Montgomery-Kurven in der Kryptographie

Bachelor-Arbeit

zur Erlangung des Grades

Bachelor of Science (B.Sc.)

Mathematik

an der Naturwissenschaftlichen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

vorgelegt am 23.08.2023

von **Nico Köslich**

Betreuer: Prof. Dr. Wolfgang Ruppert

Inhaltsverzeichnis

1 Einleitung **1**

2 Elliptische Kurven **2**

2.0.1 Implementierung von Elliptischen Kurven 3

2.0.2 Additionsgesetz 4

2.0.3 Multiplikation 4

2.0.4 Additionsalgorithmus 4

2.0.5 Implementierung des Additionsalgorithmus: 6

2.1 Elliptische Kurven über endlichen Körpern 6

2.1.1 Implementierung von elliptischen Kurven über endlichen Körpern 7

2.1.2 Addition von elliptischen Kurven auf endlichen Körpern 8

2.1.3 Implementierung der Addition auf endlichen Körpern 8

2.1.4 Satz von Hasse 9

2.1.5 Das Problem diskreter Logarithmus auf elliptischen Kurven . . . 9

2.1.6 The double-and-add Algorithm 10

3 Montgomery Kurven **11**

3.1 Projektive Koordinaten 11

3.2 Addition 11

3.3 Montgomery Ladder und der Binäralgorithmus 13

4 Anwendungen **16**

4.1 Diffie-Hellman Problem 16

4.2 Diffie-Hellman über endlichen Körpern 16

4.3 Diffie-Hellman mit elliptischen Kurven 17

4.4 ECDSA 17

5 Curve25519 **19**

5.1 Eigenschaften von Curve25519 19

5.2 Diffie-Hellman 20

5.3 Signatur 22

6 Anhänge **24**

6.1 Sage-Code 24

6.2 Python-Code 24

Literatur **26**

1 Einleitung

Die Kryptograhpie beschäftigt sich mit der Verschlüsselung von Texten. Dies ist heute relevanter denn je, da inzwischen der Hauptteil der Kommunikation elektronisch abläuft und dabei die Daten durchs Internet verbreitet werden. Da es keine Sicherheit gibt wer die Daten sieht, ist es essenziell, seine Daten zu verschlüsseln und die Kommunikation im Internet sicher zu gestalten. Hiermit beschäftigt sich die moderne Kryptographie. Interessant ist hierbei, dass ein Großteil der Kommunikation nicht mehr von Mensch zu Mensch statt findet, sondern zwischen Maschinen. Ein Beispiel hierfür wäre die Berechnung aktueller Börsenkurse, also Bereiche, in denen es wichtig ist, sich weltweit auszutauschen und Daten zusammenzufügen. Es kann leicht vorkommen, dass Daten gestört oder manipuliert werden. Um die Verhinderung dessen zu gewährleisten, wurden in den letzten Jahrzehnten etliche Protokolle entwickelt, die eben dies garantieren sollen. In der modernen Kryptographie spielen elliptische Kurven im Bereich der asymmetrischen Verschlüsselung eine wichtige Rolle, da sie im Vergleich zu RSA bei einer gleichwertigen Sicherheit deutlich kleinere Schlüssel brauchen. Elliptische Kurven sind eine algebraische Menge von Lösungen kubischer polynomialen Gleichungen. Diese Lösungen beinhalten eine Gruppenstruktur, was darauf schließen lässt, dass sich die Prinzipien der herkömmlichen asymmetrischen Verschlüsselungen, wie RSA, auf sie übertragen lassen. Interessant für die Kryptographie ist die Betrachtung von solchen Gleichungen über endlichen Körpern. 2006 hat Bernstein ein Paper veröffentlicht, in welchem er eine elliptische Kurve zur Verschlüsselung vorschlägt, namentlich Curve25519. Diese Kurve ist eine Montgomery-Kurve. Montgomery-Kurven sind eine Untermenge der elliptischen Kurven. Auf ihnen ist eine deutlich schnellere Arithmetik vorhanden. Im Jahr 1987 zeigte dies Montgomery in seiner Veröffentlichung 'Speeding the Pollard and Elliptic Curve Methods of Factorization'. Curve25519 wird aktuell unter anderem bei den Anwendungen Signal, Whatsapp oder auch bei SSH zur Verschlüsselung eingesetzt. Warum Bernstein im Jahr 2006 für eine sichere Verschlüsselung gerade diese Kurve vorgeschlagen hat, liegt an einigen Eigenschaften, wie zum Beispiel, dass jeder 32-byte String ein öffentlicher beziehungsweise privater Schlüssel ist. Außerdem wird ein Hash über den öffentlichen und privaten Schlüssel als so genanntes shared secret eingesetzt. Welches Anwendung in der Authentifizierung von Rechnern in verschiedensten Kommunikationsprotokollen findet. Viele elliptische Kurven eignen sich aus verschiedensten Gründen nicht für einen kryptographischen Einsatz. Diese Kurve allerdings ist aufgrund ihrer besonderen Arithmetik, und weiterer Eigenschaften, sicher gegen verschiedenste Arten von Angriffen und somit besonders für den praktischen Einsatz geeignet. Einige der Angriffe auf elliptische Kurven werden im Laufe dieser Arbeit vorstellen. Ziel dieser Arbeit ist es ein grundlegendes Verständnis für elliptische Kurven zu schaffen, die Anwendung in der Kryptographie exemplarisch an verschiedenen Methoden zu zeigen und schließlich eine Implementierung von elliptischen Kurven in Python3 zu erarbeiten. Ich werde meinen Ansatz darlegen, wie ich die Kurven implementiert habe und genauer auf Besonderheiten der Implementierung in Python eingehen. Zudem stelle ich am Ende noch eine mögliche Implementierung von Methoden, wie zum Beispiel des Diffie-Hellman Schlüsselaustausches, mit Curve25519 vor.

2 Elliptische Kurven

Eine elliptische Kurve wird definiert als die Menge der Lösungen von Gleichungen der Form $Y^2 = X^3+AX+B$, wobei zusätzlich gilt $4A^3+27B^2 \neq 0$. Diese zusätzliche Bedingung ist notwendig, damit es an jedem Punkt eine Tangente gibt. Die resultierende Form nennt man auch Weierstraß Form. Die beschriebene Struktur hat die Eigenschaft, dass man für Punkte, die in der Lösungsmenge einer solchen Gleichung liegen, eine Addition definieren kann. Mit dieser Addition erhält man dann eine Gruppenstruktur. Geometrisch kann man sich die Addition von Punkten P und Q folgendermaßen vorstellen. Man zeichnet die Gerade, welche die beiden Punkte verbindet. Diese Gerade schneidet dann noch einen dritten Punkt R auf der Kurve. Das Ergebnis ist dann der Punkt R', welcher dem an der x-Achse gespiegelten Punkt R entspricht. Nun muss man allerdings noch einen Punkt einführen, sodass auch P+P'eine Lösung hat, wobei P' der an der x-Achse gespiegelte Punkt P ist. Diesen neuen Punkt nennt man $\mathcal{O}$. In der folgenden Abbildung gilt $P = Q$, also ist die verbindende Gerade der Punkte die Tangente am Punkt P. [1]

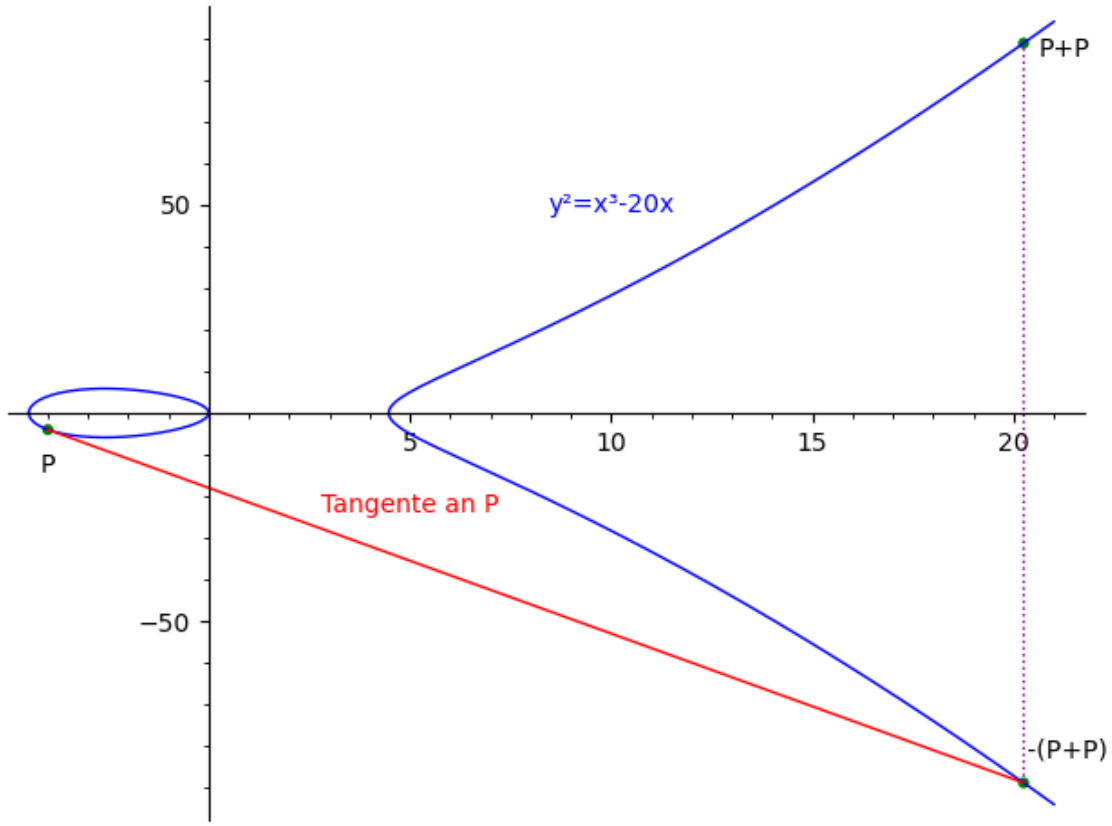

Abbildung 1 [2]

[1] [5] ab Seite 299
[2] Code siehe 6.1

2.0.1 Implementierung von Elliptischen Kurven

Zunächst wurden elliptische Kurven von Weierstraß Form als Klasse in Python3 implementiert. Die Objekte der Klasse sollen die Attribute A und B haben, diese sind die Attribute, die eine elliptische Kurve von Weierstraß Form definieren. Zusätzlich sollen Objekte der Klasse auch eine Funktion haben, um zu sehen, ob das Objekt auch die Bedingung $4A^3 + 27B^2 \neq 0$ erfüllt.

```
class elliptischeKurve:
   def __init__(self,A,B):
      self.A = A
      self.B = B

   def __str__(self):
       return "y**2 = x**3 + "+str(self.A)+"*x + "+str(self.B)
#Ueberpruefung der Bedingung das ueberall
#eine Tangente existiert
   def Korrektheit(self):
      if 4*self.A**3+27*self.B**2 == 0:
         return "Dies ist keine Kurve von Weierstrass Form"
      else:
         return True
```

Um Punkte auf einer elliptischen Kurve zu Implementieren benötigt man eine Klasse mit den Attributen x-Koordinate, y-Koordinate und schließlich noch die Kurve, auf welcher der entsprechende Punkt liegt. Man benötigt nun noch einen Punkt, der das neutrale Element darstellt. Dies wurde hier wie folgt umgesetzt, wenn die x-Koordinate als Wert den String O enthält, wird der Punkt als neutrales Element betrachtet. Eine mögliche Implementierung sieht dann wie folgt aus:

```
class punkte:
   def __init__(self,x,y,Kurve):
      self.x = x
      self.y = y
      self.Kurve = Kurve
#wenn man P eingibt soll der Punkt im Format x: x-Wert
#y: y-Wert ausgegeben werden
   def __str__(self):
      return "x: " + str(self.x) + "y: " + str(self.y)
#Addition soll mit dem + Operator aufrufbar sein
   def __add__(self, other):
      return Addition(self,other)

#liegen die Punkte wirklich auf der Kurve?
#wenn x-Koordinate O dann ist dieser Punkt
#als neutrales Element zu betrachten
#try block falls Punkt neutrales element dann Punkt.x string
```

```
#und kein float oder int
   def KorrektheitderPunkte(self):
      try:
         if self.Kurve.Korrektheit()==True and self.y**2 ==
            self.x**3+self.Kurve.A*self.x+self.Kurve.B:
            return True
         else:
            return False
      except TypeError:
         if self.x == "O":
            return True
         else:
            return False
```

2.0.2 Additionsgesetz

Sei E eine Elliptische Kurve. Dann ist das Additionsgesetz folgendermaßen definiert:

i) $P + \mathcal{O} = \mathcal{O} + P = P$ für alle $P \in E$ (Identität)

ii) $P + (-P) = \mathcal{O}$ für alle $P \in E$ (Inverse)

iii) $(P + Q) + R = P + (Q + R)$ für alle $P, Q, R \in E$ (Assoziativität)

iv) $P + Q = Q + P$ für alle $P, Q \in E$ (Kommutativität)

2.0.3 Multiplikation

Für einen Punkt $Q \in E$ sei die Multiplikation mit $n \in \mathbb{N}$ definiert als:

$$n \cdot Q = Q + Q + \cdots + Q \text{ (n-mal)}$$

2.0.4 Additionsalgorithmus

Sei $E : Y^2 = X^3 + Ax + B$ eine elliptische Kurve und seien P_1 und P_2 Punkte auf E.

i) Wenn $P_1 = \mathcal{O}$, dann gilt $P_1 + P_2 = P_2$.

ii) Wenn $P_2 = \mathcal{O}$, dann gilt $P_1 + P_2 = P_1$.

iii) Ansonsten schreibe $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$.

iv) Wenn $x_1 = x_2$ und $y_1 = -y_2$, dann gilt $P_1 + P_2 = \mathcal{O}$

v) Definiere nun λ folgendermaßen:

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{wenn } P_1 \neq P_2 \quad \text{und } P_1 \neq -P_2 \\ \frac{3x_1^2 + A}{2y_1} & \text{wenn } P_1 = P_2 \end{cases}$$

und sei nun $x_3 = \lambda^2 - x_1 - x_2$ und $y_3 = \lambda(x_1 - x_3) - y_1$. Dann ist $P_1 + P_2 = (x_3, y_3)$.

Beweis: Teile i) und ii) folgen direkt aus den Eigenschaften der Addition. Teil iv) entsteht dadurch, dass sich wenn die beiden Punkte verbunden werden, eine vertikale Linie ergibt, welche keinen weiteren Punkt auf der Kurve schneidet, da Y^2 nur maximal 2 Wurzeln hat. Für v) gilt, wenn $P_1 \neq P_2$, dann ist λ die Steigung der Geraden, die die beiden Punkte verbindet. Falls $P_1 = P_2$, dann ist λ die Steigung der Tangente an P_1. In beiden Fällen ist die Gerade G durch $Y = \lambda X + \nu$ mit $\nu = y_1 - \lambda x_1$ gegeben. Setze nun die Gleichung für G in die von E ein:

$$(\lambda X + \nu)^2 = X^3 + AX + B$$

, also

$$X^3 - \lambda^2 X^2 + (A - 2\lambda\nu)X + (B - \nu^2) = 0$$

Wir wissen nun, dass zwei der drei Lösungen x_1 und x_2 sind. Die dritte Lösung wird nun analog x_3 genannt. Die Faktoren sehen dann so aus:

$$X^3 - \lambda^2 X^2 + (A - 2\lambda\nu)X + (B - \nu^2) = (X - x_1)(X - x_2)(X - x_3).$$

Multipliziere nun die rechte Seite aus und betrachte den Koeffizient von X^2. Dieser ist $x_1 - x_2 - x_3$, was nun, in Betrachtung der linken Seite, äquivalent zu $-\lambda^2$ sein muss. Diese Gleichheit führt zu der Umstellung $x_3 = \lambda^2 - x_1 - x_2$. Dann ist die y-Koordinate des dritten Schnittpunkts von G durch E gegeben durch $\lambda x_3 + \nu$. Um $P_1 + P_2$ zu erhalten, muss noch an der x-Achse gespiegelt werden, also y mit dem additivem Inversen ersetzt werden.

Beispiel: Wähle $E : y^2 = x^3 - 20x$ und als Punkt $P = (-4, -4)$. Setze nun ein, um zu sehen, ob P auf E liegt:

$$(-4)^2 = 16 = -64 + 80 = (-4)^3 + 20 \cdot 4.$$

Damit liegt P auf E. Berechne nun $P + P$

$$\lambda = \frac{3 \cdot (-4^2) - 20}{2 * (-4)} = \frac{28}{-8}, x_3 = \frac{-28^2}{8} + 4 + 4 = \frac{81}{4},$$

$$y_3 = \frac{-28}{8}\left(-4 - \frac{81}{4}\right) + 4 = \frac{711}{8}$$

$$P + P = (x_3, y_3) = \left(\frac{81}{4}, \frac{711}{8}\right)$$

Folgendermaßen würde das Beispiel aussehen, wenn man es mit Python und der Implementierung anwendet:

```
>>> import elliptischeKurven as EK #Modul laden
>>> a = EK.elliptischeKurve(-20,0) #a als elliptische Kurve
   definieren mit Parametern (A,B)
>>> P = EK.punkte(-4,-4,a) #Punkt auf der Kurve definieren
>>> P+P #Addition durchfuehren
x: 20.25 y: 88.875
```

2.0.5 Implementierung des Additionsalgorithmus:

```
def Addition(Punkt1, Punkt2):
#ist der Algorithmus ueberhaupt anwendbar
   if not (Punkt1.Kurve == Punkt2.Kurve):
      return False
#Schritte i) und ii)
   if Punkt1.x == "O":
      Punkt3 = Punkt2
   else:
      if Punkt2.x == "O":
         Punkt3 = Punkt1
      else:
#Fall iv)
         if Punkt1.x == Punkt2.x and Punkt1.y == -Punkt2.y:
            Punkt3 = punkte("O",42,Punkt1.Kurve)
#Fall v)
         else:
            if Punkt1 == Punkt2:
               l = (3*(Punkt1.x**2)+Punkt1.Kurve.A
                  )/(2*Punkt1.y)
            else:
               l = (Punkt2.y - Punkt1.y)/(Punkt2.x - Punkt1.x)
            x3 = l**2-Punkt1.x-Punkt2.x
            y3 = l*(Punkt1.x-l**2+Punkt1.x+Punkt2.x)-Punkt1.y
            Punkt3 = punkte(x3,y3,Punkt1.Kurve)
   return Punkt3
```

2.1 Elliptische Kurven über endlichen Körpern

Die Betrachtung von Elliptischen Kurven über endlichen Grundkörpern ist essenziell für die Anwendung in der Kryptographie. [3]

Definition: Sei $p \geq 3$ eine Primzahl. Dann heißt E eine elliptische Kurve über $\mathbb{F}_p$, wenn

[3] [5] ab S. 306

E von der Form$E : Y^2 = X^3 + AX + B$ mit $A, B \in \mathbb{F}_p$ und $4A^3 + 27B^2 \neq 0$ ist. Die Menge der Punkte auf E mit Koordinaten in $\mathbb{F}_p$ ist dann gegeben durch:

$$E(\mathbb{F}_p) = \{(x, y) : x, y \in \mathbb{F}_p \text{ , so dass } y^2 = x^3 + Ax + B\} \cup \{\mathcal{O}\}.$$

2.1.1 Implementierung von elliptischen Kurven über endlichen Körpern

Im Wesentlichen ist die Implementierung analog zu 2.0.1, die Klassen bekommen jedoch zusätzlich noch eine Zahl $p \in \mathbb{N}$ als zusätzliches Attribut und alle Rechnungen sind nun modulo p.

```
class elliptischeKurveUeberEndlichemKoerper:
    def __init__(self, A, B,p):
        self.p = p
        self.A = A%self.p
        self.B = B%self.p
    def __str__(self):
        return "y**2 = x**3 + " + str(self.A) + "*x + " +
            str(self.B)
#ueberpruefung der Bedingung das ueberall eine Tangente
#existiert
    def Korrektheit(self):
        if (4 * self.A ** 3 + 27 * self.B ** 2)%self.p == 0:
            return "Dies ist keine Kurve von Weierstrass Form"
        else:
            return True
#Klasse fuer einzelne Punkte mod p
class punkteAeK:
    def __init__(self,x,y,Kurve):
        self.p = Kurve.p
        try:
            self.x = x % self.p
        except TypeError:
            if x == "O":
                self.x = x
        self.y = y % self.p
        self.Kurve = Kurve
    def __str__(self):
        return "x: " + str(self.x) + "y: " + str(self.y)
    def KorrektheitderPunkte(self):
        try:
            if self.Kurve.Korrektheit()==True and
                ((self.y**2)%self.Kurve.p)==(((self.x**3)%self.Kurve.p
                +(self.Kurve.A*self.x) % self.Kurve.p +
                self.Kurve.B % self.Kurve.p) %self.Kurve.p):
                return True
            else:
                return False
```

```
        except TypeError:
            if self.x == "O":
                return True
            else:
                return False
```

2.1.2 Addition von elliptischen Kurven auf endlichen Körpern

Sei E eine elliptische Kurve über $\mathbb{F}_p$ und seien P und Q Punkte in $E(\mathbb{F}_p)$.

i) Der Additionsalgorithmus (2.0.4), angewandt auf P und Q, führt wieder zu einem Punkt in $E(\mathbb{F}_p)$. Dieser Punkt wird als P+Q bezeichnet.

ii) Diese Addition auf $E(\mathbb{F}_p)$ erfüllt alle Eigenschaften, wie in 2.0.2 dargestellt. Also ist $E(\mathbb{F}_p)$ eine endliche Gruppe.

2.1.3 Implementierung der Addition auf endlichen Körpern

Im Prinzip erfolgt die Implementierung der Addition auf endlichen Körpern ähnlich zu 2.0.5 außer, dass nun für ein $n \in \mathbb{N}$ modulo n gerechnet wird.

```
def AdditionAeK(Punkt1, Punkt2):
#ist der Algorithmus ueberhaupt anwendbar
    if Punkt1.Kurve == Punkt2.Kurve:
        if not (Punkt1.KorrektheitderPunkte()==True and
            Punkt2.KorrektheitderPunkte()==True):
            print("Korrektheit")
            return False
    else:
        print("Kurve nicht gleiches Objekt wie Kurve")
        return False
# Additionsalgorithmus in Kapitel ueber elliptische Kurven
# Schritte i) und ii)
    if Punkt1.x == "O":
        Punkt3 = Punkt2
    else:
        if Punkt2.x == "O":
            Punkt3 = Punkt1
        else:
#Fall iv)
            if Punkt1.x == Punkt2.x and Punkt1.y ==
                pow(Punkt2.y,-1,Punkt2.Kurve.p):
                Punkt3 = punkteAeK("O",42,Punkt1.Kurve)
#Fall v)
            else:
```

```
            if Punkt1.x == Punkt2.x and Punkt1.y == Punkt2.y:
                l = 3*(Punkt1.x**2) + Punkt1.Kurve.A %
                    Punkt2.Kurve.p *
                    pow((2*Punkt1.y),-1,Punkt2.Kurve.p) %
                    Punkt2.Kurve.p
            else:
                l = ((Punkt2.y -
                    Punkt1.y)%Punkt2.Kurve.p)*pow((Punkt2.x -
                    Punkt1.x),-1,Punkt2.Kurve.p) % Punkt2.Kurve.p
            x3 = (l**2-Punkt1.x-Punkt2.x) %Punkt2.Kurve.p
            y3 = (l*(Punkt1.x-l**2+Punkt1.x+Punkt2.x)-Punkt1.y)
                % Punkt2.Kurve.p
            Punkt3 = punkteAeK(x3,y3,Punkt1.Kurve)
    return Punkt3
```

2.1.4 Satz von Hasse

Interessant ist natürlich, wie viele Punkte in endlichen Körpern die Gleichung einer elliptischen Kurve E lösen. Die Anzahl der Punkte entspricht dann der Gruppenordnung von E. Dies kann grob mir dem Satz von Hasse abgeschätzt werden. Für eine genauere Bestimmung ist dieser Satz aber leider nicht geeignet. Sei E eine elliptische Kurve über einem endlichen Körper $\mathbb{F}_q$ dann gilt nach Hasse[4]:

$$q + 1 - \sqrt{2q} \leq \#E(\mathbb{F}_q) \leq q + 1 + \sqrt{2q}$$

Dies gibt schon eine grobe Größenordnung vor, in welcher die Ordnung der Kurve liegt. Für die exakte Berechnung ist zu Erwähnen, dass effiziente Algorithmen in Sage implementiert sind. Für die exakte Berechnnung der Ordnung könnte der Algorithmus von Schoof benutzt werden, dieser ist nachzulesen in [6] ab Seite 373.

2.1.5 Das Problem diskreter Logarithmus auf elliptischen Kurven

Im Allgemeinen sind diskrete Logarithmen, also Kongruenzen der Form $h \equiv g^x mod p$ schlecht bis nicht lösbar in endlichen Körpern $\mathbb{F}_p^*$. Um eine Analogie dieses Problems zu finden, betrachte die Formel als x Multiplikationen von g.

Definition: Sei E eine elliptische Kurve über einem endlichen Körper $\mathbb{F}_p$ und seien P und Q Punkte aus $E(\mathbb{F}_p)$. Das diskrete Logarithmus Problem auf elliptischen Kurven ist folglich das Problem, eine ganze Zahl n zu finden, sodass $Q = nP$. Analog zum diskreten Logarithmus Problem in $\mathbb{F}_p^*$, schreiben wir n als $n = log_p(Q)$.[5]

[4] [3][Seite 82]
[5] [5][Seite 310]

Anzumerken zu dieser Definition ist, dass sie nicht wirklich präzise ist. So kann es sein, dass Q gar kein Vielfaches von P ist. In diesem Fall ist $log_p(Q)$ nicht definiert. Allerdings wählt man sich in kryptographischen Verfahren einen Punkt P und n ist geheim, also kommt dieser Fall nicht vor. Ein weiteres Problem ist, dass es, sobald es ein n gibt, welches $Q = nP$ erfüllt, die Lösung nicht eindeutig ist. Sei o die Ordnung von P, dann gilt $nP = Q = (o \cdot m + n)P$ für alle $m \in \mathbb{Z}$. Also ist der Wert von $log_P(Q)$ ein Element von $\mathbb{Z}/o\mathbb{Z}$. Wenn man n als ein Element von $\mathbb{Z}/o\mathbb{Z}$ betrachtet, erfüllt der diskrete Logarithmus auf elliptischen Kurven $log_P(Q_1+Q_2) = log_P(Q_1)+log_P(Q_2)$ für alle $Q_1, Q_2 \in E(\mathbb{F}_P)$.

2.1.6 The double-and-add Algorithm

Die Grundidee dieser Methode ist, dass man n binär darstellt mit n_0,..,$n_r \in \{0, 1\}$. Es wird in 3.3 darauf eingegangen wie man dies genau bestimmt. Sei $Q_i = 2^i \cdot P$, für $i \in \{0,...,r\}$.

$$nP = n_0Q_0 + \cdots + n_rQ_r$$

Mit dieser Art nP zu berechnen, benötigt man also höchstens $2 \cdot log_2(n)$ Berechnungen, was es sehr schnell macht, nP für große n zu bestimmen. [6]
Eine Implementierung in Python3, mit dem vorherigen Code, sieht dann wie folgt aus:

```
def MultiplikationAeK(n, Punkt):
    if n == int(n) and n > 0:
        Punkt1 = Punkt
        Punkt2 = punkteAeK("O", 42, Punkt.Kurve)
        while n > 0:
            if n%2 == 1:
                Punkt2 = AdditionAeK(Punkt1, Punkt2)
            Punkt1 = AdditionAeK(Punkt1,Punkt1)
            n = int(n/2)
        return Punkt2
```

[6] [5] ab Seite 312

3 Montgomery Kurven

Eine Elliptische Kurve von Montgomery Form hat folgende darstellende Gleichung:

$$E_M : y^2 = x^3 + Ax^2 + x$$

Ein Vorteil dieser Form ist, dass die Verdopplung von Punkten auf Montgomery-Kurven nur von der x-Koordinate abhängig ist. Somit fällt die Fallunterscheidung, wie sie auf Elliptischen Kurven in Weierstraß Form vorhanden ist, weg. Und die Multiplikation beziehungsweise die Vervielfachung von Punkten wird deutlich schneller.[7]

3.1 Projektive Koordinaten

Die Transformation von euklidischen Koordinaten in projektive Koordinaten funktioniert folgendermaßen. Sei P ein Punkt in $\mathbb{R}^2$, also $P = (x, y)$ mit $x, y \in \mathbb{R}$.

$$x = \frac{X}{Z}, y = \frac{Y}{Z}$$

Die Darstellung von P in projektiven Koordinaten ist dann $P = (X : Y : Z)$. Da wir uns im folgenden über endlichen Körpern bewegen, ist $Z = 1$, außer $P = \mathcal{O}$, dann definieren wir $P = (0 : 1 : 0)$.[8]

3.2 Addition

Die Addition von 2 Punkten $P, Q \in E_M$, kann man analog zu der Addition in 2.0.4 definieren:

$$P = (x_P, y_P), Q = (x_Q, y_Q), P + Q = (x_3, y_3)$$

$$\lambda = \begin{cases} \frac{y_Q - y_P}{x_Q - x_P} & \text{wenn } P \neq Q \text{ und } P \neq -Q \\ \frac{3x_P^2 + Ax_P + 1}{2y_P} & \text{wenn } P = Q \end{cases}$$

$$x_3 = \lambda^2 - x_P - x_Q - A, y_3 = \lambda(x_P - x_3) - y_P$$

In projektiven Koordinaten ist die Abbildung **x:** $E_M \to E_M / < \ominus >= \mathbb{P}^1$ gegeben durch:

$$\textbf{x: } P \mapsto \begin{cases} (x_P : 1) & \text{,wenn } P = (x_P : y_P : 1) \\ (1 : 0) & \text{,wenn } P = \mathcal{O} = (0 : 1 : 0) \end{cases}$$

Die vereinfachte Formel $\mathbf{x}((X : Y : Z)) = (X : Z)$ funktioniert nur auf der offenen Teilmenge von E_M, welche $\mathcal{O}$ nicht enthält. Montgomery beobachtete, dass für $P \neq Q$ die X-Koordinaten von $P, Q, P+Q$, und $P-Q$ in folgender Beziehung zueinander stehen:

$$x_{P+Q} x_{P-Q} (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

[7] [4]Seite 285
[8] [2]Seite 3

Für den degenerierten Fall, dass $P = Q$, ergibt sich der Zusammenhang:

$$4x_{[2]P}x_P(x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2$$

Pseudo Addition: Wir möchten $\mathbf{x}(P+Q)$ berechnen, nur mit der Kenntnis von $\mathbf{x}(Q), \mathbf{x}(P)$ und $\mathbf{x}(P-Q)$. Nun nehmen wir an, dass $P \neq Q$ und $P - Q$ eine Ordnung größer als 2 hat. Dies wäre zum Beispiel, auch im Hinblick auf noch Folgendes, der Fall für $P \neq \mathcal{O}$ und $Q = 2 \cdot P$. Dann wäre die Differenz, als $Q - P = P$ und $\mathbf{x}(Q-P) = (X_P : Z_P)$, gegeben und die Pseudoaddition wäre anwendbar. Wir wechseln nun wie folgt in projektive Koordinaten:

$$\mathbf{x}(P) := (X_P : Z_P), \mathbf{x}(Q) := (X_Q : Z_Q),$$
$$\mathbf{x}(P+Q) := (X_+ : Z_+), \mathbf{x}(P-Q) := (X_- : Z_-).$$

Mit der Bedingung, dass die Ordnung von $P - Q$ größer als 2 ist, folgt $X_- \neq 0$ und $Z_- \neq 0$. X_+ und Z_+ wird wie folgt berechnet:

$$X_+ = Z_- \left[(X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q)\right]^2, \tag{1}$$

$$Z_+ = X_- \left[(X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q)\right]^2. \tag{2}$$

Die Implementierung in Python:
Hierbei wurden die Formeln (1) und (2) benutzt und am Ende der Punkt so normiert, dass $Z = 1$ gilt. Natürlich muss man auch die Fälle berücksichtigen, bei denen die Addition zweier Punkte das neutrale Element ergibt. Dies wurde hier im Vergleich zum Pseudocode (Vgl. [2][Seite 9]) noch ergänzt.

```
def xAdd(p,q, diff):
    if p.proZ == 0 or q.proZ == 0:
        return diff
    else:
        summand1=(p.x-p.proZ)*(q.x+q.proZ)
        summand2=(p.x+p.proZ)*(q.x-q.proZ)
        xneu = diff.proZ*(summand1+summand2)**2
        zneu = diff.x*(summand1-summand2)**2
        if zneu % p.Kurve.p != 0:
            xneu = xneu * pow(zneu,-1,p.Kurve.p) % p.Kurve.p
            zneu = 1
            neuerPunkt = punkteMK(xneu, "tba", p.Kurve, zneu)
        else:
            neuerPunkt = punkteMK(0,"tba",p.Kurve,0)
    return neuerPunkt
```

Pseudo Verdopplung: Sei nun also $P = Q$. Das Ziel der Berechnung ist $\mathbf{x}(2P)$. Nun erhält man mit folgendem Satz noch eine schnelle Möglichkeit der Verdoppelung.

Satz:[9]
Sei p eine Primzahl und $p \geq 5$. Sei nun A eine ganze Zahl, sodass $A^2 - 4$ kein Quadrat modulo p ist. Sei nun E eine Montgomery-Kurve über $\mathbb{F}_p$. Sei die Abbildung **x** wie oben. Sei des Weiteren $x, z \in \mathbb{F}_{p^2}$ so gewählt, dass $(x, z) \neq (0, 0)$ gilt. Dann folgt:

$$X_{2P} = (x - z)^2(x + z)^2$$
$$Z_{2P} = 4xz(x^2 + Axz + z^2)$$

Dann ergibt sich $\mathbf{x}(2P) = \frac{x_2}{z_2}$ für alle $P \in E(\mathbb{F}_p)$.
Hier kann im Vergleich zu dem Additionsalgorithmus in 3.2 Laufzeit eingespart werden. Im Nachfolgenden findet sich eine Implementierung in Python (Vgl. [2][Seite 9]). Auch hier ist der Fall wieder zu berücksichtigen, dass die Verdoppelung eines Punktes das neutrale Element ergibt, dies wurde hier ebenfalls im Vergleich zur Quelle ergänzt.

```
def xDbl(Punkt):
    xneu = (Punkt.x + Punkt.proZ)**2 * (Punkt.x - Punkt.proZ)**2
    Faktor1 = 4*Punkt.x*Punkt.proZ
    Faktor2 = (Punkt.x - Punkt.proZ)**2 +
        (Punkt.Kurve.A+2)*pow(4,-1,Punkt.Kurve.p)*Faktor1
    zneu = Faktor1*Faktor2
    if zneu != 0:
        xneu = xneu*pow(zneu,-1,Punkt.Kurve.p) % Punkt.Kurve.p
        zneu = 1
        neuerPunkt = punkteMK(xneu, "tba", Punkt.Kurve, zneu)
    else:
        neuerPunkt = punkteMK(0,1,Punkt.Kurve,0)
    return neuerPunkt
```

Mit den Pseudo-Operationen gibt es nun eine schnelle Möglichkeit $\mathbf{x}([k]P)$ für ein $k \in \mathbb{N}$ und einen Punkt P auf einer Montgomery Kurve zu berechnen. Diese trägt den Namen Montgomery-Ladder und basiert auf dem double-and-add Algorithmus. Essenziell hierfür ist die Binärdarstellung für n. Zur Berechnung der y-Koordinate findet sich eine Formel in [4] auf Seite 286. Diese wird nachfolgend kurz vorgestellt:

$$y_k = \frac{(x_1x_k + 1)(x_1 + x_k + 2A) - 2A - (x_1 - x_k)^2x_{k+1}}{2y_1}.$$

Hierbei ist y_k schon normiert, also entspricht y_k schon der kartesischen Koordinate.

3.3 Montgomery Ladder und der Binäralgorithmus

Nun kombiniert man die Pseudoaddition und die Pseudomultiplikation und erhält die sogenannte Montgomery Ladder, die in Kombination mit double-and-add eine schnelle Berechnung von nP ermöglicht. Der implementierte Algorithmus ist an den Pseudocde aus

[9][1][B.1]

[2][Seite 11] angelehnt. Um die Binärdarstellung einer Zahl zu bekommen, berechnet man zuerst die 2-adische Darstellung und berechnet daraus die Binärdarstellung. Im Folgenden werden naive Implementierungen davon vorgestellt.

2-adische Darstellung: Man möchte nun $n \in \mathbb{N}$ als Potenzen von 2 ausdrücken. Das sieht dann folgendermaßen aus:

$$n = \sum_{i=0}^{k} a_i 2^i$$

Dabei ist $a_i \in \{0, 1\}$ und $k \in \mathbb{N}$. Ein naiver Algorithmus zum Berechnen der 2-adischen Darstellung für ein $n \in \mathbb{N}$ wäre:

```
from math import log2
def zweiadischezerlegung(n):
   listeVonAgleich1 = list()
   x=n
   while x >= 2:
#log2() ist eine Funktion im math
#Paket die den Logarithmus zur Basis 2 berechnet.
      exponent=int(log2(x))
      listeVonAgleich1.append(exponent)
      x = x - pow(2,exponent)
   if not x == 0:
      listeVonAgleich1.append(0)
#noch in die aufsteigende Reihenfolge bringen
   listeVonAgleich1.reverse()
   return listeVonAgleich1
      listeVonAgleich1.append(exponent)
      x = x - pow(2,exponent)
   if not x == 0:
      listeVonAgleich1.append(0)
#noch in die aufsteigende Reihenfolge bringen
   listeVonAgleich1.reverse()
   return listeVonAgleich1
```

Beispiel: $25 = 1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4$. In Python3, wenn das Programm als zweiAdDar.py gespeichert wurde:

```
>>> import zweiAdDar as bina
>>> bina.zweiadischezerlegung(25)
[0, 3, 4]
```

Binärdarstellung: Die Binärdarstellung einer Zahl $n \in \mathbb{N}$ ist einfach die umgekehrte Andeinanderreihung der a_i's aus der 2-adischen Zerlegung. Also im Fall $n = 25$ wäre die Binärdarstellung 11001. Und allgemein entspricht $a_k a_{k-1} \ldots a_1 a_0$ der Binärdarstellung für ein $n \in \mathbb{N}$. Eine mögliche Implementierung, wobei man die Binärdarstellung als String erhält, sähe wie folgt aus:

```
def binary(n):
```

```
    binaer = ''
    zweiad=zweiadischezerlegung(n)
    zweiad.reverse()
    bina = []
    for i in range(int(zweiad[0])+1):
        bina.append(0)
    for j in range(len(zweiad)):
        einsStelle=len(bina)-int(zweiad[j])-1
        bina.pop(einsStelle)
        bina.insert(einsStelle,1)
    for i in range(len(bina)):
        binaer = binaer + str(bina[i])
    return binaer
```

Beispiel: Hier wurde die binary Funktion der Datei zweiAdDar.py hinzugefügt. So sieht die Anwendung dann aus:

```
>>> import zweiAdDar as bina
>>> bina.binary(25)
'11001'
```

Montgomery Ladder: Die Montgomery-Ladder vereint nun die beiden Pseudooperationen und führt so zu einer effizienten Berechnung von nP, für $n \in \mathbb{N}$, und P, einem Punkt auf der entsprechenden Montgomery-Kurve. Im Folgenden ist eine Implementierung des Pseudocodes aus [2][Seite 11].

```
    def ladder(n,P):
    bina = binary(n)
    R0 = P
    R1 = xDbl(P)
    for i in range(len(bina)-1):
        if bina[i+1] == '0':
            R1 = xAdd(R0,R1,P)
            R0 = xDbl(R0)
        else:
            R0 = xAdd(R0,R1,P)
            R1 = xDbl(R1)
    return R0
```

4 Anwendungen

Dieses Kapitel beschäftigt sich mit den möglichen Anwendungen von elliptischen Kurven in der Kryptographie. Hierfür wird auf einige übliche Verfahren, wie zum Beispiel der Diffie-Hellman-Schlüsselaustausch, eingegangen, Curve25519 betrachtet sowie die Verfahren mit dem bisherigen Code implementiert.

4.1 Diffie-Hellman Problem

Diffie-Hellman befasst sich mit dem Problem, dass sich zwei Parteien auf einen gemeinsamen Schlüssel einigen wollen und die einzige Möglichkeit der Kommunikation unsicher ist. Die klassische Lösung dieses Problems ist die Ausnutzung der Tatsache, dass diskrete Logarithmen über endlichen Körpern im Allgemeinen nicht schnell lösbar sind.

4.2 Diffie-Hellman über endlichen Körpern

Der Ablauf des Schlüsselaustausches kann in drei Phasen eingeteilt werden. Zuerst veröffentlicht eine Partei eine große Primzahl p und eine Zahl $n \in \mathbb{F}_p$. n soll dabei eine hohe Ordnung in der multiplikativen Gruppe über $\mathbb{F}_p$ haben. Im nächsten Schritt wählen sich die beiden Parteien (Bob und Alice) einen privaten Schlüssel $k_i \in \mathbb{F}^*$ mit $i \in \{1, 2\}$, wobei k_1 der private Schlüssel von Bob und k_2 der private Schlüssel von Alice ist. Der öffentliche Schlüssel ist dann jeweils $k_{pi} = n^{k_i}$ mod p. Der gemeinsame Schlüssel k_p kann dann berechnet werden: Von Bob mit $k_p = k_{p2}^{k_1}$ mod p und analog von Alice $k_p = k_{p1}^{k_2}$ mod p. Also ist $k_p = n^{k_1 k_2}$ mod p.[10]

Beispiel:

Zur Nachvollziehbarkeit wurden hier kleinere Zahlen gewählt, als sie üblicherweise Anwendung finden. Die multiplikative Ordnung von 45 ist 50, also in diesem Beispiel die zweithöchste Möglichkeit, da die Elementordnung die Gruppenordnung teilt. Die Ordnung der multiplikativen Gruppe ist hier 100.

p = 101, n = 45	Alice	Bob
k_i	35	27
k_{pi}	6	96
k_p	65	65

[10] [5][ab Seite 67]

4.3 Diffie-Hellman mit elliptischen Kurven

Auf elliptischen Kurven übernimmt die Multiplikation die Exponentiation. Die Sicherheit hierbei beruht auf dem diskreten Logarithmus Problems auf elliptischen Kurven. Die Multiplikation ist hier folgendermaßen definiert:
Sei $n \in \mathbb{F}_p$ und E eine elliptische Kurve über $\mathbb{F}_p$, wobei p eine Primzahl ist. Sei nun P ein Punkt von E, dann ist $nP = \underbrace{P + \cdots + P}_{\text{n-mal}}$. Diffie-Hellman lässt sich nun analog Übertragen. Eine Partei, die nicht unbedingt Bob oder Alice ist, legt eine Kurve E, eine Primzahl p und ein Punkt Q auf E, mit hoher Ordnung, fest und veröffentlicht diese. Die Kurve wird über dem Primkörper von p betrachtet. Bob und Alice wählen sich nun jeweils $k_i \in \mathbb{F}$ mit $i \in \{1, 2\}$, wobei k_1 der private Schlüssel von Bob ist und k_2 der von Alice. Die öffentlichen Schlüssel werden nun wie folgt berechnet: $Q_{pi} = k_i Q$. Der gemeinsame Schlüssel ergibt sich dann mit folgender Gleichung $k_p = k_2 Q_{p1} = k_1 Q_{p2} = k_1 k_2 Q$.

Beispiel:

Die neutrale Partei veröffentlicht $p = 101$, $E : y^2 = x^3 + 25x + 17$ und den Punkt $P = (1, 12)$.

	Alice	Bob
k_i	9	7
Q_{pi}	(89,98)	(49,69)
k_p	(91,22)	(91,22)

4.4 ECDSA

In diesem Abschnitt kommt noch der Elliptic-Curve-Digital-Signature-Algorithm (ECDSA) zur Vorstellung. Dieser dient dazu, zu verifizieren ob eine Nachricht wirklich im Original vorliegt. Siganturen generell funktionieren nach dem Prinzip, dass eine Partei eine Signatur veröffentlicht. Mittels dieser Signatur und eines Hashwertes des signierten Dokuments, den sich jede Partei selber berechnet, kann sich die zweite Partei durch Nachrechnen davon überzeugen, dass das signierte Dokument seit der Signatur nicht verändert wurde. In nachfolgender Darstellung ist H eine Hash-Funktion und M das Dokument, welches signiert werden soll[11]:

[11] [5][Seite 322]

Parameter welche Öffentlich bekannt sind:	
Endlicher Körper $\mathbb{F}_p$, Elliptische Kurve $E(\mathbb{F}_p)$ Punkt $Q \in E(\mathbb{F}_p)$ mit hoher Primordnung q	
Alice	Bob
Schlüsselerzeugung	
Sucht sich einen zufälligen Schlüssel: $1 < z < q - 1$ Berechnet $V = z \cdot Q \in E(\mathbb{F}_p)$ Veröffentlicht V	
Signatur	
Berechne Hashwert $H(M)$ mod q Wähle ein Zufälliges e mod q Berechne $e \cdot Q$ $s_1 = x(e \cdot Q)$ mod q $s_2 \equiv (H(M) + z \cdot s_1)e^{-1}$ mod q Signatur $s = (s_1, s_2)$	
Verifikation	
	Berechne $v_1 = H(M) \cdot s_2^{-1}$ mod q $v_2 = s_1 \cdot s_2^{-1}$ mod q Berechne $N = v_1 \cdot Q + v_2 \cdot V \in E(\mathbb{F}_p)$ Verifiziere das gilt $x(N) = s_1$

5 Curve25519

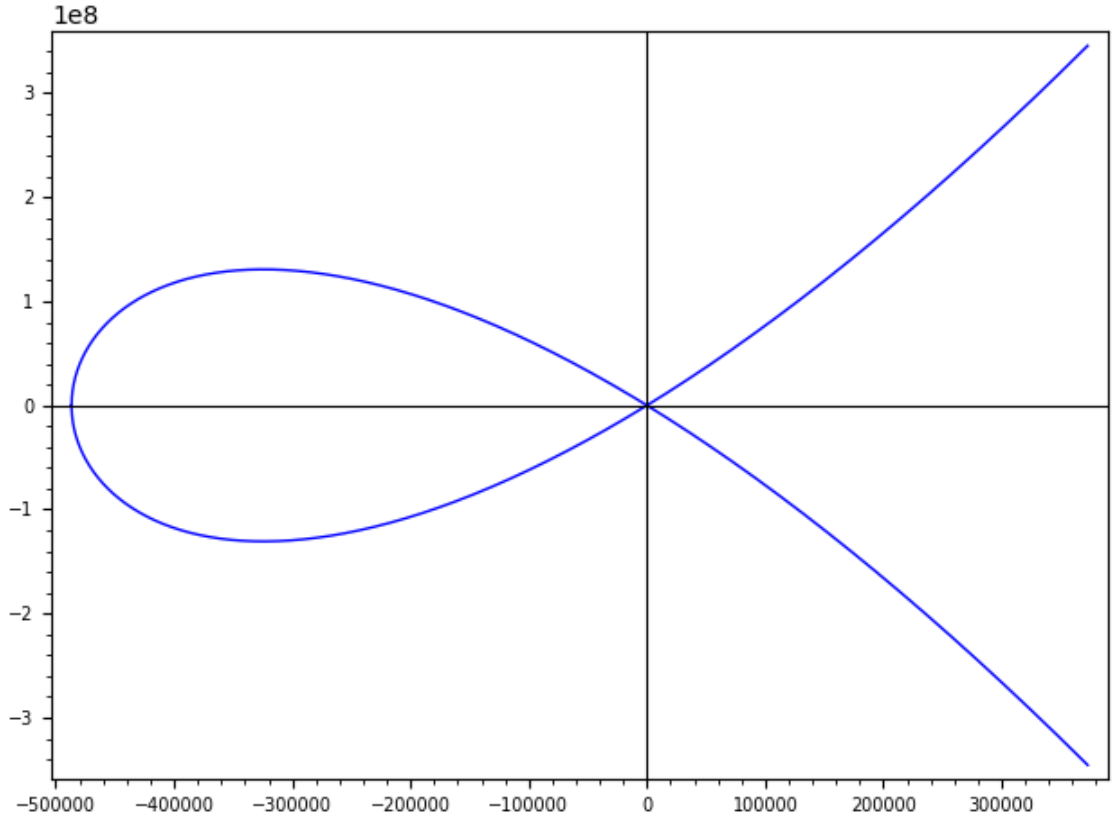

Abbildung 2 [12]. Curve25519 über den rationalen Zahlen geplottet.

Für die Anwendung bei kryptographischen Methoden hat Bernstein Curve25519 vorgeschlagen, da diese einige besondere Eigenschaften besitzt. Insbesondere ist diese Kurve von Montgomery-Form, deshalb ist mit den Pseudooperationen und der Montgomery-Ladder eine schnelle Berechnung möglich. Die Kurve, die Bernstein explizit vorgeschlagen hat, wird von der Gleichung $y^2 = x^3 + 486662x^2 + x$ beschrieben. Nun hat Bernstein noch vorgeschlagen, für Diffie-Hellman ausschließlich die x-Koordinate zu verwenden, was zusätzlich Berechnungen einspart. Im Folgenden wurde dieser Vorschlag umgesetzt und in der Implementierung auch nur mit der x-Koordinate gearbeitet.

5.1 Eigenschaften von Curve25519

Side Channel Attacks

Side Channel Attacks sind im Prinzip Angriffe, bei denen man über die Art und Weise wie Algorithmen implementiert sind, Rückschlüsse auf die Berechnungen ziehen kann. So ist es bei elliptischen Kurven von Weierstraß Form zum Beispiel möglich, durch Messung

[12] Code siehe 6.1

von Laufzeiten, die Berechnung von nP nachzuvollziehen, ohne sie aktiv zu betrachten. Dies ist möglich, da in der Addition eine Fallunterscheidung vorkommt und je nachdem welcher Fall eintritt, ist die Laufzeit des Additionsalgorithmus anders.

Sicherer Twist:

Zu Montgomery-Kurven gibt es einen Twist, der eine Montgomery-Kurve auf eine Edwards-Kurve abbildet. So ist es möglich, ein Problem auf dem Twist zu lösen und auf die eigentliche Kurve zurückzuführen. Da der Twist und die Kurve eine Ordnung haben, die von der Form $4\cdot$ Primzahl oder $8\cdot$ Primzahl, ist verhindert dies das vereinfachte Lösen des diskreten Logarithmus Problems. Im Normalfall werden dann Keys, welche leicht auf dem Twist zu anzugreifen sind, ausgeschlossen. Dies spart man sich auf dieser Kurve, da alle Keys sicher sind. [13]

5.2 Diffie-Hellman

Als Startpunkt schlägt Bernstein vor, den Punkt mit der x-Koordinate 9 zu nehmen. Als privaten Schlüssel wird empfohlen, einen 32-Byte String zu benutzen. Ein Byte besteht aus 8 Bits, welche entweder den Wert 0 oder 1 haben. Der String sollte zufällig gewählt sein, zudem schlägt er noch vor, einzelne Bits der 32-Bytes zu verändern. Die 32-Byte Schlüssel kommen daher, dass die Ordnung der Kurve im Bereich von $2^{255} = 2^{8\cdot 32-1}$ liegt. Daran sieht man auch, dass für diese Kurve höhere Schlüsselgrößen keinen erhöhten Sicherheitsfaktor bieten. Diese Bytes werden dann als 2er Potenzen betrachtet, also das erste Bit ist dann 2^0, entspricht damit also der Eins. Das letzte Bit entspricht dann $2^{8\cdot 32} = 2^{256}$, es sollte aber nach Bernstein immer 0 sein. Im Nachfolgenden wird eine Implementierung vorgestellt, in welcher zuerst 32-Byte Keys erzeugt werden. Der zugrunde liegende Zeichensatz, der Anwendung findet, ist ascii. Dieser benutzt pro Zeichen allerdings nur 7 der 8 Bit, deshalb wird an der achten Stelle immer eine Null eingefügt. Dieses Verfahren ist nicht das Optimum, da man nun die Auswahl der Keys nicht erhöht. Mit dieser Methode erhält man nicht die bestmögliche Sicherheit, sodass die Auswahl der Keys auf $2^{(256-32)} = 2^{224}$ beschränkt wird, statt auf 2^{255}, wenn man wie Bernstein vorgeschlagen hat, nur das letzte Bit auf 0 setzt. Eine sicherere Methode wäre es, den UTF-8 Zeichensatz zu verwenden, da dieser alle 8 Bit benutzt. Zusätzlich stellt Python keine direkte Methode zur Verfügung, um aus Strings, also Zeichen, die Bit Zerlegung zu bekommen. Darum wird im Code der String erst hexadezimal dargestellt, dann in binär umgewandelt. Die Binärdarstellung besitzt aber wie erwähnt nur 7 Stellen, also füge ich noch eine Null ein. Die Binärdarstellungen werden schließlich aneinandergefügt, damit man den Faktor errechnen kann, mit dem man den Punkt multipliziert. Die Funktion ladderb(binear, Punkt). Ist im Prinzip die gleiche Funktion, wie die Funktion ladder(n,Punkt) in 3.3, allerdings nimmt sie als Eingabe direkt die Binärzahl, womit wiederum Berechnungen gespart werden können. Der Code zu ladderb(binear, Punkt) ist in 6.2 zu finden.

[13] [1][Section 3]

```
import random
import string
import elliptischeKurven as EK

Kurve = EK.montgomery(486662,2**255-19)
Punkt = EK.punkteMK(9,"NN",Kurve)
def erzeugePrivKey():
   Buchstaben = string.ascii_letters
   privateKey = ''.join(random.choice(Buchstaben) for i in
      range(32))
   return privateKey
#ascii benutzt pro zeichen ein Byte und das
#8. bit ist immer leer
#fuer noch mehr Auswahl der Keys wuerde man UTF-8 benutzen

def erzeugePubKey(privateKey):
   binaer = stringToBin(privateKey)
   pubKey = EK.ladderb(binaer, Punkt)
   return pubKey.x

def stringToBin(Key):
#String to bit to zahl
   binaer = ''
#in Python gibt es leider keinen direkten Weg
#an die bits zu kommen, aber es geht
#ueber die Heximaldarstellung des ascii Zeichens
   for i in range(len(Key)):
      hexZahl = Key[i].encode("ascii").hex()
      ganzZahl = int(hexZahl,16)
      binaere = EK.binary(ganzZahl)
      binaer = binaer + binaere + "0"
#den faktor fuer den pub Key berechnen
   return binaer
```

Das string Paket wurde importiert, um die ascii Zeichen in das Programm zu laden. Das random Paket ist dann dafür da, zufällige Zeichen aus dem Zeichensatz zu wählen. So entsteht ein zufällig gewählter 32-Byte großer privater Schlüssel.

Beispiel:

Anwenden würde man das Programm dann folgendermaßen, falls es als stringgen.py gespeichert ist:

```
>>> import stringgen as s
>>> k = s.erzeugePrivKey()
>>> k
'dzmooPrNJatqARGVIoqlyBfXatWlSNPS'
>>> publickey = s.erzeugePubKey(k)
>>> publickey
```

```
5110793069909264945357952032929815783205408 1
3459405572056256216609760502645 30
```

Für den gemeinsamen Schlüssel wird zunächst der Faktor des eigenen Privaten Schlüssels berechnet und im Anschluss multipliziert mit dem öffentlichem Schlüssel. Der Code hierfür gestaltet sich mit der Vorarbeit von oben relativ trivial:

```
def berechneGemeinsamenKey(privKey,pubKey):
    key = stringToBin(privKey)
    punkt = EK.punkteMK(pubKey,"NN",Kurve)
    gemeinsam = EK.ladderb(key,punkt)
    return gemeinsam.x
```

Beispiel: Falls man den öffentlichen Schlüssel aus obigem Beispiel besitzt, kann man so den gemeinsamen Schlüssel berechnen, der zweite private Schlüssel ist hier mit k2 benannt. Im Beispiel wurde von beiden Seiten der öffentliche Schlüssel berechnet:

```
>>> s.berechneGemeinsamenKey(k2,publickey)
42298204974875853776902801843852030 26
61923644911312264736933137302495483529 13
>>> s.berechneGemeinsamenKey(k,s.erzeugePubKey(k2))
42298204974875853776902801843852030 26
61923644911312264736933137302495483529 13
```

5.3 Signatur

Eine ziemlich verbreitete Signatur Methode ist die in 4.4 erwähnte ECDSA-Methode. Bei der folgenden Implementierung ist zu beachten, dass nicht die xAdd oder xDbl Funktionen verwendet wurden, da man beim Verifizieren die Differenz der beiden Punkte nicht kennt. Stattdessen wurde die in 3.2 vorgestellte Addition benutzt. Eine Implementierung findet sich in 6.2. q ist hierbei die Ordnung des Punktes P und dieser ist wiederum der Punkt den Bernstein vorgeschlagen hat, allerdings diesmal mit y-Koordinate, da diese für die normale Addition benötigt wird. Dieser Punkt hat eine Primzahlordnung, die sehr hoch ist. Diese Tatsache kann mit Sagemath verifiziert werden. Für die Signatur wird hier die Hash-Funktion sha256 benutzt. Um Zufallszahlen zu generieren bis $q - 1$ kann das random Modul nicht benutzt werden, da q zu groß ist. Demzufolge wurde die Binärzerlegung von q betrachtet, welche 253 Stellen hat. Zur Bestimmung der Zufallszahl in einer Größenordnung wie gefordert, wurde der Ansatz gewählt, 253 mal zufällig 0 oder 1 in einen String zu schreiben und daraus die zugehörige Zahl mod q zu bestimmen. Dabei war die Rechnung Modulo q wichtig, da die Binärdarstellung nicht aus 253 mal der Eins besteht, und somit zufällig höhere Zahlen gewählt werden können.

```
import elliptischeKurven as EK
import random
from hashlib import sha256
```

```
Kurve = EK.montgomery(486662,2**255-19)
P = EK.punkteMK(9, 14781619447589544791020593568409986887264606134616475288964881837755586237401,Kurve)
q = 7237005577332262213973186563042994240857116359379907606001950938285454250989

def sign(M):
    z = 0
    while z < 1:
        z = zufall() %q
    V = z*P
    M = M.encode("ascii")
    hashInHex = sha256(M).hexdigest()
    zahl = int(hashInHex,16) % q
    e = 0
    while e < 1:
        e = zufall() %q
    s1 = (e*P).x
    s2 = (zahl+z*s1)*pow(e,-1,q)
    s = [s1,s2]
    return s, V
def verify(M,s,V):
    M = M.encode("ascii")
    hashInHex = sha256(M).hexdigest()
    zahl = int(hashInHex,16)%q
    v1 = zahl*pow(s[1],-1,q)%q
    v2 = s[0]*pow(s[1],-1,q)%q
    N = v1*P+v2*V
    if N.x == s[0]:
        return True
    else:
        return False
def zufall():
    binaer = ''
    for i in range(253):
        binaer = binaer + str(random.choice([0,1]))
    zufall = int(binaer,2)
    return zufall
```

Beispiel:
Den Code würde man dann auf einen String folgendermaßen anwenden, wenn das Programm als ecdsa.py gespeichert ist:

```
>>> import ecdsa as e
>>> k = e.sign('Hallo')
>>> e.verify('Hallo',k[0],k[1])
True
```

6 Anhänge

6.1 Sage-Code

Für sämtlichen Sage-Code wurde SageMath version 9.5 verwendet.

Abbildung 1

Der Code für Abb. 1 sieht folgendermaßen aus:

```
D = EllipticCurve([-20,0])
p = plot(D,(-7,21))
p += line([(-4,-4),(81/4,-711/8)], color="red")
p += point(((-4,-4),(81/4,711/8),(81/4,-711/8)),
color = "green", pointsize=20)
p += text("P", (-4,-12.5), color = "black")
p += text("P+P", (85/4,700/8), color = "black")
p += text(" -(P+P)", (85/4,-650/8), color = "black")
p += text("Tangente an P", (5, -22), color = "red")
p += line([(81/4,-711/8),(81/4,711/8)],
color = "purple", linestyle = ":")
p += text("y²=x³-20x",(10,50), color = "blue")
```

Abbbildung 2

```
E = EllipticCurve(QQ,[0,486662,0,1,0])
a = plot(E)
a.show(fontsize=7,frame=True)
```

6.2 Python-Code

ladderb(n,P)

```
def ladderb(n,P):
R0 = P
R1 = xDbl(P)
for i in range(len(n)-1):
   if n[i+1] == 0:
      R1 = xAdd(R0,R1,P)
```

```
            R0 = xDbl(R0)
        else:
            R0 = xAdd(R0,R1,P)
            R1 = xDbl(R1)
    return R0
```

Addition und Multiplikation von Montgomery-Kurven

```
def addMon(p,q):
    if p.Kurve != q.Kurve:
        return "Addition nicht moeglich, Kurven sind nicht gleich"
    if p.proZ == 0:
        return q
    elif q.proZ == 0:
        return p
    else:
        if p.y != q.y and p.x != q.x and p.y != p.Kurve.p -q.y:
            l = (q.y - p.y) * pow(int(q.x - p.x),-1,int(p.Kurve.p))
            xneu = (l**2 - p.x -q.x - p.Kurve.A) % p.Kurve.p
            yneu = (l*(p.x - xneu)-p.y) % p.Kurve.p
            neuerPunkt = punkteMK(xneu,yneu,p.Kurve)
        elif p.x == q.x and p.y == q.Kurve.p - q.y:
            neuerPunkt = punkteMK(0, 1, p.Kurve,0 )
        else:
            l = (3*p.x**2+2*p.Kurve.A*p.x +1) *
                pow(int(2*p.y),-1,int(p.Kurve.p))
            xneu = (l**2 - p.x -q.x - p.Kurve.A) % p.Kurve.p
            yneu = (l*(p.x - xneu)-p.y) % p.Kurve.p
            neuerPunkt = punkteMK(xneu,yneu,p.Kurve)
        return neuerPunkt

def MultiplikationMK(n, Punkt):
    if n == int(n) and n > 0:
        Punkt1 = Punkt
        Punkt2 = punkteMK(0, 1, Punkt1.Kurve,0)
        while n > 0:
            if n%2 == 1:
                Punkt2 = addMon(Punkt1, Punkt2)
            Punkt1 = addMon(Punkt1,Punkt1)
            n = n//2
        return Punkt2
```

Literatur

[1] BERNSTEIN, Daniel J.: Curve25519: new Diffie-Hellman speed records. (2006). https://cr.yp.to/ecdh/curve25519-20060209.pdf

[2] CRAIG COSTELLOM, Benjamin S.: Montgomery Curves and their arithmetic, The case of large characteristic fields. https://eprint.iacr.org/2017/212.pdf

[3] HANKERSON, Scott; Menezes A. Darrel; Vanstone V. Darrel; Vanstone: *Guide to Elliptic Curve Cryptography*. 2. Springer Science+Business Media New York, 2004

[4] HENRY COHEN, Gerhard F.: *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. 1. Chapman Hall/CRC, 2006

[5] JEFFREY HOFFSTEIN, Jill P. ; SILVERMAN, Joseph H.: *An Introduction to Mathematical Cryptography*. 2. Springer Science+Business Media New York, 2014

[6] SILVERMAN, Joseph H.: *The Arithmetic of Elliptic Curves*. 2. Springer Verlag, 2009

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Erlangen, den 22. August 2023

_____

Unterschrift