zero digit in the window $(\mu_j[a], \ldots, \mu_j[r])$, meaning that $\mu_j[l_j] \neq 0$ and $\mu_j[i] = 0$ for $i = r, \ldots, l_j - 1$, $j = 1, \ldots, k$.

Then, the transformation explained in Lemma 6.9 is applied to all rows and the b-th column becomes a zero column. Next, the algorithm sets $a \leftarrow r - 1$ and continues the scan.

According to Lemma 6.9, the value of b can be any value in the set

$$Z := \{r, \ldots, a\} \setminus \{l_1, \ldots, l_k\},$$

where l_j is set to -1, if the row $(\mu_j[a], \ldots, \mu_j[r])$ is all zero. If Z contains more than one element, b is chosen as large as possible. Hence, a convertible block is the smallest block of columns such that $Z \neq \emptyset$ holds. Algorithm 13 shows an implementation of this strategy.

Algorithm 13 MOF to ltrJSF

Require: k n-bit scalars d_j in their MOF $(\mu_j[n], \mu_j[n-1], \ldots, \mu_j[0])$, $j = 1, \ldots, k$.
Ensure: The ltrJSF $(\delta_j[n], \delta_j[n-1], ..., \delta_j[0])$ of the k scalars, $j = 1, \ldots, k$.

1: $a \leftarrow n$
2: **while** $a \geq 0$ **do**
3: $\quad r \leftarrow a$
4: $\quad Z \leftarrow \emptyset$
5: $\quad$ **while** $Z = \emptyset \wedge r \geq 0$ **do**
6: $\quad\quad Z \leftarrow \{r, \ldots, a\} \setminus \{l_1, \ldots, l_k\}$
7: $\quad\quad$ **if** $Z \neq \emptyset$ **then**
8: $\quad\quad\quad b \leftarrow \max\{z : z \in Z\}$
9: $\quad\quad\quad$ **for** $j = 1$ to k **do**
10: $\quad\quad\quad\quad$ **if** $\mu_j[b] = 0$ **then**
11: $\quad\quad\quad\quad\quad (\delta_j[a], \ldots, \delta_j[r]) \leftarrow (\mu_j[a], \ldots, \mu_j[r])$
12: $\quad\quad\quad\quad$ **else**
13: $\quad\quad\quad\quad\quad (\delta_j[a], \ldots, \delta_j[b+1]) \leftarrow (\mu_j[a], \ldots, \mu_j[b+1])$
14: $\quad\quad\quad\quad\quad (\delta_j[b], \delta_j[b-1], \ldots, \delta_j[w_j]) \leftarrow (0, \overline{\mu_j[b]}, \ldots, \overline{\mu_j[b]})$
15: $\quad\quad\quad\quad\quad (\delta_j[w_j-1], \ldots, \delta_j[r]) \leftarrow (\mu_j[w_j-1], \ldots, \mu_j[r])$
16: $\quad\quad\quad\quad$ **end if**
17: $\quad\quad\quad$ **end for**
18: $\quad\quad$ **else**
19: $\quad\quad\quad r \leftarrow r - 1$
20: $\quad\quad$ **end if**
21: $\quad$ **end while**
22: $\quad a \leftarrow r - 1$
23: **end while**
24: **return** $(\delta_j[n], \delta_j[n-1], ..., \delta_j[0])$.

Example 6.11. *This example shows how Algorithm 13 generates the ltrJSF of the four scalars* $d_1 = 2716, d_2 = 801, d_3 = 3742$ *and* $d_4 = 3395$. *The convertible blocks are marked bold.*

$$\begin{pmatrix} \mathbf{1} & \mathbf{\bar{1}} & \mathbf{1} & \bar{1} & 1 & \bar{1} & 0 & 1 & 0 & 0 & \bar{1} & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & 0 & \bar{1} & 0 & 1 & \bar{1} & 0 & 0 & 0 & 1 & \bar{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \bar{1} & 1 & \bar{1} & 0 & 1 & 0 & 0 & 0 & \bar{1} & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{\bar{1}} & 1 & \bar{1} & 1 & \bar{1} & 0 & 0 & 0 & 1 & 0 & \bar{1} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & \bar{1} & \mathbf{\bar{1}} & \mathbf{1} & \bar{1} & 0 & 1 & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & \mathbf{0} & \mathbf{\bar{1}} & 0 & 1 & \bar{1} & 0 & 0 & 0 & 1 & \bar{1} \\ 1 & 0 & 0 & \mathbf{\bar{1}} & \mathbf{1} & \bar{1} & 0 & 1 & 0 & 0 & 0 & \bar{1} & 0 \\ 1 & 0 & \bar{1} & \mathbf{1} & \mathbf{\bar{1}} & 1 & \bar{1} & 0 & 0 & 0 & 1 & 0 & \bar{1} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & \bar{1} & 0 & \bar{1} & \mathbf{\bar{1}} & \mathbf{0} & \mathbf{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & \mathbf{0} & \mathbf{1} & \mathbf{\bar{1}} & 0 & 0 & 0 & 1 & \bar{1} \\ 1 & 0 & 0 & 0 & \bar{1} & \mathbf{\bar{1}} & \mathbf{0} & \mathbf{1} & 0 & 0 & 0 & \bar{1} & 0 \\ 1 & 0 & \bar{1} & 0 & 1 & \mathbf{1} & \mathbf{\bar{1}} & \mathbf{0} & 0 & 0 & 1 & 0 & \bar{1} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & \bar{1} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & \mathbf{0} & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & 0 & 1 & \bar{1} & \mathbf{0} & 0 & 0 & 1 & \bar{1} \\ 1 & 0 & 0 & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & \mathbf{0} & 0 & 0 & \bar{1} & 0 \\ 1 & 0 & \bar{1} & 0 & 1 & 0 & 1 & 0 & \mathbf{0} & 0 & 1 & 0 & \bar{1} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & \bar{1} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & \mathbf{0} & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & 0 & 1 & \bar{1} & 0 & \mathbf{0} & 0 & 1 & \bar{1} \\ 1 & 0 & 0 & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & \mathbf{0} & 0 & \bar{1} & 0 \\ 1 & 0 & \bar{1} & 0 & 1 & 0 & 1 & 0 & 0 & \mathbf{0} & 1 & 0 & \bar{1} \end{pmatrix}$$

After applying the algorithm, the ltrJSF of the scalars $d_1, \ldots, d_4$ *is given as*

$$\begin{pmatrix} 1 & 0 & \bar{1} & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & \bar{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & 0 & 1 & \bar{1} & 0 & 0 & 0 & 1 & \bar{1} \\ 1 & 0 & 0 & 0 & \bar{1} & 0 & \bar{1} & \bar{1} & 0 & 0 & 0 & \bar{1} & 0 \\ 1 & 0 & \bar{1} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \bar{1} \end{pmatrix}$$

In [HKPR04], the authors also proved that the JHW of k scalars given in their ltrJSF is minimal. This implies, that the AJHD of the ltrJSF is minimal amongst all $\mathcal{D}$-representations which use the digit set $\mathcal{D} = \{0, \pm 1\}$. Therefore no other $\mathcal{D}$-representation using this digit set provides fewer non-zero columns than the ltrJSF. This leads to the following theorem.

Theorem 6.12. *The AJHD of the ltrJSF is*

$$\mathcal{AJHD}_k(\text{ltrJSF}) = 1 - \frac{1}{c_k},$$

where c_k is given by the recursive formula

$$c_k = \frac{1}{2^k}\left(3 + \sum_{j=1}^{k-1} \binom{k}{j}(c_j + 1)\right)$$

and $c_1 = 1.5$.

Furthermore, in [DOT05a], the authors proposed an algorithm to compute the AJHD of the ltrJSF for certain values of k. Table 6.1 shows some values computed by this algorithm which underline the correctness of Theorem 6.12.

k	$\mathcal{AJHD}_k(\text{ltrJSF})$	
1	1/3	≈ 0.3333
2	1/2	$= 0.5000$
3	23/39	≈ 0.5897
4	115/179	≈ 0.6424
5	4279/6327	≈ 0.6763
6	152821/218357	≈ 0.6998

Table 6.1: Example values of $\mathcal{AJHD}_k(\text{ltrJSF})$

According to this table, the ltrJSF of one scalar has the same AHD as the 2MOF. In fact, the output of both algorithms is exactly the same.

7 Computing a Multi-Scalar Multiplication

In Chapter 4, two efficient algorithms to compute a multi-scalar multiplication were introduced, namely the Interleave method and the Shamir method. It turned out, that those algorithms can be sped up by deploying $\mathcal{D}$-representations of the scalars which provide a low AHD or AJHD. Such $\mathcal{D}$-representations were introduced in Chapters 5 and 6.

The purpose of this chapter is to examine, in what way those $\mathcal{D}$-representations can speed up the computation of a multi-scalar multiplication compared to the binary representation. Further, the $\mathcal{D}$-representations are compared based on the required number of ECADD operations and the number of points to precompute.

7.1 Speeding up the Interleave Method

According to Section 4.2, the Interleave method on average requires

$$n \text{ ECDBL} + n \cdot k \cdot \mathcal{AHD}(\mathcal{X}) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, where the scalars are represented in the $\mathcal{D}$-representation $\mathcal{X}$. This implies, that the required number of ECADD operations can be reduced by applying the scalars in a $\mathcal{D}$-representation with a low AHD. Further, the Interleave method requires the precomputation of

$$k \cdot \frac{(|\mathcal{D}| - 1)}{2} - k$$

points if the scalars are represented in a signed representation and

$$k \cdot (|\mathcal{D}| - 1) - k$$

points otherwise, as explained in Section 4.4.

The binary representation uses the digit set $\mathcal{D} = \{0, 1\}$ and provides an AHD of $1/2$. Therefore, no points must be precomputed and the Interleave method on average requires

$$n \text{ ECDBL} + n \cdot k \cdot \frac{1}{2} \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, if the scalars are represented in the binary representation.

$\mathcal{D}$-representations which specifically aim on minimizing the AHD of the scalars were introduced in Sections 5.1 and 6.2, namely the wNAF and the wMOF. Both representations use the digit set

$$\mathcal{D}_w = \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\},$$

which consists of $2^{w-1} + 1$ elements, and provide the same AHD, namely

$$\mathcal{AHD}(w\text{NAF}) = \mathcal{AHD}(w\text{MOF}) = \frac{1}{w+1}.$$

Therefore, both $\mathcal{D}$-representations decrease the number of ECADD operations required by the Interleave method in the same way. For a given w, the Interleave method on average requires

$$n \text{ ECDBL} + n \cdot k \cdot \frac{1}{w+1} \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, if the scalars are represented in either the wNAF or the wMOF. Further, the precomputation of

$$k \cdot (2^{w-2} - 1)$$

points is required, since both the wNAF and the wMOF are signed representations. Note, that in the case $w = 2$, no points have to be precomputed. The points to precompute if $w \geq 3$ are all points of the form tP_j, $t \in \mathcal{D}_w$, $t > 1$, $j = 1, \ldots, k$, namely

$$\begin{gathered} 3P_1, 5P_1, \ldots, (2^{w-1} - 1)P_1 \\ 3P_2, 5P_2, \ldots, (2^{w-1} - 1)P_2 \\ \vdots \\ 3P_k, 5P_k, \ldots, (2^{w-1} - 1)P_k \end{gathered}$$

The difference between the wNAF and the wMOF is the direction in which they are recoded. The wNAF is recoded from right-to-left, while the wMOF is recoded from left-to-right. Since the Interleave method is applied to the scalars from left-to-right, it is inevitable to perform the recoding in a separate stage if the scalars are to be represented in their wNAF. Therefore, the whole recoded scalars must be stored in memory, which requires memory of the order of magnitude of $k \cdot n$ bits.

If the scalars are represented in their wMOF, it is possible to perform the recoding on-the-fly during the evaluation. In other words, the scalars are only

recoded as much as required at once. The algorithm to generate the wMOF, Algorithm 12, outputs at most w digits of the recored scalar at once. Therefore, only memory of the order of magnitude of $k \cdot w$ bits is required in total, which is very small compared to $k \cdot n$.

This significant saving of memory is the great advantage of the wMOF over the wNAF and also the reason why the wMOF should be favored.

7.2 Speeding up the Shamir Method

According to Section 4.3, the Shamir method on average requires

$$n \text{ ECDBL} + n \cdot \mathcal{AJHD}_k(\mathcal{X}) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, where the scalars are represented in the $\mathcal{D}$-representation $\mathcal{X}$. This implies that the required number of ECADD operations can be reduced by applying the scalars in a $\mathcal{D}$-representation with a low AJHD. Further, the Shamir method requires the precomputation of

$$\frac{|\mathcal{D}|^k - 1}{2} - k$$

points if the scalars are represented in a signed representation and

$$|\mathcal{D}|^k - 1 - k$$

points otherwise, as explained in Section 4.4.

The binary representation uses the digit set $\mathcal{D} = \{0, 1\}$ and provides a AJHD of $1 - 1/2^k$. Therefore, the Shamir method on average requires

$$n \text{ ECDBL} + n \cdot \left(1 - \frac{1}{2^k}\right) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, if the scalars are represented in the binary representation. Further, the precomputation of

$$2^k - 1 - k$$

points is required. Those points are all points of the form $t_1 P_1 + \ldots + t_k P_k$, where $t_1, \ldots, t_k \in \{0, 1\}^k$, such that at least two of the t_j are non-zero. For example, if $k = 3$ holds, the four points to precompute are

$$P_1 + P_2, P_1 + P_3, P_2 + P_3, P_1 + P_2 + P_3.$$

$\mathcal{D}$-representations which specifically aim on minimizing the AJHD of the scalars were introduced in Sections 5.2 and 6.3, namely the JSF and the ltrJSF. Both representations use the digit set

$$\mathcal{D} = \{0, \pm 1\},$$

which consists of 3 elements, and provide the same AJHD, namely

$$\mathcal{AJHD}_k(\text{JSF}) = \mathcal{AJHD}_k(\text{ltrJSF}) = 1 - \frac{1}{c_k},$$

where

$$c_k = \frac{1}{2^k}\left(3 + \sum_{j=1}^{k-1}\binom{k}{j}(c_j + 1)\right)$$

and $c_1 = 1.5$. Therefore, both $\mathcal{D}$-representations decrease the number of ECADD operations required by the Shamir method in the same way. On average, the Shamir method requires

$$n \text{ ECDBL} + n \cdot \left(1 - \frac{1}{c_k}\right) \text{ ECADD}$$

operations to compute a multi-scalar multiplication $\sum_{j=1}^{k} d_j P_j$, if the scalars are represented in either the JSF or the ltrJSF. Further, the precomputation of

$$\frac{3^k - 1}{2} - k$$

points is required, since both the JSF and the ltrJSF are signed binary representations. Those points are all points of the form $t_1P_1 + \ldots + t_kP_k$, where $t_1, \ldots, t_k \in \{0, \pm 1\}^k$, such that at least two of the t_j are non-zero and the leftmost non-zero entry equals 1. For example, if $k = 3$ holds, the ten points to precompute are

$$P_1 + P_2, P_1 - P_2, P_1 + P_3, P_1 - P_3, P_2 + P_3, P_2 - P_3,$$
$$P_1 + P_2 + P_3, P_1 - P_2 + P_3, P_1 + P_2 - P_3, P_1 - P_2 - P_3.$$

The difference between the JSF and the ltrJSF is the direction in which they are recoded. The JSF is recoded from right-to-left, while the ltrJSF is recoded from left-to-right. Since the Shamir method is applied to the scalars from left-to-right, it is inevitable to perform the recoding in a separate stage if the scalars are to be represented in their JSF. Therefore, the whole recoded scalars must be stored in memory, which requires memory of the order of magnitude of $k \cdot n$ bits.

If the scalars are represented in their ltrJSF, it is possible to perform the recoding on-the-fly during the evaluation. In other words, the scalars are only recoded as much as required at once. The algorithm to generate the ltrJSF, Algorithm 13, outputs at most $(k + 1)$ digits of the recored scalar at once. Therefore, only memory of the order of magnitude of $k \cdot (k + 1)$ bits is required in total, which is very small compared to $k \cdot n$.

This significant saving of memory is the great advantage of the ltrJSF over the JSF and also the reason why the ltrJSF should be favored.

7.3 Comparison

This section compares the $\mathcal{D}$-representations introduced so far. The comparison is based on the average number of ECADD operations required and the number points which have to be precomputed to compute a multi-scalar multiplication with the Interleave method and the Shamir method. In this section, the scalars are assumed to be 160-bit.

Table 7.1 shows those values for the Interleave method were the scalars are represented in the binary representation and the wMOF for different values of w. Those values are visualized in Figures 7.1 and 7.2 for different values for k.

Table 7.2 shows those values for the Shamir method were the scalars are represented in the binary representation and the ltrJSF for different values of k. Those values are visualized in Figures 7.3 and 7.4.

$\mathcal{D}$-representation	avg. # ECADD	# points
Binary	$80.00k$	0
2MOF	$53.33k$	0
3MOF	$40.00k$	k
4MOF	$32.00k$	$3k$
5MOF	$26.67k$	$7k$
6MOF	$22.86k$	$15k$

Table 7.1: Costs for the Interleave method

	avg. # ECADD		# points	
k	ltrJSF	Binary	ltrJSF	Binary
1	53.33	80.00	0	0
2	80.00	120.00	2	1
3	94.39	140.00	10	4
4	102.79	150.00	36	11
5	108.21	155.00	116	26
6	111.98	157.50	358	57

Table 7.2: Costs for the Shamir method

The first thing that draws attention when examining Tables 7.1 and 7.2 is, that compared to the binary representation, both the wMOF and the ltrJSF significantly decrease the average number of ECADD operations required. Those tables also show, that decreasing the number of ECADD operations always results in an increased number of points which have to be precomputed. Thus,

there is always a trade-off between the speed for the evaluation and the number of points which have to be precomputed.

In the case of the wMOF, increasing the value of w further decreases the average number of ECADD operations required. While this reduction gets very small if w gets large, the number of points which have to be precomputed grows exponentially with the value of w. Therefore, increasing the value of w does not automatically yields a better total performance of the multi-scalar multiplication, since additional ECADD and ECDBL operations are required for the precomputation. Also, the number of points to precompute and the required number of ECADD operations increase linear with the number of scalars.

In the case of the ltrJSF, the number of ECADD operations required does not increase linear with the number of scalars k, but converges against 160 for $k \to \infty$. This is because the AJHD converges against 1, and therefore at most n ECADD operations are required. The same holds for the binary representation. For both representations, the number of points to precompute increases exponential with the number of scalars.

It is also interesting, that in the case where the number of points to precompute is the same, the wMOF and the ltrJSF decrease the required number of ECADD operations in the same way. In the case where $k = 1$, no points have to be precomputed if the scalar is represented in the 2MOF or the ltrJSF. Here, the Interleave method as well as the Shamir method on average require 53.33 ECADD operations. In the case where $k = 2$, two points must be precomputed if the scalars are represented in the 3MOF or the ltrJSF. Those points are $3P_1, 3P_2$ and $P_1 + P_2, P_1 - P_2$, respectively. Here, both methods on average require 80 ECADD operations.

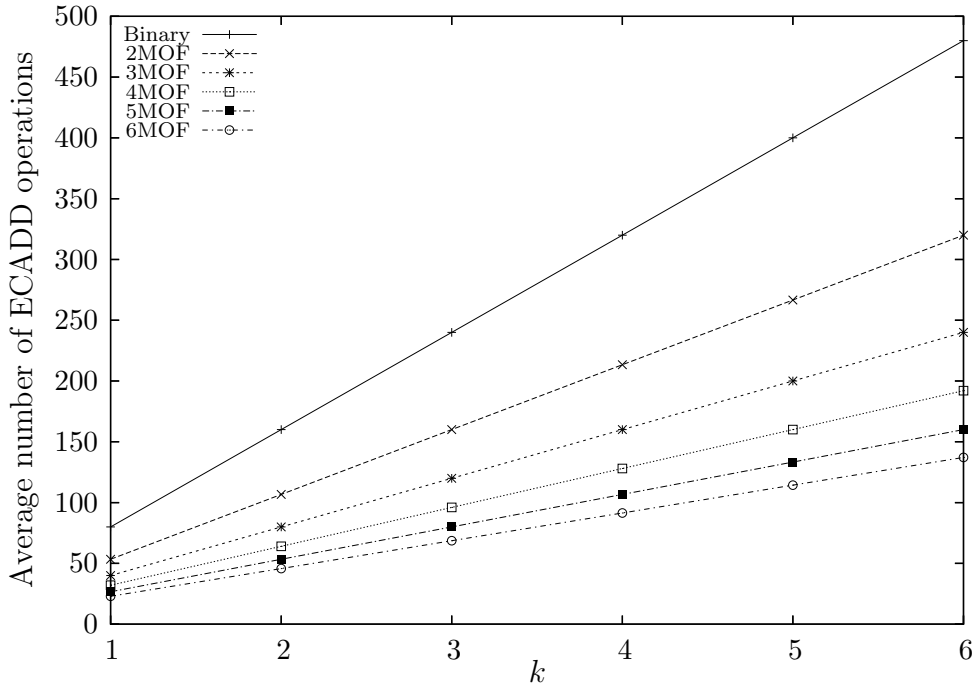


Figure 7.1: ECADD operations required by the Interleave method

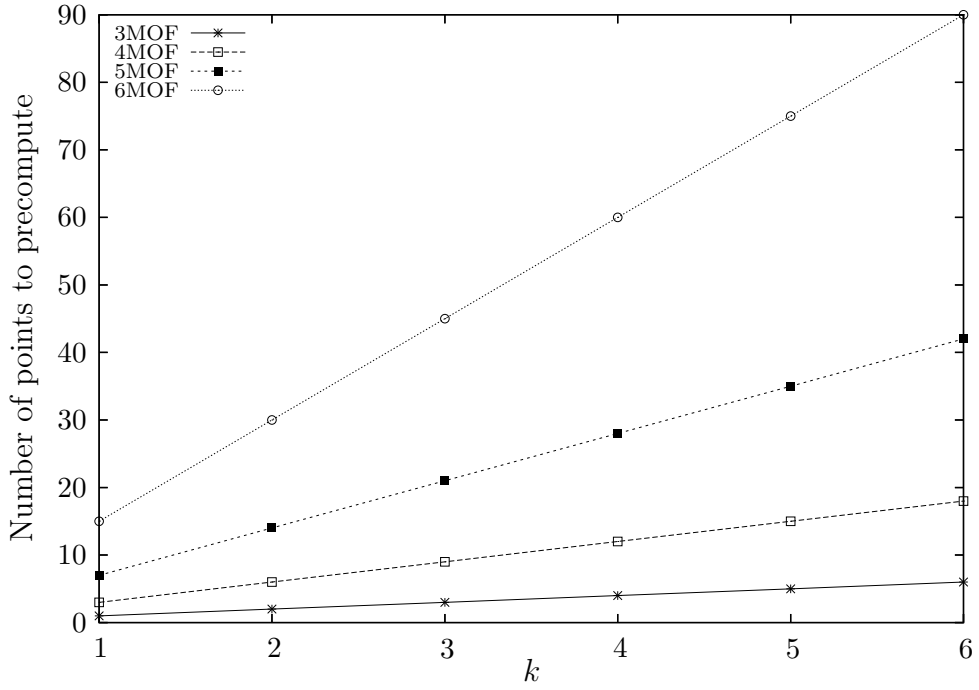


Figure 7.2: Points to precompute for the Interleave method

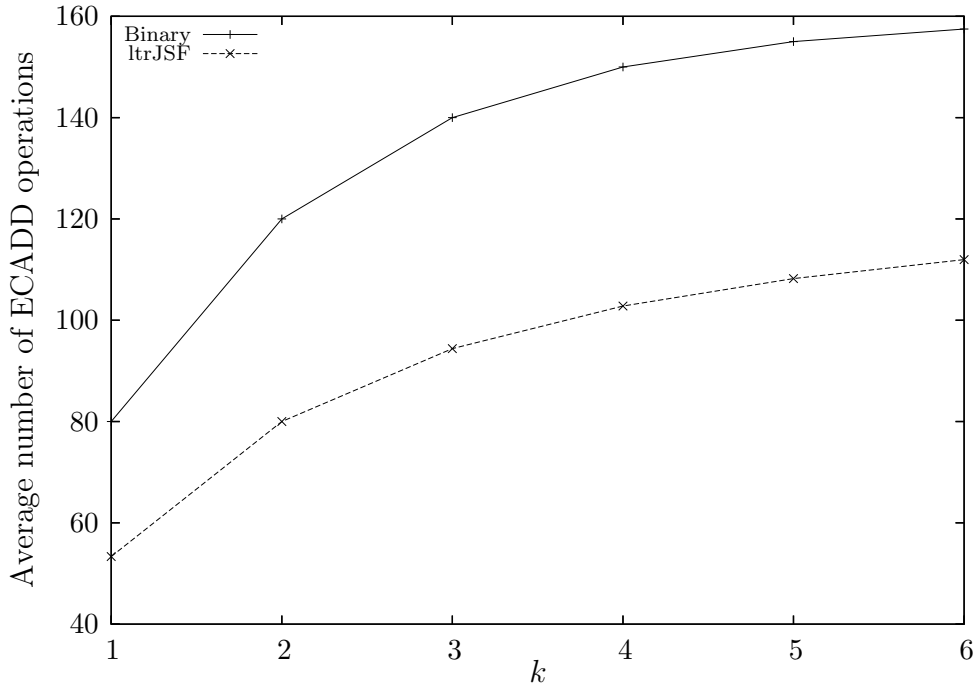


Figure 7.3: ECADD operations required by the Shamir method

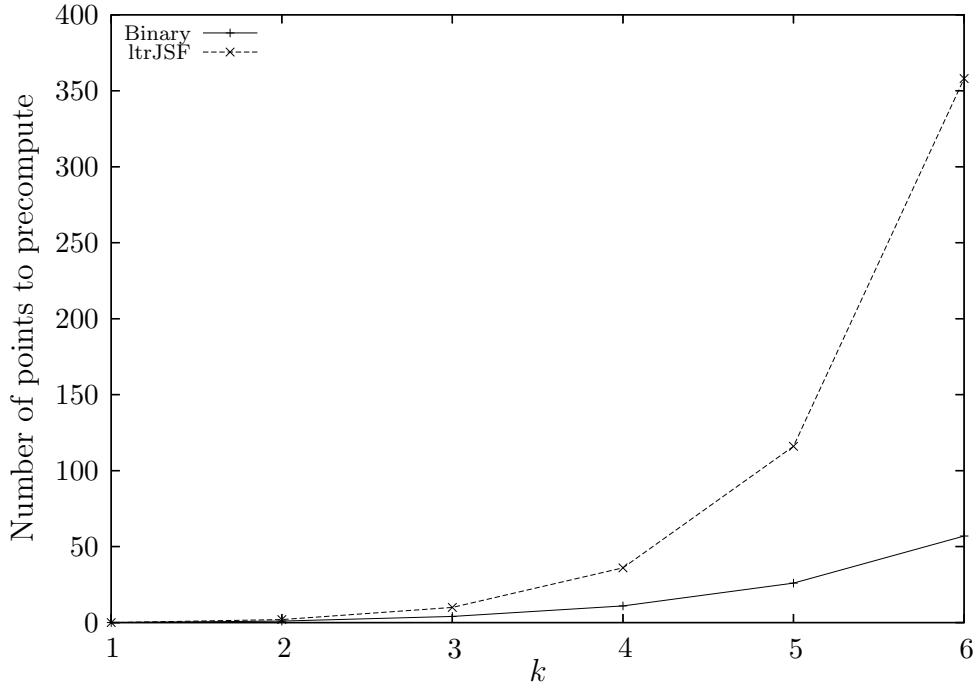


Figure 7.4: Points to precompute for the Shamir method

8 Field Operations

According to the last chapter, the computation of a multi-scalar multiplication is done in two stages. Those stages are

Precomputation Stage Precompute the required points and store them in memory.

Evaluation Stage For the evaluation stage, two methods exist.

Interleave Method Evaluate the multi-scalar multiplication by applying Algorithm 8. If the scalars are to be represented in the wMOF, call Algorithm 12 to partially recode the scalars as required.

Shamir Method Evaluate the multi-scalar multiplication by applying Algorithm 9. If the scalars are to be represented in the ltrJSF, call Algorithm 13 to partially recode the scalars as required.

In the last chapter, the number of ECADD and ECDBL operations required for the evaluation stage as well as the number of points to precompute during the precomputation stage were estimated.

In order to decide which method offers the best total performance, it is necessary to estimate the number of field operations required by those stages. In other words, it has to be considered how the required ECADD and ECDBL operations are actually computed. As explained in Section 2.2, the number of field multiplications, squarings and inversions required for an ECADD or ECDBL operation depends on the coordinate system used to represent the points.

Section 8.1 estimates the number of field operations required for the evaluation stage, where the points are represented using mixed coordinates as proposed in [CMO98].

Section 8.2, at first introduces a straight forward method for the precomputation stage of both the Interleave and the Shamir method. Then, the required number of field operations is estimated.

Finally, Section 8.3 states the total costs required to compute a multi-scalar multiplication and compares the introduced $\mathcal{D}$-representations based on those values.

Throughout this section, the ratio of squarings and multiplications S/M is set to $S = 0.8M$ and the ratio of inversions and multiplications I/M is set to $I = 30M$. Further, the scalars are assumed to be 160-bit.

8.1 Evaluation Stage

To estimate the required number of field operations, it is at first necessary to divide the evaluation stage of the Interleave method and the Shamir method into four steps. Those steps are distinguished by the current operation and the succeeding operation. Hence, there are four possibilities:

DD An ECDBL operation is followed by an ECDBL operation.

This happens, if the current column is all zero.

DA An ECDBL operation is followed by an ECADD operation.

This happens, if the current column is non-zero.

AA An ECADD operation is followed by an ECADD operation.

This step occurs only in the Interleave method, namely if the current column is non-zero and the entry which is currently examined is not the last non-zero entry in this column.

AD An ECADD operation is followed by an ECDBL operation.

In the case of the Interleave method, this happens if the current column is non-zero and the entry which is currently examined is the last non-zero entry in this column.

In the case of the Shamir method, this happens if the current column is non-zero. Here, an ECADD operation is always followed by an ECDBL operation, since only one ECADD operation is performed for each non-zero column.

Example 8.1. *Consider* $d_1 = 18$ *and* $d_2 = 3$ *with binary representations* $(1,0,0,1,0)$ *and* $(0,0,0,1,1)$*, respectively. While computing* $d_1P_1+d_2P_2$*, the sequences of the four steps (DD, DA, AA, AD) introduced above for both methods are:*

Interleave method					Shamir method				
1	0	0	1	0	1	0	0	1	0
0	0	0	1	1	0	0	0	1	1
DA	*DD*	*DD*	*DA*	*DA*	*DA*	*DD*	*DD*	*DA*	*DA*
AD			*AA*	*AD*	*AD*			*AD*	*AD*
			AD						

In [CMO98], the authors proposed an optimal choice for the coordinate system to use for each step as shown in Table 8.1. Here, the same notation as in Section 2.2.4 is used to denote the chosen coordinate system. Further, $\mathcal{A}$, $\mathcal{J}$ and $\mathcal{J}^m$ stands for Affine, Jacobian and modified Jacobian coordinates, respectively.

Step	Coordinates			Costs		
DD	$2\mathcal{J}^m$	$\rightarrow$	$\mathcal{J}^m$	$4M + 4S$	$=$	$7.2M$
DA	$2\mathcal{J}^m$	$\rightarrow$	$\mathcal{J}$	$3M + 4S$	$=$	$6.2M$
AD	$\mathcal{J} + \mathcal{A}$	$\rightarrow$	$\mathcal{J}^m$	$9M + 5S$	$=$	$13M$
AA	$\mathcal{J}^m + \mathcal{A}$	$\rightarrow$	$\mathcal{J}^m$	$9M + 5S$	$=$	$13M$

Table 8.1: Coordinate systems for the evaluation stage

This explains also, why the precomputed points should be represented in Affine coordinates as mentioned in Section 4.1.3. The next step is to estimate, how often those four steps on average occur during the evaluation of a multi-scalar multiplication. Here it is assumed that the scalars are represented in a $\mathcal{D}$-representation $\mathcal{X}$.

In the case of the Interleave method, the probability that the current column is all zero is given as $(1 - \mathcal{AHD}(\mathcal{X}))^k$. This implies, that the current column is non-zero with probability $1 - (1 - \mathcal{AHD}(\mathcal{X}))^k$. This gives the probabilities for the DD and DA steps, respectively. In the case of the ECADD operations, note that the steps AD and AA have the same costs and can therefore be considered equal. The probability that either of those steps occurs is given as $\mathcal{AHD}(\mathcal{X})$. Therefore, the average costs of the evaluation stage of the Interleave method in terms of the four steps is given as

$$n\left(1 - \mathcal{AHD}(\mathcal{X})\right)^k \text{ DD} + n\left(1 - \left(1 - \mathcal{AHD}(\mathcal{X})\right)^k\right) \text{ DA} + n \cdot k \cdot \mathcal{AHD}(\mathcal{X}) \text{ AD}$$

The case of the Shamir method is less complicated. Here, the probability that the current column is all zero is given as $1 - \mathcal{AJHD}_k(\mathcal{X})$. Therefore, the probability that the current column is non-zero is given as $\mathcal{AJHD}_k(\mathcal{X})$. An AD step occurs every time the current column is non-zero, hence with probability $\mathcal{AJHD}_k(\mathcal{X})$. Therefore, the average costs of the evaluation stage of the Shamir method in terms of the four steps is given as

$$n(1 - \mathcal{AJHD}_k(\mathcal{X})) \text{ DD} + n \cdot \mathcal{AJHD}_k(\mathcal{X}) \text{ DA} + n \cdot \mathcal{AJHD}_k(\mathcal{X}) \text{ AD}$$

Table 8.2 shows the average number of field multiplications required by the Interleave method and the Shamir method for different values of k and different representations of the scalars. These numbers were obtained by substituting the respective AHD and AJHD as well as the costs for the four steps according to Table 8.1 in the above formulas.

k	1	2	3	4	5	6
Int. + Binary	2112.00	3112.00	4132.00	5162.00	6197.00	7234.50
Int. + 2MOF	1792.00	2449.78	3119.41	3796.94	4479.74	5166.05
Int. + 3MOF	1632.00	2122.00	2619.50	3122.63	3629.97	4140.49
Int. + 4MOF	1536.00	1926.40	2321.92	2721.54	3124.43	3529.94
Int. + 5MOF	1472.00	1796.44	2124.59	2455.83	2789.63	3125.58
Int. + 6MOF	1426.29	1703.84	1984.19	2266.94	2551.74	2838.31
Sha. + Binary	2112.00	2592.00	2832.00	2952.00	3012.00	3042.00
Sha. + ltrJSF	1792.00	2112.00	2284.31	2385.52	2450.51	2495.75

Table 8.2: Field multiplications for the evaluation stage

The differences between the compared methods shown by this table are similar to what was shown by Tables 7.1 and 7.2.

Interesting is, that although both the ltrJSF and the 3MOF require the same amount of ECADD operations in the case of $k = 2$, the ltrJSF requires less field multiplications. This is because the more expensive DD step occurs more frequent when using the 3MOF.

8.2 Precomputation Stage

Before the number of field operations required for the precomputation stage can be estimated, it is at first necessary to estimate the number of ECADD and ECDBL operations required to precompute those points. In other words, an explicit method for the precomputation stage is required.

Recall, that the Interleave method requires no precomputation if the digit set is given as $\mathcal{D} = \{0, 1\}$. In the case of a digit set of the form $\mathcal{D}_w = \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$, the precomputation of $k \cdot (2^{w-2} - 1)$ points is required by the Interleave method. Those are all points of the form tP_j, where $t \in \mathcal{D}_w$ and $t > 1$, $j = 1, \ldots, k$. The precomputation is done using the chain:

$$\begin{array}{ccccccccccc}
P_1 & \rightarrow & 2P_1 & \rightarrow & 3P_1 & \rightarrow & 5P_1 & \rightarrow & \ldots & \rightarrow & (2^{w-1}-1)P_1 \\
P_2 & \rightarrow & 2P_2 & \rightarrow & 3P_2 & \rightarrow & 5P_2 & \rightarrow & \ldots & \rightarrow & (2^{w-1}-1)P_2 \\
 & & & & \vdots & & & & & & \\
P_k & \rightarrow & 2P_k & \rightarrow & 3P_k & \rightarrow & 5P_k & \rightarrow & \ldots & \rightarrow & (2^{w-1}-1)P_k
\end{array}$$

where $(2i + 1)P_j$ is computed as $(2i - 1)P_j + 2P_j$. Hence,

$$k \text{ ECDBL} + k \cdot (2^{w-2} - 1) \text{ ECADD}$$

operations are required in total.

In the case of the Shamir method and the digit set $\mathcal{D} = \{0, 1\}$, the precomputation of $2^k - 1 - k$ points is required. Those are all points of the form $t_1P_1 + \ldots + t_kP_k$, where $t_1, \ldots, t_k \in \{0, 1\}^k$, such that at least two of the t_j are non-zero. This is done using the chain:

$$\begin{array}{lll} & P_{j_1} + P_{j_2}, & j_1, j_2 = 1, \ldots, k : j_1 < j_2 \\ \rightarrow & P_{j_1} + P_{j_2} + P_{j_3}, & j_1, j_2, j_3 = 1, \ldots, k : j_1 < j_2 < j_3 \\ & \vdots & \\ \rightarrow & P_{j_1} + P_{j_2} + \ldots + P_{j_k}, & j_1, j_2, \ldots, j_k = 1, \ldots, k : j_1 < j_2 < \ldots < j_k \end{array}$$

In the first step, $\binom{k}{2}$ points are computed and the same amount of ECADD operations is required. In the second step $\binom{k}{3}$ points are computed also by using only one ECADD operation per point. This is because two of the required summands were already added in the first step. In general, each point in each step can be computed with only one ECADD operation by reusing the points computed in the preceding step. Hence,

$$\sum_{j=2}^{k} \binom{k}{j} \text{ ECADD} = 2^k - 1 - k \text{ ECADD}$$

operations are required in total.

If $\mathcal{D} = \{0, \pm 1\}$, the precomputation of $(3^k - 1)/2 - k$ points is required by the Shamir method. Those are all points of the form $t_1P_1 + \ldots + t_kP_k$, where $t_1, \ldots, t_k \in \{0, \pm 1\}^k$, such that at least two of the t_j are non-zero and the leftmost non-zero entry equals 1. This is done using the chain:

$$\begin{array}{lll} & P_{j_1} \pm P_{j_2}, & j_1, j_2 = 1, \ldots, k : j_1 < j_2 \\ \rightarrow & P_{j_1} \pm P_{j_2} \pm P_{j_3}, & j_1, j_2, j_3 = 1, \ldots, k : j_1 < j_2 < j_3 \\ & \vdots & \\ \rightarrow & P_{j_1} \pm P_{j_2} \pm \ldots \pm P_{j_k}, & j_1, j_2, \ldots, j_k = 1, \ldots, k : j_1 < j_2 < \ldots < j_k \end{array}$$

Here, $P_{j_1} - P_{j_2}$ is computed as $P_{j_1} + (-P_{j_2})$ and the costs to invert P_{j_2} are neglected, see Section 4.4. In the first step, $\binom{k}{2}2$ points are computed, each with one ECADD operation. In the second step $\binom{k}{3}2^2$ points are computed. Here, also only one ECADD operation is required for each point, since two of the required summands were already added in the first step. In the j-th step $\binom{k}{j}2^{j-1}$ points are computed, each by using only one ECADD operation, since the points computed in the preceding step can be reused. Hence,

$$\sum_{j=2}^{k} \binom{k}{j} 2^{j-1} \text{ ECADD} = \frac{3^k - 1}{2} - k \text{ ECADD}$$

operations are required in total.

Because of the coordinate systems chosen for the evaluation stage in the last section, the precomputed points must be represented in Affine coordinates. Table 8.3 shows the number of field operations required for the ECADD and ECDBL operations of the precomputation stage as explained in Section 2.2.

Operation	Coordinates			Costs		
ECDBL	$2\mathcal{A}$	$\rightarrow$	$\mathcal{A}$	$2M + 2S + I$	$=$	$33.6M$
ECADD	$\mathcal{A} + \mathcal{A}$	$\rightarrow$	$\mathcal{A}$	$2M + S + I$	$=$	$32.8M$

Table 8.3: Coordinate systems for the precomputation stage

Table 8.4 shows the number of field multiplications required to precompute the points for different values of k and different representations of the scalars. These numbers were obtained by substituting the costs shown in Table 8.3 in the above formulas.

k	1	2	3	4	5	6
Int. + Binary	0.0	0.0	0.0	0.0	0.0	0.0
Int. + 2MOF	0.0	0.0	0.0	0.0	0.0	0.0
Int. + 3MOF	66.4	132.8	199.2	265.6	332.0	398.4
Int. + 4MOF	132.0	264.0	396.0	528.0	660.0	792.0
Int. + 5MOF	263.2	526.4	789.6	1052.8	1316.0	1579.2
Int. + 6MOF	525.6	1051.2	1576.8	2102.4	2628.0	3153.6
Sha. + Binary	0.0	32.8	131.2	360.8	852.8	1869.6
Sha. + ltrJSF	0.0	65.6	328.0	1180.8	3804.8	11742.4

Table 8.4: Field multiplications for the precomputation stage

The differences between the compared methods shown by this table are similar to what was shown by Tables 7.1 and 7.2.

Interesting is, that although both the ltrJSF and the 3MOF require the same amount precomputed points in the case of $k = 2$, the ltrJSF requires less field multiplications to compute them. This is because the precomputation of $3P_1$ and $3P_2$ requires two additional ECDBL operations.

8.3 Total Costs

To estimate the total number of field operations required to compute a multi-scalar multiplication, the number of field operations required for the evaluation stage and the precomputation stage obtained in the last two sections have to

be added. The total number of field multiplications required for the different methods is shown in Table 8.5. The smallest and therefore optimal value for each number of scalars is marked bold.

k	1	2	3	4	5	6
Int. + Binary	2112.00	3112.00	4132.00	5162.00	6197.00	7234.50
Int. + 2MOF	1792.00	2449.78	3119.41	3796.94	4479.74	5166.05
Int. + 3MOF	1698.40	2254.80	2818.70	3388.23	3961.97	4538.88
Int. + 4MOF	**1668.0**	2190.40	2717.92	**3249.5**	**3784.4**	**4321.9**
Int. + 5MOF	1735.20	2322.84	2914.19	3508.63	4105.63	4704.78
Int. + 6MOF	1951.89	2755.04	3560.99	4369.34	5179.74	5991.91
Sha + Binary	2112.00	2624.80	2963.20	3312.80	3864.80	4911.60
Sha + ltrJSF	1792.00	**2177.6**	**2612.3**	3566.32	6255.31	14238.15

Table 8.5: Total number of field multiplications

In the case of the Interleave method, this table shows that the wMOF leads to a better performance than the binary representation for any number of scalars and any value of w considered here. This table also shows, that the choice $w = 4$ is optimal for any number of scalars. For larger values of w, the costs for the precomputation stage are too high and for smaller values the evaluation stage is too expensive. Of course, the value $w = 4$ can only be chosen if there is sufficient memory to store the precomputed points. In the case where no memory is available for precomputed points, the choice $w = 2$ still provides a significant improvement compared to the binary representation.

In the case of the Shamir method, the ltrJSF only results in a better performance than the binary representation if $k \leq 3$. After that, the number of points to precompute is too large and the superior evaluation stage of the ltrJSF cannot compensate the immense costs of the precomputation stage. Therefore, the binary representation has a better performance if $k > 3$.

In total, this comparison shows that for $k = 2, 3$ the Shamir method in conjunction with the ltrJSF provides the best performance, while for any other number of scalars the Interleave method in conjunction with the 4MOF is the fastest.

However, the above values only apply in this specific scenario. For example, if the bit length n of the scalars is longer than 160, the evaluation stage becomes more expensive. Then, the value $w = 4$ is not necessarily optimal anymore, since it might be better to precompute more points and in exchange save some ECADD operations in the evaluation stage. Also, the method used to precompute the points and the ratio of multiplications and inversions I/M plays an important role. If the costs for the precomputation stage can be reduced, larger

values than $w = 4$ might become optimal. Of course, it is also possible that the wMOF becomes optimal in the case of $k = 2, 3$ or that the ltrJSF becomes optimal for other values of k.

The scenario also changes if some or all of the points to precompute are already known. For example in the signature generation of the ECDSA (Algorithm 2) the point P is publicly known and doesn't change. Therefore the precomputation stage must be performed only once and the points can be stored in non-volatile memory. In this case, the method that utilizes the available memory best should be chosen.

Summarizing, the task of finding the best method to compute a multi-scalar multiplication in a specific scenario depends on many factors and requires a lot of fine tuning. It was the authors intention to present some values which apply in a very general scenario and give a good impression of the differences of the introduced methods.

9 Conclusion

This thesis presented several measures which can be taken to efficiently implement cryptosystems on smart cards.

As explained in Chapter 1, one of the most critical issues concerning cryptosystems is the security of the secret key which is used for signing and decrypting messages. Due to their tamper resistance and mobility, smart cards are a good choice to serve as host for the secret keys and the cryptosystems. However, since the computational power and the available memory on smart cards is very limited, efficient implementations are needed.

The first measure to reduce the memory and computational power required, is to use cryptosystems that are based on the additive group of points on an elliptic curve, as explained in Chapter 2. The main advantage of elliptic curves over commonly used groups is, that the same level of security can be achieved with much smaller key sizes, i.e. 160-bit instead of 1024-bit.

As it turned out, the most basic operation used in elliptic curve cryptosystems is a multi-scalar multiplication

$$\sum_{j=1}^{k} d_j P_j,$$

where d_j are the scalars and P_j are points on an elliptic curve. The remaining chapters of the thesis dealt with the efficient computation of such multi-scalar multiplications.

In Chapter 4, two basic algorithms for the efficient computation of a multi-scalar multiplication were introduced. Those were the Interleave method and the Shamir method. Here, the fact that points on an elliptic curve can be inverted at negligible costs proved very useful, namely the effort for precomputing the required points can be reduced by more than 50%, if the scalars are represented in a signed representation. It also turned out, that the average number of ECADD operations required by the Interleave method and the Shamir method depends on the AHD and the AJHD of the scalars, respectively.

In Chapter 5 two $\mathcal{D}$-representations which minimize the AHD and the AJHD of the scalars were presented, namely the wNAF and the JSF, respectively. While those representations speed up the Interleave method and the Shamir method in the best possible way, i.e. the resulting AHD and AJHD is minimal, there still is a drawback. The generation of the wNAF and the JSF is only possible starting at the least significant bit, i.e. right-to-left. Therefore, the

recoding of the n-bit scalars must be performed in a separate stage and the whole recoded scalars must be stored, which requires memory of the order of magnitude of $n \cdot k$ bits for both $\mathcal{D}$-representations.

A solution to this problem was proposed in Chapter 6, namely the wMOF and the ltrJSF. Both those $\mathcal{D}$-representations provide the same, minimal AHD and AJHD as the wNAF and the JSF, respectively. Their great advantage is, that they can be generated from left-to-right which means, that the recoding doesn't have to be done in a separate stage, but can be performed on-the-fly during the evaluation. As a result, it is no longer necessary to store the whole recoded scalars, but only small parts at once. In detail, the wMOF requires only memory of the order of magnitude of $k \cdot w$ bits and the ltrJSF requires only memory of the order of magnitude of $k \cdot (k+1)$ bits, which is very small compared to $n \cdot k$ bits.

Chapter 7 showed in detail, in what way the introduced $\mathcal{D}$-representations improve the speed of the Interleave method and the Shamir method. A comparison was made based on the average number of ECADD operations required and the number of points which have to be precomputed. It turned out, that compared to the binary representation, the introduced $\mathcal{D}$-representations significantly reduce the average number of ECADD operations required. However, it was also shown that there is a trade-off between the number of points to precompute and the number of ECADD operations required.

To decide, which method offers the best trade-off, the total number of field operations required to compute a multi-scalar multiplication was estimated explicitly in Chapter 8. It turned out, that in the chosen scenario the Shamir method in conjunction with the ltrJSF provides the best performance if $k = 2, 3$, while for any other number of scalars the Interleave method in conjunction with the 4MOF is the fastest.

9.1 Outlook and Further Research

In this thesis two different kinds of $\mathcal{D}$-representations were discussed. On the one hand two $\mathcal{D}$-representations that use digit sets of the form $\mathcal{D}_w = \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$ and provide a low AHD were introduced. The problem with such digit sets is, that their size grows exponentially with the value of w.

To provide more flexibility, there also exist $\mathcal{D}$-representations which use a digit set of the form $\mathcal{D}_x = \{0, \pm 1, \pm 3, \ldots, \pm x\}$, for any odd $x \geq 1$ [SST04, Möl02, Möl04]. It has also been proven that the AHD provided by those $\mathcal{D}$-representations is minimal, which implies that if $x = 2^{w-1} - 1$ holds for some w, the AHD is the same as of the wMOF and the wNAF. The $\mathcal{D}$-representations that use such digit sets can also be generated from left-to-right and from right-to-left likewise.

On the other hand, two $\mathcal{D}$-representations that use the digit set $\mathcal{D} = \{0, \pm 1\}$ and provide a low AJHD were introduced. Naturally, the next step here is to consider $\mathcal{D}$-representations which use the next larger digit set $\mathcal{D} = \{0, \pm 1, \pm 3\}$. In the case of two scalars, this has been done in [Ava02, DOT05b, KZZ04]. Different to the other digit sets, a $\mathcal{D}$-representation which provides a minimal AJHD amongst all $\mathcal{D}$-representations using the digit set $\{0, \pm 1, \pm 3\}$ is still unknown. However, the $\mathcal{D}$-representation that currently provides the lowest AJHD was proposed in [DOT05b], namely $\mathcal{AJHD}_2([\text{DOT05b}]) = 239/661 \approx 0.3615$. Further, this $\mathcal{D}$-representation can also be generated from left-to-right. Research that has to be done here is to find minimal representations for an arbitrary number of scalars, not only for the digit set $\mathcal{D} = \{0, \pm 1, \pm 3\}$, but also for any larger digit set.

Also, the research in the direction of the precomputation stage is not completed. The methods for precomputing the required points introduced in Section 8.2 are straight forward and not very complicated. A superior approach, which significantly decreases the number of field inversions required was proposed in [CMO98]. However, a method that minimizes the computational costs for the precomputation stage is still unknown.

Bibliography

[Ava02] Avanzi, R., *On multi-exponentiation in cryptography*, Cryptology ePrint Archive, Technical Report 2002/154, 2002, available at `http://eprint.iacr.org/2002/154/`.

[Ava04] Avanzi, R., *A Note on the Signed Sliding Window Integer Recoding and a Left-to-Right Analogue*, Selected Areas in Cryptography - SAC 2004, LNCS 3357, Springer, 2004, pp. 130-143.

[Boo51] Booth, A., *A signed binary multiplication technique*, Quarterly Journal of Mechanics and Applied Mathematics, vol. 4, no. 2, 1951, pp. 236-240.

[BSS99] Blake, I., Seroussi, G., and Smart, N., *Elliptic Curves in Cryptography*, London Mathematical Society, Lecture Note Series 265, Cambridge University Press, 1999.

[CMO98] Cohen, H., Miyaji, A., Ono, T., *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, Advances in Cryptology - ASIACRYPT '98, LNCS 1514, Springer, 1998, pp. 51-65.

[DH76] Diffie, W., and Hellman, M., *New directions in cryptography*, IEEE Transactions on Information Theory, vol. IT-22, no. 6, 1976, pp. 644-654.

[DOT05a] Dahmen, E., Okeya, K. and Takagi, T., *Efficient Left-to-Right Multi-Exponentiations*, Technical University of Darmstadt, Technical Report TI-2/05, 2005, to appear, will be available at `http://www.cdc.informatik.tu-darmstadt.de/reports/README.TR.html`.

[DOT05b] Dahmen, E., Okeya, K. and Takagi, T., *An Advanced Method for Joint Scalar Multiplications on Memory Constraint Devices*, 2nd European Workshop on Security and Privacy in Ad hoc and Sensor Networks - ESAS 2005, LNCS 3813, Springer, 2005, to appear.

[ElG85] ElGamal, T., *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, Advances in Cryptology - CRYPTO '84, LNCS 196, Springer, 1985, pp. 10-18.

[GHPT03] Grabner, P., Heuberger, C., Prodinger, H., Thuswaldner J., *Analysis of linear combination algorithms in cryptography*, ACM Transactions on Algorithms - TALG, vol. 1, no. 1, 2005, pp. 123-142.

[Gor98] Gordon, D., *A survey of fast exponentiation methods*, Journal of Algorithms, vol. 27, no. 1, 1998, pp. 129-146.

[HKPR04] Heuberger, C., Katti, R., Prodinger, H., Ruan, X., *The Alternating Greedy Expansion and Applications to Left-To-Right Algorithms in Cryptography*, Theoretical Computer Science, vol. 341, 2005, pp. 55-72.

[JM99] Johnson, D., and Menezes, A., *The Elliptic Curve Digital Signature Algorithm (ECDSA)* University of Waterloo, Technical Report CORR 99-34, 1999, available at `http://www.cacr.math.uwaterloo.ca`.

[Kob87] Koblitz, N., *Elliptic Curve Cryptosystems*, Mathematics of Computation, vol. 48, no. 177, 1987, pp. 203-209.

[Kob99] Koblitz, N., *Algebraic Aspects of Cryptography*, Algorithms and Computation in Mathematics, vol. 3, 2nd printing, Springer, 1999.

[KZZ04] Kuang, B., Zhu, Y., Zhang, Y., *An Improved Algorithm for uP+vQ using* JSF_3, Applied Cryptography and Network Security - ACNS 2004, LNCS 3089, Springer, 2004, pp. 467-478.

[LL94] Lim, C., Lee, P., *More flexible exponentiation with precomputation*, Advances in Cryptology - CRYPTO '94, LNCS 839, Springer, 1994, pp. 95-107.

[Mil86] Miller, V.S., *Use of Elliptic Curves in Cryptography*, Advances in Cryptology - CRYPTO '85, LNCS 218, Springer, 1986, pp. 417-426.

[MOC97] Miyaji, A., Ono, T., and Cohen, H., *Efficient Elliptic Curve Exponentiation*, Information and Communication Security - ICICS 1997, LNCS 1334, Springer, 1997, pp. 282-291.

[MO90] Morain, F., Olivos, J., *Speeding Up the Computations on an Elliptic Curve using Addition-Subtraction Chains*, Theoretical Informatics and Applications, vol. 24, no. 6, 1990, pp.531-543.

[MOV93] Menezes, A., Okamoto, T., Vanstone, S., *Reducing elliptic curve logarithms to logarithms in a finite field*, IEEE Transactions on Information Theory, vol. 39, no. 5, 1993, pp. 1639-1646.

[Möl01] Möller, B., *Algorithms for Multi-exponentiation*, Selected Areas in Cryptography - SAC 2001, LNCS 2259, Springer, 2001, pp. 165-180.

[Möl02] Möller, B., *Improved Techniques for Fast Exponentiation*, Information Security and Cryptology - ICISC 2002, LNCS 2587, Springer, 2003, pp. 298-312.

[Möl04] Möller, B., *Fractional Windows Revisited: Improved Signed-Digit Representations for Efficient Exponentiation*, Information Security and Cryptology - ICISC 2004, LNCS 3506, Springer, 2005, pp. 137-153.

[MS04a] Muir, J., Stinson, D., *Minimality and Other Properties of the Width-w Nonadjacent Form*, University of Waterloo, Technical Report CORR 2004-08, 2004, available at `http://www.cacr.math.uwaterloo.ca`.

[MS04b] Muir, J., Stinson, D., *New Minimal Weight Representations for Left-to-Right Window Methods*, University of Waterloo, Technical Report CORR 2004-19, 2004, available at `http://www.cacr.math.uwaterloo.ca`.

[NIST01] Daley, W., Kammerer, R., *Digital Signature Standard (DSS)*, National Institute of Standards and Technology - NIST, Federal Information Processing Standards - FIPS 186-2, 2001, available at `http://csrc.nist.gov/publications/fips/index.html`.

[Odl84] Odlyzko, A., *Discrete Logarithms in Finite Fields and Their Cryptographic Significance*, Advances in Cryptology - EUROCRYPT '84, LNCS 209, Springer, 1984, pp. 224-314.

[OSST04] Okeya, K., Schmidt-Samoa, K., Spahn, C., Takagi, T., *Signed Binary Representations Revisited*, Advances in Cryptology - CRYPTO 2004, LNCS 3152, Springer, 2004, pp. 123-139, full version available at `http://eprint.iacr.org/2004/195/`

[Phi05] *Philips products - SmartMX*, available at `http://www.semiconductors.philips.com/`

[Pol78] Pollard, J.M., *Monte Carlo methods for index computation (mod p)*, Mathematics of Computation, vol. 32, no. 143, 1978, pp. 918-924.

[Pro03] Proos, J., *Joint Sparse Forms and Generating Zero Columns when Combing*, University of Waterloo, Technical Report CORR 2003-23, 2003, available at `http://www.cacr.math.uwaterloo.ca`.

[Rei60] Reitwiesner, G. W., *Binary arithmetic*, Advances in Computers, vol. 1, 1960, pp. 231-308.

[Ren05] *Renesas Technology - Product range*, available at `http://eu.renesas.com/`

[Sha69] Shanks, D., *Class number, a theory of factorization, and genera*, Proceedings of Symposia in Pure Mathematics, vol. 20, 1969, pp. 415-440.

[SST04] Schmidt-Samoa, K., Semay, O., Takagi, T., *Analysis of Some Fractional Window Recoding Methods and their Application to Elliptic Curve Cryptosystems*, IEEE Transactions on Computers, vol. 55, no. 1, 2006, pp. 1-10.

[Sol00] Solinas, J.A., *Efficient Arithmetic on Koblitz Curves*, Design, Codes and Cryptography, vol. 19, 2000, pp. 195-249.

[Sol01] Solinas, J.A., *Low-weight binary representations for pairs of integers*, University of Waterloo, Technical Report CORR 2001-41, 2001, available at `http://www.cacr.math.uwaterloo.ca`.

[Van92] Vanstone, S., *Responses to NIST's proposal*, Communications of the ACM, vol. 35, no. 7, 1992, pp. 41-54, communicated by John Anderson.