TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Bachelor's Thesis

Parallelism investigation for elliptic curve key exchange

Henning Basold

November 30th, 2010

Institut für Datentechnik und Kommunikationsnetze
Prof. Dr. Berekovic

supervised by:

Matthias Hanke

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, November 30th, 2010

_____

Abstract

In this bachelor thesis the operations on elliptic curves, which are needed for key agreement, are analyzed for possible parallelization. The goal is a parallelization at a very small granularity suitable for hardware implementation.

This thesis consists of an introduction to the needed mathematical background and the derivation of the necessary algorithms from this background. Afterwards the algorithms are analyzed for possible parallelization.

These steps involve an implementation using SystemC and Verilog to get a model which is used to measure the speedups through altering algorithms. This model is suitable to be synthesized as e.g. an ASIP (Application Specific Instruction Set Processor).

The model consists of a software and a hardware part. The operations on elliptic curves and below are implemented in hardware. The protocol for key exchange is implemented in software (that means it should be executed on a general purpose processor).

Keywords Elliptic Curve Cryptography, Parallelization, Key Agreement

Technische Universität Braunschweig

Institut für Datentechnik
und Kommunikationsnetze

Hans-Sommer-Strasse 66
D-38106 Braunschweig

Germany

Fon +49(0)531/391-3734
Fax +49(0)531/391-4587

Prof. Dr.-Ing. Mladen Berekovic

Fon +49(0)531/391-3166

berekovic@ida.ing.tu-bs.de

30.06.2010

Aufgabenstellung zur Bachelorarbeit

Student: Henning Basold (Informatik)
Betreuer: Matthias Hanke

Titel: **Parallelism investigation for elliptic curve key exchange**

Der Lehrstuhl für VLSI Design entwickelt im Rahmen des Artemis-Projektes "SMART - Secure Mobile Visual Sensor Network Architecture" einen rekonfigurierbaren Prozessor (RASIP - Reconfigurable Application Specific Instruction Set Processor) für drahtlose Sensorknoten. Dieser soll für unterschiedliche kryptographische Algorithmen möglichst energieeffizient und performant arbeiten.

Für eine hohe Effizienz des Prozessors ist insbesondere die Parallelisierung einzelner Algorithmenabschnitte erfolgversprechend. Insbesondere Spezialinstruktionen, die auf mehreren Datenpfaden gleichzeitig ausgeführt werden, wird große Bedeutung beigemessen.

In dieser Bachelorarbeit soll zunächst ein sequentieller Algorithmus für den Schlüsselaustausch mit elliptischen Kurven in C entwickelt werden. Darauf aufbauend sind parallelisierbare Abschnitte zu identifizieren. Die Ergebnisse sollen zu einer parallelisierten Lösung des Algorithmus führen, der von seiner Ausführungszeit her mit der sequentiellen Variante zu vergleichen ist. Als Maß soll dabei die Anzahl der Zyklen, die für einen Pragrammablauf notwendig sind verwendet werden. Für diese Gegenüberstellung sind entsprechende Testvektoren zu erzeugen und eine Testumgebung aufzubauen. Abschließend müssen die Ergebnisse dokumentiert werden.

Abgabetermin ist der 02.12.2010

Prof. Dr.-Ing. Mladen Berekovic Student

Contents

List of Figures

List of Abbreviations

ASIP Application Specific Instruction Set Processor

CDH Computational Diffie-Hellman assumption

DAG Directed Acyclic Graph

DLP Discrete Logarithm Problem

EC Elliptic Curve

ECC Elliptic Curve Cryptography

GF Galois Field

1 Mathematical Background

1.1 Introduction

In this chapter all necessary mathematical background will be given. This includes a short introduction into group theory and some operations using this foundation. Afterwards we will see this theory at work in a simple encryption algorithm: the ElGamal encryption.

For the introduction to elliptic curves we need a small fraction of field theory. Of special interest here are finite fields which are very useful for cryptography.

Last but not least we will take a look at the key ingredient: the elliptic curves.

1.2 Groups

Definition 1.1 (Group)
Let G be a non-empty set and $\cdot : G \times G \to G$ an operation on G. The pair $(G, \cdot)$ is named *group* if $\cdot$ has the following properties.

1. Associativity: $(\forall a, b, c \in G) : (a \cdot b) \cdot c = a \cdot (b \cdot c)$
2. Neutral element $e \in G$: $(\forall a \in G) : e \cdot a = a = a \cdot e$
3. Existence of an inverse: $(\forall a \in G)(\exists a' \in G) : a' \cdot a = e = a \cdot a'$

Note: *The order of the group G is the cardinality $|G|$ of the underlying set. If $|G| \in \mathbb{N}$ this is the number of elements in G.*

Note: *If $a \cdot b = b \cdot a$ for all $a, b \in G$ then the group $(G, \cdot)$ is called* abelian *or* commutative.

Note: *The operation $\cdot$ may for example be $+$. To use known symbols, the neutral element of a group $(G, \cdot)$ will be called 1 and of a group $(H, +)$ it will be called 0.*

If the combination of operation and set is clear from the context the group $(G, \cdot)$ will be referenced only by G.

Also the dot in $a \cdot b$ will be left out (ab) if it enhances the reading experience. This can be done due to the associativity.

Note: *Since a' in $a \cdot a' = e$ is uniquely determined we write a^{-1} for a'.*

If not differently noted the group G will be multiplicative ($(G, \cdot)$) in the following. So it has the unit $1 \in G$.

Example 1.1
$\mathbb{Z} = (\mathbb{Z}, +)$ is a group with the neutral element 0.

Definition 1.2 (Exponentiation)
Let $(G, \cdot)$ and $(H, +)$ be groups and $a \in G$, $b \in H$ and $k \in \mathbb{N}_0$. The *exponentiation* a^k and the *multiple* $k \cdot b$ are defined as follows:

$$a^k := \underbrace{a \cdot a \cdots a}_{\text{k times}} \qquad k \cdot b := \underbrace{b + b + \cdots + b}_{\text{k times}}$$
$$a^0 := 1 \in G \qquad 0 \cdot b = 0 \in H$$

Definition 1.3 (Element order)
Let G be a group and $a \in G$. The order is a function $ord : G \to \mathbb{N} \cup \{\infty\}$ defined as follows:

$$ord(a) = \begin{cases} \infty, & (\forall i \neq j) : a^i \neq a^j \\ \min_{n \in \mathbb{N}}\{a^n = 1\}, & else \end{cases}$$

Definition 1.4 (Generator/cyclic group)
If there exists an element $g \in G$ with $G = \{g^k : g \in \mathbb{Z}\}$ then g is called the *generator* of G and G is a *cyclic group*. This is written as $G = \langle g \rangle$.

If $ord(g) = n \in \mathbb{N}$ (that is $ord(g)$ is finite) it follows that $\langle g \rangle = G = \{1 = g^0, g = g^1, g^2, ..., g^{n-1}\}$. So $ord(g) = |G|$.

Note: *In the following we will consider only finite cyclic groups.*

In figure 1.1 the concept of a cyclic group is illustrated. The composition $g^1 \cdot g^3 = g^4$ is illustrated. No composition of arbitrary elements of the group ever leaves the cycle.

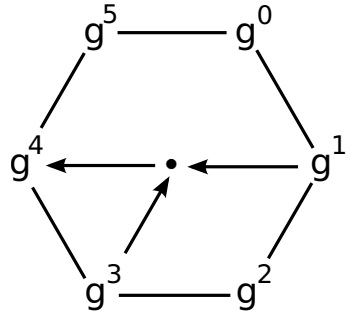

Figure 1.1: Visualization of a cyclic group $\langle g \rangle$ with $|\langle g \rangle| = 6$

The composition of g^1 and g^3 to g^4 motivates the following lemma. It follows directly from the associativity.

Lemma 1.5 (Exponent rules)
Let G be a group and $a \in G$ an element. Then the following holds for all $k, l \in \mathbb{N}_0$:

1. $a^k a^l = a^{k+l}$
2. $(a^k)^l = a^{kl} = (a^l)^k$

For the ElGamal method the last thing we need is the inverse of the exponentiation. That is we want to reduce the fraction $\frac{a^k}{a^k} = a^k \cdot a^{-k}$ to 1. To achieve this we extend the definition of the exponentiation to $\mathbb{Z}$. From that the inverse follows naturally.

Definition 1.6
Let G be a finite cyclic group, $a \in G$ and $k \in \mathbb{N}_0$. Then

$$a^{-k} := (a^{-1})^k.$$

Lemma 1.7
a^{-k} is the inverse to a^k that is $a^k a^{-k} = 1 = a^{-k} a^k$.

Proof. The proof is performed inductively:

$$\begin{aligned}
&a^k a^{-k} \\
&\quad \overset{1.5}{=} (a^1)^k \cdot (a^{-1})^k \\
&\quad = \underbrace{a \cdots a}_{\text{k times}} \cdot \underbrace{a^{-1} \cdots a^{-1}}_{\text{k times}} \\
&\quad = \underbrace{a \cdots a}_{\text{k-1 times}} \cdot (a \cdot a^{-1}) \cdot \underbrace{a^{-1} \cdots a^{-1}}_{\text{k-1 times}} \\
&\quad = (a^1)^{k-1} \cdot 1 \cdot (a^{-1})^{k-1} \\
&\quad = (a^1)^{k-1} \cdot (a^{-1})^{k-1} \\
&\quad \vdots \\
&\quad = 1
\end{aligned}$$

□

1.3 ElGamal encryption

In the following the ElGamal encryption is introduced. We look at the more abstract version which operates on arbitrary groups.

The algorithm itself is not used here because the goal is not to encrypt data but to exchange keys. But the algorithm shows nicely the principles of cryptography using elliptic curves.

Definition 1.8 (Generalized ElGamal encryption)
Let $G = \langle g \rangle$ be a finite cyclic group and $n = ord(g)$. $M \in G$ is the message which Alice (A) wants to transport to Bob (B).

g and n are the domain parameters on which A and B agree beforehand.

- *Key generation*:
  B chooses $d \in 2 \dots n-1$ and assigns $a := g^d$. a is B's *public key* and d is his *private key.*

- *Encryption*:
  A chooses $e \in 2 \dots n-1$ and assigns $b := g^e$. To encrypt M A calculates $C := M \cdot a^e$. b and e are *ephemeral* keys used only for one session.

  A sends (C, b) to B.

- *Decryption*:
  Let $s = b^{-d}$. From that it follows that $M = s \cdot C$.

Theorem 1.9
The generalized ElGamal encryption is correct.

Proof. The symbols are the same as in 1.8.

$$\begin{aligned} s \cdot C &\overset{s,C}{=} b^{-d} \cdot a^e \cdot M \\ &\overset{a,b}{=} (g^e)^{-d} \cdot (g^d)^e \cdot M \\ &\overset{1.5}{=} g^{-de} \cdot g^{de} \cdot M \\ &\overset{1.7}{=} 1 \cdot M \\ &= M \end{aligned}$$

□

Note: *In 1.8 d, e are chosen from $\mathbb{Z}_n \setminus \{0, 1\}$ because $ord(g) = n$ and $g^0 = 1$ and $g^1 = g$. So numbers above $n-1$ don't generate new keys and for 0 and 1 the encryption is trivial to break.*

If an attacker Eve (E) can eavesdrop everything A and B exchange, she only gets $g, a = g^d, b = g^e$ and C. But to decrypt C she has to calculate g^{de} from a and b. In certain groups this holds and is called the *computational Diffie-Hellman assumption (CDH).*

A counter example is $(\mathbb{Z}_p, +)$. A group in which the CDH assumption holds is $(\mathbb{Z}_p, \cdot)$. In $(\mathbb{Z}_p, \cdot)$ the CDH assumption is equivalent to the *discrete logarithm problem (DLP).* That is to retrieve d from $a = g^d$.

For more about the CDH assumption see [3, p.132 et seq.].

1.4 Finite fields

To define elliptic curves for cryptographic use one concept is needed: finite fields.

Definition 1.10 (Field)
Let $K \neq \emptyset$ be a nonempty set and $+ : K \times K \to K$ $\cdot : K \times K \to K$ two operations on K.

$(K, +, \cdot)$ is called a *field* if the following holds:

- $(K, +)$ and $(K \setminus \{0\}, \cdot)$ are commutative groups
- Distributivity: $(\forall a, b, c \in K) : a(b + c) = ab + ac$ and $(a + b)c = ac + bc$

If $|K| \in \mathbb{N}$ K is a *finite field*. Finite fields are also called *Galois fields* (GF).

Note: *$K \setminus \{0\}$ are all elements which have an inverse under $\cdot$. Generally $K^\times \subseteq K$ denotes all elements which have an inverse under some operation.*

Finite fields have a very special structure:

Theorem 1.11
If K is a finite field $|K| = p^n$ where p is a prime number and $n \in \mathbb{N}_0$.

Proof. See [5, p.264]. □

Because of this special structure the class of all finite fields with order p^n is denoted as $GF(p^n)$. From the finiteness of the elements in $GF(p^n)$ follows that all finite fields of order p^n are *isomorphic*. That is there is a one-to-one correspondence between the elements. An element from $GF(p^n)$ is called a *representation*. This will be used in 1.4.1. $\mathbb{F}_{p^n}$ stands for one such representation.

The different fields can be structured into classes by looking at their internal structure. This is needed to choose the correct equations for elliptic curves.

Definition 1.12
Let K be a finite field and $1 \in K$ the neutral element with respect to the multiplication. Then the *characteristic* of K is defined as

$$char(K) = min\{n \in \mathbb{N} : n \cdot 1 = 0\}$$

where $\cdot$ is the multiplication from definition 1.2.

Note: *It is important to note that the above definition is only valid for finite fields. In infinite fields it may happen that $char(K) = \infty$.*

Example 1.2
$char(\mathbb{Z}_3) = 3$ because $3 \cdot 1 = 1 + 1 + 1 = 3 \equiv 0 \pmod 3$.

1.4.1 Polynomial fields

Now we use the fact that all finite fields are isomorphic. One example for a representation are polynomials over another field modulus a reduction polynomial:

Definition 1.13
Let K be a field. Then $K[x]$ is the set of polynomials with factors from K:

$$K[x] = \{f(x) : f(x) = a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0,\ \ a_0, \ldots, a_n \in K, n \in \mathbb{N}\}$$

n is the *degree* of f denoted as $deg(f) := n$.

Definition 1.14 (Irreducible)
Let $f, g, h \in K[x]$. If

$$f = g \cdot h \ \ \Rightarrow \ \ g \in K^\times \text{ or } h \in K^\times$$

then f is named *irreducible*.

Note: *Note that the condition $g \in K^\times$ is equivalent to $deg(g) = 0$.*

Definition 1.15
Let $f, g \in K[x]$. There exist $q, r \in K[x]$ so that

$$f = qg + r \text{ and } deg(r) < deg(g)$$

With this we extend the modulus operation: $f \bmod g := r$.

Theorem 1.16 (Polynomial field)
Let K be a field with $|K| = k$ and $p \in K[x]$ be irreducible with $deg(p) = n$. The set

$$K[x]/(p) = \{f \bmod p : f \in K[x]\}$$

is a finite field. It has the order $|K[x]/(p)| = k^n$.

Proof. See [5]. □

Example 1.3
For the purpose of implementing the operations in binary logic (on a computer or in transistor logic) the field of binary polynomials is the most interesting.

This field from $GF(2^n)$ is represented by $\mathbb{Z}_2/(p)$ where p is a polynomial with factors from $\{0, 1\}$. A polynomial can be represented by a tuple of factors if one agrees on the exponent of each x^k corresponding to the position of a factor in the tuple. So a binary polynomial can be represented by a bit vector directly in memory.

The characteristic of $\mathbb{Z}_2/(p)$ is 2 since $1 + 1 \bmod = 0$.

1.5 Elliptic Curves

In the following we will introduce elliptic curves and define geometrical operations on them. These operations will form an additive operation which in turn forms a group structure.

Elliptic curves are defined by the points that fulfill the following equation

$$y^2 + a_1 xy + a_2 y = x^3 + a_3 x^2 + a_4 x + a_5$$

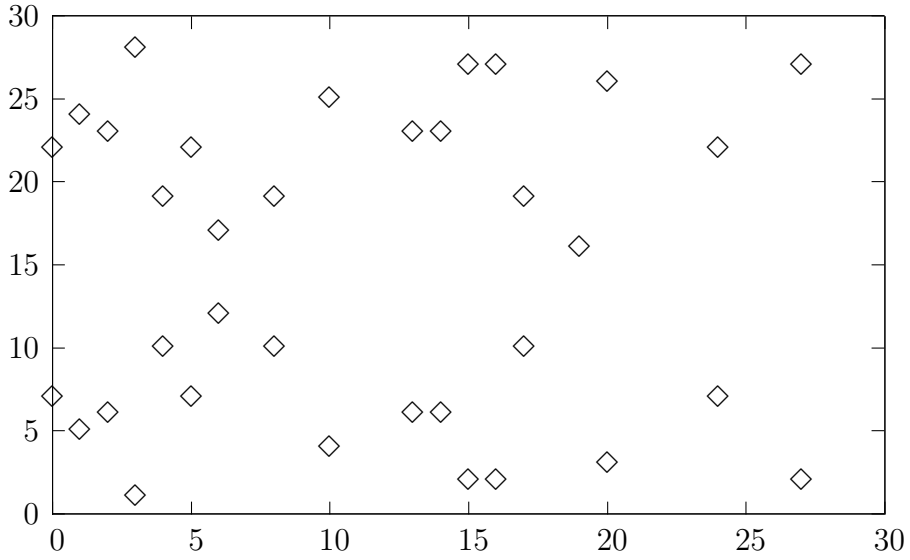

Figure 1.2: Elliptic curve over finite field $\mathbb{F}_{29}$

where $a_1, \ldots a_5$ are factors from an (mostly) arbitrary field. The "mostly" refers to the characteristic of the field. For fields of characteristic 2 or 3 we have to use slightly reduced equations. But more on that later.

In figure 1.3 a typical elliptic curve has been sketched. They look like this over fields like $\mathbb{R}$. But those would be useless for cryptographic usage. Over finite fields like $\mathbb{Z}_2/(p)$ they look like in figure 1.2. This is a lot more chaotic. The structure of a curve can be seen to some extend. It is repeated if the curve reaches the right border. But very little differences at the left border lead to big differences going to the right. We might think of this as a good pseudo random number generator if we use bigger fields for the curve.

Now we go for the combination of the points on the curves.

1.5.1 Geometrical operations

In the following we denote points (in the Euclidean plane) as $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. Those are assumed to be on the curve.

Let $P \neq Q$. The addition of P and Q is done by drawing a line through P and Q. The line hits the curve in exactly one point R' which is not equal to P or Q. To get $R = P + Q$ we now have to mirror R' at the x-axis.

This process is visualized in figure 1.3.

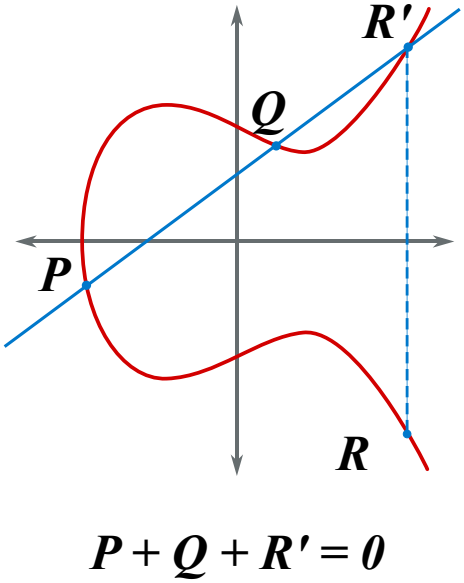

Figure 1.3: Addition on elliptic curves

If $P = Q$ then $P + Q = P + P$ thus this is called point doubling. It is done by drawing the tangent at P. We then again get a point R' in which the tangent hits the curve again. To get $R = P + P$ R' has to be mirrored again.

The point doubling is visualized in figure 1.4.

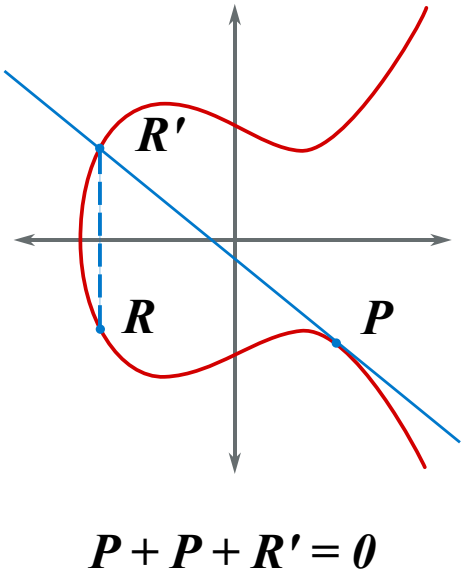

Figure 1.4: Point doubling on elliptic curves

It is worth noting that the operands to the addition may be exchanged without changing the result. This is pretty obvious from the geometrical point of view. So the addition is commutative.

For now we have an operation on the points of an elliptic curve that is well defined (i.e. for every pair P, Q $P + Q$ lies on the curve). If we want to interpret this as a group structure we need an inverse to the addition and a neutral element.

For this let us look at figure 1.5. There is a line drawn through P and its mirroring called $-P$. The line never hits the curve again. And this is the only situation in which that happens if we connect points on an elliptic curve. We define an imaginary point ∞ which is "hit" by the line as the result of this operation. If we add ∞ to an arbitrary point on the curve using the normal addition defined above we get always get the point back. This is because connecting an arbitrary point with ∞ always results into a line parallel to the y-axis.

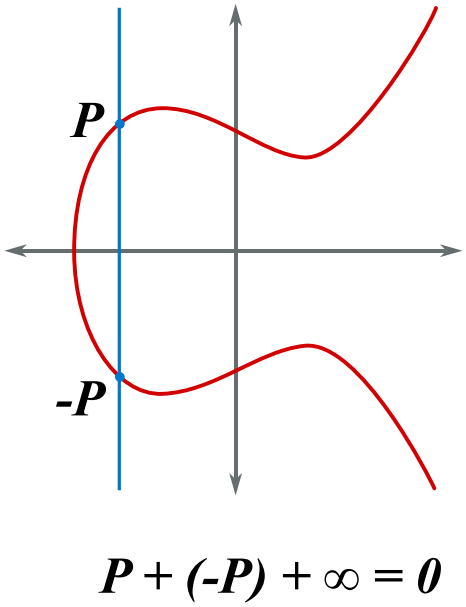

Figure 1.5: Addition of an inverse on elliptic curves

This way we have defined a neutral element ∞ and an inverse $-P$.

To sum up: an elliptic curve E over a field K defines a set of points

$$E(K) := \{P : \text{P lies on E}\} \cup \{\infty\}$$

and an operation $+$ with the neutral element ∞ and the inverse of a point P denoted as $-P$. Together $(E(K), +)$ form a commutative group.

1.5.2 Algebraic operations

Now we look at the algebraic definitions. To ease those we only define elliptic curves over fields of characteristic 2 because those are used in the implementation.

Definition 1.17
Let K be a finite field with $char(K) = 2$. Then

$$E(K) := \{(x, y) \in K \times K : y^2 + xy = x^3 + ax^2 + b\} \cup \{\infty\}, \quad a, b \in K$$

is the set of points on the elliptic curve defined by $y^2 = x^3 + ax + b$.

Let further be $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ then define the following:

- $-P := (x_1, x_1 + y_1)$
- $P + Q := (x_3, y_3)$, where

$$x_3 = s^2 + s + x_1 + x_2 + a, \quad y_3 = s(x_1 + x_3) + x_3 + y_1, \quad s = \frac{y_1 + y_2}{x_1 + x_2}$$

- $P + P := 2P := (x_3, y_3)$, where

$$x_3 = s^2 + s + a, \quad y_3 = x_1^2 + sx_3 + x3, \quad s = \frac{x_1 + y_1}{x_1}$$

Note: *The subtraction $a - b$ and division $\frac{c}{d}$ are just short forms of $a + (-b)$ and $(c \cdot d^{-1})$.*

Theorem 1.18
$(E(K), +)$ forms a commutative group.

Proof. The proof is of very technical nature. It is a straightforward construction from the geometrical imagination. It can partly be found in [10, p. 247].

But to get a first impression one may interpret s as the slope of the line/tangent in each case. □

2 Algorithms

2.1 Introduction

In this chapter all necessary algorithm for an encryption with elliptic curves will be collected and implemented.

The algorithms are divided into four parts:

1. operations on finite fields $\mathbb{F}_{2^n}$ from $GF(2^n)$ (2.2)
2. operations in $\mathbb{Z}$ (not covered here)
3. operations on $E(\mathbb{F}_{2^n})$ (2.3)
4. algorithms for key exchange (2.4)

1 and 2 are foundations for 3 whilst 2 and 3 are needed for 4.

2.2 Operations in $\mathbb{F}_{2^n}$

2.2.1 Definitions

Used values:

- $\mathbb{F}_{2^n} = \mathbb{Z}_2[x]/(p),\ \ p \in \mathbb{Z}_2[x]$ irreducible
- $p(x) = x^n + R(x),\ \ deg(p) = n,\ deg(R) < n$
- $R(x) = \sum_{i=0}^{n-1} r_i x^i \in \mathbb{Z}_2[x]$
- $f, g, h \in \mathbb{F}_{2^n}$ where

$$f = f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0 = \sum_{i=0}^{n-1} a_i x^i$$
$$g = \sum_{i=0}^{n-1} b_i x^i$$
$$h = \sum_{i=0}^{n-1} c_i x^i$$
$$a_i, b_i, c_i \in \mathbb{Z}_2, i = 0, \ldots, n-1$$

Used symbols:

- $F, G, H \in \mathbb{Z}_2^n$ (bit vectors of f, g, h) where

$$F = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$$
$$G = (b_{n-1}, \ldots, b_0)$$
$$H = (c_{n-1}, \ldots, c_0)$$

- $F[i] = a_i, G[i] = b_i, H[i] = c_i, i = 0, \ldots n-1$ (bit access)

Used operations:

- xor: $a \oplus b = (a + b) \mod 2, \ a, b \in \mathbb{Z}_2$
- left shift: $F \ll k = (a_{n-k-1}, a_{n-k-2}, \ldots, a_k, 0, \ldots, 0)$
- swapping: $swap(a, b) = (b, a)$

2.2.2 Addition

$$\begin{aligned}
h &= (f + g) \mod p \\
&= \left(\sum_{i=0}^{n-1} a_i x^i + \sum_{i=0}^{n-1} b_i x^i \right) \mod p \\
&= \left(\sum_{i=0}^{n-1} ((a_i + b_i) \bmod 2) \cdot x^i \right) \mod p \\
&= \left(\sum_{i=0}^{n-1} (a_i \oplus b_i) x^i \right) \mod p \\
&= \sum_{i=0}^{n-1} (a_i \oplus b_i) x^i \qquad \text{since } deg(f+g) < deg(p) \\
&\Rightarrow H[i] = F[i] \oplus G[i], i = 0, \ldots n-1
\end{aligned}$$

Algorithm 2.1 Addition in $\mathbb{F}_{2^n}$

```
for i = 0 to n − 1 do
    H[i] ← F[i] ⊕ G[i]
end for
return h
```

2.2.3 Substraction

From $a - b \equiv a + b \pmod 2$ immediately follows $f - g \equiv f + g \pmod p$.

2.2.4 Multiplication

$$
\begin{aligned}
h &= f \cdot g \bmod p \\
&= \left(\sum_{i=0}^{n-1} a_i x^i\right) \cdot g \bmod p \\
&= \left(\sum_{i=0}^{n-1} (a_i x^i \cdot g)\right) \bmod p \\
&= \left(\sum_{i=0}^{n-1} (a_i x^i \cdot g \bmod p)\right) \bmod p \\
&= \sum_{i=0}^{n-1} (a_i x^i \cdot g \bmod p) \qquad\qquad deg(a_i x^i g \bmod p) < deg(p)
\end{aligned}
$$

If $a_j = 1$:

$$
\begin{aligned}
a_j x^j \cdot g \mod p & \\
&= x^j \cdot g \mod p \\
&= gx x^{j-1} \mod p \\
&\equiv (gx \mod p) \cdot (x^{j-1} \mod p) \\
&\vdots \\
&\equiv \underbrace{[((gx \mod p) \cdot x \mod p) \ldots] \cdot x \mod p}_{\text{j times}}
\end{aligned}
$$

$$
\begin{aligned}
gx \mod p & \\
&= \left(\sum_{i=0}^{n-1} b_i x^i\right) \cdot x \mod p \\
&= \sum_{i=0}^{n-1} b_i x^{i+1} \mod p
\end{aligned}
$$

By polynomial long division we get:

$$
\begin{aligned}
(b_{n-1}x^n + b_{n-2}x^{n-1} + \ldots + b_0 x) &: (x^n + R(x)) \\
&= b_{n-1}R(x) + b_{n-2}x^{n-1} + \ldots + b_0 x
\end{aligned}
$$

which leads to

$$
\begin{aligned}
gx \mod p & \\
&\equiv b_{n-1}R(x) + b_{n-2}x^{n-1} + \ldots + b_0 x \mod p \\
&= \left(b_{n-1}\sum_{i=0}^{n-1} r_i x^i\right) + \sum_{i=0}^{n-2} b_i x^{i+1},\ deg(R) < n \\
&= \begin{cases} \sum_{i=0}^{n-1} (r_i + b'_i)x^i, & b_{n-1} = 1 \\ \sum_{i=0}^{n-1} b'_i x^i, & b_{n-1} = 0 \end{cases}
\end{aligned}
$$

where $b'_i = b_{i-1}, b'_0 = 0, i = 1, \ldots n-1$, that is $G' = G \ll 1$. This leads to the following algorithm.

Algorithm 2.2 $g(x) \cdot x \mod p$ in $\mathbb{F}_{2^n}$

```
if G[n − 1] = 1 then
  return (G ≪ 1) + R
else
  return G ≪ 1
end if
```

Based on this we get

$$\begin{aligned} h &= \sum_{i=0}^{n-1} (a_i x^i \cdot g \bmod p) && \text{see above} \\ &= \sum_{i=0}^{n-1} (a_i d_i) \end{aligned}$$

where

$$d_i = x^i \cdot g \bmod p = d_{i-1} x \bmod p,\ d_0 = g$$

This corresponds to the following algorithm.

Algorithm 2.3 $f(x) \cdot g(x) \mod p$ in $\mathbb{F}_{2^n}$

```
h ← 0
for i = 0 to n − 1 do
  h ← h + F[i] · g
  g ← g · x mod p {Algorithm 2.2}
end for
return h
```

The multiplication $F[i] \cdot g$ can be reduced to a simple check since $F[i] \in \mathbb{Z}_2$.

2.2.5 Division

Since $\frac{f}{g} = f \cdot g^{-1}$ we have to calculate the inverse of g. This means:

$$\begin{aligned} & g \cdot g^{-1} \equiv 1 \pmod p \\ & \Leftrightarrow g \cdot g^{-1} = m \cdot p + 1\ , m \in \mathbb{N}_0 \\ & \Leftrightarrow g \cdot g^{-1} - m \cdot p = 1 \\ & \Rightarrow (\exists u, v \in \mathbb{F}_{2^n}) : u \cdot g + v \cdot p = 1 \qquad (*) \end{aligned}$$

u and v can be found using the extended Euclidean Algorithm (EEA). Now we want to derive an efficient version of the EEA for polynomials. The EEA relies on

the fact that $\gcd(a,b) = \gcd(a, b-ca)$. In $\mathbb{F}_{2^n}$ $b-ca = b+ca$. So now we have to seek for a c so that $deg(b+ca) < deg(b)$ otherwise the algorithm would not terminate.

Let $0 < d = deg(b) - deg(a)$. Such a d exists, otherwise change a and b.

$$\begin{aligned}
&\Rightarrow deg(a \cdot x^d) = deg(b) \\
&\Rightarrow deg(b + a \cdot x^d) < deg(b) \\
&\Rightarrow \gcd(a,b) = \begin{cases} \gcd(a, b + ax^d) & deg(a) \leq deg(b) \\ \gcd(b,a) & deg(a) > deg(b) \\ b & a = 0 \end{cases}
\end{aligned}$$

This leads to algorithm 2.4. The recursion there is already transformed into an iteration.

Algorithm 2.4 $\gcd(f,g)$ in $\mathbb{F}_{2^n}$ (Euclidean algorithm)

Require: $deg(f), deg(g) < deg(p)$
  while $f \neq 0$ **do**
    $d \leftarrow deg(g) - deg(f)$
    if $d \geq 0$ **then**
      $g \leftarrow g + f \cdot x^d$
    else
      $(f,g) \leftarrow swap(f,g)$
    end if
  end while
  return g

The precondition ensures that $deg(f \cdot x^d) < deg(p)$. This can be used to implement the multiplication by a simple shift (algorithm 2.5).

Algorithm 2.5 $f \cdot x^d$ in $\mathbb{F}_{2^n}$

Require: $deg(f) + d < deg(p)$
  return $F \ll d$

The Euclidean algorithm for $\gcd(a,b)$ can be extended by the following invariant:

$$f \cdot u_1 + g \cdot v_1 = a \tag{2.1}$$
$$f \cdot u_2 + g \cdot v_2 = b \tag{2.2}$$

where $\gcd(a,b)$ is calculated with the initial arguments f and g. It can be proven by induction over the number of steps that this is really invariant if $u_{1,2}$ and $v_{1,2}$ undergo the same transformations as a and b.

This leads to the following recursion:

$$\gcd((a,u_1,v_1),(b,u_2,v_2)) = \begin{cases} \gcd((a,u_1,v_1),(b',u_2',v_2')), & deg(a) \leq deg(b) \\ \gcd((b,u_2,v_2),(a,u_1,v_1)), & deg(a) > deg(b) \\ (b,u_2,v_2), & a = 0 \end{cases}$$

where

$$\begin{aligned} d &= deg(b) - deg(a) \\ b' &= b + ax^d \\ u_2' &= u_2 + u_1 x^d \\ v_2' &= v_2 + v_1 x^d \end{aligned}$$

This function can be used in the following: $(b,u,v) = \gcd((f,1,0),(g,0,1))$ (cf. invariant). With this we get: $f \cdot u + g \cdot v = b$. This is exactly what we were looking for in $(*)$ (p. 15).

The algorithm can be retrieved analogously to algorithm 2.4.

Algorithm 2.6 $deg(f)$ in $\mathbb{F}_{2^n}$

Require: $f \neq 0$
  for $i = n-1$ to 0 **do**
    if $F[i] \neq 0$ **then**
      return i
    end if
  end for

The EEA uses the degree of a polynomial. It can be retrieved by algorithm 2.6. In algorithm 2.4 the exact degree has to be calculated twice. Which potentially means walking through all bits. There is another algorithm known as "Stein's algorithm". It only has to calculate if $deg(f) < deg(g)$. For this only one traversal is needed instead of two.

It is based on the following facts:

$$\gcd(a,b) = \begin{cases} x \gcd(a/x, b/x), & x \mid a,b \\ \gcd(a/x, b), & x \mid a,\ x \nmid b \\ \gcd((a-b)/x, b), & x \nmid a,b,\ deg(a) \geq deg(b) \\ \gcd(b,a), & x \nmid a,b,\ deg(a) < deg(b) \\ b, & a = 0 \end{cases}$$

From this algorithm 2.7 can be constructed. The invariant from the EEA is already attached to it. Some cases have been eliminated to make the algorithm more compact. The algorithm stops at $a \neq 1$ because we know that $gcd(f,p) = 1$. So we don't have to compute the next step.

Algorithm 2.7 f^{-1} in $\mathbb{F}_{2^n}$ (Stein's algorithm)

Require: $deg(f), deg(g) < deg(p)$
  $a \leftarrow f$
  $b \leftarrow p$
  $u \leftarrow 1$
  $v \leftarrow 0$
  while $a \neq 1$ **do**
    if $x \nmid a$ **then**
      if $deg(a) < deg(b)$ **then**
        $(a, b) \leftarrow swap(a, b)$
        $(u, v) \leftarrow swap(u, v)$
      end if
      $a \leftarrow a + b$
      $u \leftarrow u + v$
    end if
    while $x \mid a$ **do** {make a not divisible by x}
      $a \leftarrow a/x$
      if $x \nmid u$ **then**
        $u \leftarrow u + p$
      end if
      $u \leftarrow u/x$
    end while
  end while
  return u

For this to work there are two additional algorithms needed: $deg(f) < deg(g)$ (2.8) and f/x (2.9).

Algorithm 2.8 $deg(f) < deg(g)$ in $\mathbb{F}_{2^n}$

  for $i = n - 1$ to 0 **do**
    if $F[i] \neq G[i]$ and $F[i] = 0$ **then**
      return true
    else if $F[i] \neq G[i]$ and $F[i] = 1$ **then**
      return false
    else if $F[i] = G[i]$ **then**
      return false
    end if
  end for
  return false

Algorithm 2.9 f/x in $\mathbb{F}_{2^n}$

Require: $x \mid f$
  return $F \gg 1$

The *swap* operation in algorithm 2.7 prevents parallelization because it introduces a data dependency. This can be eliminated in regard to the implementation in hardware circuits. This is shown in algorithm 2.10.

Algorithm 2.10 f^{-1} in $\mathbb{F}_{2^n}$ (Stein's algorithm)

Require: $deg(f), deg(g) < deg(p)$
  $a \leftarrow f$
  $b \leftarrow p$
  $u \leftarrow 1$
  $v \leftarrow 0$
  while $a \neq 1$ **do**
    if $x \nmid a$ **then**
      if $deg(a) < deg(b)$ **then**
        $b \leftarrow a + b$
        $v \leftarrow u + v$
      else
        $a \leftarrow a + b$
        $u \leftarrow u + v$
      end if
    end if
    while $x \mid a$ **do** {make a not divisible by x}
      $a \leftarrow a/x$
      if $x \nmid u$ **then**
        $u \leftarrow u + p$
      end if
      $u \leftarrow u/x$
    end while
    while $x \mid b$ **do** {make b not divisible by x}
      $b \leftarrow b/x$
      if $x \nmid v$ **then**
        $v \leftarrow v + p$
      end if
      $v \leftarrow v/x$
    end while
  end while
  return u

Now the variables a/u and b/v can be divided independently. The addition in the if-part of the loop is just multiplexing instead of copying.

To use this algorithm in hardware the while loop has to be eliminated. To do this we transform the loops into a state machine. For this we write the algorithm more compact:

Algorithm 2.11 f^{-1} in $\mathbb{F}_{2^n}$ (Stein's algorithm compact)

Require: $deg(f), deg(g) < deg(p)$
  $a \leftarrow f$
  $b \leftarrow p$
  $u \leftarrow 1$
  $v \leftarrow 0$
  while $a \neq 1$ **do** $\{\mathbf{1}\}$
    $(a, b, u, v) \leftarrow f(a, b, u, v)$
    while $x \mid a$ **do** $\{\mathbf{2}\}$
      $(a, u) \leftarrow g(a, u)$
    end while
    while $x \mid b$ **do** $\{\mathbf{3}\}$
      $(b, v) \leftarrow g(b, v)$
    end while
  end while
  return u $\{\mathbf{4}\}$

where

$$f(a,b,u,v) = \begin{cases} (a,b,u,v), & x \mid a \\ (a+b,b,u+v,v), & x \nmid a,\ deg(a) < deg(b) \\ (a,a+b,u,u+v), & x \nmid a,\ deg(a) \geq deg(b) \end{cases}$$

$$g(c,w) = \begin{cases} (c/x,(w+p)/x), & x \nmid w \\ (c/x,w/x), & x \mid w \end{cases}$$

In algorithm 2.11 four lines are numbered. They are branching nodes inside the algorithm and are to be transformed into states:

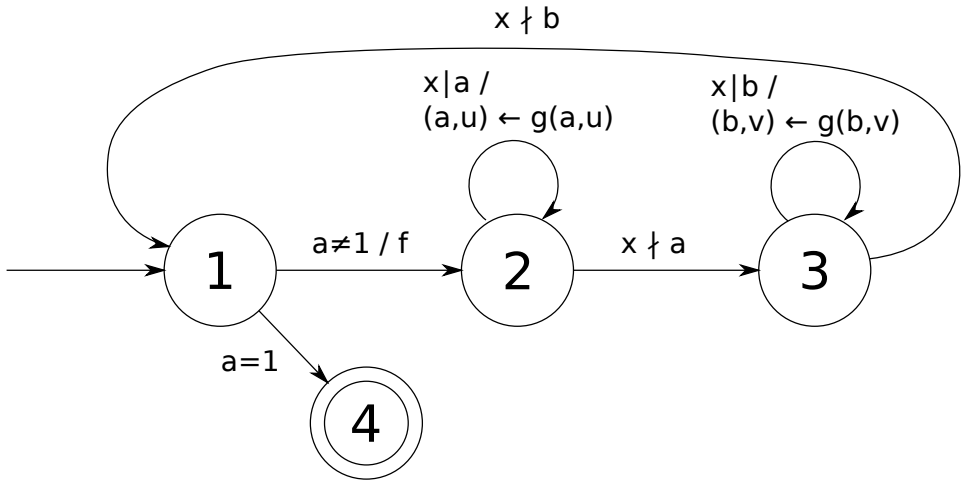

Figure 2.1: State machine for algorithm 2.11

The semantic of this state machine is that the conditions at the transitions are tested on fixed steps (i.e. a clock). When a transition is taken, the action after the slash is executed. State 1 is the initial state and 4 is a final state.

Because state 2 and 3 have no data dependency they can be driven in parallel. That means on each step the transitions which lead back to state 2 and 3 respectively can be taken in parallel. This idea is shown in figure 2.2.

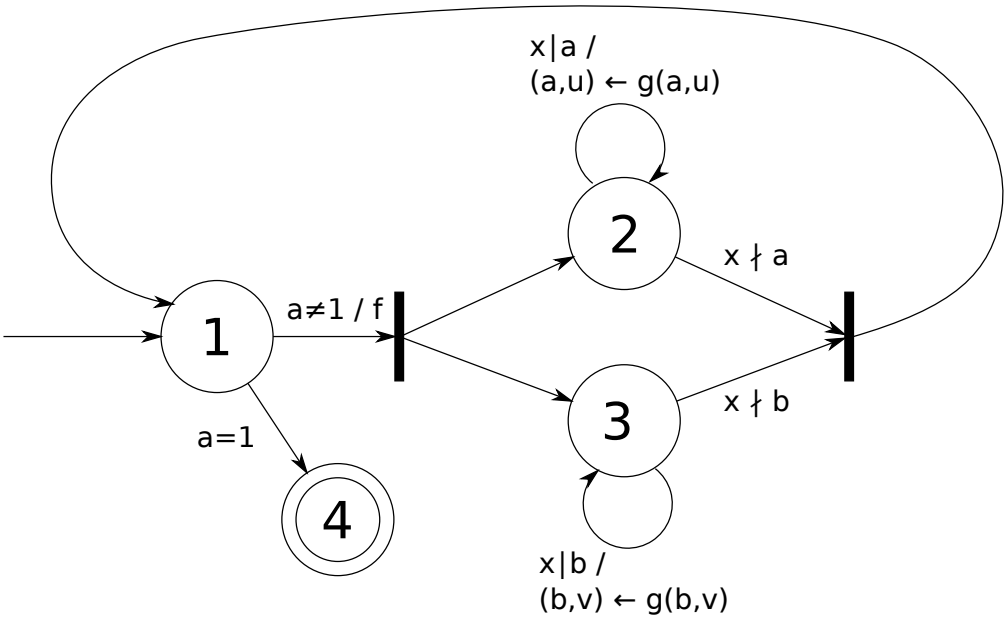

Figure 2.2: State machine for algorithm 2.11 (parallelized)

To use any of the presented inversion algorithms to calculate $\frac{f}{g} = f \cdot g^{-1}$ one can use the result as parameter for a multiplication. But the invariant can also be used to calculate the result of the division directly. See algorithm 2.12.

Algorithm 2.12 f/g in $\mathbb{F}_{2^n}$

$(a, u, v) \leftarrow \gcd((g, f, 0), (p, 0, 1))$
return u

2.3 Operations in $E(\mathbb{F}_{2^n})$

2.3.1 Affine coordinates

The concept of *affine points* refers to points in the Euclidean plane represented by a pair of elements from the underlying field. The operations are just the defined formulas in 1.17.

The only interesting part is the point multiplication. That is the implementation of definition 1.2 (exponentiation). The addition and point doubling are just direct implementations of the formulas given in 1.17.

The point multiplication is an implementation of the so called "fast multiplication".

Algorithm 2.13 aP in $E(\mathbb{F}_{2^n})$ (Point multiplication)

if $a < 0$ **then**
  $a \leftarrow -a$
  $P \leftarrow -P$
end if
$Q \leftarrow \infty$
while $a \neq 0$ **do**
  if $2 \nmid a$ **then**
    $Q \leftarrow Q + P$
  end if
  $P \leftarrow 2P$
  $a \leftarrow \lfloor a/2 \rfloor$
end while
return Q

2.3.2 Projective coordinates

The motivation of using another representation of coordinates is that the operations on elliptic curves (addition and point doubling) use one inversion in $\mathbb{F}_{2^n}$ each. Relatively to the multiplication an inversion may be very expensive. This can be seen in chapter 4.

Using projective coordinates those inversions can be replaced by multiple multiplications.

Definition 2.1
Let K be a field and $c, d \in \mathbb{Z}$. Now $\sim$ is an equivalence relation over $K^3 \setminus \{(0, 0, 0)\}$

defined by

$$(x_1, y_1, z_1) \sim (x_2, y_2, z_2) \Leftrightarrow (\exists \lambda \in K^\times) : x_1 = \lambda^c x_2, y_1 = \lambda^d y_2, z_1 = \lambda z_2 \,.$$

The equivalence classes are notated by the coordinates of a representative. That is (x, y, z) lies in the class

$$\begin{aligned}[x : y : z] &:= \{(x', y', z') \in K^3 \setminus \{(0,0,0)\} : (x', y', z') \sim (x, y, z)\} \\ &= \{(\lambda^c x, \lambda^d y, \lambda z) : \lambda \in K^\times\}\end{aligned}$$

The quotient space $K^3 \setminus \{(0,0,0)\}/\sim = \{[x : y : z] : (x, y, z) \in K^3 \setminus \{(0,0,0)\}\}$ is denoted as $\mathbf{K}P^3$.

If $[x : y : z] \in \mathbf{K}P^3$ every element in it can be used as representative especially the element $[x/z^c : y/z^d : 1]$ for some c and d and $z \neq 0$. That motivates the following theorem where $(\mathbf{K}P^3)^\times = \{[x : y : z] : (x, y, z) \in K^3, z \in K^\times\}$. Note that $K^\times = K \setminus \{0\}$ so the only condition for z is $z \neq 0$.

Theorem 2.2
Let K be a field and $f : K^2 \to (\mathbf{K}P^3)^\times$ with $f((x, y)) = [x : y : 1]$. Then f is a bijection with the inverse $\pi := f^{-1}([x : y : z]) = (x/z^c, y/z^d), c, d \in \mathbb{Z}$.

Proof. Since a bijective map has an uniquely determined inverse map we only have to show, that π is the inverse of f. That means showing that $\pi \circ f = id_{K^2}$ and $f \circ \pi = id_{(\mathbf{K}P^3)^\times}$ holds.

1. $\pi \circ f = id_{K^2}$

$$\pi(f((x, y))) = \pi([x : y : 1]) = \left(\frac{x}{1^c}, \frac{y}{1^d}\right) = (x, y)$$

2. $f \circ \pi = id_{(\mathbf{K}P^3)^\times}$

$$\begin{aligned}f(\pi([x : y : z])) & \\ &= f(\left(\frac{x}{z^c}, \frac{y}{z^d}\right)) \\ &= \left[\frac{x}{z^c} : \frac{y}{z^d} : 1\right] \\ &= \left[z^c \frac{x}{z^c} : z^d \frac{y}{z^d} : z \cdot 1\right] \qquad \text{by 2.1} \\ &= [x : y : z]\end{aligned}$$

□

Note: *The points $\mathbf{K}P^3 \setminus (\mathbf{K}P^3)^\times = \{[x : y : z] : (x, y, z) \in K^3, z = 0\}$ are called the line at infinity. They do not correspond to any point in K^2.*

Now we choose fixed c and d for the mapping and define the elliptic curves and the corresponding operations. Again this only done for fields of characteristic 2.

Definition 2.3 (López-Dahab projective coordinates, [6])
Let K be a finite field with $char(K) = 2$ and $c = 1, d = 2$. Then $\pi : (\mathbf{K}P^3)^\times \to K^2$, $\pi([x : y : z]) = (x/z, y/z^2)$.

The corresponding EC operations are:

- Inverse $-[x_1 : y_1 : z_1] = [x_1 : x_1 z_1 + y_1 : z_1]$
- Point doubling $2[x_1 : y_1 : z_1] = [x_3 : y_3 : z_3]$ where

$$\begin{aligned} z_3 &= z_1^2 \cdot x_1^2, \\ x_3 &= x_1^4 + b \cdot z_1^4, \\ y_3 &= bz_1^4 \cdot z_3 + x_3 \cdot (az_3 + y_1^2 + bz_1^4). \end{aligned}$$

- Point adding $[x_1 : y_1 : z_1] + [x_2 : y_2 : z_3] = [x_3 : y_3 : z_3]$ where

$$\begin{aligned} A_1 &= y_2 \cdot z_1^2, & D &= B_1 + B_2, & H &= C \cdot F, \\ A_2 &= y_1 \cdot z_2, & E &= z_1 \cdot z_2, & x_3 &= C^2 + H + G, \\ B_1 &= x_2 \cdot z_1, & F &= D \cdot E, & I &= D^2 \cdot B_1 \cdot E + x_3, \\ B_2 &= x_1 \cdot z_2, & z_3 &= F^2, & J &= D^2 \cdot A_1 + x_3, \\ C &= A_1 + A_2, & G &= D^2 \cdot (F + aE^2), & y_3 &= H \cdot I + z_3 \cdot J. \end{aligned}$$

In the following lemma the notation $E(\,\cdot\,)$ gets an overloaded meaning. $E(K^2)$ and $E(\mathbf{K}P^3)$ describe curves with $E(K^2) \subseteq K^2$ any $E(\mathbf{K}P^3) \subseteq \mathbf{K}P^3$ respectively. That is we describe the structure of the used coordinates.

Lemma 2.4
With $c = 1, d = 2$ is

$$\pi([x : y : z]) = \left(\frac{x}{z}, \frac{y}{z^2}\right)$$

an isomorphism from $E(\mathbf{K}P^3)$ to $E(K^2)$.

Proof. That the in 2.3 defined formulas are correct can be found in [6]. So π is a homomorphism. By theorem 2.2 π is an isomorphism. □

Now we have a transformation which is reversible from affine to projective coordinates. In projective coordinates the operations on elliptic curves do not involve the division in the underlying field. It is replaced by a some more multiplications. As we will see this has a lot more potential for parallelization.

2.4 ECMQV

The last algorithm that is introduced is for the key agreement. The implementation is taken directly from [4, p.195]. The algorithm is called "Elliptic Curve MQV" (ECMQV) after it's inventors Menezes, Qu and Vanstone. In [7] an improved version is presented which is said to fix some security issues. But as it is relatively new it has not been as much analyzed as ECMQV.

The protocol does not have much potential for parallelization. It is just presented for completeness.

The protocol uses the following parameters:

- *Cofactor* h where $|E(\mathbb{F}_{2^n})| = ord(P) \cdot h$.
- Identifiers for the participants: A and B.
- Key pairs for A and B: (Q_A, d_A) and (Q_B, d_B) respectively.

The used functions and notations are the following:

- $KDF(k)$ – a key derivate function (algorithm 2.16).
- $MAC_k(s)$ – a message authentication code function (algorithm 2.17).
- $\overline{P} = (\overline{x} \mod 2^{\lceil f/2 \rceil}) + 2^{\lceil f/2 \rceil}$, where $\overline{x}$ is the x-coordinate of P interpreted as integer and $f = \lfloor \log_2 n \rfloor + 1$ (this is roughly $\overline{x} \mod \sqrt{n}$ or halving of bit size).

Protocol 2.14 ECMQV

1. A:
   1.1 **Select** $k_A \in \{1, \dots n-1\}$ and **set** $R_A \leftarrow k_A P$.
   1.2 **Send** A and R_A to B.
2. B:
   2.1 **Validate** R_A (see algorithm 2.15).
   2.2 **Select** $k_B \in \{1, \dots n-1\}$ and **set** $R_B \leftarrow k_B P$.
   2.3 **Set** $s_B \leftarrow (k_B + \overline{R_B} d_B) \bmod n$.
   2.4 **Set** $Z = (x_Z, y_Z) \leftarrow h s_B (R_A + \overline{R_A} Q_A)$ and **verify** $Z \neq \infty$.
   2.5 **Set** $(k_1, k_2) \leftarrow KDF(x_Z)$.
   2.6 **Set** $t_B \leftarrow MAC_{k_1}(2 \parallel B \parallel A \parallel R_B \parallel R_A)$.
   2.7 **Send** B, R_B and t_B to A.
3. A:
   3.1 **Validate** R_B (see algorithm 2.15).
   3.2 **Set** $s_A \leftarrow (k_A + \overline{R_A} d_A) \bmod n$.
   3.3 **Set** $Z = (x_Z, y_Z) \leftarrow h s_A (R_B + \overline{R_B} Q_B)$ and **verify** $Z \neq \infty$.
   3.4 **Set** $(k_1, k_2) \leftarrow KDF(x_Z)$.
   3.5 **Set** $t \leftarrow MAC_{k_1}(2 \parallel B \parallel A \parallel R_B \parallel R_A)$ and **verify** $t = t_B$.
   3.6 **Set** $t_A \leftarrow MAC_{k_1}(3 \parallel A \parallel B \parallel R_A \parallel R_B)$.
   3.7 **Send** t_A to B.
4. B:
   4.1 **Set** $t \leftarrow MAC_{k_1}(3 \parallel A \parallel B \parallel R_A \parallel R_B)$ and **verify** $t = t_A$.

In the following the required algorithms are presented. The operation $\parallel$ stands for the concatenation.

Algorithm 2.15 Validate public key $Q = (x, y)$

Ensure: Returns true if Q is a valid public key.

```
if x, y ∉ F_{2^n} then
  return false
else if Q = ∞ then
  return false
else if Q ∉ E(F_{2^n}) then
  return false
else
  return true
end if
```

Algorithm 2.16 $KDF(k)$

Require: $H(s)$ – hash function with l_H bits output
Ensure: Key of l bits returned
  $m \leftarrow \left\lceil \frac{l}{l_H} \right\rceil$ {number of needed hashes}
  $d \leftarrow \epsilon$ {derived key (initially empty)}
  for $i = 1$ to m **do**
    $s \leftarrow s \parallel H(k, i)$
  end for
  return $s[0 : l - 1]$

Algorithm 2.17 $HMAC_k(m)$ – MAC based on a hash algorithm ([3, p.193])

Require: $H(m)$ – hash function
Require: $ipad = (36)_{16}$ and $opad = (5c)_{16}$
  return $H\left((k \oplus opad) \parallel H((k \oplus ipad) \parallel m)\right)$

3 Parallelization

3.1 Introduction

In this chapter we try to parallelize the algorithms that we introduced in chapter 2. The most interesting part here are the operations on elliptic curves because they utilize computational costly operations on the underlying field. So the operations take enough time that having multiple field implementations is legitimated.

3.2 Operations in $\mathbb{F}_{2^n}$

Here we have seen a small parallelization inside the loop of the division. More parallelization is not possible for the chosen algorithms because every step of the loop depends on the results of the previous step. The same holds for the multiplication.

3.3 Operations in $E(\mathbb{F}_{2^n})$

In the following we draw the operations in $E(\mathbb{F}_{2^n})$ as directed acyclic graphs (DAG). The figures have been generated using *dot* from the *Graphviz* suite ([1]).

Each node stands for the result of *one* operation. Exponentiation (x^2, x^4 etc.) is implemented in terms of multiplication. As in this case the exponent is only 2 or 4 the operation is unrolled.

The graphs have four types of nodes: domain parameter (boxes), input parameter (green ellipses), result nodes (red ellipses) and intermediate results (black ellipses). An edge (a, b) is added to the graph if there is a dependency of the value of b on the value of a.

Another "trick" lets us reason about parallelization more easily: the nodes are ordered by dependencies from top to bottom. That means if there is an edge (a, b) the node b is drawn on a lower level than a.

3.3.1 Affine coordinates

In affine coordinates two units for operations in $\mathbb{F}_{2^n}$ could be utilized in parallel.

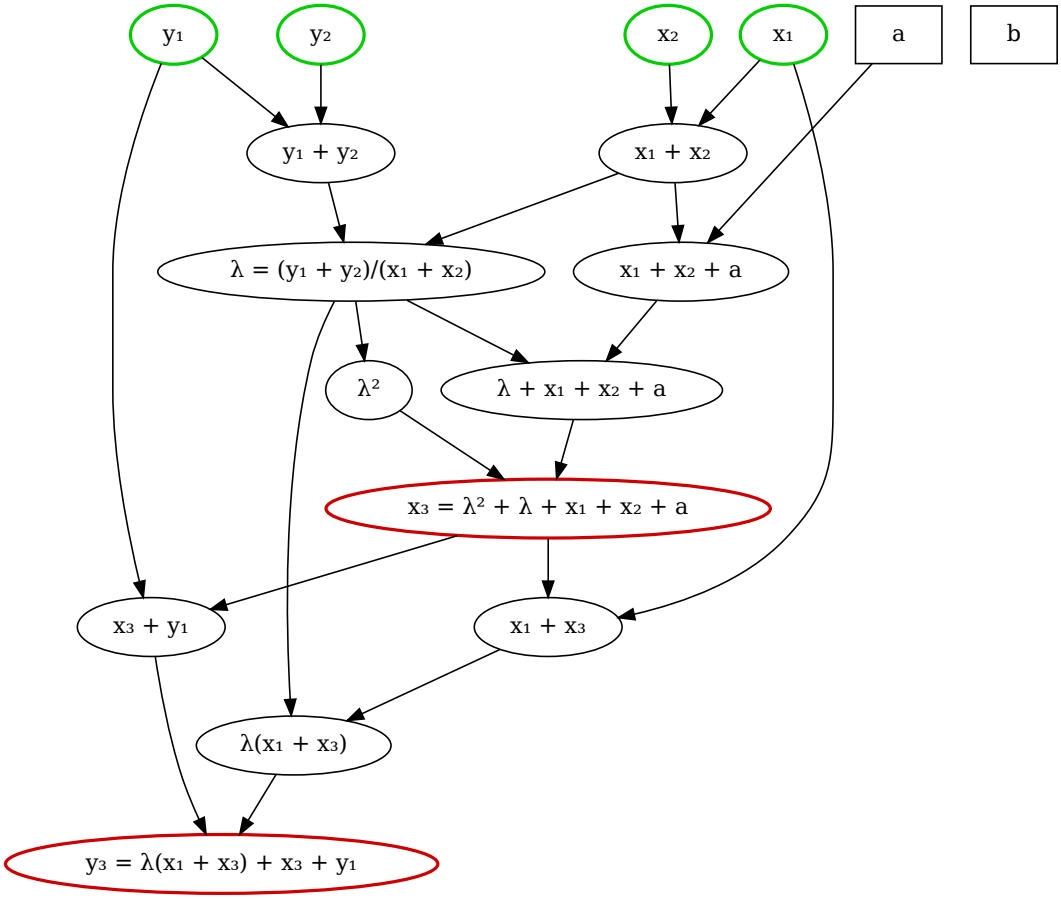

Figure 3.1: Data dependencies of the point addition using affine coordinates

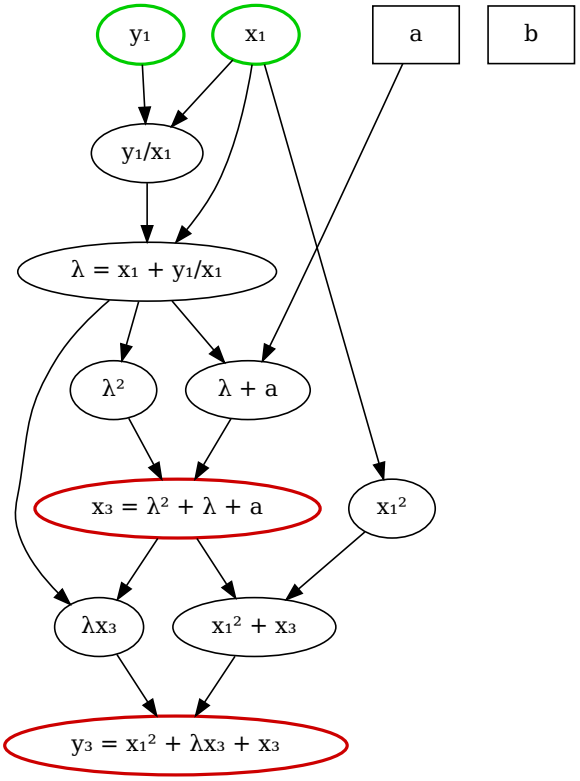

Figure 3.2: Data dependencies of the point doubling using affine coordinates

3.3.2 Projective coordinates

In projective coordinates four units for operations in $\mathbb{F}_{2^n}$ could be utilized in parallel. Five units would have been possible but the critical path would have been only one step shorter.

For projective coordinates only addition and multiplication is needed in those units. Division has to be done only once in the conversion from projective to affine coordinates after all necessary calculation have been done. So the needed space for those four units should not be very different from those two needed in affine coordinates (see chapter 4).

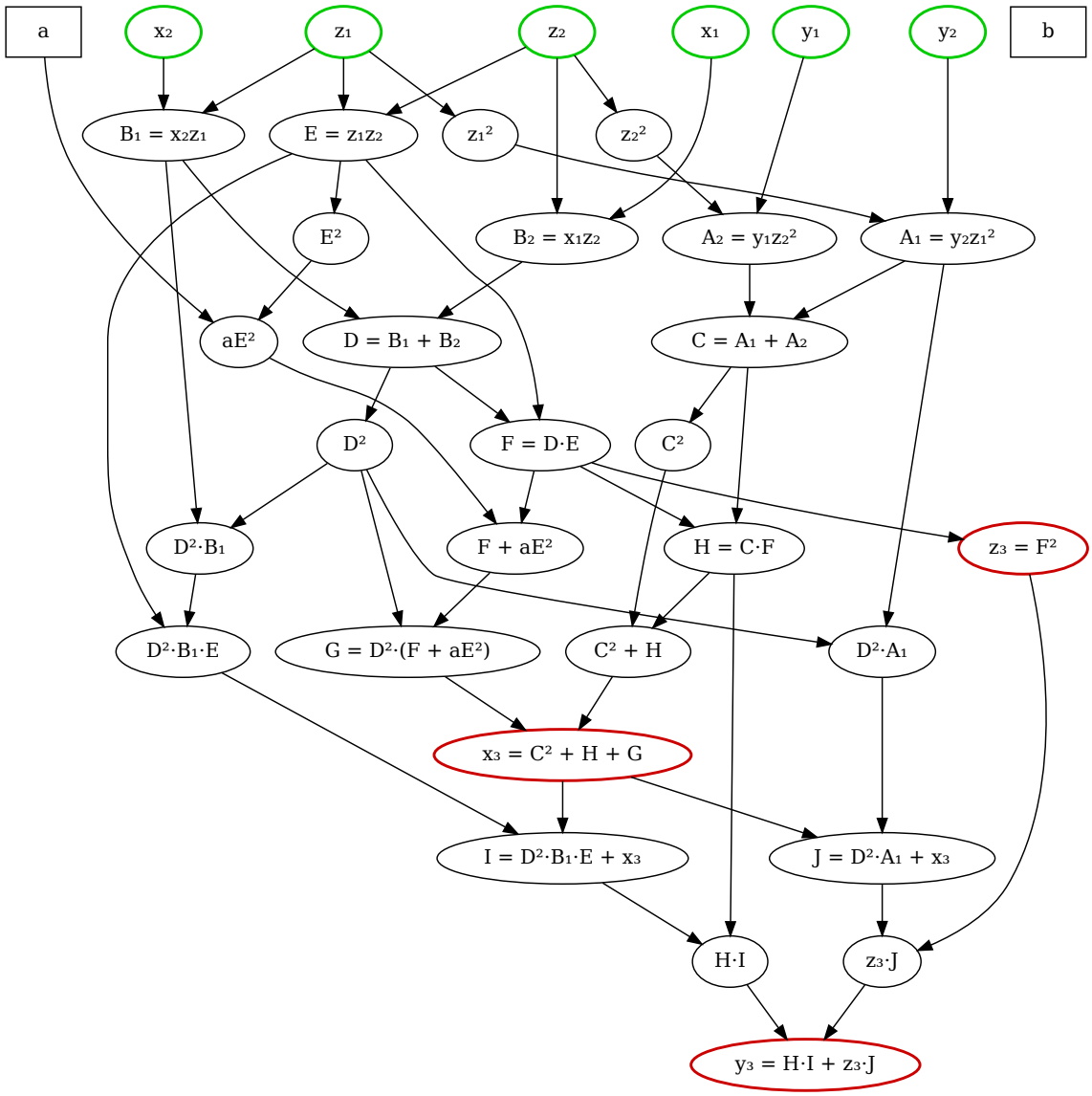

Figure 3.3: Data dependencies of the point addition using projective coordinates

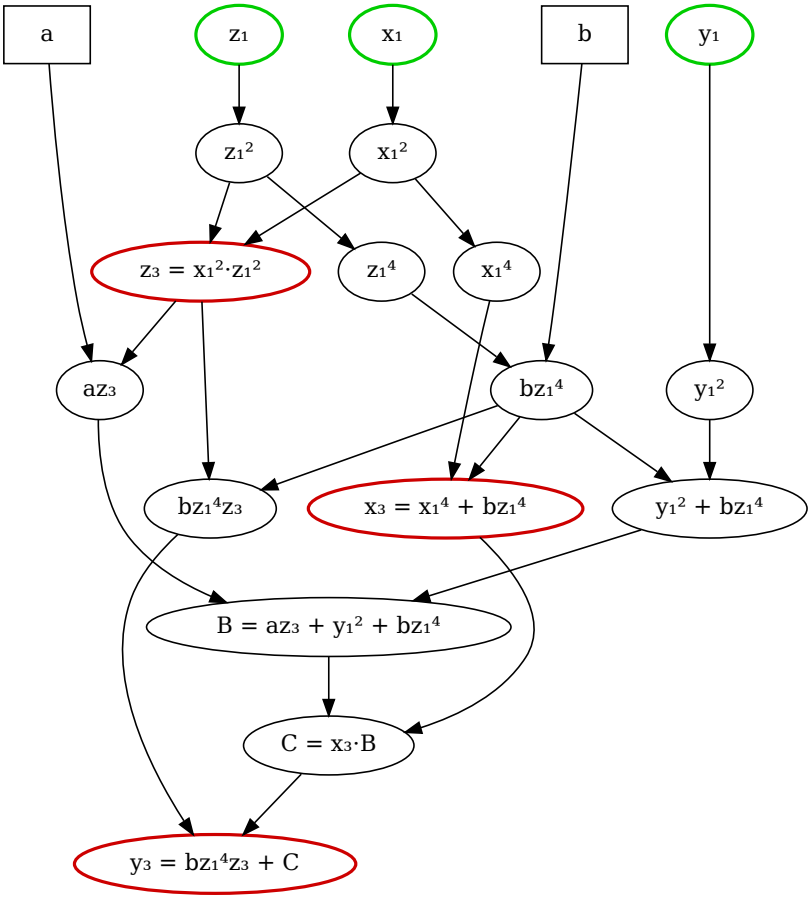

Figure 3.4: Data dependencies of the point doubling using projective coordinates

4 Results

4.1 Implementation

The implementation which has been used for the measurements consists of two parts. The operations in $\mathbb{F}_{2^n}$ have been realized using Verilog and the operations in $E(\mathbb{F}_{2^n})$ in SystemC. ECMQV will be run in Software so it has been realized in SystemC.

The Verilog implementation on the one hand allows a more fine grained measurement of the operations using a concrete technology library. On the other hand the SystemC implementation allows an easier exploration of the different algorithms and the parallelization.

4.2 Environment

4.2.1 Testing

To ensure a correct transformation of the theory into programming code a test suite has been implemented. It consists of hand written tests of special cases and automatically generated test vectors. The test vectors have been randomly generated using the algebra system *Sage* ([2]). All tests are based on parameters for for the elliptic curve "B-163" from [9].

The following table lists the implemented test cases for each module. Where P is a random point on the curve and Q is the generator chosen.

Finite field	Elliptic curve
• $0 + 1 = 1$ • $1 + 1 = 0$ • $1 \cdot x = x$ • $0 \cdot x = 0$ • $x/1 = x$ • $0/x = 0$ • 1000 vectors for $+$, $\cdot$ and $/$	• $P + (-P) = \infty$ • $0 \cdot P = \infty$ • $1 \cdot P = P$ • $kP + (-k)P = \infty$ • $ord(Q) \cdot P = \infty$ • en- and decryption using ElGamal • $P \in E(\mathbb{F}_{2^n})$ for a point on the curve and one not • 1000 vectors for $+$ and $\cdot$

Table 4.1: Test cases for the operations in $\mathbb{F}_{2^n}$ and $E(\mathbb{F}_{2^n})$

4.2.2 Measurements

The measurements have been done using ModelSim. It allows a simulation of SystemC and Verilog implementations side by side. To get concrete clock speeds the $\mathbb{F}_{2^n}$ operations in Verilog have then been synthesized by a RTL compiler from Cadence using a technology library for 250nm silicon germanium structures. These are used in the "SMART" project.

The measured critical path allowed the clock to be run at around 900MHz.

The ratio in size of a unit for finite fields with and without division is about three (9000 logic blocks to 3000 logic blocks).

4.3 Timings

In table 4.2 different timings are listed for an implementation of operations in $E(\mathbb{F}_{2^n})$. The additional conversion from projective back to affine coordinates has been added to a separate row. These timings are presented in graphical form in the figures 4.1 and 4.2.

Type	Time Addition in NS (clock cycles)	Time Multiplication in NS (clock cycles)
Affine Sequential	97,039 (10,782)	4,565,589 (507,288)
Affine Parallelized	96,908 (10,768)	4,561,301 (506,811)
Projective Sequential	21,706 (2,412)	880,817 (97,869)
Projective Seq. + Conversion	224,563 (24,951)	1,084,287 (120,476)
Projective Parallel	10,743 (1194)	430,439 (47,826)
Projective Par. + Conversion	213,599 (23,733)	633,884 (70,432)

Table 4.2: Timings for different implementations of operations in $E(\mathbb{F}_{2^n})$

Looking at these timings one can see that the improvement using parallelization in affine coordinates is negligible. This is because the operating time is dominated by a long running division which can't be parallelized.

On the other hand the operations in projective coordinates are build of a lot of small operations which can be parallelized very good. Comparing the sequential implementation of affine coordinates with the parallelized projective coordinates we see a speedup of around 10. The parallelization makes a speedup of 2.

As expected one addition in projective coordinates with a subsequent conversion into affine coordinates (involving a division) is slower than the same operation in plain affine coordinates. Since the ECMQV key agreement protocol uses an addition only after a multiplication (see protocol 2.14) this is negligible because here only one conversion has to be done.

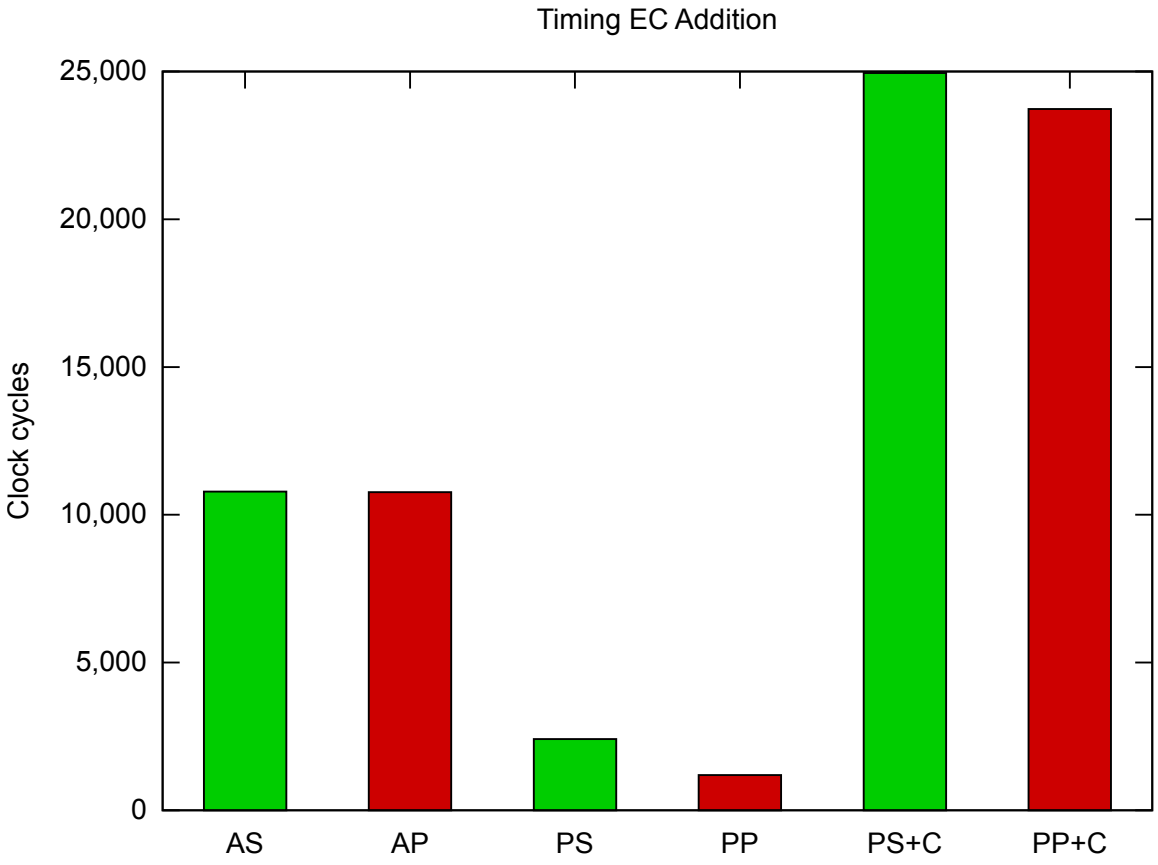

Figure 4.1: Timings for EC addition

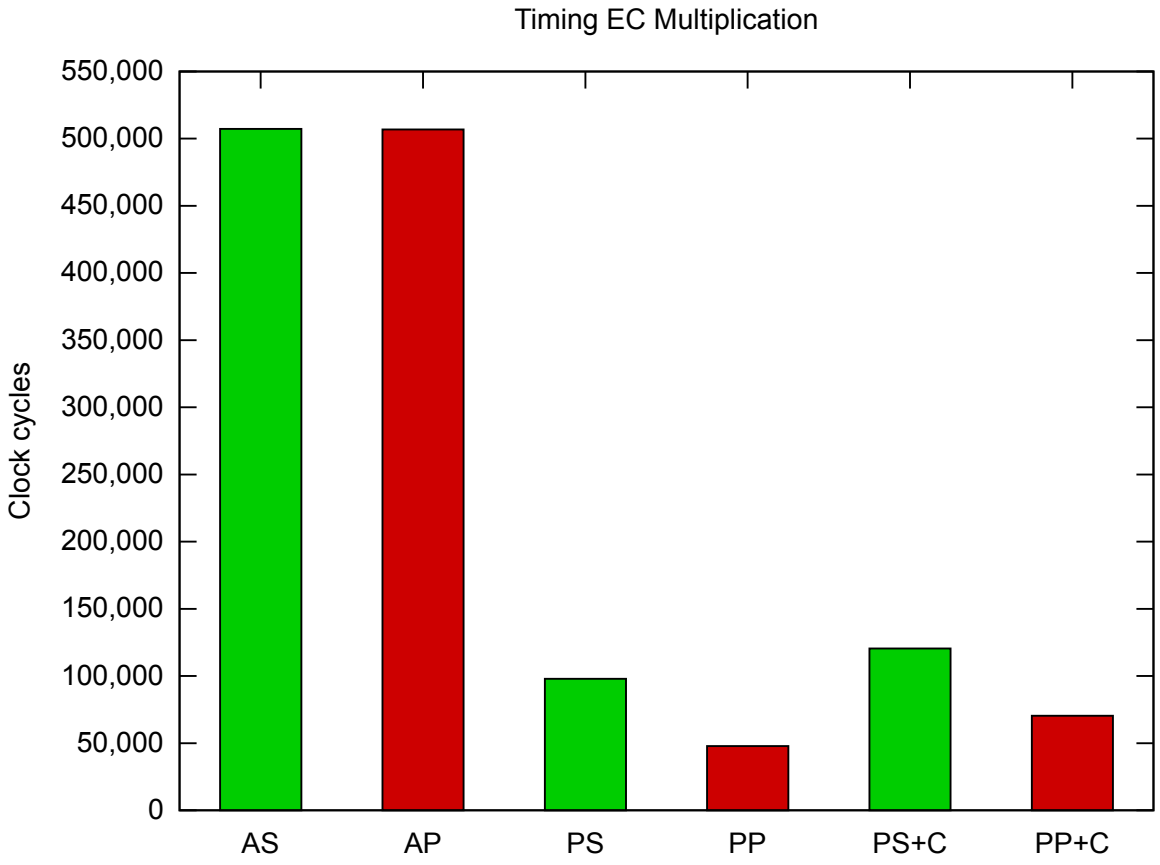

Figure 4.2: Timings for EC multiplication

To proof that there is a speed up in the ECMQV protocol, it has been measured, too. The times include all protocol stages on one core. But both participants do the same work so the ratios should be preserved. The timings are listed in table 4.3. The necessary conversion from projective to affine coordinates in before sending data have been included.

Type	Running time in μs
Affine Sequential	126,333
Projective Parallel	12,967
Speedup	9.74

Table 4.3: Comparison of timings for ECMQV

So the speedup of the single operations are propagated almost completely through the protocol.

4.4 Suggestions for future research

In principal the improvements are very good. Maybe another type of projective coordinates would scale better in parallelization. The ratio of speedup to the number of used units is about $\frac{1}{2}$ (4.3).

For the project the general running times have to be improved. The cause of the long running times (12 seconds for one ECMQV pass) is the too much simplified handling of bit operations. The processing of single bits in algorithm 2.3 and 2.10 costs in the current implementation one clock cycle each. So there is a lot of room for improvement.

Some more engineering has to be done to transform the EC implementation from SystemC to Verilog. With this at hand the implementations could be tested on a FPGA.

Last but not least the parameters for the used elliptic curve have to be chosen. The parameters used in the tests construct the smallest curve which is suggested by the NIST ([9]). To have a key space that is big enough not be searched completely in the future the next larger curve may be used.

Bibliography

[1] Graphviz – graph visualization software. `http://www.graphviz.org/`. 27

[2] Sage: Open source mathematics software. `http://www.sagemath.org/`. 32

[3] Albrecht Beutelspacher, Heike B. Neumann, and Thomas Schwarzpaul. *Kryptographie in Theorie und Praxis*. Vieweg+Teubner, Wiesbaden, 2. edition, 2010. 4, 26

[4] Darrel Hankerson, Scott Vanstone, and Alfred J. Menezes. *Guide to Elliptic Curve Cryptography*. Springer, Berlin, 1. edition, 2004. 24

[5] Christian Karpfinger and Kurt Meyberg. *Algebra: Gruppen - Ringe - Körper*. Spektrum Akademischer Verlag, 1. edition, 2008. 5, 6

[6] Julio López and Ricardo Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, Lecture Notes in Computer Science, pages 632–632. Springer Berlin / Heidelberg, 1999. 23

[7] P. Augustin Sarr, Philippe Elbaz-Vincent, and Jean-Claude Bajard. A secure and efficient authenticated diffie-hellman protocol. Cryptology ePrint Archive, Report 2009/408, 2009. `http://eprint.iacr.org/2009/408`. 24

[8] SECG. SEC 1: Elliptic curve cryptography. `http://www.secg.org/download/aid-780/sec1-v2.pdf`, September 2000. [Online; accessed 07-June-2010].

[9] SECG. SEC 2: Recommended elliptic curve domain parameters. `http://www.secg.org/download/aid-784/sec2-v2.pdf`, September 2005. [Online; accessed 07-June-2010]. 32, 36

[10] Dietmar Wätjen. *Kryptographie - Grundlagen, Algorithmen, Protokolle*. Spektrum Akademischer Verlag, 2. edition, 2008. 10