UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe Codes und Kryptographie

Implementation and Comparison of Elliptic Curve Algorithms in Java

Master's Thesis

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by
Swante Scholz

submitted to:
Prof. Dr. Johannes Blömer
and
Jun. Prof. Dr. Sevag Gharibian

Paderborn, August 1, 2019

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date

Signature

Contents

Acronyms

CDHP computational Diffie-Hellman problem. 11

DDHP decisional Diffie-Hellman problem. 11, 12

DLP discrete logarithm problem. 6

ECC elliptic curve cryptography. 3, 5

ECDLP elliptic curve discrete logarithm problem. 6, 11

IBE identity-based encryption. 13, 14

JIT just-in-time. 40

NAF non-adjacent form. 26

SAS sequential aggregate signature. 14

TTP trusted third tarty. 13, 14

wNAF width-w non-adjacent form. 26, 27

Notation

$\mathbb{N}$	the natural numbers, including 0: $\mathbb{N} \coloneqq \{0, 1, \dots\}$
$A[i]$	the element of an array/list with index i; indices start at 0.
$A[i...j]$	the subarray of A from index i to index j (inclusive); if either index is outside of A's range, a smaller subarray containing only the elements of A whose indices are within $[i...j]$ will be returned
$\log x$	the binary logarithm of x
p	a (usually large) prime
q	a prime power: $q = p^k$, $k \in \mathbb{N}_{>0}$
$\mathbb{F}$	placeholder for an arbitrary (not necessarily finite) field
$\mathbb{G}$	placeholder for an arbitrary group
$0_{\mathbb{G}}$	neutral element of additive group $\mathbb{G}$
$1_{\mathbb{G}}$	neutral element of multiplicative group $\mathbb{G}$
$\mathbb{F}_q$	finite field with exactly $q = p^k$ elements; if q is prime ($k = 1$), $\mathbb{F}_q = \{0, 1, ..., q-1\}$
$\mathbb{F}_q^*$	multiplicative group $\mathbb{F}_q \setminus \{0\}$
E	the equation defining an elliptic curve in the short Weierstrass form: $$E : y^2 = x^3 + ax + b$$
$E(\mathbb{F})$	set of points with coordinates from field $\mathbb{F}$ lying on an elliptic curve E, including ∞
∞	neutral element of elliptic curve $E(\mathbb{F})$, the *point at infinity*
$\#E(\mathbb{F})$	number of points on curve over field $\mathbb{F}$, including ∞
r	a large prime dividing $\#E(\mathbb{F})$, thus also the size of a subgroup of $E(\mathbb{F})$
$P + Q$	general addition of two points on an elliptic curve E
$P + P = 2P$	doubling of a point P on an elliptic curve E

kP $P + P + P + ...$ (k times), the kth scalar multiple of elliptic curve point P

G a designated point on $E(\mathbb{F}_{p^k})$, the *generator*; G generates the cyclic elliptic curve subgroup that is used for ECC

$E_G(\mathbb{F}_{p^k})$ the cyclic elliptic curve subgroup of $E(\mathbb{F}_{p^k})$, generated by the generator $G \in E(\mathbb{F}_{p^k})$: $E_G(\mathbb{F}_{p^k}) = \{eG : e \in \{1 \dots n_G\}\} \subseteq E$

n_G smallest positive integer such that $n_G G = \infty$ for a generator $G \in E(\mathbb{F}_{p^k})$; thus, n_G is also the size of the subgroup generated by G; if G was specifically chosen to generate a subgroup of given size r, we have $n_G = r$

h cofactor of curve E with generator G: $h = \#E(\mathbb{F}_{p^k})/n_G \in \mathbb{N}$

k the exponent of the prime power $q = p^k$; in the context of groups, k is thus also the *embedding degree* of $\mathbb{F}_{p^k}$: the smallest integer s.t. $n_G \mid p^k - 1$

t the *trace of Frobenius*: $t = \#E(\mathbb{F}_q) - q - 1$

$\mathrm{char}(\mathbb{F}_{p^k}) = p$ the *characteristic* of $\mathbb{F}_{p^k}$, defined as

$$\min\Big\{n \in \mathbb{N} : n \cdot a = 0 \; \forall \, a \in \mathbb{F}_{p^k}\Big\}$$

$E(\mathbb{F}_{p^k})[r]$ the $r-$torsion subgroup of curve $E(\mathbb{F}_{p^k})$:

$$E(\mathbb{F}_{p^k})[r] \coloneqq \{P \in E : rP = \infty\}$$

(x, y) affine coordinates, equivalent to $(x, y, 1)$ in projective and Jacobian coordinates

(X, Y, Z) projective or Jacobian coordinates; if $Z = 0$, (X,Y,Z) represents ∞, otherwise it's equivalent to affine point $(X/Z, Y/Z)$ (in the projective case) or $(X/Z^2, Y/Z^3)$ (in the Jacobian case)

e bilinear pairing function which maps from the source groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to the target group $\mathbb{G}_T$

$\pi_p(Q)$ Frobenius endomorphism of (affine) elliptic curve point Q: $\pi_p(Q) = \pi_p((Q_x, Q_y)) \coloneqq (Q_x^p, Q_y^p)$

Abstract

Cryptography has a huge impact on our everyday lives: Private messaging, online banking, E-commerce, etc.; none of this would work securely without proper cryptographic protocols. But with rising computational power of potential attackers, the key lengths required to keep popular encryption schemes like RSA secure have become inconveniently large.

This has given rise to interest in alternative protocols that require shorter keys for the same level of security. Elliptic curve cryptography (ECC) is one way to implement such protocols. Together with appropriately defined bilinear maps (called *pairings*), the advancement of ECC led to new achievements like identity-based cryptography.

There are many subtopics related to ECC (point addition, scalar multiplication, and pairings, among other things), with numerous possible ways to approach them each and it is often not clear when to use which technique, and how to efficiently implement it in Java.

Over the course of our work on this thesis, we looked at all these topics, implemented the algorithms in Java, and evaluated their relative performance on laptops and Android-smartphones.

1 Introduction

Elliptic curve cryptography (ECC) is based on operations on points on a two-dimensional curve defined by the Weierstrass equation

$$y^2 = x^3 + ax + b$$

over a finite field $\mathbb{F}_{p^k}$.

Subexponential algorithms exist for solving the discrete logarithm problem in finite fields, requiring RSA protocols to use large keys in order to remain secure. Meanwhile, no subexponential algorithms have been found for the equivalent problem for elliptic curves, making them attractive in particular for environments where memory or bandwidth is limited, for example on smart cards (see [9, p. xxx]).

The main goal of this master thesis is to give an overview of various important algorithms that can be used for ECC, show how to implement them efficiently in Java, and evaluate which ones are best suited for given tasks and environments. We are primarily focusing on the practical aspects of implementing these algorithms, rather than dwelling on the theoretical aspects and providing rigorous proofs. For a more detailed analysis of the mathematical subtleties of the subject matter, please refer to [9], [12], and [4].

All code written over the course of our work on this thesis built upon `upb.crypto.math`, the open source elliptic curve library of the University of Paderborn (`https://github.com/upbcuk/upb.crypto.math/`), adapting and extending it at appropriate places in order to accommodate the new algorithms and techniques presented here.

The remainder of this thesis is structured as follows: In Chapter 2, we present the mathematical foundations of ECC – finite fields, elliptic curves, and bilinear pairings. All operations mentioned in that chapter are already part of the `upb.crypto.math` library. In Chapter 3, we extend on the previously described operations, showing how they can be more efficiently computed using more advanced techniques, for example by using projective or Jacobian coordinates instead of affine ones, or by replacing the simple double-and-add method for scalar multiplication with faster, windowed methods. Chapter 4 then continues with outlining how these optimization ideas have been implemented in Java and integrated in the `upb.crypto.math` library. In Chapter 5, we show results of running performance tests for all new algorithms in comparison to the existing ones, both on a laptop and on an Android smartphone, with various combinations of parameters. Chapter 6 contains some final thoughts on how meaningful the improvements were overall and which areas could be further researched in future work.

2 Fundamentals

In this chapter we will describe fundamental structures and operations required for ECC. All operations mentioned in this chapter were already part of the `upb.crypto.math` library at the start of this project.

In Section 2.1 we will summarize relevant characteristics of finite fields. Section 2.2 continues with the definition of elliptic curves and their properties. Finally, in Section 2.3 we will explain the details of bilinear pairings in general, and the Tate pairing in particular.

2.1 Finite fields

Finite fields are the basic underlying building blocks used for elliptic curves. Thus, understanding their basic properties is crucial in understanding ECC.

2.1.1 Definition

A *field* is a triple $(K, +, \cdot)$ such that

- $(K, +)$ is an abelian group with neutral element denoted by 0
- $(K^*, \cdot)$ is also an abelian group, with its neutral element denoted by 1
- Distributivity holds: $a \cdot (b + c) = a \cdot b + a \cdot c$

If a field is of finite order q, it is called a *finite field*.

The only finite fields that exist are of order $q = p^k$ with p prime and $k \in \mathbb{N}_{\geq 1}$ (cf. [12, p. 26]). Finite fields of prime order are called *prime fields*. Finite fields with order $q = p^k$, with $k > 1$ are called *extension fields*.

Any two finite fields of the same order q are isomorphic to each other (cf. [9, p. 31]). A finite field of prime power order is therefore essentially unique and denoted by $\mathbb{F}_q$.

2.1.2 Extension fields

While an element x of a prime field $\mathbb{F}_p$ can simply be represented by an integer $i \in \{0, ..., p-1\}$, it is not immediately obvious how to represent an element of an extension field $\mathbb{F}_{p^k}$. As outlined in [9, p. 34], one possibility is to use polynomials of degree k with all the coefficients being from $\mathbb{F}_p$. Addition, subtraction and multiplication are performed modulo an irreducible polynomial $m(X)$ (also of degree k), while division/inversion can be implemented using a variation of the extended GCD algorithm.

It should be noted that a trivial multiplication implementation on the polynomials will incur a runtime cost that is quadratic in k. Thus it is desirable to work with lower-degree polynomials if possible in order to optimize runtime performance.

2.1.3 Discrete logarithm problem

For a (multiplicative) group $\mathbb{G}$ and a randomly chosen integer $x \in [1, |\mathbb{G}| - 1]$, given $\alpha \in \mathbb{G}$ and $y = \alpha^x \in \mathbb{G}$, the task of computing x is referred to as the discrete logarithm problem (DLP). Many popular encryption schemes like RSA (see [23]) depend (among other things) on the assumption that the DLP is difficult. However, subexponential algorithms for solving the DLP in finite fields $\mathbb{F}_{p^k}$ have been found (e.g. index calculus algorithms like [1]), leading to inconveniently large key sizes in order to achieve a sufficient degree of security.

For other groups though, no subexponential algorithms for the DLP are known, which makes them very attractive to be used in cryptographic schemes. The probably most notable of those groups, elliptic curves, are presented in the next section. In an (additively-written) elliptic curve group $\mathbb{G}$, the so-called elliptic curve discrete logarithm problem (ECDLP) can be formulated as: Given only the elliptic curve points $P \in \mathbb{G}$ and $Q = kP \in \mathbb{G}$ for a randomly chosen integer k, find k. The ECDLP is commonly assumed to be intractable [12, pp. 153-154].

2.2 Elliptic curves

Elliptic curves are algebraic structures well suited for cryptographic purposes, allowing for encryption with a high level of security, even when using a key size that is significantly lower than those used in other encryption schemes.

2.2.1 Definition

Given elements a, b of a field $\mathbb{F}$ (usually $\mathbb{F}_q$), an elliptic curve $E(\mathbb{F})$ (in the *Weierstrass short form*) is defined as the set of all points $P = (x, y) \in \mathbb{F}^2$ satisfying the *Weierstrass equation* (cf. [4, p. 31])

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

with the *discriminant* $\Delta := -16(4a^3 + 27b^2) \neq 0$, a special identity point ∞ (sometimes referred to as "point at infinity"), and an addition operation

$$+ : \quad (E(\mathbb{F}))^2 \to E(\mathbb{F}), \quad (P, Q) \mapsto P + Q$$

which is defined in the following subsections. If the field $\mathbb{F}$ used for the elliptic curve is clear from context, we will just write E instead of $E(\mathbb{F})$.

In addition to a and b, another parameter used for specifying elliptic curves over finite fields is the *generator* $G \in E(\mathbb{F})$. Rather than looking at the complete group $E(\mathbb{F})$, we

often just look at the subgroup E_G "generated" by G:

$$E_G \coloneqq \{eG : e \in \{1 \dots n_G\}\} \subseteq E(\mathbb{F})$$

where n_G is the order of G (i.e. the smallest $e \in \mathbb{Z}_{\geq 1}$ such that $eG = \infty$). This way, any point $P \in E_G$ can be uniquely represented by an integer $\alpha \in \{1 \dots n_G\}$ with $P = \alpha G$.

If $E(\mathbb{F})$ has prime order $n = \#E(\mathbb{F})$ (as is the case with the Barreto-Naehrig curves described in Section 2.2.8), every point $P \in E(\mathbb{F})$ has order n and thus the subgroup E_P generated by P is equal to $E(\mathbb{F})$ (cf. [10]).

2.2.2 Inverse

The inverse $-P$ of point $P = (x, y) \in E(\mathbb{F})$ is given by $(x, -y)$. Compared to inversions in finite fields, computing the inverse of elliptic curve points is trivial, which can be utilized by certain algorithms that work with inverses in order to minimize the number of operations (like the one described in Section 3.2.3).

2.2.3 General point addition

If $P \neq Q$, we can think of the addition operation in the following way: Draw a straight line through P and Q. This line will intersect E in exactly one other, third point: $-R$. Mirroring this point across the x-axis gives the result $R = P + Q$. Translating this procedure into equation gives:

$$P + Q = (x_1, y_1) + (x_2, y_2) = (x_3, y_3) = (s^2 - x_1 - x_2, s(x_1 - x_3) - y_1) \tag{2.2}$$

with $s = \frac{y_1 - y_2}{x_1 - x_2}$ being the "slope" of the line through P and Q.

Figure 2.1 shows how addition would look like on elliptic curves over $\mathbb{Q}$. While this visualization doesn't work well when using $\mathbb{F}_q$, the equation above works just the same in both cases, no matter what underlying field is used. Counting the number of field operations required to compute $P+Q$ via Equation 2.2, we get: one division for computing s (which is equivalent to a field inversion, followed by a multiplication), one squaring, one general multiplication, and six subtractions.

2.2.4 Point doubling

If $P = Q$, it is not clear how the line as described in the previous section should be drawn. Instead, as depicted in Figure 2.2, the tangent slope at that point is used, which is computed as $s = \frac{3x^2 + a}{2y}$. Otherwise, the procedure is very similar to the general addition formula 2.2:

$$P + P = 2P = 2 \cdot (x, y) = (x_3, y_3) = (s^2 - 2x, s(x - x_3) - y) \tag{2.3}$$

Thus, the used field operations consist of: one division (i.e. a field inversion, followed by a multiplication), two squarings, one general multiplication, two multiplications by

Figure 2.1: General point addition over $\mathbb{Q}$

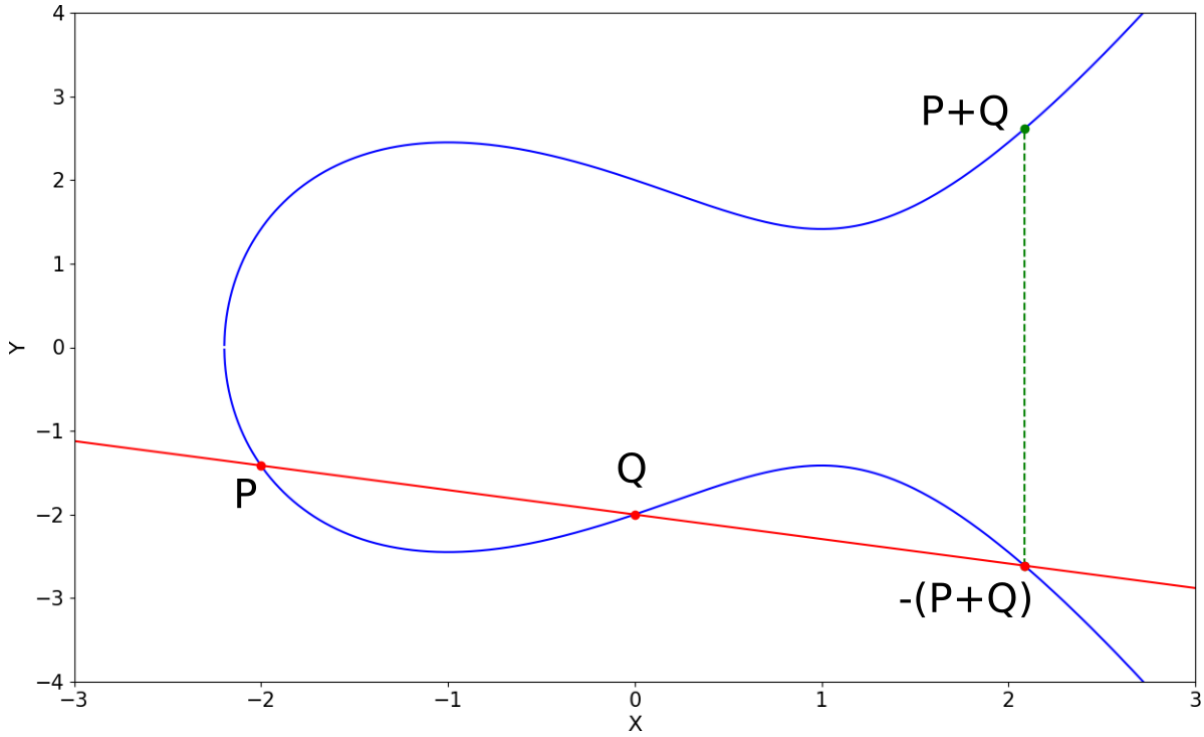
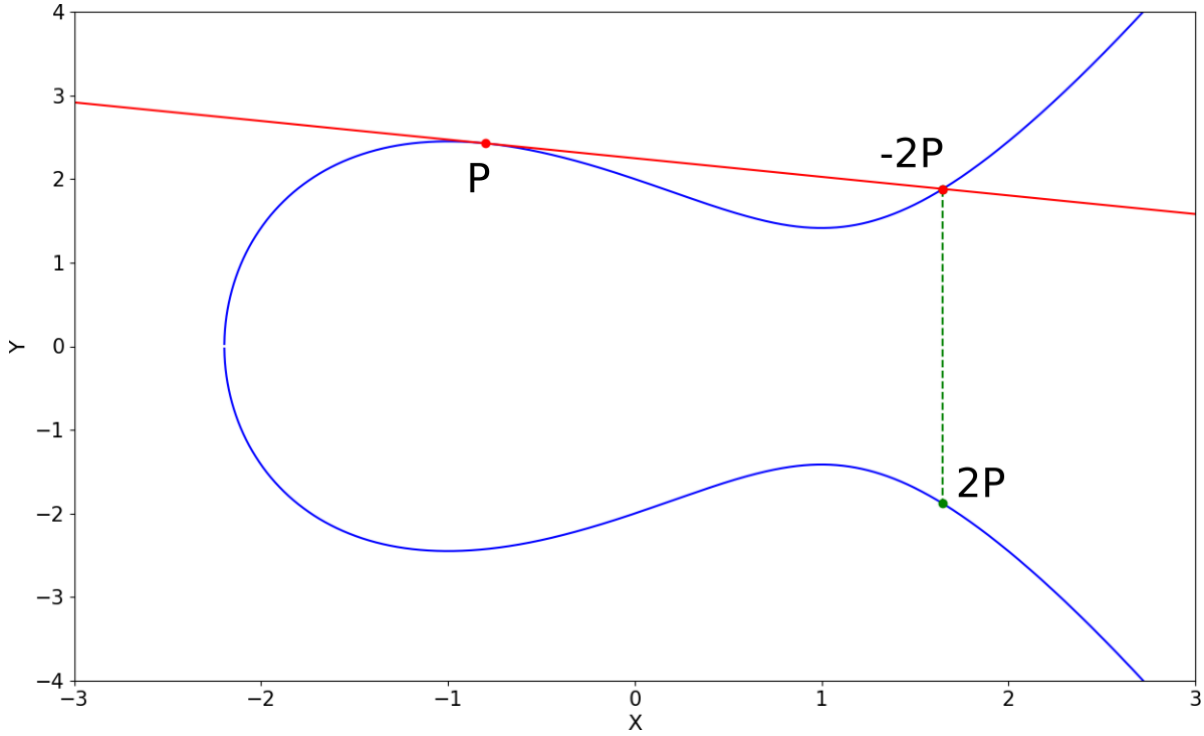


Figure 2.2: Point doubling over $\mathbb{Q}$



2, one multiplication by 3, and four additions/subtractions.

2.2.5 Corner cases

Addition with the neutral element ∞ leaves a point $P \in E(\mathbb{F})$ unchanged:

$$P + \infty = \infty + P = P$$

∞ is its own inverse:

$$-\infty = \infty$$

For any point $P = (x, y) \in E(\mathbb{F})$ we have:

$$P + (-P) = (x, y) + (x, -y) = \infty$$

I.e.: All vertical lines intersect E at $-\infty = \infty$.

2.2.6 Scalar multiplication

In order to compute the scalar multiple

$$tP = \overbrace{P + P + \cdots + P}^{t \text{ addends}}$$

you could simply perform all additions separately, which would require $t - 1$ additions in total (one of which being a doubling operation).

A common optimization is based on the well-known square-and-multiply method for fast exponentiation of elements from multiplicative groups (cf. [9, p.181]). Replacing squarings with doublings and multiplications with general additions, we get a "double-and-add" algorithm for computing tP:

Listing 2.1: Computing tP using the basic "double-and-add" technique

```
1 doubleAndAdd(P, t = (t_{L-1} t_{L-2} ... t_0)_2):
2     R ← P
3     for i from L-2 downto 0:
4         R ← 2 · R   # doubling operation
5         if t_i = 1:
6             R ← R + P   # general addition
7     return R
```

Since we have one doubling per loop iteration and one general addition per 1 in the binary representation of t, we have just $\mathcal{O}(\log t)$ group operations in total, a significant improvement to the trivial approach. This can be seen in the following example: With $t = 19 = (10011)_2$, the "double-and-add" algorithm would compute

$$19P = 2(2(2(2P)) + P) + P$$

which requires just four doublings and two general additions, rather than 18 additions.

2.2.7 Group order

The order of an elliptic curve group $E(\mathbb{F}_q)$ is given by

$$n \coloneqq \#E(\mathbb{F}_q) = q + 1 - t$$

with $t \in \mathbb{Z}$ being the so-called *trace of Frobenius*. According to *Hasse's theorem* (cf. [4, pp. 34-35]) we have

$$|t| \leq 2\sqrt{q}. \tag{2.4}$$

Since $|t|$ is thus significantly smaller than q, we could say there are "roughly" q points on an elliptic curve over $\mathbb{F}_q$.

2.2.8 Barreto-Naehrig curves

A specific class of elliptic curves that have some useful properties are *Barreto-Naehrig curves* (or *BN-curves* for short), where the trace of Frobenius t, the group order n and the characteristic p are parameterized as

$$\begin{aligned} t(s) &= 6s^2 + 1 \\ n(s) &= 36s^4 - 36s^3 + 18s^2 - 6s + 1 \\ p(s) &= 36s^4 - 36s^3 + 24s^2 - 6s + 1 \end{aligned}$$

with s chosen such that $n(s)$ and $p(s)$ are primes (see [3]). Then, a suitable value for b is computed, such that the elliptic curve $E(\mathbb{F}_p) : y^2 = x^3 + b$ conforms to the chosen t, n, and p (cf. [3]).

BN-curves have the advantage that their embedding degree $k = 12$ is larger than that of previous approaches, which is desirable in order to thwart index-calculus attacks (cf. [10]). The parameterized nature of BN-curves also allows for efficient transmission of it (you only need to send s), as well as for some optimizations during the computation of pairings. Lastly, since n is prime, we have $E(\mathbb{F}_p) = E_P$ for all $P \in E(\mathbb{F}_p)$, i.e. all points in $E(\mathbb{F}_p)$ are viable generators (of prime order n) and are in the cyclic subgroup. This makes various ECC algorithms significantly easier as we don't need to check if a given point from $E(\mathbb{F}_p)$ is also part of the desired cyclic subgroup.

2.2.9 Application example: elliptic curve Diffie–Hellman key exchange

In this section we will see how the aforementioned operations on elliptic curve points can be utilized for exchanging secret keys: The elliptic curve Diffie-Hellman key exchange is very similar to the "normal" Diffie-Hellman key exchange over finite fields, and can be described as follows (cf. [18]):

We assume Alice and Bob want to agree on a secret key, having only an insecure connection between them available. They have already agreed on a elliptic curve over a prime field $\mathbb{F}_p$ that is defined by the following *domain parameters*:

- a prime number p, the order of the underlying prime field $\mathbb{F}_p$
- the parameters a and b of the elliptic curve (in the short Weierstrass form)
- the generator $G \in E(\mathbb{F}_p)$ that can "generate" all elements of the elliptic curve subgroup
- the order n_G of the elliptic curve subgroup generated by G

Now, Alice generates her private key α uniformly at random from the range $[1, n_G - 1]$ and computes her public key $A = \alpha G$. Similarly, Bob computes his key pair (β, B), with $B = \beta G$. After exchanging their public keys A and B via the insecure channel, Alice can compute $\alpha B = \alpha\beta G =: K = (K_x, K_y)$, while Bob can compute $\beta A = \beta\alpha G = \alpha\beta G = K = (K_x, K_y)$ and they can use the value K_x (which was never transmitted via the insecure channel) to derive a symmetric key.

Computational/Decisional Diffie-Hellman Problem

After a successful elliptic curve Diffie-Hellman key exchange between Alice and Bob, a potential eavesdropper Eve could now know the values $G, \alpha G, \beta G$, and might want to try to determine the secret key K based only on this information. This is called the computational Diffie-Hellman problem (CDHP), and is assumed to not be efficiently solvable as long as the order n of the elliptic curve is sufficiently large (cf. [12, p. 171]). Fortunately, as described in Section 2.2.7, elliptic curves over large finite fields also have a similarly large order. It should be noted, however, that as the CDHP can easily be reduced to the ECDLP by first computing α as the discrete logarithm of $A = \alpha G$ and then evaluating $K = \alpha B = \alpha\beta G$. Thus the CDHP is at most as difficult as the ECDLP, which has not (yet) been proven to be truly intractable.

If you already have a candidate solution C and you only want to check if C is indeed equal to $K = \alpha\beta G$, this is called the decisional Diffie-Hellman problem (DDHP) and can be solved efficiently with bilinear pairings which are described in the next section.

2.3 Bilinear pairings

Bilinear pairings are functions that map two elements from source groups (usually elliptic curve groups) to a "simpler" target group (usually $\mathbb{F}_{p^k}^*$). Originally, these pairings were used to attack the DDHP, but since then numerous other applications for them have been found.

After a quick review of the definition of bilinear pairings (Section 2.3.1), we will summarize some of those useful applications in Section 2.3.2, and then conclude in Section 2.3.3 with the one type of pairing that is currently implemented in the `upb.crypto.math` library: the Tate-pairing.

2.3.1 Definition

A bilinear pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, with source groups $\mathbb{G}_1, \mathbb{G}_2$ (written additively), and a target group $\mathbb{G}_T$ (written multiplicatively), satisfying (cf. [18]):

Bilinearity e is linear in both of its arguments:

$$e(A + C, B) = e(A, B) \cdot e(C, B)$$

$$e(A, B + C) = e(A, B) \cdot e(A, C)$$

which implies for any $x, y \in \mathbb{Z}$:

$$e(xA, yB) = e(A, B)^{xy}$$

Non-degeneracy

$$(e(P, Q) = 1 \text{ for all } Q \in \mathbb{G}_2) \to P = \infty$$

Computability e is efficiently computable

Bilinear pairings are classified into three separate types, depending on the relationship of groups $\mathbb{G}_1$ and $\mathbb{G}_2$ (cf. [11, p. 3115]):

Type 1 There exists an efficiently computable isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ and vice versa (or simply $\mathbb{G}_1 = \mathbb{G}_2$)

Type 2 There exists an efficiently computable isomomorphism from $\mathbb{G}_2$ into $\mathbb{G}_1$, but as far as we know not from $\mathbb{G}_1$ into $\mathbb{G}_2$

Type 3 There is no currently known, efficiently computable isomomorphism from either source group to the other

While type 1 pairings have been preferred in the past due to their simplicity, it has been shown that they are not sufficiently secure (cf. [11, p. 3118]). Therefore, nowadays type 2 and in particular type 3 pairings are preferable.

2.3.2 Applications of pairings

Here is a list of just a few applications of bilinear pairings in order to give an overview about their usefulness in cryptography:

Using pairings to solve the DDHP

If there is a bilinear pairing $e : \mathbb{G}_1^2 \to \mathbb{G}_T$, then the DDHP on the group $\mathbb{G}_1$ becomes easy to solve: Given a generator G for $\mathbb{G}_1$, and the transferred values $A = \alpha G$ and $B = \beta G$, and a candidate value $C = \gamma G$, we have

$$\alpha\beta = \gamma \Leftrightarrow e(\alpha G, \beta G) = e(G, G)^{\alpha\beta} = e(G, G)^{\gamma} = e(G, \gamma G) \Leftrightarrow e(A, B) = e(G, C)$$

which can be checked trivially.

Joux's three-party one-round key agreement

Before the discovery of bilinear pairings, there was no known algorithm for a Diffie-Hellman-like key agreement for three parties (Alice, Bob and Charlie), that requires just one round of communications. Given a bilinear pairing $e : \mathbb{G}_1^2 \to \mathbb{G}_T$ with G being the generator of $\mathbb{G}_1$, however, this can be done like this (cf. [16, p. 387-388]):

- Alice generates her private key α uniformly at random from the range $[1, n_G - 1]$ and broadcasts $A = \alpha G$ to Bob and Charlie. Similarly, Bob and Charlie broadcast $B = \beta G$ and $C = \gamma G$, respectively.
- Now Alice can compute the secret shared key K as $K = e(G, G)^{\alpha\beta\gamma} = e(\beta G, \gamma G)^{\alpha} = e(B, C)^{\alpha}$, and Bob and Charlie can compute K in a similar manner.
- Now, K can be used to derive a symmetric secret key for further communications.

BLS short signatures

In the BLS short signature scheme (named after the creators Boneh, Lynn and Shacham, [5]), signatures consist of a single element on an elliptic curve E, which can represented by a single natural number (usually simply its x-coordinate). Given a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, a generator G of $\mathbb{G}_1$, and a hash function $H : \{0,1\}^* \to \mathbb{G}_2 \setminus \{0_{\mathbb{G}_2}\}$, the scheme works as follows (cf. [18]):

- Alice generates her private key α uniformly at random from the range $[1, n_G - 1]$ and computes her public key $A = \alpha G \in \mathbb{G}_1$
- Given a message $m \in \{0,1\}^*$, Alice computes her signature as $S = \alpha H(m) \in \mathbb{G}_2$
- Given Alice's public key A, the message m, and her signature S on it, anybody can now verify it by checking the following equality using Alice's public key A:

$$e(G, S) = e(G, \alpha H(m)) = e(\alpha G, H(m)) = e(A, H(m))$$

Identity-based encryption

Bilinear pairings also opened the door for entirely new encryption schemes, most notably identity-based encryption (IBE), where keys are generated by a trusted third tarty (TTP) based on the recipient's identifying information (cf. [18]):

- First, the TTP decides on a secure pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with $\mathbb{G}_2$ being a group of order n with generator G. Then, it generates a secret private key $s \in [1, n-1] \subseteq \mathbb{Z}$ and a public key $S = sG \in \mathbb{G}_2$. Lastly, the TTP also defines a cryptographic hashing function $H : \{0,1\}^* \to \mathbb{G}_1 \setminus \{0_{\mathbb{G}_1}\}$ which maps an arbitrary binary message to an element of the source group $\mathbb{G}_1$. Except for s, all of these values are made available publicly.

- Let's say, Alice now wants to send a message m to Bob with Bob's identifying information (e.g. email, name, or address) given by $\mathrm{ID}_B \in \{0,1\}^*$. Next, Alice generates a random integer $r \in [1, n-1]$ and $R = rG$. Then she encrypts the message m to E_K using a symmetric encryption scheme with a key derived from $K \coloneqq e(H(\mathrm{ID}_B), S)^r$, and sends (E_K, R, ID_B) to Bob via a potentially insecure channel.

- Bob now needs to connect to the TTP via a secure channel in order to receive $d_B \coloneqq sH(\mathrm{ID}_B)$ from it. If he has stored this value (e.g. because he obtained it already for a previous message that used the same ID), he can skip this step. Given E_K and R, Bob can then compute $e(d_B, R) = e(sH(\mathrm{ID}_B), rG) = e(H(\mathrm{ID}_B), sG)^r = e(H(\mathrm{ID}_B), S)^r = K$ and use K to decrypt E_K in order to get the original message m.

As summarized in [28] and [18], IBE has numerous advantages: Encrypted messages can be sent without the need of any prior input from Bob. Also, Alice could add further constraints to ID_B, for example a minimum age requirement: Assuming the TTP knows Bob's birthday, it could then refuse to send the decryption key for this ID to Bob until he has reached the required age, making it impossible for him to decrypt the message as long as he is too young. In order to avoid the overhead of needing to contact the TTP for every new message though, these IDs should ideally be chosen from a small discrete set.

These advantages, however, come at the cost of a high degree of dependency towards the TTP, who is being in control of all keys: The TTP could decrypt any messages that Alice sends to Bob via insecure channels, rendering all communications insecure if the TTP gets compromised. Also, the TTP needs to be online whenever a new identity is used. If the aforementioned method of adding additional constraints to messages is used, Bob might be unable to decrypt any of his messages unless the TTP is online.

Sequential Aggregate Signatures

Another application of pairings, which explicitly requires type 3 pairings in order to work securely, are the sequential aggregate signatures (SAS) introduced in [21]. In contrast to the Camenisch-Lysyanskaya signatures [6], whose sizes grow linearly with the number r of parties, the SAS approach results in a final signature consisting of only two values, regardless of r. The procedure can be summarized in the following way:

Initialization There is a type 3 pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with groups of prime order p. $G_1 \in \mathbb{G}_1$ and $G_2 \in \mathbb{G}_2$ are the generators of the two source groups. An integer x is then chosen at random from $[1, p-1]$ and $X_1 \coloneqq xG_1$ and $X_2 \coloneqq xG_2$ are computed. Except for x, all of these values are public parameters available to all r parties. Each party i has a (private) signing key $y_i \in [1, p-1]$ and a (public) verification key $Y_i \coloneqq y_i G_2$. The initial, "empty" signature σ_0 is defined as $\sigma_0 \coloneqq (\alpha_0, \beta_0) \coloneqq (G_1, X_1) \in \mathbb{G}_1^2$.

Signing Starting with an empty message $M = \langle\ \rangle$ and the "empty" certificate σ_0, M is passed from party 1 to party 2 to party 3, etc. Each party i adds a message $m_i \in [1, p-1]$ to M and updates the current certificate $\sigma_{i-1} = (\alpha_{i-1}, \beta_{i-1})$ based on that message, their signing key y_i, and a random integer $t_i \in [1, p-1]$:

$$\sigma_i = (t_i\alpha_{i-1}, t_i(\beta_{i-1} + y_i m_i \alpha_{i-1}))$$

Thus we have

$$\alpha_r = t_r\alpha_{r-1} = t_r t_{r-1}\alpha_{r-2} = ... = \left(\prod_{i=1}^{r} t_i\right) G_1 = tG_1, \qquad \text{with } t \coloneqq \prod_{i=1}^{r} t_i$$

and

$$\begin{aligned}\beta_r &= t_r(\beta_{r-1} + y_r m_r \alpha_{r-1}) = t_r\beta_{r-1} + t y_r m_r G_1 \\ &= t_r t_{r-1}(\beta_{r-2} + y_{r-1} m_{r-1}\alpha_{r-2}) + t y_r m_r G_1 \\ &= t_r t_{r-1}\beta_{r-2} + t y_{r-1} m_{r-1} G_1 + t y_r m_r G_1 \\ = ... &= t\left(\beta_0 + \left(\sum_{i=1}^{r} y_i m_i\right) G_1\right) = \left(x + \sum_{i=1}^{r} y_i m_i\right) tG_1\end{aligned}$$

Verification Given the public keys Y_i of all parties, the final message $M = \langle m_1, ..., m_r\rangle$ and the aggregated signature $\sigma_r = (\alpha_r, \beta_r)$, the authenticity of the signature can now be verified by checking if $e(\alpha_r, X_2 + \sum_i m_j Y_i) = e(\beta_r, G_2)$ holds, since

$$\begin{aligned} e\left(\alpha_r, X_2 + \sum_i m_i Y_i\right) &= e\left(tG_1, X_2 + \sum_i m_i y_i G_2\right) \\ &= e\left(tG_1, \left(x + \sum_i m_i y_i\right) G_2\right) \\ &= e\left(\left(x + \sum_i m_i y_i\right) tG_1, G_2\right) \\ &= e(\beta_r, G_2). \end{aligned}$$

2.3.3 Reduced Tate pairing

One of the most popular pairings is the reduced Tate pairing. Given an elliptic curve $E(\mathbb{F}_p)$, a large prime r dividing $\#E(\mathbb{F}_p)$, and the embedding degree k of r, the source and target groups of the reduced Tate pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ are defined as (cf. [10]):

- $\mathbb{G}_1$ is the r-torsion subgroup of $E(\mathbb{F}_p)$:

$$\mathbb{G}_1 = E(\mathbb{F}_p)[r] = \{P \in E(\mathbb{F}_p) : rP = \infty\},$$

- $\mathbb{G}_2$ is the quotient group of equivalence classes of elements from $E(\mathbb{F}_{p^k})$ under the

equivalence equation $a \equiv b \Leftrightarrow (a-b) \in rE(\mathbb{F}_{p^k}) = \left\{rP : P \in E(\mathbb{F}_{p^k})\right\}$:

$$\mathbb{G}_2 = E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k}),$$

- the target group $\mathbb{G}_T$ is the set of the r-th roots of unity (over $\mathbb{F}_{p^k}$):

$$\mathbb{G}_T = \mu_r := \left\{x \in \mathbb{F}_{p^k} : x^r = 1\right\}.$$

The pairing e now maps two elliptic curve points $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$ to an element of $\mathbb{G}_T$ by evaluating $e(P,Q) = f_{r,P}(Q)^{(p^k-1)/r}$, where f is defined to be a function with r zeros at P, one pole at rP, and $r-1$ poles at ∞, which can be expressed as (see [26]):

$$(f_{r,P}) = r(P) - (rP) - (r-1)\infty$$

One of main reasons why the Tate pairing has become so widespread is that it is efficiently computable using *Miller's algorithm* [10] detailed in Listing 2.2, where $l_{A,B}$ is the function defining the line passing through A and B (or the tangent line if $A = B$), s.t. Q is on that line if and only if $l_{A,B}(Q) = 0$.

Listing 2.2: Compute $e(P,Q) = f_{r,P}(Q)^{(p^k-1)/r}$ via the Miller algorithm

```
1  tateMillerAlgorithm(r = (r_L r_{L-1} ... r_0)_2, P, Q):
2    T ← P
3    f ← 1
4    for i from L-2 downto 0:   # "Miller loop"
5      f ← f^2 · l_{T,T}(Q)
6      T ← 2T
7      if r_i = 1:
8        f ← f · l_{T,P}(Q)
9        T ← T + P
10   return f^{(p^k-1)/r}   # final exponentiation
```

3 Optimization ideas

While Chapter 2 summarized structures and algorithms that were already implemented in the `upb.crypto.math` library, this chapter is about ideas for new algorithm implementations that might improve the library's performance.

In Section 3.1 we will describe how the performance of the basic elliptic curve operations (addition and doubling) can be significantly improved by working with other coordinate systems. In Section 3.2 we present the multiple windowed approaches for fast exponentiation in multiplicative groups (which is equivalent to scalar multiplication in elliptic curve groups), and in Section 3.3 we will expand on this topic by describing more involved strategies that can be employed to further increase performance when computing the product of powers of group elements.

Section 3.4 details another type of pairing, the *ate pairing*, which can be used in order to improve the performance of Miller loop, and Section 3.5 closes with some ideas about how parts of the existing code from the `upb.crypto.math` library could be rewritten in a more efficient manner.

When determining the costs of certain operations in this chapter, we will ignore field additions/subtractions and multiplications with small constants as those operations induce only marginal costs compared to field inversions, squarings and general multiplications.

3.1 Projective and Jacobian coordinates

The two-dimensional representation $P = (x, y)$ of a an elliptic curve point P, as introduced in Section 2.2, is also referred to as *affine coordinates*. A major disadvantage when working with affine coordinates is that any addition or doubling operation requires one division, which is equivalent to a costly field inversion, followed by a multiplication. Field inversions, which are usually computed using the *extended Euclidean algorithm*, are commonly estimated to require as much computation as around 50-100 general field multiplications (see [22]). It's thus desirable to find a way to avoid having to perform inversions for each operation.

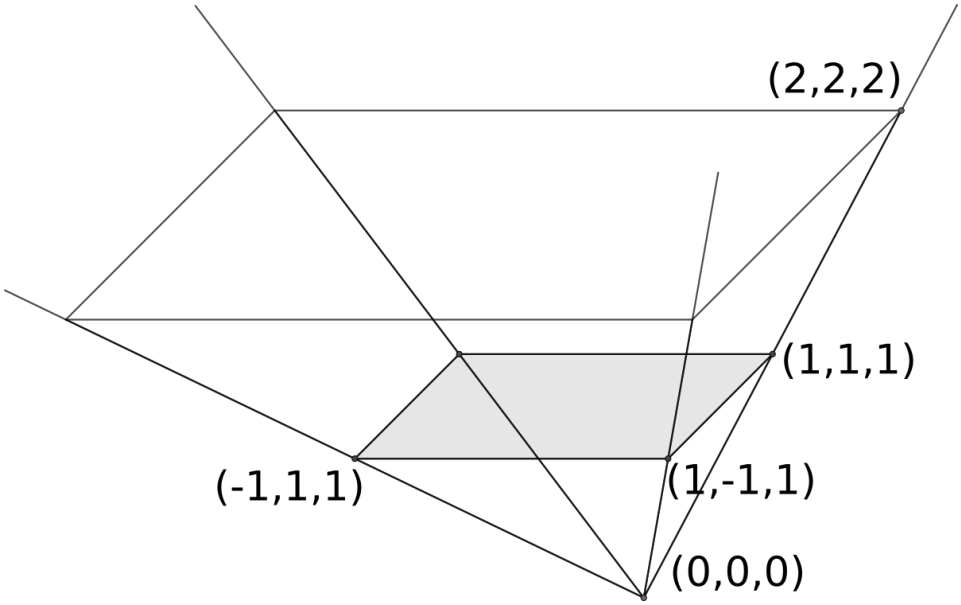
Various alternative coordinate systems have been proposed, many of which share the idea of adding a third coordinate Z to each point, which essentially "accumulates" all necessary divisions. This way, if you perform multiple addition/doubling operations in a row (as is the case when computing scalar multiples), you will only need at most one division, at the end, when you transform the point back to affine coordinates. In case you don't require affine coordinates as an end result, no divisions at all are needed. Two of the most common coordinate systems are projective coordinates and Jacobian

coordinates:

Projective coordinates A projective point $P = (X, Y, Z)$, with $Z \neq 0$, represents the affine point $(x, y) = (X/Z, Y/Z)$. In Figure 3.1 you can see how the projective point $(2, 2, 2)$ is equivalent to the point $(1, 1)$ in the affine plane with $Z = 1$ (i.e. the line from the origin to $(2, 2, 2)$ intersects the affine plane at point $(1, 1, 1)$).

Jacobian coordinates A Jacobian point $P = (X, Y, Z)$, with $Z \neq 0$, represents the affine point $(x, y) = (X/Z^2, Y/Z^3)$. Using the square and cube of Z here instead of using Z directly – as it is done with projective coordinates – allows for certain optimizations when doubling a point (see Section 3.1.5).

Figure 3.1: Projective coordinates and the affine plane (with $Z = 1$)



In the following sections we will describe how to implement transformation, addition and doubling operations in these alternative coordinate systems. In each case, only the most efficient implementation we have found from various sources is mentioned.

Section 3.1.7 contains a final overview of all operations mentioned here, and their respective costs.

3.1.1 Transformations

An affine point (x, y) can be mapped to the equivalent projective/Jacobian point $(x, y, 1)$ without any further operations required. On the other hand, in order to transform a projective point (X, Y, Z) to an affine point $(x, y) = (X/Z, Y/Z)$ requires two divisions by Z. This can be implemented by first computing the inverse Z^{-1} and then doing two multiplication: $(x, y) = (X \cdot Z^{-1}, Y \cdot Z^{-1})$.

Similarly, transforming a Jacobian point (X, Y, Z) to an affine point $(x, y) = (X/Z^2, Y/Z^3)$ also requires a field inversion to compute Z^{-1}. Before we can multiply with X and Y, however, we need one squaring to compute Z^{-2}, and one multiplication to get Z^{-3}. Thus, one inversion, one squaring, and three general multiplications in total.

A projective/Jacobian point (X, Y, Z) with $Z = 1$ is called *normalized* and can be transformed to the equivalent affine point (X, Y) without requiring any further operations. If two projective or Jacobian points P and Q have the same coordinates when normalized, we write $P = Q$.

3.1.2 Projective addition

For this and the following sections, we will assume we want to add the projective or Jacobian points $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$ to get the sum $R = P + Q = (X_3, Y_3, Z_3)$, with $Q = P$ in case of doubling. As described in [8], we can compute the sum of two projective points in the following manner:

$$
\begin{aligned}
W_X &\leftarrow X_1 Z_2, & W_Y &\leftarrow Y_1 Z_2, & W_Z &\leftarrow Z_1 Z_2,\\
u &\leftarrow Y_2 Z_1 - W_Y, & u_2 &\leftarrow u^2, & &\\
v &\leftarrow X_2 Z_1 - W_X, & v_2 &\leftarrow v^2, & v_3 &\leftarrow v_2 v,\\
R &\leftarrow v_2 W_X, & A &\leftarrow u_2 W_Z - v_3 - 2R, & &\\
X_3 &\leftarrow vA, & Y_3 &\leftarrow u(R - A) - v_3 W_Y, & Z3 &\leftarrow v_3 W_Z
\end{aligned}
$$

This approach requires 12 multiplications, 2 squarings. Even though no divisions were performed, the resulting projective point is equivalent to the affine point we would have computed if we had normalized the coordinates first and then applied the normal affine addition as described in Section 2.2.3; for example, for the X-coordinate we have:

$$
\begin{aligned}
\frac{X_3}{Z_3} &= \frac{vA}{v^3 Z_1 Z_2}\\
&= \frac{u^2 Z_1 Z_2 - v^3 - 2v^2 X_1 Z_2}{v^2 Z_1 Z_2}\\
&= \left(\frac{u}{v}\right)^2 - \frac{X_2 Z_1 - X_1 Z_2}{Z_1 Z_2} - \frac{2X_1 Z_2}{Z_1 Z_2}\\
&= \left(\frac{Y_2 Z_1 - Y_1 Z_2}{X_2 Z_1 - X_1 Z_2}\right)^2 - \frac{X_2}{Z_2} + \frac{X_1}{Z_1} - \frac{2X_1}{Z_1}\\
&= \left(\frac{Y_2/Z_2 - Y_1/Z_1}{X_2/Z_2 - X_1/Z_1}\right)^2 - \frac{X_1}{Z_1} - \frac{X_2}{Z_2} = x_3
\end{aligned}
$$

With $x_1 \coloneqq \frac{X_1}{Z_1}, y_1 \coloneqq \frac{Y_1}{Z_1}, x_2 \coloneqq \frac{X_2}{Z_1}$ and $y_2 \coloneqq \frac{Y_2}{Z_2}$ being the normalized affine coordinates, this equivalent to the affine addition operation. A similar computation for the Y-coordinate would show that $\frac{Y_3}{Z_3} = y_3$.

Detecting special cases

For the above procedure to work, we need P and Q to have different X-coordinates when normalized, because otherwise v would be zero, resulting in an invalid point with $Z_3 = 0$:

$$\frac{X_1}{Z_1} = \frac{X_2}{Z_2} \Leftrightarrow X_1 Z_2 = X_2 Z_1 \Leftrightarrow v = X_2 Z_1 - X_1 Z_2 = 0 \Leftrightarrow Z_3 = 0$$

Similarly, we can detect if the two points have different Y-coordinates by looking at u:

$$\frac{Y_1}{Z_1} = \frac{Y_2}{Z_2} \Leftrightarrow Y_1 Z_2 = Y_2 Z_1 \Leftrightarrow u = Y_2 Z_1 - Y_1 Z_2 = 0$$

Since both of these values are computed as part of the normal projective addition procedure, no significant overhead is caused in cases where $v \neq 0$ and we just continue with the general addition procedure. If $v = 0$ and $u \neq 0$, we have a vertical line, and we can immediately return ∞ without any further costs. If, on the other hand $v = 0 = u$, we have $P = Q$ and we must perform a doubling operation, which would require all the costs associated with that, in addition to the five already performed multiplications. It is, however, very unlikely to encounter a doubling when a general multiplication was intended (e.g., for the "double-and-add"-algorithm, we would just call the doubling procedure directly, skipping these five unnecessary multiplications), thus that special case can be ignored when determining the average costs of the general projective addition operation.

A very similar approach for detecting special cases can also be employed for addition of Jacobian coordinates.

3.1.3 Projective doubling

[22] details a efficient method for doubling a projective point P, requiring only 11 multiplications/squarings, by performing following operations in the given order:

$$\begin{aligned}
X_{11} &\leftarrow X_1^2, & Z_{11} &\leftarrow Z_1^2, & & \\
w &\leftarrow aZ_{11} + 3X_{11}, & s &\leftarrow 2Y_1 Z_1, & s_2 &\leftarrow s^2, \\
R &\leftarrow Y_1 s, & R_2 &\leftarrow R^2, & & \\
B &\leftarrow (X_1 + R)^2 - X_{11} - R_2, & h &\leftarrow w^2 - 2B, & & \\
X_3 &\leftarrow hs, & Y_3 &\leftarrow w(B - h) - 2R_2, & Z_3 &\leftarrow s \cdot s_2
\end{aligned}$$

This approach requires 5 multiplications and 6 squarings.

3.1.4 Jacobian addition

[15] gives the following procedure for computing the sum of two Jacobian points:

$$
\begin{aligned}
Z_{11} &\leftarrow Z_1^2, & Z_{22} &\leftarrow Z_2^2, & & \\
U_1 &\leftarrow X_1 Z_{22}, & U_2 &\leftarrow X_2 Z_{11}, & H &\leftarrow U_2 - U_1, \\
S_1 &\leftarrow Y_1 Z_2 Z_{22}, & S_2 &\leftarrow Y_2 Z_1 Z_{11}, & r &\leftarrow 2 \cdot (S_2 - S_1), \\
I &\leftarrow (2H)^2, & J &\leftarrow HI, & V &\leftarrow U_1 I, \\
X_3 &\leftarrow r^2 - J - 2V, & Y_3 &\leftarrow r \cdot (V - X_3) - 2S_1 \cdot J, & & \\
Z_3 &\leftarrow ((Z_1 + Z_2)^2 - Z_{11} - Z_{22}) \cdot H & & & &
\end{aligned}
$$

This approach requires 11 multiplications and 5 squarings, thus two operations more than were necessary for projective addition.

3.1.5 Jacobian doubling

[15] also shows the following very efficient doubling procedure for an Jacobian point P:

$$
\begin{aligned}
X_{11} &\leftarrow X_1^2, & Y_{11} &\leftarrow Y_1^2, & Z_{11} &\leftarrow Z_1^2, \\
F &\leftarrow Y_{11}^2, & S &\leftarrow 2 \cdot ((X_1 + Y_{11})^2 - X_{11} - F), & & \\
M &\leftarrow 3X_{11} + aZ_{11}^2, & X_3 &\leftarrow M^2 - 2S, & & \\
Y_3 &\leftarrow M(S - X_3) - 8F, & Z_3 &\leftarrow (Y_1 + Z_1)^2 - Y_{11} - Z_{11} & &
\end{aligned}
$$

This approach requires just one multiplications and 8 squarings, which is two operations less than required for doubling using projective coordinates. Also, almost all operations here are squarings, which are generally preferable to multiplications as they can be performed slightly faster for large numbers (see [7]).

3.1.6 Mixed additions

The procedures described above work for any projective/Jacobian points, without further restrictions. Sometimes however, we know that the points adhere to certain constrains, which might allow for some optimizations when computing additions (or doublings). In particular, we might know that one of the points (w.l.o.g. the second) is normalized (i.e. $Z_2 = 1$). Since the second point is now essentially in affine coordinates, such an addition operation is also called "mixed" (see [8]), as we are working with points from two different coordinate systems.

[22] and [15] show how to implement mixed additions requiring only 9 multiplications and 2 squarings (if P is a projective point) and 7, multiplications and 4 squarings (if P is given in Jacobian coordinates). The algorithms are essentially equivalent to those presented in Sections 3.1.2 and 3.1.4, just that multiplications with Z_2 are simply left out.

Table 3.1: Overview of costs of various elliptic curve point operations (S=squaring, M=general multiplication, I=field inversion, C=50I+M+S)

Operation	I	M	S	C
affine addition	1	2	1	53
affine doubling	1	2	2	54
projective to affine transformation	1	2	0	52
Jacobian to affine transformation	1	3	1	54
projective addition	0	12	2	14
projective doubling	0	5	6	11
Jacobian addition	0	11	5	16
Jacobian doubling	0	1	8	9
mixed addition (projective+affine)	0	9	2	11
mixed addition (Jacobian+affine)	0	7	4	11

3.1.7 Comparison of costs

Taking the costs for affine coordinates (discussed in Sections 2.2.3 and 2.2.4) and the numbers from the previous sections, we might want to compare these operations with each other, ideally based on a simple cost function $C(I, M, S)$, where I, M, and S represent the number of field inversions, general multiplications, and squarings, respectively.

With a conservative estimate of an inversion being equivalent to about 50 multiplications, and assuming – for simplicity – that squarings cost about as much as general multiplications ($S = M$), we would get $C(I, M, S) = 50I + M + S$, giving the final results presented in Table 3.1.

A few things of note:

- As projective/Jacobian still require on field inversion in the end in order to transform the point to affine coordinates, if only a single doubling or addition operation is needed, staying in the affine coordinate system is most efficient. However, the numbers from Table 3.1 suggest that as soon as you want to perform more than one doubling or addition operation in a row, working in either of the three-dimensional coordinate systems will be significantly faster.

- While the normal projective addition operation is faster than the Jacobian one, the Jacobian coordinates perform significantly better in regards to the doubling operation. When performing scalar multiplication (via Algorithm 2.1), doublings are expected to be performed about twice as often as additions; thus, it is to

be expected that in these situations, Jacobian coordinates are faster than projective ones. This advantage should be even more pronounced when further optimizations are employed: Using mixed coordinates would make Jacobian additions about as fast as projective additions, while sliding-window techniques (see Section 3.2) would further decrease the number of required additions. With multiexponentiation evaluations that share the doubling operation though (see Section 3.3), projective coordinates might perform again better.

- Even if the point Q is not already normalized, mixed additions might still be preferable to normal projective additions in cases where you perform many additions with Q in a row, e.g. when computing a scalar multiple kQ. If k is large enough, the savings from using the cheaper addition procedure could compensate for the costly normalization (i.e. one inversion and some multiplications), which is required in the beginning.

3.2 Windowed exponentiation

As mentioned in Section 2.2.6, the "double-and-add"-algorithm for computing a scalar multiple $kP \in \mathbb{G}_A$ with $k \in \mathbb{Z}_{\geq 1}$ in a additive group $\mathbb{G}_A$ is equivalent to "square-and-multiply"-algorithm for computing the power $g^e \in \mathbb{G}_M$ in a multiplicative group $\mathbb{G}_M$. Many variations of that basic algorithm exist in the literature, most of them commonly written in multiplicative notation. Therefore, we will also employ the multiplicative notation within this section and the next (Section 3.3), where we will describe more general versions of the "double-and-add"/"square-and-multiply"-algorithm that can be used when exponentiating multiple bases at the same time.

3.2.1 2^w-ary exponentiation

Given a random element g from a group $\mathbb{G}$ and a random L-bit exponent $e = (e_{L-1}e_{L-2}\dots e_0)_2 \in \mathbb{N}$, the basic "square-and-multiply" algorithm would require L squarings and $L/2$ general multiplications on average (see [9, p. 147]).

The idea of the 2^w-ary exponentiation method is to reduce the number of required multiplications by precomputing x^d for every $d \in [0, 2^w - 1]$ and iterating through the bits of e in d-bit chunks F (referred to as *windows*), multiplying by the precomputed power x^{F_i} only if $F_i \neq 0$ (cf. [17]):

Listing 3.1: Compute x^e using the 2^w-ary method

```
1 2^w-ary-exponentiation(x,w, e=(e_{L-1}e_{L-2}...e_0)_2):
2   if w does not divide L:
3     pad e with zeros so that its new length L is a multiple
        ↪ of w
4   # precomputations:
5   X ← new Array  # indexed by [1...2^w - 1]
6   X[1] ← x
```

```
7   for i in 2...2^w − 1:
8     X[i] ← X[i-1] · x   # 2^w − 2 multiplications
9   R ← X[(e_{L−1}...e_{L−d})_2]
10
11  # main loop:
12  i ← L-w
13  while i > 0:
14    for j in 1...w:
15      R ← R^2
16    F ← (e_{i−1}...e_{i−w})_2
17    if F ≠ 0:
18      R ← R · X[F]
19    i ← i - w
20  return R
```

For example, with $w = 2$ and $e = (\underline{11}00\underline{11}00\underline{11}00\underline{11})_2$, only the four marked chunks would trigger a multiplication in this algorithm. Including the two multiplications for the precomputation of x^2 and x^3, and the 12 squarings we get a total of 18 operations. The normal square-and-multiply method, on the other hand, would require 13 squarings and 8 multiplications (for the 8 1s), thus 21 operations in total. In general, we need about 2^w multiplications (and 1 squaring) for the precomputation, and about L squarings and $L(1 - 2^{-w})/w$ multiplications during the evaluation phase (see [19]).

3.2.2 Basic sliding window exponentiation

The sliding window approach is very similar to the 2^w-ary exponentiation, the only difference being how the bit string is split into chunks: Instead of always taking exactly k bits per chunk, we skip an arbitrary number of zeros until we reach the first 1-bit. From there, we now take the longest chunk of length at most w which ends with a one. This process is repeated until we have processed the whole bit string (cf. [9, p. 150]):

Listing 3.2: Compute x^e using the basic sliding window method

```
1  sliding-window-exponentiation(x,w, e=(e_{L−1}e_{L−2}...e_0)_2):
2    # precomputations:
3    X ← new Array   # indexed by [1,3,5,...,2^w − 1] (even powers
       ↪ are not necessary)
4    X[1] ← x
5    x2 ← x·x   # one squaring
6    for i from 3 to 2^w − 1 step 2:
7      X[i] ← X[i-2]·x2   # 2^w/2 multiplications
8    R ← 1   # neutral element of 𝔾
9
10   # mail loop:
11   i ← L-1
```

```
12  while i ≥ 0:
13    if e_i = 0: # skip zeros
14      R ← R^2
15      i ← i - 1
16    else: # non-zero chunk starts
17      s ← max(i-w+1, 0)  # most rightmost possible end of
          ↪ chunk
18      while e_i = 0:
19        s ← s + 1  # search for first 1
20      for h from 1 to i-s+1:
21        R ← R^2  # shift result by length of chunk
22      R ← R · X[(e_i e_{i-1} ... e_s)_2]  # multiply by chunk value
23      i ← s - 1
24  return R
```

By allowing more flexibility on where the chunks start and end, the sliding window method will often result in fewer multiplications, for example with $w = 2$ and $e = (1110011001100111)_2$, the sliding window approach will result in only six non-zero chunks:

$$(\underline{11}\,\underline{1}\,00\,\underline{11}\,00\,\underline{11}\,00\,\underline{11}\,\underline{1})_2$$

while the 2^w-ary method would result in eight non-zero chunks:

$$(\underline{11}\,\underline{10}\,\underline{01}\,\underline{10}\,\underline{01}\,\underline{10}\,\underline{01}\,\underline{11})_2$$

A further advantage of this approach is that because we only consider non-zero chunks ending with a 1-bit, we don't need to compute the even powers of x during the precomputation phase. In total we have about $2^w/2$ multiplications (and one squaring) for the precomputation, about L squarings during the evaluation, as well as $L/(w+1)$ multiplications on average (see [19]).

3.2.3 Signed-digit methods

If inversions are cheap in $\mathbb{G}$ (as it is for example in elliptic curve groups), exponentiation algorithms employing divisions become interesting. For example, g^{63} could be computed as

$$g^{63} = g^{64}/g^1 = g^{(1000000)_2} \cdot g^{-1},$$

which can be computed very quickly since 64 is a power of two. In the following sections we will assume that the costs of inversions are at most as expensive as additions/subtractions and therefore similarly ignore them when summarizing the costs for exponentiations.

Binary NAF

The *signed binary representation* consists only of the digits $0, 1$ and $\overline{1} = -1$. In general, this representation is not unique (e.g. $(111)_2 = 4 + 2 + 1 = 7 = 8 - 1 = (100\overline{1})_2$). However, when we add the restriction that two adjacent digits cannot both be non-zero, there is a unique representation for every number (see [12, p. 98]). This is called the non-adjacent form (NAF) for that number.

The NAF can be computed by the following algorithm (cf. [9, p. 151]):

Listing 3.3: Compute the NAF representation for e

```
1 computeNAF(e=(e_{L-1}e_{L-2}...e_0)_2):
2   e_L ← 0  # pad e with a zero on the left
3   c ← 0
4   R ← new Array  # indexed by [0...L]
5   for i in 0 to L:
6     d ← floor((c + e_i + e_{i+1}) / 2)
7     R[i] ← c + e_i - 2·d
8     swap c and d
9   return (R[L]R[L-1]...R[0])_NAF
```

For evaluating a power x^e with e being in non-adjacent form, our sliding-window Algorithm 3.2 can be used, requiring L squarings and $L/3$ multiplications on average (see [12, p. 98]). Since the computation of the NAF representation doesn't require any expensive operations, the costs for precomputations remain about the same.

wNAF

A more general signed-digit representation of an exponent e is the width-w non-adjacent form (wNAF). A sequence $(n_{L-1}n_{L-2}...n_0)$ is the wNAF-representation of e if and only if (see [19]):

- every $n_i \in \mathbb{Z}$ is either zero or odd
- $|n_i| < 2^{w-1}$ for all i
- $e = \sum_{i=0}^{L-1} n_i 2^i$
- at most w consecutive digits are non-zero

Due to the last constraint, many digits in the wNAF are zero, reducing the number of required multiplications. The wNAF can be computed by the following algorithm (cf. [9, p. 153]):

Listing 3.4: Compute the wNAF representation for e

```
1 compute-w-NAF(w, e=(e_{L-1}e_{L-2}...e_0)_2):
2   i ← 0
3   R ← new Array  # indexed by [0...L-1]
```

```
4   while e > 0:
5     if e is odd:
6       R[i] ← n mod 2^w
7       e ← n - R[i]
8     else:
9       R[i] ← 0
10    e ← e / 2
11    i ← i + 1
12  return (R[L-1]R[L-2]...R[0])_wNAF
```

For evaluating the computed wNAF, the original square-and-multiply algorithm can be used, with the modification that instead of multiplying with base g^1, we need a multiplication with the precomputed power g^{e_i}. As summarized in [19], the precomputations require again one squaring and $2^w/2$ multiplications, while the evaluation requires L squarings but only $L/(w+2)$ multiplications on average.

3.2.4 Comparison of costs

Here is a table summarizing the costs of the presented exponentiation methods:

Table 3.3: Overview of expected costs of various exponentiation methods; S=squaring, M=general multiplication

Method	Precomputation	Evaluation
Basic square-and-multiply	-	$LS + (L/2)M$
2^w-ary exponentiation	$\approx 1S + 2^w M$	$\approx LS + (L(1-2^{-w})/w)M$
Basic sliding window method	$\approx 1S + (2^w/2)M$	$\approx LS + (L/(w+1))M$
Signed-digit method (binary NAF)	$\approx 1S + (2^w/2)M$	$\approx LS + (L/3)M$
Signed-digit method (wNAF)	$\approx 1S + (2^w/2)M$	$\approx LS + (L/(w+2))M$

3.3 Multiexponentiation

A multiexponentiation is a product of $r \geq 2$ powers in multiplicative groups:

$$g_1^{e_1} g_2^{e_2} \dots g_r^{e_r} = \prod_{i=1}^{r} g_i^{e_i},$$

which is equivalent to sums of scalar multiples in additive groups:

$$e_1 g_1 + e_2 g_2 \dots e_r g_r = \sum_{i=1}^{r} e_i g_i.$$

Just as in the previous section though, we will stick with the first notation, as it is more common in the literature.

In various cryptographic schemes, including the sequential aggregate signature scheme described in Section 2.3.2, multiexponentiations with two or more bases are computed. For simplicity, let's assume that all exponents e_i have the same bit-length L. (If this is not the case, the shorter exponents can be padded with zeros.) A trivial evaluation would be to compute all powers separately (requiring at least L squarings each) and then multiplying the results ($r - 1$ further multiplications) in order to get the final result. Clearly, this is not yet optimal; thus, we will discuss more efficient alternatives in the following sections.

First, the next section will describe the most effective optimization for multiexponentiation computation. Afterwards we will go into detail in regards to two categories of further optimizations: Simultaneous methods, where precomputations of products of different bases (with various powers) are computed, and interleaved methods where the precomputed powers for each base are calculated separately, thus allowing for more flexibility.

3.3.1 Basic idea: merging the squarings

A major optimization is to perform the squarings not for each base separately, but instead for all bases in parallel. Following example illustrates the idea:

$$a^5 b^6 c^3 d^7 = ((abd)^2 bcd)^2 acd$$

Rather than computing the powers for all bases separately, this way all required squaring operations are only performed once, reducing the overall cost to a fraction. Algorithm 3.5 shows how this method can be implemented:

Listing 3.5: Compute multiexponentiation with merged squarings

```
1 computeMultiExpoBasic(x=(x_1, ..., x_r), e=
    ↪ (e_1, ..., e_r) = ((e_{1,L-1}e_{1,L-2}...e_{1,0})_2, ..., (e_{r,L-1}e_{r,L-2}...e_{r,0})_2)):
2   R ← 1   # neutral element of 𝔾
3   for i in L-1 downto 0:
4     R ← R^2
5     for j in 1 to r:
6       if e_{j,i} = 1:
7         R ← R · x_j
8   return R
```

3.3.2 Simultaneous methods

Simultaneous methods (see [19]) precompute combinations of powers. Given a window size w, simultaneous methods will precompute and store all relevant products of bases powered to at most $2^w - 1$. In principle, all exponentiation techniques described in Section 3.2 should also work for simultaneous multiexponentiation.

As an example, here is how simultaneous 2^w-ary exponentiation would look like: It would be very similar to 2^w-ary single-exponentiation, but at the beginning, products of powers will be precomputed:

$$T_{E_1,\ldots,E_r} \coloneqq \prod_{i=1}^{r} g_i^{E_i} \text{ for all } (E_1,\ldots,E_r) \in [-(2^w-1), 2^w-1]^r \subseteq \mathbb{Z}^r$$

Then, exponentiation is performed "simultaneously", with block-size w. With $w = 2$ this might look like:

$$a^{62}b^{35}c^{20} = ((a^3b^2c)^4a^3c)^4a^2b^3 = (T_{3,2,1}^4 \cdot T_{3,0,1})^4 \cdot T_{2,3,0},$$

with e.g. $T_{3,2,1}$ being the precomputed power product $a^3b^2c^1$.

This procedure would require about 2^{rw} multiplications/squarings to precompute the 2^{rw} small power products, and additional $\frac{L}{w+2^{-r}}$ multiplications/squarings to compute the final result.

3.3.3 Interleaved methods

Just like with the simultaneous methods, basically all of the previously mentioned exponentiation algorithms from Section 3.2 could also be used for interleaved multiexponentiation.

In contrast to simultaneous methods, interleaved multiexponentiation methods only precompute powers of the bases separately, without any combinations (see [19]). The exponentiations of the various bases are thus very much independent from each other, they just share their squaring operation with each other.

This has multiple advantages:

- The number of required precomputations grows linear in r, rather than exponential, making it feasible to compute even multiexponentiations with more than 10 or 20 different bases.
- Since the precomputations are done for each base separately, precomputed powers could be reused for further multiexponentiations that share some (but not necessarily all) of the bases with the first multiexponentiation.
- Different window sizes could be chosen for different base/exponent-pairs, allowing for choosing the right parameter for each individual exponentiation.

As an example, here is how an interleaved sliding windows approach would look like for computing $a^{e_1}b^{e_2}$ with the two different window sizes $w_1 = 2, w_2 = 1$:

$$a^{e_1}b^{e_2} = a^{9831}b^{15} = a^{10011001100111_2}b^{1111_2} = (((((((a)^{16}G_{1,3})^{16}G_{1,3})^4b)^2b)^2G_{1,3}b)^2ab$$

with $G_{1,3} \coloneqq a^3$ having been precomputed. Assuming – for simplicity – all bases have the same bit length L and the same window size w is used for each base, interleaved sliding window multiexponentiation as described in [19] requires only $r2^{w-1}$ multiplications/squarings for precomputing the powers, and about $\frac{Lr}{w+1}$ multiplications/squarings for the evaluation itself.

3.3.4 Comparison

The costs of the sliding window examples above for the simultaneous and the interleaving approach are similar also when using other exponentiation algorithms like 2^w-ary exponentiation or wNAF (see [19]). In general, it holds that the simultaneous approach will consist of a precomputation requiring time and space that grows exponentially in r, and an evaluation phase which doesn't significantly increase as r grows. The interleaving approach, on the other hand, will result in a precomputation phase with space and time costs that are linear in r and an evaluation phase with costs that are linear in r as well.

Thus, if the number of bases r is rather small and resources for the heavy precomputation phase are available, the simultaneous approach should be used, in particular if the set of bases will used more than one multiexponentiation. If on the other hand, r is rather large, and the specific set of bases will not be used for further multiexponentiations, then the interleaving approach should be used, due to it's cheaper precomputation phase.

3.4 Ate pairing

Multiple variations of the Tate pairing (described in Section 2.3.3) have been devised in order to improve its runtime. One of these variations is the *Ate pairing*, which uses a different source set G_2, which reduces the required number of iterations of the Miller loop (Algorithm 2.2).

Formally, the Ate pairing is a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ defined as:

$$\begin{aligned}\mathbb{G}_1 &\coloneqq E(\mathbb{F}_{p^k})[r] \cap \ker(\pi_p - [1]) = E(\mathbb{F}_p)[r]\\ \mathbb{G}_2 &\coloneqq E(\mathbb{F}_{p^k})[r] \cap \ker(\pi_p - [p])\\ \mathbb{G}_T &\coloneqq \mu_r\end{aligned}$$

with $\pi_p((x, y)) \coloneqq (x^p, y^p)$ being the *Frobenius endomorphism*, and $[x]$ representing the scalar multiplication function.

This definition of the source groups has the interesting property that $E(\mathbb{F}_{p^k})[r] = \mathbb{G}_1 \oplus \mathbb{G}_2$ (see [20, p. 25]), i.e. each point $Q \in E(\mathbb{F}_{p^k})[r]$ can be represented as a sum

$Q_1 + Q_2$ with $Q_1 \in \mathbb{G}_1$ and $Q_2 \in \mathbb{G}_2$.

Since $\mathbb{G}_2$ has changed compared to its definition for the Tate pairing, we need to find a generator for it in order to be able to generate random elements for it (which is needed for procedures like the SRS signatures discussed in Section 2.3.2).

Given a point Q chosen randomly from $E(\mathbb{F}_{p^k})[r]$ (which can be represented as $Q = Q_1 + Q_2$ with $Q_1 \in \mathbb{G}_1, Q_2 \in \mathbb{G}_2$), we can find a generator $R \in \mathbb{G}_2$ by computing

$$\begin{aligned}\pi_p(Q) - Q &= \pi_p(Q_1) + \pi_p(Q_2) - Q_1 - Q_2 \\ &= Q_1 + pQ - Q_1 - Q_2 \\ &= (p-1)Q_2 =: R \in \mathbb{G}_2\end{aligned}$$

with $\pi_p(Q_1) = Q_1$ and $\pi_p(Q_2) = pQ_2$ because they are in the kernel of $\pi_p - [1]$ and $\pi_p - [p]$, respectively. And we have $R = (p-1)Q_2 \in \mathbb{G}_2$ because $Q_2 \in \mathbb{G}_2$.

The pairing function itself can be evaluated just like for the Tate pairing, using the Miller function f. However, now it suffices to pass $t-1$ instead of r as the loop parameter, and P and Q need to be swapped (see [3]):

$$e(P, Q) = f_{t-1,Q}(P)^{(p^k-1)/r}$$

.

As we've seen in Section 2.2.7, we have $t \leq 2\sqrt{p^k}$ for the trace of Frobenius t. Since then number of iterations for the Miller loop depends on the number of bits in the given loop parameter, the cost for the Miller loop is essentially halved, which might significantly improve the performance of this approach, depending on how costly the final exponentiation step is in comparison to the Miller loop.

3.5 Implementation-specific optimizations

We have so far described algorithmic optimizations at great lengths, but in practice, specific implementation details can easily be just as relevant for the final performance.

When running elliptic curve point operation tests, a performance analysis with the VisualVM profiler [27] showed that the `BigInteger.mod` call in the constructor of the class `ZnElement` (which represents numbers from $\mathbb{F}_p$) is a major hotspot, with runtime costs exceeding even the total costs of all field multiplications performed elsewhere in the `upb.crypto.math` library during that test.

A simple optimization would be to only call the `BigInteger.mod` if the value `v` is actually outside of the range $[0, p-1]$. Elliptic curve point addition and doubling consists of numerous field subtractions, the result of which should on expectation be within the range $[0, p-1]$ in about 50% of cases, which might lead to a significant speedup when the `BigInteger.mod` calls are skipped.

Another cause of suboptimal performance might be due to the complex structure of the `upb.crypto.math` library: There is a deep inheritance tree with numerous wrapper functions that often do little else than calling another method from a superclass. Also,

there are many type cast, which could potentially be avoided. A flatter class hierarchy, which fewer method calls and less type casts might significantly improve the performance.

Implementation and evaluation of these two optimization ideas will be discussed in Sections 4.5 and 5.6, respectively.

4 Implementation

In this chapter, we discuss how we implemented the ideas presented in the previous chapter and integrated the code into the existing `upb.crypto.math` library.

First, Section 4.1 will give an overview about the general architecture of the existing code, and at which places the new code was added. Then, Sections 4.2 and 4.3 describe details of the implementation of the new elliptic point operations and (multi)exponentiation algorithms, while Section 4.4 describes how the Ate pairing was implemented.

Next, Section 4.5 describes a mostly separate implementation that avoids the complex existing class hierarchy, Section 4.6 delves into the intricacies of testing the correctness and performance of the new code, and Section 4.7 closes with a short summary of how tests can be run on Android as well.

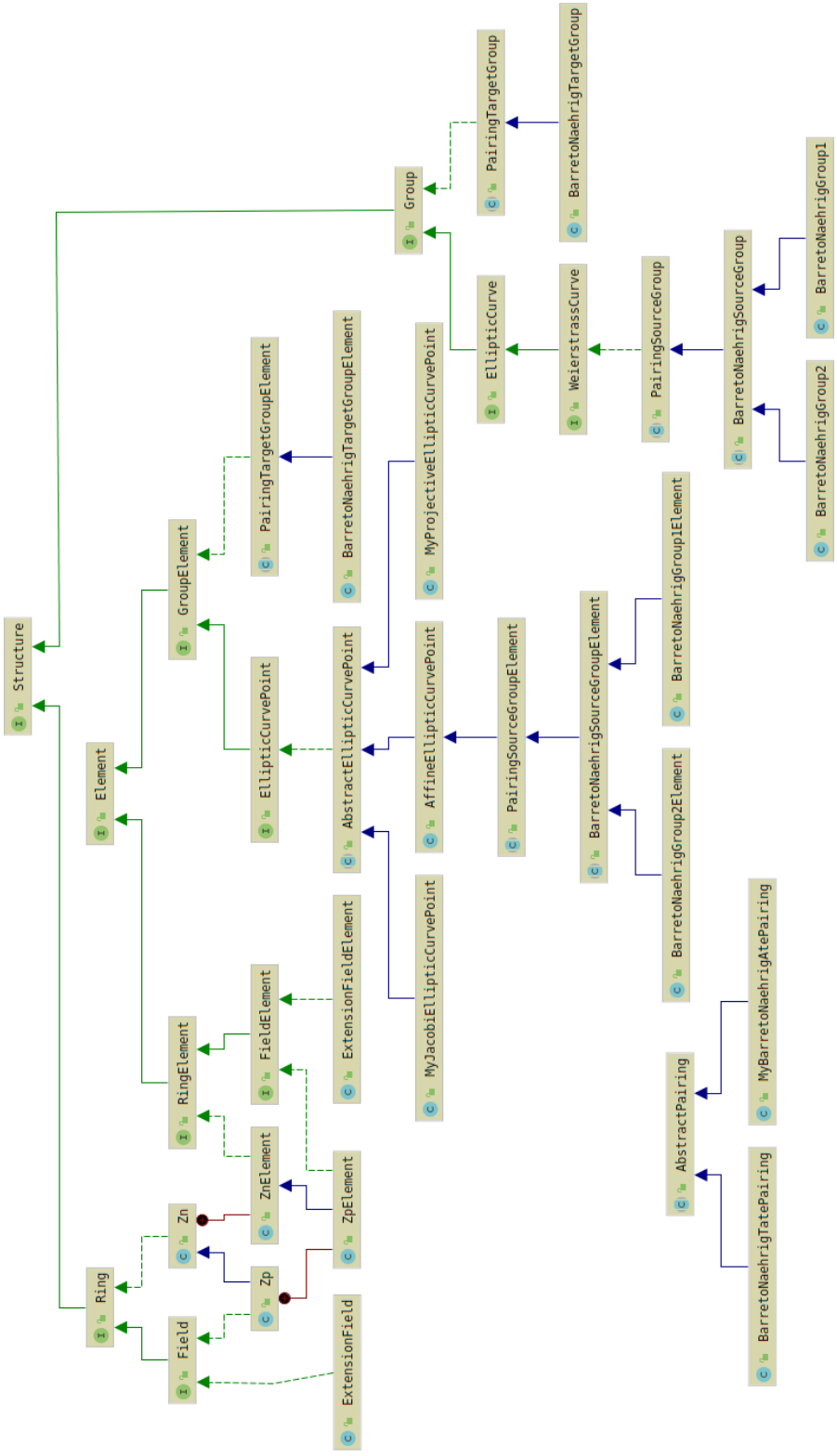
4.1 General architecture

In Figure 4.1 we can see an overview over the hierarchy of the most relevant classes from the `upb.crypto.math` library, including new ones we have added over the course of writing this thesis. At the very top we have the `Structure` interface that represents any algebraic mathematical structure, with declared methods for getting the size of the structure and taking an element of it uniformly at random. Subinterfaces are `Group` and `Ring`, declaring – among other things – methods for getting neutral elements and for inquiring whether or not they are commutative.

Further down the `Ring` tree we have the `Field` interface with the implementing classes `Zp` (representing $\mathbb{F}_p$) and `ExtensionField` (representing $\mathbb{F}_{p^k}$). Meanwhile, in the `Group` subtree we have the interface `WeierstrassCurve` declaring methods for getting the parameters defining a Weierstrass curve, and its subclass `PairingSourceGroup` which is the base class for the source groups used for bilinear pairings, containing attributes for e.g. the generator, cofactor, and size of the source group.

Parallel to the tree for these structures, we also have a separate hierarchy for the elements of these structures: `Element` as the base interface, and then `GroupElement`, `RingElement`, `FieldElement`, `ZpElement` (which is an inner class of `Zp`), and `ExtensionFieldElement`, in a hierarchy equivalent to the structures they belong to. These interfaces/classes contain methods for actually performing valid mathematical operations on them, e.g. `GroupElement` has the `op` method for performing the group operation, while `ZpElement` contains the methods `add`, `sub`, `mul`, `inv`, and `pow`, for addition, subtraction, etc.

Figure 4.1: Inheritance hierarchy of the relevant classes



Down the `GroupElement` tree we have the abstract base class `AbstractEllipticCurvePoint` with attributes for the X, Y, and Z coordinates and declared methods for adding two points and computing the line equation for the line passing through two given points (required for the miller algorithm 2.2). Subclasses which actually implement these methods are `AffineEllipticCurvePoint`, which was already part of the `upb.crypto.math` library), as well as `MyProjectiveEllipticCurvePoint` and `MyJacobiEllipticCurvePoint` which we implemented in order to improve the performance of the elliptic curve point operations. The prefix `"My"` was chosen in order to make clear which components where added over the course of writing this thesis, and can be removed once they are integrated into the main branch of the library.

Subclasses for the elements of the source groups for pairings over BN curves inherit from `AffineEllipticCurvePoint`. Pairing computations with projective or Jacobian coordinates are not currently supported.

Separate from the inheritance trees of the `Structure` and `Element` interfaces, we have the `AbstractPairing` class (defining the miller function, among other things) with the subclass `BarretoNaehrigTatePairing` for the previously implemented Tate pairing, and the new class `MyBarretoNaehrigAtePairing` for the Ate pairing, which is equivalent to the Tate pairing class except that it passes different parameters to the miller function (P and Q are swapped, and $t-1$ is passed as loop parameter instead of r), and it computes an appropriate generator as described in Section 3.4.

Classes for single- and multiexponentiation, as well as for the simplified implementation do not inherit from any existing class and are therefore not included in Figure 4.1. Instead, they are described in more detail in Sections 4.3 and 4.5.

4.2 Elliptic curve point operations

In this section, we'll discuss how we implemented the elliptic curve point operations in projective and Jacobian coordinates.

For projective coordinates, all logic for the implementation of the curve point operations can be found in the class `MyProjectiveEllipticCurvePoint`. It contains the methods `add` for addition, `square` for doubling (its name inspired by the multiplicative notation, as it overrides a method from the `GroupElement` interface which already used the multiplicative notation also for the `pow` method), and `addAssumingZ2IsOne` which performs a mixed addition (it assumes that the second point's Z coordinate is equal to 1). All these methods essentially just implement the mathematical procedures as outlined in Sections 3.1.2 and 3.1.3.

The perhaps most interesting part of the implementation is how the `add` method deals with special cases, which is shown in Listing 4.1: If `this` or `Q` is the neutral element, the other point is returned (since $P + \infty = P = \infty + P$). If `Q` is normalized (i.e. its z-coordinate is 1), the specialized `addAssumingZ2IsOne` is called.

At the very beginning, we check if the second point (`Q`) is identical to the first one (`this`), in which case we want to perform a doubling operation rather than continuing with the addition procedure. This simple reference equality checking is insufficient,

however. Even checking the actual coordinates separately for equality would not be enough, since two projective points with different coordinates can still represent the same (affine) point. Thus, as described in Section 3.1.2, after computing the first four temporary variables (`x1z2`, `v`, `y1z2`, and `u`), we can detect this special case, as well as the case that they lie on the same vertical line, by checking if `v` and/or `u` are zero. Since `u` and `v` are also required for the rest of the normal addition procedure, this implementation doesn't cause significant additional computational costs.

Listing 4.1: Corner case checking in the `MyProjectiveEllipticCurvePoint.add` method

```
1  public AbstractEllipticCurvePoint add(
       ↪ AbstractEllipticCurvePoint Q) {
2    if (Q == this)
3      return this.square();
4    if (Q.isNeutralElement())
5      return this;
6    if (this.isNeutralElement())
7      return Q;
8    if (Q.isNormalized())
9      return addAssumingZ2IsOne(Q); // mixed addition
         ↪ applicable
10   FieldElement x1z2 = x.mul(Q.z);
11   FieldElement v = Q.x.mul(z).sub(x1z2);
12   FieldElement y1z2 = y.mul(Q.z);
13   FieldElement u = Q.y.mul(z).sub(y1z2);
14   if (v.isZero()) { // x coordinates are equal
15     if (u.isZero()) { // y coordinates are also equal →
         ↪ double the point
16       return this.square();
17     }
18     // vertical line → sum is neutral element
19     return (AbstractEllipticCurvePoint)structure.
         ↪ getNeutralElement();
20   }
21   [...] // no special case applies; compute sum and return
       ↪ result point in the normal way
22 }
```

The implementation for Jacobian coordinates in the class `MyJacobiEllipticCurvePoint` is basically equivalent to the `MyProjectiveEllipticCurvePoint` class, except that the computation for addition and doubling uses the appropriate Jacobian formulas introduced in Sections 3.1.4 and 3.1.5.

4.3 Exponentiations

The next subsections describe the implementation details of the algorithms for single exponentiation (with only one base), and multiexponentiation, respectively.

4.3.1 Single exponentiation

The class `MySingleExponentiationAlgorithms` contains implementations of all the previously discussed single exponentiation methods: `simpleSquareAndMultiplyPow` implements the simple default square-and-multiply method, while the methods `powUsing2wAryMethodMethod`, `powUsingSlidingWindow`, and `powUsingWNafMethod` implement the corresponding advanced methods. Small powers of the base in question can be precomputed by corresponding static methods and are passed as arguments to the evaluation methods. Bit operations were used wherever possible, since they're very fast.

4.3.2 Multiexponentiation

In order to group a set of bases into an object so they can potentially be evaluated with different exponents multiple times in a convenient manner, we decided to create the class `MyBasicPowProduct` to represent such an expression. The previously existing class `PowProductExpression` served a similar purpose. However, it used very costly routines intended to optimize the evaluation order of the power product which slowed the total execution time down significantly. Also, it employed a `HashMap` for storing the bases which made it inconvenient for precomputing small power product combinations (for simultaneous multiexponentiation approaches) and might also be significantly slower than just using plain arrays, in particular because for projective and Jacobian points there is no trivial hashing function that would avoid a costly normalization.

The class `MyBasicPowProduct` contains a method `evaluate` which returns the result when computing the multiexponentiation with the exponents, which are passed as arguments. This `evaluate` method intentionally employs the trivial strategy for computing the multiexponentiation – computing the powers of each base separately and then multiplying the results – and is intended to be overridden by methods from subclasses implementing more advanced algorithms by employing more elaborate precomputations:

The subclass `MySimplePowProductWithSharedDoublings` simply overrides the `evaluate` function, replacing it with an simple implementation that share the doubling operation for all bases, without any precomputation; meanwhile, the subclasses `MyInterleavingSlidingWindowPowProduct` and `MySimultaneousSlidingWindowPowProduct` precompute the small powers required for the interleaving and the simultaneous sliding window approach, respectively, and efficiently compute the multiexponentiation in the `evaluate` method. For best performance, bit operations are used wherever possible.

4.4 Ate pairing

Implementation-wise, the Ate pairing is very similar to the Tate pairing and we've been able to reuse most of the existing code. Just like the existing class `BarretoNaehrigTatePairing`, our new class `MyBarretoNaehrigAtePairing` inherits from `AbstractPairing`, and the only major required changes consisted of swapping `P` and `Q`, and passing `t-1` instead of `r` when calling the Miller function. Additionally, the `evaluateLine` method needed to be appropriately adapted, which turned out to be a rather non-trivial task: It was not immediately obvious how elements of the source and target groups were represented, in particular because *sextic twists* were employed for increased efficiency, allowing an elliptic curve point P on E over $\mathbb{F}_{p^{12}}$ to be represented on a "twisted" curve E' by a point P' over the much smaller extension field $\mathbb{F}_{p^2}$. Twists are a very complex subject and outside the scope of this thesis. For a detailed account on twists, see [20, pp. 13-16].

After a thorough analysis of the existing code tough, we deduced that the used representation is the one described in [20, pp. 64-65]. Implementing the line functions for the Ate pairing as detailed in that paper finally led to a working pairing computation.

4.5 Simplified implementation

As discussed in Section 3.5, the existing library components might cause performance penalties due to the potentially unnecessary call `BigInteger.mod` that happen each time a new `ZpElement` is created, as well as by the complex class hierarchy with many type casts and long chains of method calls.

In order to evaluate the impact these aspects of the `upb.crypto.math` library have, we created an entirely independent class `MyProjectiveTriple` for performing rudimentary elliptic curve operations (in projective coordinates) by directly working with `BigIntegers` (while performing the modulo operation manually, only when necessary) and avoiding any overhead from unnecessary method calls. The implementation is limited to performing addition, doubling and scalar multiplication on projective elliptic curve points, since reimplementing all existing functionality in a simplified way would certainly be outside the scope of this thesis. But by comparing the performance of this simplified implementation with the original one that was integrated into the existing class hierarchy, we should get a good estimate about the general costs associated with it.

Listing 4.2 shows how the `MyProjectiveTriple` class implements the modulo operations when performing a point addition:

The private function `modp` manages the task of performing a modulo operation (for simplicity, `p` is saved an attribute of the instance) operation, only calling the expensive `BigInteger.mod` method when the argument `x` is outside the range $[0, p-1]$. (The first three lines of the `modp` function could be commented out, in order to mimic the behavior of the modulo operation performed by the `upb.crypto.math` library.)

The `add` function performs the point addition (checking of corner cases has been

removed to improve readability). Instead of performing a `modp` operation after each field operation, it only calls `modp` after each multiplication (in order to keep the values reasonable small), thereby saving even more modulo computations. Since we call `modp` for the result coordinates `rx`, `ry`, and `rz` though, the returned point is still correct.

Listing 4.2: Modulo and addition implementation in the `MyProjectiveTriple` class

```
1  private BigInteger modp(BigInteger x) {
2    if (x.compareTo(p) < 0 && x.signum() ≥ 0) {
3      return x;
4    }
5    return x.mod(p);
6  }
7
8  public MyProjectiveTriple add(MyProjectiveTriple q) {
9    BigInteger x1z2 = modp(x.multiply(q.z));
10   BigInteger v = modp(q.x.multiply(z)).subtract(x1z2);
11   BigInteger y1z2 = modp(y.multiply(q.z));
12   BigInteger u = modp(q.y.multiply(z)).subtract(y1z2);
13   BigInteger uu = modp(u.multiply(u));
14   BigInteger vv = modp(v.multiply(v));
15   BigInteger vvv = modp(v.multiply(vv));
16   BigInteger r = modp(vv.multiply(x1z2));
17   BigInteger z1z2 = modp(z.multiply(q.z));
18   BigInteger a = modp(uu.multiply(z1z2)).subtract(vvv).
       ↪ subtract(r.shiftLeft(1));
19   BigInteger rx = modp(v.multiply(a));
20   BigInteger ry = modp(u.multiply(r.subtract(a)).subtract(
       ↪ vvv.multiply(y1z2)));
21   BigInteger rz = modp(vvv.multiply(z1z2));
22   return new MyProjectiveTriple(p, curveParameterA, rx, ry,
       ↪ rz);
23 }
```

4.6 Testing

The next two subsections detail how the new code described in the last sections has been tested for correctness and efficiency.

4.6.1 Unit tests

There were already quite a few unit tests present in the `upb.crypto.math` library to begin with. In particular, the `GroupTests` and the `PairingTests` unit tests were very useful: The `GroupTests` unit tests were parameterized, so that we could check the

correctness of basic group operations of the new projective and Jacobian elliptic curve points by simply adding the corresponding structure to a list containing all groups that are to be tested; meanwhile, the `PairingTests` unit tests were copied to a separate file `AtePairingTests` in order to test the correctness of the Ate pairing.

In addition to these tests, we also created the entirely new unit test files `SingleExponentiationTests`, `MultiExponentiationTests`, and `ProjectiveTripleTests` in order to test the correctness of the new single- and multiexponentiation algorithms, and the independent `ProjectiveTriple` class.

In the beginning, these unit tests were also used to roughly measure the performance of the components under test, but all final evaluations used in this thesis were done with separate `main`-functions in the main code in order to allow for easier execution via Bash scripts and to avoid potential overhead from the IDE or testing framework.

4.6.2 Performance tests

In this section, we will discuss relevant aspects of the way we performed performance tests: Letting the JVM "warm-up" before doing our measurements, and automating the testing procedure by writing Bash scripts.

Warm-up

An important aspect to consider when measuring the performance of Java is the "warm-up" phase of the JVM. As discussed in [14], performance-heavy tasks like class loading and just-in-time (JIT)-compilation can occur in particular in the beginning of the run, skewing the results.

In order to avoid this problem, we decided to perform twice as many iterations than we intend to measure, and then discarding the measurements from the first half of them. This way, we can assume that the JVM has already been sufficiently warmed up by the time we start measuring (which is usually a couple seconds after the run started).

Bash scripts

Even if performing warm-up iterations as described in the previous sections there might still be undesired side-effects when running performance tests with different parameters right one after the other in the same JVM instance. For example, garbage collection of space used for the previous test round might decrease the performance of the test with the current parameters.

In order to make comparisons as fair as possible, we decided to run a fresh JVM instance for each parameter configuration we wanted to test. Because starting tests for each configuration manually would be too time-consuming, we wrote Bash scripts to automate these tests. Listing 4.3, for example, shows a Bash script that we used to measure the performance of simple multiexponentiation algorithms for different number of bases.

Listing 4.3: An example Bash script that executes performance tests for the simple multiexponentiation algorithms (with window size of 1 and no caching)

```
#!/bin/bash
runcommand="/usr/lib/jvm/java-8-openjdk-amd64/bin/java [
   ↪ arguments to set classpath, etc...] de.upb.crypto.math
   ↪ .swante.profiling.ThesisMultiExpo"
basepath="/[absolute path to library source code]/upb.crypto
   ↪ .math"
outputfile="outputs/out1.txt"
myrun() {
  echo $1
  eval "$runcommand $1" >> $outputfile
}
echo "outputs:" > $outputfile
iters="100" # number of repetitions
securityParam="128" # half of the curve's bit length
windowSize="1" # doesn't change for the simple algorithms
cacheSmallPowers="False" # small powers are never cached for
   ↪  the simple algorithms
for coordType in jacobi projective
do
    for algorithmIndex in 0 1 2
    do
      for numberOfBases in 1 2 3 4 5 6 7 8 9 10
      do
        myrun "$securityParam $coordType $numberOfBases
           ↪ $iters $windowSize $algorithmIndex
           ↪ $cacheSmallPowers"
      done
    done
done
```

4.7 Smartphone implementation

Considering the widespread use of mobile messaging apps like WhatsApp and Telegram, developers should make sure that the encryption algorithms they wish to employ also work on smartphones, which tend to have significantly less memory and computing power than PCs.

Given that this thesis is specifically aimed towards Java implementations, we decided to build a small Android app in order to test the performance of our elliptic curve algorithms. We started with a default app template using the Android Studio IDE [2], added the `upb.crypto.math` library, and then wrote performance tests similar to those

described above. Since running an app with different parameters via a Bash script is non-trivial, we decided to run all tests within a single launch of the app via the IDE instead. The computations are run in a separate background thread, which is started during the creation of the main GUI activity.

5 Evaluation

This chapter shows the most relevant results we got when evaluating the performance of the algorithms presented in the previous chapters.

Section 5.1 opens with a description of the environment in which the evaluations were run, including used hardware, software and curve parameters. Section 5.2 shows measurements for basic field operations, followed by Section 5.3 which compares different coordinate types with each other. Sections 5.4 and 5.5 deal with single- and multiexponentiation, respectively, Section 5.6 shows how much the performance of the existing code by employing a simplified implementation, and Section 5.7 compares the Ate pairing with the previously implemented Tate pairing. Lastly, Section 5.8 closes with a quick look at how well the code performs on a smartphone, in comparison to a laptop.

5.1 Set-up

Unless otherwise noted, all tests in this chapter are run on a Lenovo ThinkPad E580 Laptop (Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 16GB RAM), running the Linux Mint 19.1 operating system. The smartphone tests were executed on a Honor 7X Smartphone with a HiSilicon Kirin 659 processor (4 x 2.36 GHz + 4 x 1.7 GHz), 4 GB RAM, and 64 GB memory, running Android version 8.0.

The code was compiled and run with Java version 1.8 and didn't require any further libraries except for those dependencies the `upb.crypto.math` library already had (e.g. `junit` and `log4j`), as well as the Android Java SDK (for the smartphone tests). Tests were performed on Barreto-Naehrig curves, since this was the kind of curve the `upb.crypto.math` library was specialized for, with a group order with a bit length of either 256 or 512 bit, as recommended curve parameters suggest values in this range for secure cryptography (see [25]).

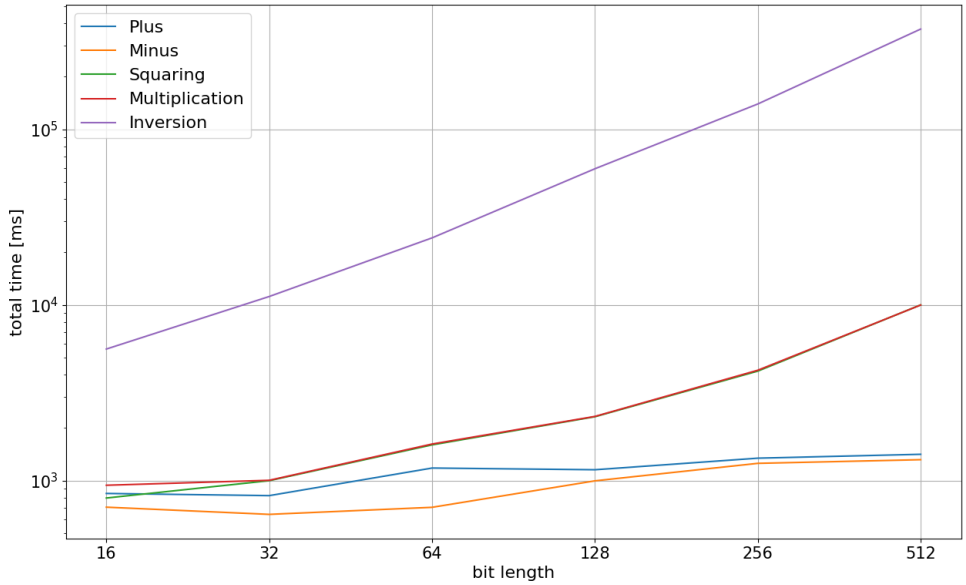
Tests were run with as few open applications as possible, in order to minimize potential noise.

5.2 Basic field operations evaluations

Figure 5.1 depicts the runtime costs of the basic field operation `add`, `mul`, etc. when executed via the `Zp` class from the `upb.crypto.math` library.

We can see that multiplication and squaring are almost equally performant (probably because the optimized algorithms for squaring that are implemented in the `BigInteger` class are only applicable for significantly larger values), while both addition and

Figure 5.1: Total cost (in ms) of performing 10^7 operations of the given kind on random field elements of given bit length



subtraction take considerably less time. Costs for inversions are more than an order of magnitude higher than for multiplications/squarings. While this difference is not quite as pronounced as guessed in Section 3.1.7, it is still high enough to justify why we would like to avoid inversions by using projective or Jacobian coordinates instead of affine coordinates.

5.3 Affine vs Jacobian vs projective coordinates

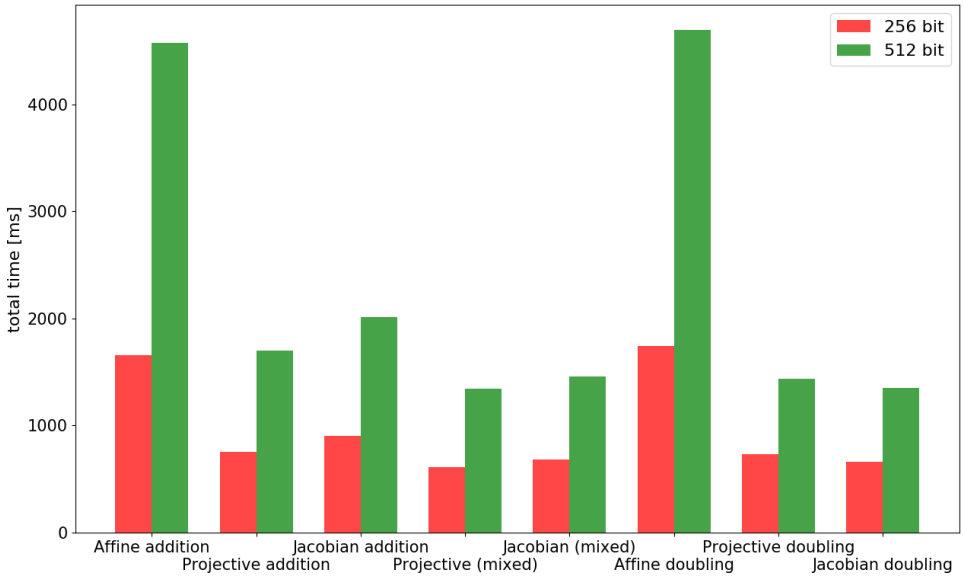
Figure 5.2 shows the costs of the basic elliptic curve operations – addition and doubling – over 256- and 512-bit curves for different coordinate systems.

The affine coordinates are easily outperformed by the other coordinate systems by a factor of $2x$-$3x$. As predicted in Section 3.1.7, projective coordinates perform better for the normal addition, while Jacobian coordinates are faster when it comes to doubling. Using mixed coordinates improves both projective and Jacobian addition, but has a slightly larger effect on Jacobian coordinates than on projective ones, which also aligns with previous predictions.

5.4 Comparison of single exponentiation techniques

In Figure 5.3 we can see for Jacobian coordinates how well various single exponentiation algorithms (both with and without caching precomputed small powers) perform in comparison to the simple double-and-add procedure, with varying values for the window size. With an appropriately chosen window size (4 seems to be the optimal value in at

Figure 5.2: Total cost (in ms) of performing 10^5 operations of the given kind on random elements of 256- and 512-bit BN-curves (over $\mathbb{F}_p$)



least this example configuration), even if no caching of small powers is used (i.e. the small powers need to be recomputed again for each exponentiation), all three depicted algorithms perform significantly better than the simple double-and-add technique, with wNAF being fastest. If caching is allowed (i.e. the precomputation of the small powers is only done once, at the beginning, and not included in the runtime costs), the exponentiations get ever cheaper with increasing window size.

Figure 5.4 shows the equivalent plot for projective coordinates, with slightly different results: With caching, Jacobian coordinates perform significantly better, for window sizes larger than 2. Without caching, projective coordinates perform faster for both small and large window sizes, while for a window size of 4 both approaches perform about equally well. This can be explained by considering that the precomputation phase consists of general point addition operations, which are significantly faster in projective coordinates: For large window sizes, the expected number of additions during the evaluation phase is very small, but large for the precomputation phase. For small window sizes, on the other hand, the precomputation phase is insignificant, but the expected number of additions during the evaluation phase is still rather high, making projective coordinates faster.

Thus we can conclude, in cases where caching of small powers makes sense because the same base will be used multiple times (e.g. generators or public keys like in the Diffie-Hellman key exchange), the window size should be chosen as large as possible, as long as the space and runtime side effects from the required precomputation don't become excessive. In cases where each base is used only once, a smaller window size should be used, ideally around 4. The wNAF algorithm is to be preferred over the other ones, and Jacobian coordinates should be used when caching small powers, otherwise projective coordinates are preferable.

Figure 5.3: Total cost (in ms) of performing 5000 exponentiations using different algorithms on random elements of a 256-bit BN-curve (over $\mathbb{F}_p$) using Jacobian coordinates, with exponents chosen uniformly at random from $\mathbb{F}_p$

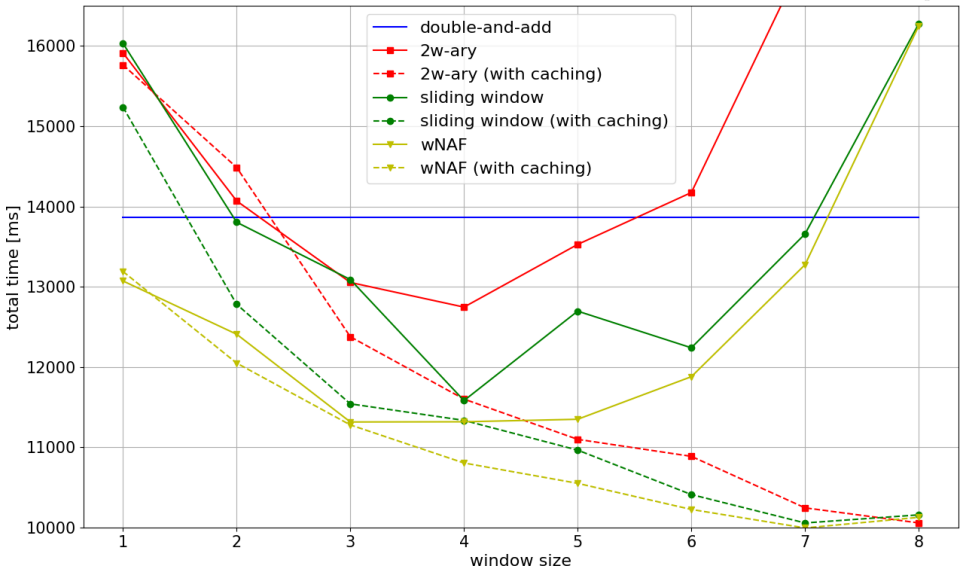
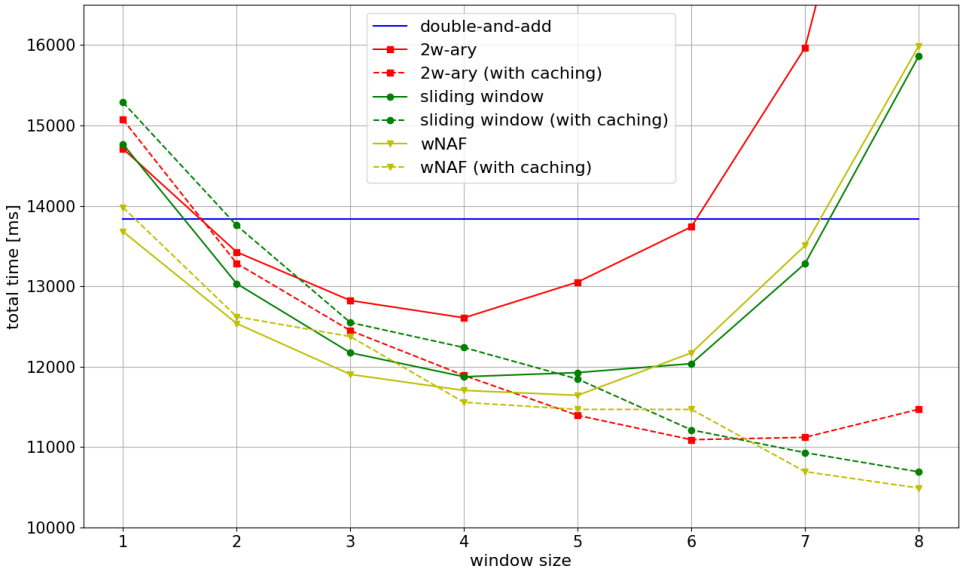


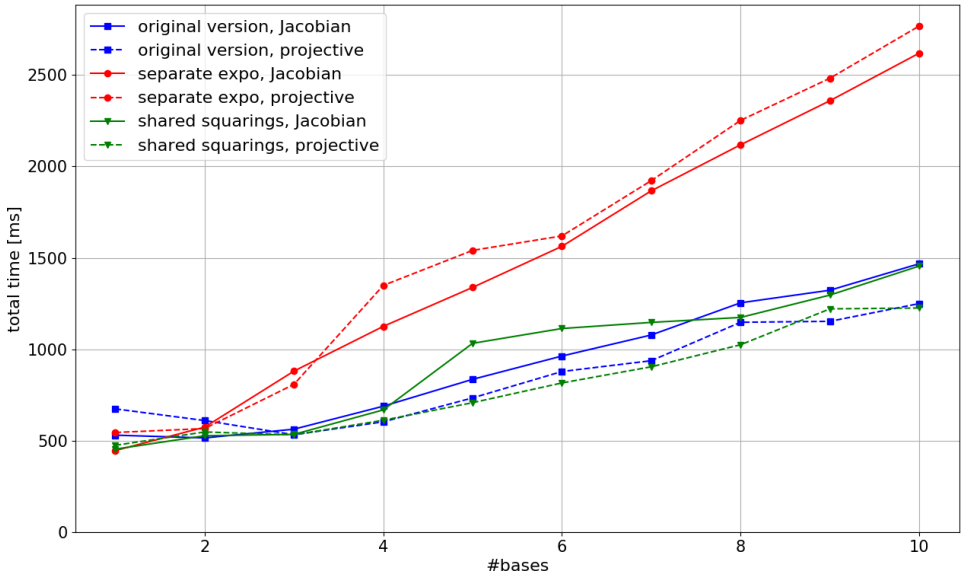
Figure 5.4: Total cost (in ms) of performing 5000 exponentiations using different algorithms on random elements of a 256-bit BN-curve (over $\mathbb{F}_p$) using projective coordinates, with exponents chosen uniformly at random from $\mathbb{F}_p$



5.5 Comparison of multiexponentiation techniques

Figure 5.5 shows performance measurements for the trivial multiexponentiation algorithm (where the power of each base is computed separately), the original implementation using `HashMaps`, as well as our equivalent simplified array-based implementation.

Figure 5.5: Total cost (in ms) of performing 100 multiexponentiations of the given number of random bases from a 256-bit BN-curve (over $\mathbb{F}_p$) in projective coordinates using basic algorithms, with exponents chosen uniformly at random from $\mathbb{F}_p$



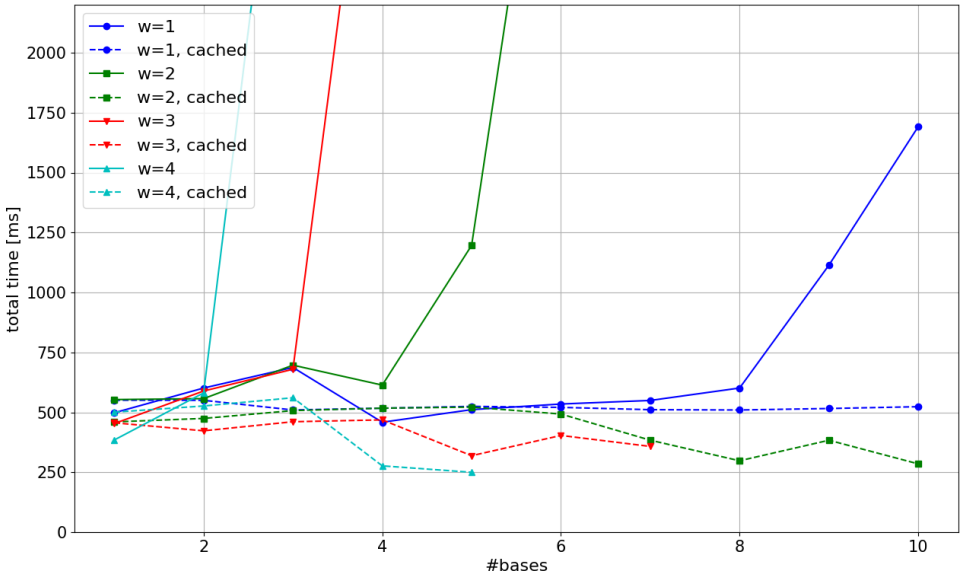
You can observe that the original version and the array-based implementation are almost equally fast, both of them being significantly faster than the trivial implementation, because they share the doubling operations when there are multiple bases. (For this test, the costly call of the `dynamicOptimization` method from the original `PowProductExpression` has been commented out; with the `dynamicOptimization` call, the original version would perform significantly slower than the simplified array-based version.)

For the trivial implementation, Jacobian coordinates are slightly faster than projective coordinates, which makes sense considering that this implementation basically just performs a few single exponentiations separately, and as discussed in the previous section, Jacobian coordinates perform better than projective coordinates when performing single exponentiations.

For the other two multiexponentiation algorithms that share the doubling operations, though, projective coordinates are again faster than the Jacobian ones, because fewer doublings are performed, and projective coordinates perform additions faster than Jacobian ones. In order to improve the readability of the plots for the following sliding window multiexponentiation algorithms, only results for projective coordinates are shown. (Measurements in Jacobian coordinates are very similar, just slightly slower.)

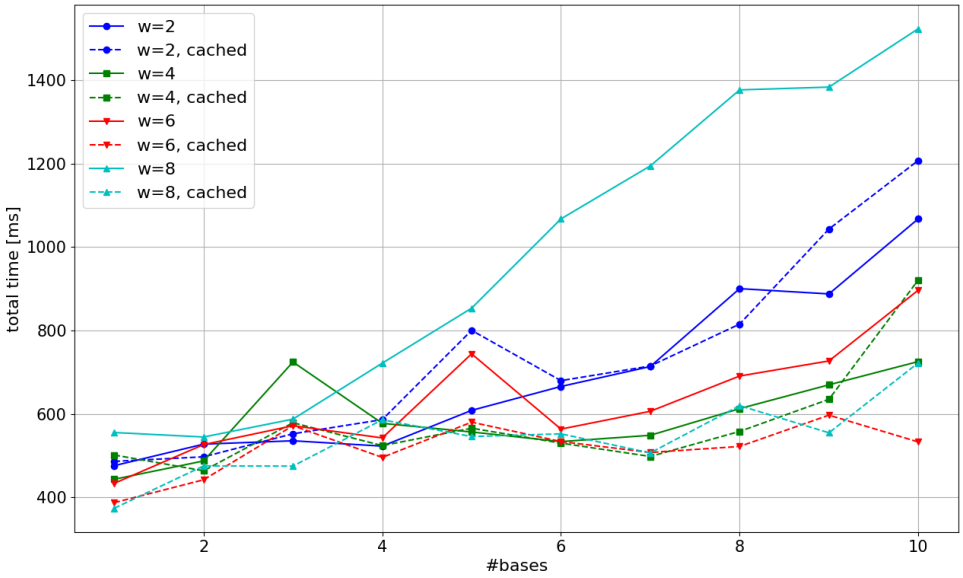
Figure 5.6 depicts the performance of the simultaneous sliding window approach for multiexponentiation with window sizes ranging from 1 to 4. (For window sizes 3 and 4, the data is truncated, since the Laptop's space limitations were reached.) If the precomputed power combinations can be cached and reused, this approach is extremely fast and evaluations don't cost significantly more as the number of bases grows. If the precomputation needs to be done for every multiexponentiation though (because no set

Figure 5.6: Total cost (in ms) of performing 100 multiexponentiations of the given number of random bases from a 256-bit BN-curve (over $\mathbb{F}_p$) in projective coordinates using the simultaneous sliding window technique, with exponents chosen uniformly at random from $\mathbb{F}_p$



of bases is used more than once), the simultaneous approach becomes quickly unfeasible for a larger number of bases r, as it grows exponentially in r.

Figure 5.7: Total cost (in ms) of performing 100 multiexponentiations of the given number of random bases from a 256-bit BN-curve (over $\mathbb{F}_p$) in projective coordinates using the interleaved sliding window technique, with exponents chosen uniformly at random from $\mathbb{F}_p$



In Figure 5.7, on the other hand, we can see how the interleaving approach results in linear complexity for both the precomputation as well as the evaluation phase. Up until a window size of about 4, the runtime improves both for the version with caching as well as for the one without. With a larger window size, however, the precomputation does start to cause significant costs and only if caching and reuse of small powers is possible, such a large window size would make sense. (Note that the precomputation is not nearly as costly here as for the simultaneous approach, since only the direct powers of the bases rather than combinations of them are computed.)

In conclusion, we can give the recommendation to use the simultaneous approach (with window size of 1 or at most 2 because otherwise the precomputation become to time- and space- consuming) when the number of bases doesn't exceed 10 and power combinations can be cached and will be reused. In all other cases, the interleaving approach should be employed, with a window size of about 4 if no caching is applicable, and as large a value as possible otherwise (as long as the precomputation phase uses not more time or space than available).

5.6 Performance gains via implementation-specific optimizations

Figure 5.8 shows the performance impact of the `BigInteger.mod` operation from the `Zn` constructor and the total overhead from the complex structure of the `upb.crypto.math` library by the example of projective scalar multiplication performed by different components: We can see that the original implementation is already significantly improved when optimizing the ZnElement creation by not performing the `mod` operation when the passed `BigInteger` is already in the desired range $[0, p-1]$. The right columns shows how the simplified implementation from the `MyProjectiveTriple` class can even further improve the performance.

These effects are significantly more pronounced for the shorter 256-bit curve than for the 512-bit curve, presumably because the overhead from the `ZnElement` creation doesn't increase as much with a larger number of bits as the time consumed by the main computations.

5.7 Tate vs Ate pairing

In Table 5.1 we can see the runtimes of Tate pairing and Ate pairing computation on 256- and 512-bit BN-curves. The Ate pairing ran about 6.5% faster than the Tate pairing for the 256-bit curve, but only about 1.5% faster on the 512-bit BN-curve. Apparently, the reduced number of required miller loop iterations leads to significantly improved performance as long as the curve size is not too large. With 512-bit, or even larger curve sizes, the improvement is less significant, presumably because the final exponentiation (which requires expensive $\mathbb{F}_{p^k}$-arithmetic) has larger relative costs for large bit lengths.

Figure 5.8: Relative improvement (in comparison to the original version) of performing 10^4 simple double-and-add scalar multiplications of random projective points from a 256- or 512-bit BN-curve (over $\mathbb{F}_p$), using the slightly modified version (with optimized `ZnElement` creation), and `MyProjectiveTriple` implementation

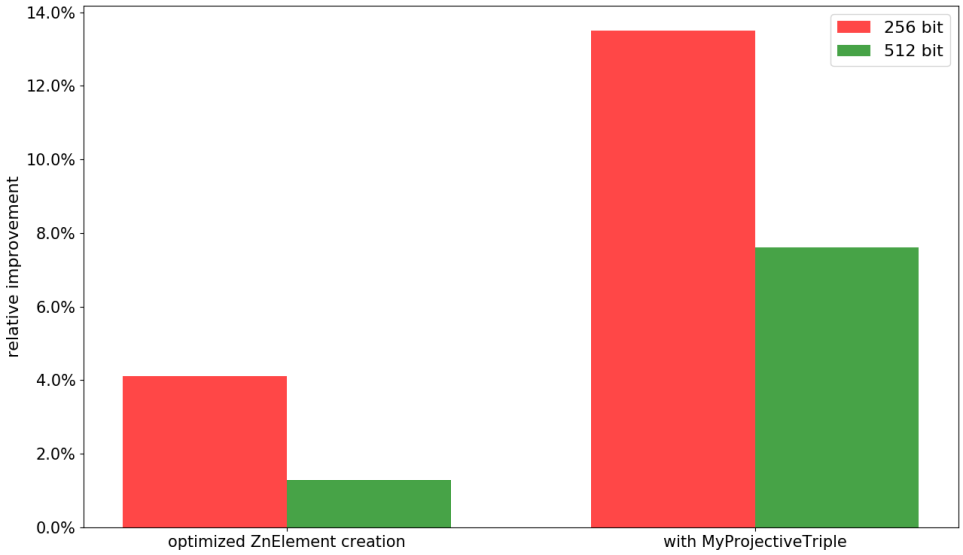


Table 5.1: Runtimes (in ms) of 50 pairing computation of the given type over a BN-curve of given bit length

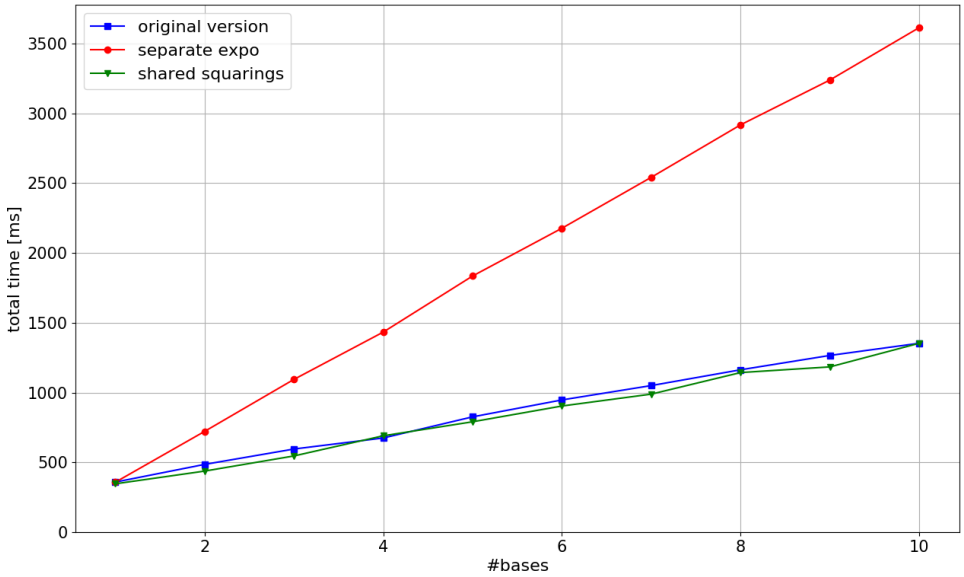
	256-bit	512-bit
Tate pairing	76 133	192 691
Ate pairing	71 324	189 837

5.8 Performances on a smartphone

Rather than running all tests from above anew on the Android phone, we decided to focus on multiexponentiation: It's the most complex problem presented so far, and its high requirement for memory and computing power will show us the limits of performing elliptic curve algorithms on smartphones. Since we already established that projective coordinates perform slightly faster for multiexponentiation with all algorithms except the trivial one, for simplicity, we'll just show the results for projective coordinates.

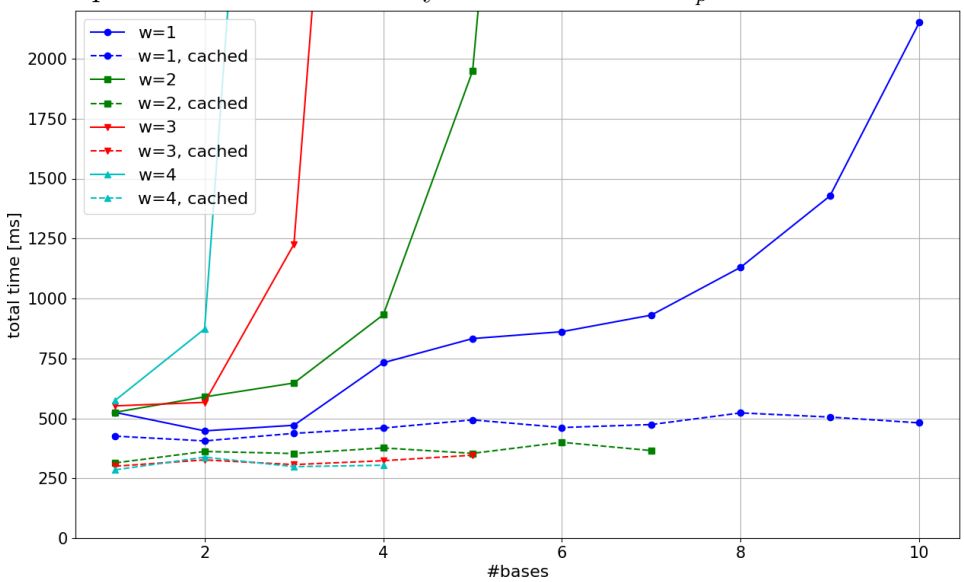
Figure 5.9 shows how fast multiexponentiations perform when employing the three basic implementations. Note that only 5 iterations were performed, in contrast to the 100 iterations performed for the laptop tests from Section 5.5. The average runtime of performing a single multiexponentiation is about 25 times slower on the Honor 7X Android phone than on the ThinkPad E580 laptop (see Section 5.1 for the exact hardware

Figure 5.9: Total cost (in ms) of performing 5 multiexponentiations of the given number of random bases from a 256-bit BN-curve (over $\mathbb{F}_p$) in projective coordinates using basic algorithms on a smartphone, with exponents chosen uniformly at random from $\mathbb{F}_p$



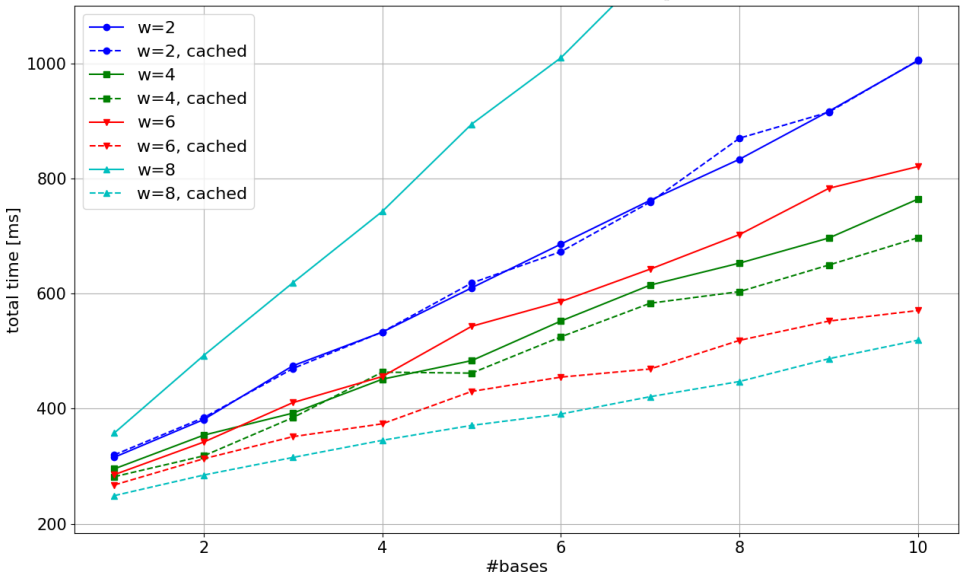
specifications of these devices). Except for this astoundingly large slowdown, the three algorithms exhibit behavior equivalent to the one seen in the laptop tests.

Figure 5.10: Total cost (in ms) of performing 5 multiexponentiations of the given number of random bases from a 256-bit BN-curve (over $\mathbb{F}_p$) in projective coordinates using the simultaneous sliding window technique on a smartphone, with exponents chosen uniformly at random from $\mathbb{F}_p$



Results for simultaneous sliding window multiexponentiation are shown in Figure 5.10. We can see how very quickly the precomputation phase leads to exceedingly high computation times, even with rather small bases. With appropriate calls to the Android SDK we determined the total memory available for the app to be 384MB. While all other tested algorithms only used between 30 and 90MB the simultaneous sliding window multiexponentiation required more than 300MB when the window size and the number of bases was relatively large, leading to much overhead due to memory allocation / garbage collection. This is why, compared to the laptop tests, the computation costs grow more quickly on the smartphone; and for e.g. five bases and a window size of four, it was not even possible at all to do the required precomputation, while this was no problem on the laptop.

Figure 5.11: Total cost (in ms) of performing 5 multiexponentiations of the given number of random bases from a 256-bit BN-curve (over $\mathbb{F}_p$) in projective coordinates using the interleaved sliding window technique on a smartphone, with exponents chosen uniformly at random from $\mathbb{F}_p$



In Figure 5.11 we the evaluation results for interleaved sliding window multiexponentiation, which are just as for the laptop: Without caching small powers, a window size of four performs best, and with caching, larger window sizes can be used to improve performance further.

In conclusion, we can say that the limited available memory of smartphones should be taken into account when choosing the extent of precomputations to be performed. More importantly, the performance on the phone in general seems to be much slower than on the laptop, even in cases where little memory was used. Thus, all elliptic curve algorithms should be implemented as efficiently as possible in order to ensure that they execute sufficiently fast even on these platforms.

6 Conclusion & future work

In conclusion, we would like to emphasize in which areas putting time and effort into optimizing performance was most worthwhile: Clearly, using projective or Jacobian coordinates instead of affine ones was a great performance enhancement, similarly using sliding window techniques for single- and multiexponentiation significantly improved upon the existing code. Considering, on the other hand, how much time we have spent analyzing the difference between projective and Jacobian coordinates, or the between sliding window and wNAF exponentiation, leading to only small improvements of performance, we can't help but wonder if some of that time could have been spent elsewhere more effectively.

Bearing in mind the large improvement that we achieved by a simple additional `if`-clause to avoid unnecessary modulo operations or by the simplified `ProjectiveTriple` code, we would like to point out that the notion of "premature optimization" – a derogatory term commonly used to describe the premature attempt to improve performance by working on low-level implementation details – can sometimes also be applied in the reverse direction: Optimizing the theoretical runtime of the algorithms by using new and rather complicated approaches from a vast set of scientific papers might often just improve performance by a minuscule amount, while working on the specific implementation details might lead to much more significant improvements in less time. In the end, it is of the essence to find the right middle ground between these two extremes.

Regarding future work, it might thus be worthwhile to have an even closer look at how the `BigInteger` class is used: A specialized wrapper class might be able to encapsulate optimizations that might further reduce the number of performed `mod` operations, e.g. by just subtracting `p` once (if necessary) after an addition of two values from $\mathbb{F}_p$ thus saving the expensive `mod` operation. Using the internal `MutableBigInteger` class directly might also improve performance significantly.

Further experiments about other types of pairing (like the *Eta pairing* [13]) as well as how the final exponentiation step could be improved (see [24]) might be useful. Entirely new topics of interest could include how to perform elliptic curve algorithms in a parallelized fashion, or how to make them secure against side-channel attacks. Finally, our new code could get better integrated into the existing library, with automated heuristics deciding when to use which algorithm.

Bibliography

[1] Leonard Adleman. "A subexponential algorithm for the discrete logarithm problem with applications to cryptography". In: *20th Annual Symposium on Foundations of Computer Science (SFCS 1979)*. IEEE. 1979, pp. 55–60.

[2] *Android Studio website*. URL: `https://developer.android.com/studio` (visited on 07/20/2019).

[3] Paulo SLM Barreto and Michael Naehrig. "Pairing-friendly elliptic curves of prime order". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2005, pp. 319–331.

[4] Ian Blake et al. *Elliptic curves in cryptography*. Vol. 265. Cambridge university press, 1999.

[5] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short signatures from the Weil pairing". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2001, pp. 514–532.

[6] Jan Camenisch and Anna Lysyanskaya. "Signature schemes and anonymous credentials from bilinear maps". In: *Annual International Cryptology Conference*. Springer. 2004, pp. 56–72.

[7] Mathieu Ciet et al. "Trading inversions for multiplications in elliptic curve cryptography". In: *Designs, codes and cryptography* 39.2 (2006), pp. 189–206.

[8] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. "Efficient elliptic curve exponentiation using mixed coordinates". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 1998, pp. 51–65.

[9] Henri Cohen et al. *Handbook of elliptic and hyperelliptic curve cryptography*. Chapman and Hall/CRC, 2005.

[10] Augusto Jun Devegili, Michael Scott, and Ricardo Dahab. "Implementing cryptographic pairings over Barreto-Naehrig curves". In: *International Conference on Pairing-Based Cryptography*. Springer. 2007, pp. 197–207.

[11] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. "Pairings for cryptographers". In: *Discrete Applied Mathematics* 156.16 (2008), pp. 3113–3121.

[12] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. "Guide to elliptic curve cryptography". In: *Computing Reviews* 46.1 (2005), p. 13.

[13] Florian Hess, Nigel P Smart, and Frederik Vercauteren. "The eta pairing revisited". In: *IEEE Transactions on Information Theory* 52.10 (2006), pp. 4595–4602.

[14] Vojtěch Horky et al. “Dos and don’ts of conducting performance measurements in java”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM. 2015, pp. 337–340.

[15] *Jacobian coordinates for short Weierstrass curves*. URL: `http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian.html` (visited on 07/20/2019).

[16] Antoine Joux. “A one round protocol for tripartite Diffie–Hellman”. In: *International algorithmic number theory symposium*. Springer. 2000, pp. 385–393.

[17] Cetin K Koç. “Analysis of sliding window techniques for exponentiation”. In: *Computers & Mathematics with Applications* 30.10 (1995), pp. 17–24.

[18] Alfred Menezes. “An introduction to pairing-based cryptography”. In: *Recent trends in cryptography* 477 (2009), pp. 47–65.

[19] Bodo Möller. “Algorithms for multi-exponentiation”. In: *International Workshop on Selected Areas in Cryptography*. Springer. 2001, pp. 165–180.

[20] Michael Naehrig. “Constructive and computational aspects of cryptographic pairings”. In: (2009).

[21] David Pointcheval and Olivier Sanders. “Short randomizable signatures”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2016, pp. 111–126.

[22] *Projective coordinates for short Weierstrass curves*. URL: `http://www.hyperelliptic.org/EFD/g1p/auto-shortw-projective.html` (visited on 07/20/2019).

[23] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[24] Michael Scott et al. “On the final exponentiation for calculating pairings on ordinary elliptic curves”. In: *International Conference on Pairing-Based Cryptography*. Springer. 2009, pp. 78–88.

[25] *Standards for Efficient Cryptography Group*. URL: `https://www.secg.org/` (visited on 07/20/2019).

[26] Frederik Vercauteren. “Optimal pairings”. In: *IEEE Transactions on Information Theory* 56.1 (2009), pp. 455–461.

[27] *VisualVM website*. URL: `https://visualvm.github.io/` (visited on 07/20/2019).

[28] Carl Youngblood. “An introduction to identity-based cryptography”. In: *CSEP 590TU* (2005), pp. 1–7.