Fortgeschrittene Techniken der Kryptographie

Ein Begleitheft zur Vorlesung und zu den Übungen

von

Jürgen Fuß

Bachelorstudiengang Sichere Informationssysteme
Fakultät für Informatik, Kommunikation und Medien,
FH OÖ, Campus Hagenberg

Dritte, überarbeitete Auflage, 2023

Inhaltsverzeichnis

1 RSA **1**

1.1 Mathematische Werkzeuge . . . 1

1.1.1 Euklids Algorithmus . . . 2

1.1.2 Die Eulersche φ-Funktion . . . 2

1.1.3 Der chinesische Restsatz . . . 4

1.1.4 Die Sätze von Fermat und Euler . . . 7

1.2 Performance des RSA-Verfahrens . . . 9

1.2.1 Square and Multiply . . . 10

1.2.2 RSA-CRS . . . 13

1.3 Angriffe auf RSA bei falscher Wahl der Schlüssel . . . 15

1.3.1 Wieners Attacke über Kettenbrüche . . . 15

1.3.2 Fermat-Faktorisierung . . . 22

2 Auf diskreten Logarithmen basierende Verfahren **25**

2.1 Diffie-Hellman-Key-Exchange . . . 25

2.2 Gruppen . . . 27

2.3 Diffie-Hellman mit Gruppen . . . 29

2.4 Mehr zur Ordnung . . . 31

2.5 Verfahren zur Berechnung diskreter Logarithmen . . . 34

2.5.1 Baby Step Giant Step . . . 34

2.5.2 Pohlig-Hellman . . . 36

2.5.3 Der Index-Calculus-Algorithmus . . . 38

2.6 DSA-Signaturen . . . 41

2.6.1 Performance . . . 42

2.6.2 Attacken auf DSA-Signaturen . . . 43

Nonce Reuse . . . 43

2.7 Schnorr-Signaturen . . . 44

3 Elliptische Kurven **47**

3.1 Elliptische Kurven über $\mathbb{R}$ . . . 48

3.2 Elliptische Kurven modulo p . . . 53

3.2.1 Der Satz von Hasse . . . 54
3.3 Kryptografische Verfahren mit elliptischen Kurven . . . 57
3.3.1 ECDH . . . 57
3.3.2 ECDSA . . . 57
Ed25519 . . . 58
3.4 Vergleich von Schlüssellängen . . . 60
3.5 Effizienz elliptischer Kurven . . . 60
3.5.1 Performancesteigerung . . . 61
Punktkompression . . . 61
Signed Double-&-Add . . . 62
Projektive Koordinaten . . . 62
Montgomery Curves . . . 63

4 Hashfunktionen **67**
4.1 Merkle Trees . . . 67
4.1.1 Erstellen eines Merkle Tree . . . 68
4.1.2 Verfizieren eines Datensatzes . . . 68
4.2 One-Time Signatures . . . 70
4.2.1 Lamport-Signaturen . . . 70
Schlüsselerzeugung . . . 70
Signieren . . . 70
Verifizieren . . . 70
4.2.2 Winternitz One-Time Signatures (WOTS) . . . 71
Schlüsselerzeugung . . . 71
Signieren . . . 71
Verfizieren . . . 72
4.3 Stateful Signatures . . . 73
4.3.1 Merkle Signatures . . . 73
Schlüsselerzeugung . . . 73
Signieren . . . 73
Verifizieren . . . 74
4.3.2 Weitere Verbesserungen . . . 74
XMSS . . . 74
XMSS^{MT} . . . 74
HORS und HORST . . . 75
SPHINCS+ . . . 75

5 Kryptographische Verfahren auf Basis von Polynomen **77**
5.1 Polynome . . . 77
5.2 Endliche Körper . . . 78
5.3 AES . . . 81
5.3.1 MixColumns . . . 82
5.3.2 SubBytes . . . 83
5.4 Der Galois Counter Mode (GCM) . . . 85
5.5 Post-Quantum-Verfahren . . . 86
5.5.1 Allgemeines . . . 86
5.5.2 Das Problem . . . 86
5.5.3 Digitale Signatur: CRYSTALS-Dilithium . . . 87
5.5.4 KEM: CRYSTALS-Kyber . . . 89

A Details **91**
A.1 Chinesischer Restatz . . . 91
A.2 Satz von Wiener . . . 92
A.3 Kleinste gemeinsame Vielfache und größte gemeinsame Teiler . . . 92
A.4 Punktaddition auf elliptischen Kurven . . . 93
A.5 Addition in projektiven Koordinaten . . . 95
A.6 Die Montgomery Ladder . . . 96
A.7 Quadratwurzeln in endlichen Körpern . . . 97

B Faktorisieren **99**
B.1 Die Pollard-Methode . . . 99
B.2 Das quadratische Sieb . . . 100
B.3 Faktorisieren mit elliptischen Kurven . . . 102

Vorwort

Dies ist ein Skriptum zur Vorlesung „Fortgeschrittene Techniken der Kryptographie“ am Bachelorstudiengang „Sichere Informationssysteme“ der FH OÖ an der Fakultät für Informatik, Kommunikation und Medien in Hagenberg.

Aufbauend auf die Lehrveranstaltung „Grundlagen der Kryptographie“ im zweiten Semester werden in dieser Lehrveranstaltung vor allem Public-Key-Verfahren genauer unter die Lupe genommen. Der Fokus liegt dabei auf den Themen Sicherheit und Performance. In Bezug auf die Sicherheit ist bei Public-Key-Verfahren dabei besonders die Schlüsselerzeugung bzw. die Wahl der Domain Parameter von Interesse. Performance ist für Public-Key-Verfahren ein zweites wichtiges Thema, da diese Verfahren im Vergleich zu symmetrischen Verfahren signifikant langsamer sind und damit zu einem Flaschenhals werden können. Aus diesem Grund werden Public-Key-Verfahren oft mit einer Reihe von Optimierungen implementiert, die hier auch wenigstens im Überblick betrachtet werden. Neben heutzutage gängigen werden auch Verfahren vorgestellt, die zukünftig interessant sein könnten. Insbesondere Verfahren aus dem Bereich der Post-Quantum-Kryptographie werden hier betrachtet.

Gerade im Bereich der Public-Key-Kryptographie bauen die Verfahren stark auf mathematische Grundlagen, die im Rahmen dieser Lehrveranstaltung bei Bedarf mitbehandelt werden. Eine genaue Trennung zwischen Kryptographie und Mathematik ist hier schwierig und wird auch gar nicht versucht.

Die Vorlesung folgt in weiten Teilen dem Buch „Introduction to Modern Cryptography“ von Katz und Lindell [KL20] und „A Course in Number Theory and Cryptography“ von Koblitz [Kob94]. Genauere Beschreibungen vieler Algorithmen finden sich in „Serious Cryptography“ von Aumasson [Aum18], das auch weitere praktische Aspekte beleuchtet. Detailliertere mathematische Grundlagen sind bspw. in „Einführung in die Zahlentheorie“ von Bundschuh [Bun92] oder in „Elementary Number Theory“ von Stein [Ste09] zu finden. Wer mathematisch wirklich in die Tiefe gehen möchte und die unmittelbare Verbindung zur Kryptographiesucht, liest „Mathematics of Public Key Cryptography“ von Galbraith [Gal12]. Eine nützliche Sammlung von zahlentheoretischen Algorithmen findet sich in „A Course in Computational Algebraic Number Theory“ von Cohen [Coh93]. Sehr zu empfehlen ist auch „Introduction to Cryptography with Coding Theory“ von Trappe [TW06], die Abschnitte zu Kettenbrüchen folgen dieser Vorlage. Wer nach einem deutschen Buch zum Thema sucht, findet in „Kryptographie verständlich“ von Paar [Paa16] zu-

mindest Basisinfos zum Thema. Einige der Inhalte werden in aktuellen Büchern noch nicht ausreichend behandelt, hier habe ich zu den wissenschaftlichen Arbeiten oder den Standards/RFCs gegriffen.

Historische Chiffren werden in dieser Lehrveranstaltung nicht behandelt. Eine sehr genaue historische Behandlung der Kryptographie fast bis in die 1970er Jahre liefert Kahn in „The Codebreakers“ [Kah67]. Etwas unterhaltsamer und auch für's Nachtkasterl geeignet ist „Geheime Botschaften“ von Singh [Sin00].

Dieses Skriptum wurde mit viel Hingabe von Anna Vymazal Korrektur gelesen, die viele Fehler entdeckt hat. Was noch an Fehlern in diesem Skriptum steckt, liegt in meiner Verantwortung. Lass mich wissen, wenn du einen Fehler findest, damit ich ihn korrigieren kann.

Viel Vergnügen!

Kapitel 1

RSA

Ziele 1.1

In diesem Kapitel lernst du,

- warum das RSA-Verschlüsselungsverfahren funktioniert.
- warum und unter welchen Bedingungen das RSA-Verschlüsselungsverfahren sicher oder unsicher ist. Du lernst Attacken wie die Wiener-Attacke kennen.
- wie sich das RSA-Verschlüsselungsverfahren effizient implementieren lässt. Du lernst die Verfahren Square-and-Multiply und RSA-CRS kennen.
- wie die Geschwindigkeit des RSA-Verschlüsselungsverfahrens mit der Schlüssellänge skaliert.

1.1 Mathematische Werkzeuge

Ich setze für dieses Kapitel voraus, dass die Leserin oder der Leser vertraut ist mit den mathematischen Grundlagen des RSA-Verfahrens. Diese können im Bedarfsfall in Kapitel 5 des Begleithefts zur LVA „Diskrete Mathematik und lineare Algebra“ nachgelesen werden.

Im diesem Kapitel wird das RSA-Verfahren genauer unter die Lupe genommen. Dabei soll untersucht werden, welche Teile welche (Rechen-)Aufwände erzeugen. In der Folge werden gängige Optimierungen betrachtet.

In diesem Skriptum wird „mod“ in zwei verschiedenen Bedeutungen verwendet.

1. Als arithmetischer Operator wie in

$$17 \bmod 3 = 2$$

berechnet „mod“ den Rest bei der Division.

2. Als Kongruenzrelation wie in

$$7^{14} = 7 + 2 \pmod{10}$$

wird im Gegensatz dazu ausgesagt, dass diese Gleichung „modulo 10 stimmt“. Alternativ wird auch das Relationensymbol $\equiv_n$ verwendet, wenn dies satztechnisch einfacher ist:

$$7^{14} \equiv_{10} 7 + 2$$

Zur Unterscheidung zwischen dem Operator und der Relation wird bei der Relation stets die Schreibweise mit Klammern verwendet.

1.1.1 Euklids Algorithmus

Basis für den größten Teil der Methoden der Public-Key-Kryptographie ist Euklids Algorithmus[1] zur Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen. Die Anzahl der benötigten Schritte für den Euklidschen Algorithmus lässt sich recht gut abschätzen.

Satz 1.1: Lamé, 1844

Sind a und b natürliche Zahlen, ist $a > b$ und ist n die Bitlänge von b, so endet der Euklidsche Algorithmus zur Berechnung von $\text{ggT}(a, b)$ nach spätestens $17 \cdot n$ Schritten.

Im Durchschnitt geht es bedeutend schneller.

Satz 1.2: [Knu77, S. 356]

Es sei c eine natürliche Zahl mit n Bit Länge. Für zufällig gewählte Zahlen a und b zwischen 1 und c ist die erwartete Anzahl an Schritten zur Berechnung von $\text{ggT}(a, b)$ mit dem Euklidschen Algorithmus $0{,}584 \cdot n + 0{,}06$.

Da die Zahlen im Euklidschen Algorithmus stets kleiner werden, liegt der Hauptaufwand in den ersten Modulooperationen. Der erwartete Gesamtaufwand für die Berechnung eines größten gemeinsamen Teilers liegt in der Regel beim 10- bis 20-fachen einer Modulooperation.

1.1.2 Die Eulersche φ-Funktion

Wie viele Zahlen zwischen 0 und $n - 1$ sind relativ prim zu n? Diese Frage wollen wir jetzt beantworten.

[1] Vgl. Skriptum zur Vorlesung „Diskrete Mathematik und lineare Algebra“.

Definition 1.3

Für $n \in \mathbb{N}$ sei

$$\mathbb{Z}_n^* := \{k \in \mathbb{Z} \mid 0 \leq k < n \text{ und } \operatorname{ggT}(n,k) = 1\}.$$

Die Funktion

$$\begin{aligned} \varphi : \mathbb{N} &\to \mathbb{N} \\ n &\mapsto \mid \mathbb{Z}_n^* \mid \end{aligned}$$

heißt Eulersche φ–Funktion.

Ist $n \in \mathbb{P}$, so lässt sich $\varphi(n)$ recht einfach berechnen. Da n keine Primfaktoren (außer sich selbst) besitzt, gilt für jede ganze Zahl $k \in \{1, \ldots, n-1\}$: $\operatorname{ggT}(n,k) = 1$. Lediglich $\operatorname{ggT}(n,0) = n > 1$. Daher ist $\varphi(n) = n - 1$.

Im Fall $n = p \cdot q$ und $p \neq q \in \mathbb{P}$ ist $\varphi(n)$ ebenfalls recht einfach zu ermitteln. Dazu zählt man jene $k \in \{0, \ldots, n-1\}$, für welche $\operatorname{ggT}(n,k) > 1$ ist. Zum einen sind dies p und Vielfache davon, also $p, 2p, 3p, \ldots, (q-1)p$. Das nächstgrößere Vielfache qp ist bereits gleich n und damit größer als $n-1$. Zum anderen sind es die Vielfachen von q, $q, 2q, 3q, \ldots, (p-1)q$. Sind p und q verschieden, so gibt es auch keine gemeinsamen Vielfachen, das kleinste gemeinsame Vielfache wäre $pq = n$. Weitere Kandidaten für k mit $\operatorname{ggT}(n,k) > 1$ gibt es nicht, denn n besitzt nur zwei Primfaktoren (p und q). Als ggT kommen daher nur die Werte 1, p, q und $pq = n$ in Frage, für k demnach nur Vielfache von p oder q. Es gibt nun $q-1$ Vielfache von p und $p-1$ Vielfache von q, in Summe $q-1+p-1 = p+q-2$ Vielfache. Da die Null hier noch nicht gezählt wurde, ergeben sich $p+q-1$ Zahlen k mit $\operatorname{ggT}(n,k) > 1$. Mithin ist $\varphi(pq) = pq - (p+q-1)$, was sich schöner in der Form $\varphi(pq) = (p-1)(q-1)$ schreiben lässt.

Der folgende Satz verallgemeinert diese Ergebnisse für beliebige natürliche Zahlen.

Satz 1.4

Der Wert $\varphi(n)$ lässt sich effizient berechnen, wenn man die Primfaktorzerlegung von n kennt, denn es gilt:

1. Ist $p \in \mathbb{P}$, dann ist
$$\varphi(p) = p - 1.$$
2. Ist $p \in \mathbb{P}$ und $d \in \mathbb{N}$, dann ist
$$\varphi(p^d) = p^{d-1}(p-1).$$

3. Sind $m, n \in \mathbb{N}$ und $\text{ggT}(m, n) = 1$, dann ist

$$\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n).$$

4. Ist $n = p_1^{e_1} \cdots p_r^{e_r}$ die Primfaktorzerlegung von $n \in \mathbb{N}$, dann ist

$$\varphi(n) = n(1 - \frac{1}{p_1}) \cdots (1 - \frac{1}{p_r}).$$

Eine wichtige Voraussetzung in Satz 1.4 ist, dass die Primfaktorenzerlegung bekannt ist. Für den im RSA-Verfahren so wichtigen Fall $n = pq$ überlegen wir nun noch, dass es sich um eine notwendige Voraussetzung handelt, m.a.W.: Sind die Primfaktoren von n unbekannt, kann $\varphi(n)$ nicht berechnet werden. Um dies zu beweisen, wird angenommen, dass $\varphi(n)$ (wie auch immer) berechnet werden kann, und gezeigt, dass aus der Kenntnis von n und $\varphi(n)$ einfach die Primfaktoren p und q berechnet werden können.

Es seien also n und $\varphi(n)$ bekannt, weiterhin sei bekannt, dass n das Produkt zweier Primzahlen p und q ist. Dann gilt

$$pq = n \quad \text{und}$$
$$(p-1)(q-1) = \varphi(n)$$

Aus der ersten Gleichung ergibt sich $q = {}^{n}/_{p}$ und eingesetzt in die zweite Gleichung

$$(p-1)(\frac{n}{p} - 1) = \varphi(n).$$

Nun lässt sich diese Gleichung (in einer Variable p) umformen.

$$n - p - \frac{n}{p} + 1 = \varphi(n)$$
$$p + \varphi(n) - n - 1 + \frac{n}{p} = 0$$
$$p^2 + p(\varphi(n) - n - 1) + n = 0$$

Diese quadratische Gleichung in p lässt sich einfach lösen, da n und $\varphi(n)$ bekannt sind.

$$p_{1,2} = -\frac{\varphi(n) - n - 1}{2} \pm \sqrt{\left(\frac{\varphi(n) - n - 1}{2}\right)^2 - n}$$

Somit ist das Berechnen von $\varphi(n)$ bestimmt nicht einfacher als das Faktorisieren von n.

1.1.3 Der chinesische Restsatz

Das Rechnen modulo n hat unter anderem den Vorteil, dass man immer mit kleinen Zahlen rechnen kann, weil man ja immer den Rest modulo n nehmen darf[2], wenn man will. Das Ergebnis erfährt man dafür aber auch nur modulo n. Das lässt sich ändern.

[2] In Kürze werden wir sehen, dass auch im Exponenten immer mit kleinen Zahlen gerechnet werden kann. Korollar 1.12 wird das noch klären.

Beispiel 1.5

Die Reduktion modulo n ist eine recht einfache Operation, schnell ergibt sich

$$42 \bmod 3 = 0$$
$$42 \bmod 4 = 2$$
$$42 \bmod 5 = 2$$

Umgekehrt stellen wir uns jetzt die Frage, ob und wie sich aus den Restklassengleichungen

$$z = 0 \pmod 3$$
$$z = 2 \pmod 4$$
$$z = 2 \pmod 5$$

z bestimmen lässt. Schon wieder ein schwieriges Problem. Müssen wir hier probieren? Ist 42 die einzige Lösung?

Tatsächlich ist dieses Problem effizient lösbar und der erweiterte Euklidsche Algorithmus ist – wieder einmal – das Werkzeug der Wahl.

Algorithmus 1.6: Chinesischer Restsatz für zwei Gleichungen

Es seien $n_1, n_2 \in \mathbb{N}$, so dass $\text{ggT}(n_1, n_2) = 1$. Weiterhin seien z_1 und z_2 ganze Zahlen. Dann erhält man alle Lösungen des Restklassengleichungssystems

$$z = z_1 \pmod{n_1}$$
$$z = z_2 \pmod{n_2}$$

auf folgende Weise:

1. Berechne $n = n_1 \cdot n_2$.
2. Berechne mithilfe des erweiterten Euklidschen Algorithmus ganze Zahlen x_1 und x_2, so dass $n_1 x_1 + n_2 x_2 = 1$.
3. Berechne
$$z := z_1 n_2 x_2 + z_2 n_1 x_1 \bmod n.$$
Dieses z ist die eindeutige Lösung des Restklassengleichungssystems modulo n. Die Menge aller Lösungen ist $\{z + kn \mid k \in \mathbb{Z}\}$.

Ein Beweis für diesen Satz finden interessierte Leserinnen und Leser im Anhang in Abschnitt A.1.

Für Restklassengleichungssysteme mit mehr als zwei Gleichungen kann man mit zwei Gleichungen beginnen und dann nach und nach je eine Gleichung hinzunehmen. Schneller geht es mit dem folgenden Rezept.

Algorithmus 1.7: Chinesischer Restsatz

Es seien $n_1, n_2, \ldots, n_s$ paarweise relativ prime natürliche Zahlen. Weiterhin seien $z_1, z_2, \ldots, z_s$ ganze Zahlen. Dann erhält man alle Lösungen des Restklassengleichungssystems

$$\begin{aligned} z &= z_1 \pmod{n_1} \\ z &= z_2 \pmod{n_2} \\ &\vdots \\ z &= z_s \pmod{n_s} \end{aligned}$$

auf folgende Weise:

1. Berechne $n = n_1 \cdot n_2 \cdots n_s$. Modulo n ist die Lösung des Restklassengleichungssystems eindeutig.

2. Berechne für $i = 1, \ldots, s$ die Zahlen
$$q_i := \frac{n}{n_i}.$$

3. Jedes q_i ist relativ prim zu n_i und besitzt daher ein inverses Element modulo n_i. Berechne für $i = 1, \ldots, s$ das inverse Element r_i von q_i modulo n_i (mit dem erweiterten Euklidschen Algorithmus), also
$$r_i := q_i^{-1} \bmod n_i.$$

4. Berechne
$$z := z_1 q_1 r_1 + z_2 q_2 r_2 + \cdots + z_s q_s r_s \bmod n.$$
Dieses z ist die eindeutige Lösung des Restklassengleichungssystems modulo n. Die Menge aller Lösungen ist $\{z + kn \mid k \in \mathbb{Z}\}$.

Beispiel 1.8: Fortsetzung von Beispiel 1.5

Wir lösen das Restklassengleichungssystem

$$z = 0 \pmod 3$$
$$z = 2 \pmod 4$$
$$z = 2 \pmod 5$$

1. $n_1 = 3$, $n_2 = 4$ und $n_3 = 5$, also ist $n = 3 \cdot 4 \cdot 5 = 60$.
2. $q_1 = 20$, $q_2 = 15$, $q_3 = 12$.
3. $q_1^{-1} \bmod 3 = 2^{-1} \bmod 3 = 2$, $q_2^{-1} \bmod 4 = 3^{-1} \bmod 4 = 3$ und $q_3^{-1} \bmod 5 = 2^{-1} \bmod 5 = 3$, also sind $r_1 = 2$, $r_2 = 3$ und $r_3 = 3$.
4. $z = 0 \cdot 20 \cdot 2 + 2 \cdot 15 \cdot 3 + 2 \cdot 12 \cdot 3 = 162 = 42 \mod 60$.

Mit der Funktion `chinese_remainder` aus dem Modul `si` lässt sich dieses Restklassengleichungssystem lösen. Übergeben werden eine Liste mit den Moduln (3, 4 und 5) und eine Liste mit den Resten (0, 2 und 2).

```
> from si import chinese_remainder
> chinese_remainder( [3,4,5], [0,2,2] )
42
```

Bemerkung 1.9

Der mit Abstand aufwendigste Teil des Algorithmus ist Schritt 3. Man beachte jedoch, dass die Schritte 1–3 nicht erneut durchgeführt werden müssen, solange sich im Restklassengleichungssystem die Werte der Moduln $n_1, \ldots, n_s$ nicht ändern.

1.1.4 Die Sätze von Fermat und Euler

Satz 1.10: Kleiner Satz von Fermat

Ist $p \in \mathbb{P}$, ist $z \in \mathbb{Z}$ und ist $\mathrm{ggT}(z, p) = 1$, dann gilt

$$z^{p-1} = 1 \pmod p.$$

Beweis. Es sei p eine Primzahl und z eine ganze Zahl, die relativ prim zu p ist. Sind a und b ganze Zahlen, so dass $a \neq b \pmod{p}$, dann ist auch $az \neq bz \pmod{p}$, denn wäre $az = bz \pmod{p}$, so könnte man diese Gleichung auf beiden Seiten mit $z^{-1} \bmod p$ multiplizieren und erhielte $a = b \pmod{p}$.

Berechnet man

$$\prod_{i=1}^{p-1} (i \cdot z \bmod p)$$

so erhält man das Produkt von $p-1$ verschiedenen Zahlen außer 0 modulo p. Dies ist also das Produkt aller Zahlen außer 0 modulo p. So erhält man

$$\prod_{i=1}^{p-1} ((i \cdot z) \bmod p) = \prod_{i=1}^{p-1} i \pmod{p}$$

und weiter

$$z^{p-1} \cdot \prod_{i=1}^{p-1} i = \prod_{i=1}^{p-1} i \pmod{p}.$$

Multipliziert mit $\left(\prod_{i=1}^{p-1} i\right)^{-1} \bmod p$ erhält man

$$z^{p-1} = 1 \pmod{p}.$$

□

Der kleine Satz von Fermat lässt sich verallgemeinern. Einen Beweis dafür werden wir erst später liefern, dann aber gleich noch allgemeiner für Satz 2.9.

Satz 1.11: Euler

Sind $n \in \mathbb{N}$ und $z \in \mathbb{Z}$ und ist $\text{ggT}(z, n) = 1$, dann gilt

$$z^{\varphi(n)} = 1 \pmod{n}.$$

Korollar 1.12

Sind $n \in \mathbb{N}$ und $z, a, b \in \mathbb{Z}$ und ist $\text{ggT}(z, n) = 1$, dann gilt:

1. Ist $a = b \pmod{\varphi(n)}$, dann ist $z^a = z^b \pmod{n}$.
2. $z^a = z^{a \bmod \varphi(n)} \pmod{n}$.

Aus dem Satz von Fermat lässt sich mit Hilfe des chinesischen Restsatzes jedoch einfach der Satz von Euler für den Fall $n = pq$ gewinnen. Dies ist auch der einzige Fall, der im Zusammenhang mit RSA von Interesse sein wird.

Satz 1.13: Satz von Euler für $n = pq$

Sind $p, q \in \mathbb{P}$, $n = pq$ und $z \in \mathbb{Z}$, dann gilt für alle $z \in \mathbb{Z}$ mit $\text{ggT}(z, n) = 1$:

$$z^{(p-1)(q-1)} = 1 \pmod{n}.$$

Beweis. Wir berechnen zunächst mit Hilfe des Satzes von Fermat

$$\left(z^{(p-1)(q-1)} \bmod n\right) \bmod p = z^{(p-1)(q-1)} \bmod p = \left(z^{q-1}\right)^{p-1} \bmod p = 1.$$

Analog ergibt sich

$$\left(z^{(p-1)(q-1)} \bmod n\right) \bmod q = 1.$$

Somit ist $z^{(p-1)(q-1)} \bmod n$ die modulo pq eindeutige Lösung des Restklassengleichungssystems

$$x = 1 \pmod{p}$$
$$x = 1 \pmod{q}$$

Offensichtlich ist aber $x = 1$ eine Lösung dieses Restklassengleichungssystems, somit ist $z^{(p-1)(q-1)} = 1 \pmod{n}$. □

1.2 Performance des RSA-Verfahrens

In aller Kürze soll zunächst das RSA-Verfahren wiederholt werden. Hier und weiter im Rest des Skriptums steht „$x \xleftarrow{R} Y$“ für „das Element x wird zufällig aus der Menge Y gewählt“.

Algorithmus 1.14: RSA

Schlüsselerzeugung:

- $p, q \xleftarrow{R} \mathbb{P}$
- $n := pq$, $\varphi(n) = (p-1)(q-1)$
- $e \in \mathbb{Z}^*_{\varphi(n)}$ (beliebig)
- $d := e^{-1} \bmod \varphi(n)$

$\rightarrow$ Public Key (n, e)

$\rightarrow$ Private Key (p, q, d)

Verschlüsseln:

$$c := m^e \bmod n$$

Entschlüsseln:

$$m := c^d \bmod n$$

An einem kleinen Beispiel (mit Zahlen, die natürlich viel zu klein sind) soll zunächst demonstriert werden, was bei einer RSA-Verschlüsselung zu berechnen ist.

Beispiel 1.15

Alice wählt $p = 7$ und $q = 11$, also $n = 77$ und $\varphi(77) = 6 \cdot 10 = 60$. Nun wählt sie $e = 13$ und sendet Bob ihren Public Key $(n, e) = (77, 13)$. Weiterhin berechnet sie den Kehrwert von 13 modulo 60 mit dem erweiterten Euklidschen Algorithmus und erhält $13^{-1} \bmod 60 = 37$. Der Private Key von Alice ist daher $(p, q, d) = (7, 11, 37)$.

Bob verschlüsselt die Nachricht $m = 4$. Er berechnet

$$c := m^e = 4^{13} = 67108864 = 53 \pmod{77}$$

und schickt Alice $c = 53$. Alice entschlüsselt

$$\begin{aligned} m = c^d &= 53^{37} = \\ &= 628358038363668332248635694548393830494073197366814679114902621 3 \\ &= 4 \pmod{77}. \end{aligned}$$

1.2.1 Square and Multiply

Das kleine Beispiel 1.15 zeigt bereits, dass das Potenzieren – selbst mit kleinen Zahlen – zu sehr großen Zahlen führt. Es wird nun überlegt, wie eine Berechnung „$x^e \pmod{n}$“ effizient durchgeführt werden kann. Dies erfolgt in zwei Schritten. Zunächst werden Potenzen m^e berechnet für Exponenten e, die Potenzen von 2 sind. In einem zweiten Schritt werden diese Ergebnisse benutzt, um allgemeine Potenzen zu berechnen.

1. Ist $e = 2^k$, so kann $y := x^e$ durch k-maliges Quadrieren von x berechnet werden:

$$x^{2^k} = \underbrace{\left(\left(\left(x^2\right)^2\right)^2 \dots\right)^2}_{k\text{-mal quadrieren}}$$

2. Ist $(b_m, b_{m-1}, \dots, b_0) \in \mathbb{Z}_2^{m+1}$ die Binärdarstellung von e, also $e = \sum_{k=0}^{m} b_k \cdot 2^k$,dann ist

$$x^e = x^{\sum_{k=0}^{m} b_k \cdot 2^k} = \prod_{k=0}^{m} x^{b_k \cdot 2^k} = \prod_{k=0}^{m} \left(x^{2^k}\right)^{b_k}.$$

   Es sind also alle jene Potenzen x^{2^k} zu multiplizieren, wo $b_k = 1$ ist.

Dies führt zu folgendem Algorithmus.

Algorithmus 1.16: Square and Multiply

Gegeben sind $x \in \mathbb{Z}$ und $n, e \in \mathbb{N}$, gesucht ist $y = x^e \bmod n$.

1. Bestimme die Binärdarstellung $e = \sum_{k=0}^{m} b_k \cdot 2^k$ ($b_0, \ldots, b_m \in \{0,1\}$) von e.
2. Berechne durch fortgesetztes Quadrieren die Zahlen $x, x^2, x^4, x^{2^3}, \ldots, x^{2^m}$ (jeweils modulo n).
3. Multipliziere (modulo n) die Potenzen x^{2^k} von x, für jene k, für die $b_k = 1$.

Das folgende Beispiel soll illustrieren, wie effizientes Potenzieren mit der Square-and-Multiply-Methode funktioniert. Dabei lassen wir zunächst die Moduloperation außer Acht.

Beispiel 1.17

Um 53^{37} zu berechnen, werden zunächst Potenzen für Zweierpotenzen berechnet.

$$
\begin{aligned}
53^{2^0} &= 53^1 = 53,\\
53^{2^1} &= 53^2 = 2809,\\
53^{2^2} &= 53^4 = (53^2)^2 = 2809^2 = 7890481,\\
53^{2^3} &= 53^8 = (53^4)^2 = 7890481^2 = 62259690411361,\\
53^{2^4} &= 53^{16} = (53^8)^2 = 62259690411361^2 = 3876269050118516845397872321,\\
53^{2^5} &= 53^{32} = (53^{16})^2 =\\
&= 1502546174890670885945286107013099326955379687381792704 1
\end{aligned}
$$

Mit diesen Ergebnissen lässt sich nun 53^{37} berechnen. Es ist $37 = 32 + 4 + 1 = 2^5 + 2^2 + 2^0$. Die Binärdarstellung von 37 ist $(100101)_2$. Somit ist

$$
\begin{aligned}
53^{37} =& 53^{2^5} \cdot 53^{2^2} \cdot 53^{2^0} =\\
=& 1502546174890670885945286107013099326955379687381792704 1\\
& \cdot 7890481 \cdot 53 = 628358038363\ldots49026213.
\end{aligned}
$$

Für diese Berechnung waren fünf Multiplikationen in Schritt 1 und zwei Multiplikationen in Schritt 2 nötig, in Summe also sieben Multiplikationen.

Im Beispiel RSA ist eigentlich die Potenz einer Restklasse zu berechnen, am Ende interessiert uns das Ergebnis modulo n. Daher ist es gestattet, auch alle Zwischenergebnisse als Restklassen modulo n zu behandeln, d. h. den Rest bei der Division durch n weiterzuverwenden. Dies führt

dazu, dass die Zahlen, mit denen zu rechnen ist, kleiner (jedenfalls kleiner als n) werden.

Beispiel 1.18

Um $53^{37} \bmod 77$ zu berechnen, werden zunächst Potenzen für Zweierpotenzen berechnet. Dabei wird sofort modulo 77 reduziert.

$$\begin{aligned}
53^{2^0} &= 53^1 = 53,\\
53^{2^1} &= 53^2 = 37 \pmod{77},\\
53^{2^2} &= 53^4 = (53^2)^2 = 37^2 = 60 \pmod{77},\\
53^{2^3} &= 53^8 = (53^4)^2 = 60^2 = 58 \pmod{77},\\
53^{2^4} &= 53^{16} = (53^8)^2 = 58^2 = 53 \pmod{77},\\
53^{2^5} &= 53^{32} = (53^{16})^2 = 53^2 = 37 \pmod{77}
\end{aligned}$$

Mit diesen Ergebnissen lässt sich nun $53^{37} \bmod 77$ berechnen. Es ist $37 = 32+4+1 = 2^5+2^2+2^0$. Die Binärdarstellung von 37 ist $(100101)_2$. Somit ist

$$53^{37} = 53^{2^5} \cdot 53^{2^2} \cdot 53^{2^0} = 37 \cdot 60 \cdot 53 = 4.$$

Für diese Berechnung waren fünf Multiplikationen in Schritt 1 und zwei Multiplikationen in Schritt 2 nötig, in Summe also sieben Multiplikationen (modulo 77).

In Python lassen sich modulare Potenzen einfach mit der Funktion `pow` berechnen.

```
> pow( 53, 37, 77 )
4
```

Allgemein lässt sich erkennen: Ist die Binärdarstellung des Exponenten $m + 1$ Bits lang und enthält w Einsen, so sind im ersten Schritt m und im zweiten Schritt $w - 1$ Multiplikationen erforderlich. Verdoppelt sich die Bitlänge des Exponenten, so steigt der Aufwand für das Potenzieren auf das Doppelte.

Der Aufwand für eine einzelne Multiplikation hängt ebenfalls von der Größe der Zahlen ab. Für bis zu 64 Bit lange Zahlen kann eine Multiplikation als eine Instruktion aufgefasst werden. Für längere Zahlen kann die Multiplikation von 64-Bit-Zahlen als das „kleine Einmaleins“ aufgefasst werden, dann muss jede (64-Bit-)Ziffer der ersten mit jeder der zweiten Zahl multipliziert werden. Verdoppelt sich die Länge der zu multiplizierenden Zahlen, so steigt damit der Aufwand auf das Vierfache.

Kann man also (durch die Reduktion modulo n) mit kleineren Zahlen rechnen, dann geht dies schneller. Allerdings müssen dann auch nach jedem Rechenschritt Modulo-Operationen – also Divisionen mit Rest – durchgeführt werden. Wie bei den Multiplikationen steigt auch hier mit doppelter Länge des Moduls der Aufwand auf das Vierfache.

Auch der Aufwand für Additionen steigt mit der Länge der Zahlen. Allerdings wächst dieser Aufwand weniger stark, bei einer Verdopplung der Länge der Zahlen verdoppelt sich der Aufwand nur. Nach Additionen ist auch die Modulo-Operation weniger aufwendig. Werden zwei Zahlen addiert, die kleiner sind als n, so ist ihre Summe kleiner als $2n$. Es braucht hier keine Division durchgeführt werden: Ist die Summe kleiner als n, dann ist nichts zu tun. Ist die Summe größer als n, muss nur n subtrahiert werden. Im Vergleich zu Multiplikationen fallen Additionen daher nicht sehr ins Gewicht und es reicht, sich mit den Multiplikationen zu beschäftigen.

Praktisch stellt sich das RSA-Verfahren erst bei Schlüssellängen (Bitlänge von n) ab 3000 Bit als sicher heraus. Entsprechend aufwendig gestaltet sich dort das Ver- und Entschlüsseln. Um das Verschlüsseln effizient zu gestalten, kann ein kleiner Exponent e gewählt werden, typischerweise ist dies $e = 65537$. Betrachtet man die Binärdarstellung $(10000000000000001)_2$ dieser Zahl, so wird klar, warum – sowohl in Schritt 1 wie auch in Schritt 2 – Square and Multiply besonders effizient durchgeführt werden kann ($m = 16$, $w = 2$). Da es sich überdies bei 65537 um eine Primzahl handelt, ist mit großer Wahrscheinlichkeit auch $\text{ggT}(\varphi(n), e) = 1$.

Für das Ver- und Entschlüsseln im RSA-Verfahren lässt sich somit festhalten: Eine Verdopplung der Schlüssellänge führt zu einer Vervierfachung des Aufwands beim Verschlüsseln und zu einer Verachtfachung des Aufwands beim Entschlüsseln. Überdies ergibt sich ein großer Geschwindigkeitsunterschied zwischen Ver- und Entschlüsseln, da beim Verschlüsseln stets ein kleiner (65537; Länge: 17 Bit) Exponent verwendet werden kann, beim Entschlüsseln der Exponent jedoch immer groß (>3000 Bit) ist, was einen Faktor von zumindest ca. $3000/17 \approx 180$ für die Geschwindigkeit bedeutet.

Der aufwendigste Teil der Schlüsselerzeugung ist, zwei große Primzahlen zu finden. Dazu werden zufällig gewählte Zahlen mit dem Miller-Rabin-Test auf Primalität getestet. Der Aufwand eines solchen Tests entspricht dem modularen Potenzieren, auch hier steigt der Aufwand – wie beim Entschlüsseln – bei einer Verdopplung der Schlüssellänge auf das Achtfache. Zusätzlich ergibt sich aus den Primzahlverteilungssätzen[3] jedoch, dass doppelt so lange Primzahlen nur etwa halb so häufig sind, d. h. dass die Suche aus diesem Grund etwa doppelt so lange dauert. Der Aufwand für die Schlüsselerzeugung steigt daher um den Faktor 16, wenn die Schlüssellänge verdoppelt wird.

1.2.2 RSA-CRS

Der chinesische Restsatz erlaubt es, die Entschlüsselung um den Faktor 4 zu beschleunigen. Üblicherweise wird die RSA-Entschlüsselung in dieser Form implementiert. Es geht darum, $m = c^d \bmod pq$ zu berechnen. Alternativ lassen sich

$$m_p := c^d \bmod p = (c^d \bmod pq) \bmod p = m \bmod p \text{ und}$$
$$m_q := c^d \bmod q = (c^d \bmod pq) \bmod q = m \bmod q$$

[3] Vgl. die Vorlesung „Grundlagen der Kryptographie“.

berechnen und da p und q relativ prim sind, ergibt sich m (modulo pq) mit dem chinesischen Restsatz aus m_p und m_q. Der Vorteil dieser Art der Berechnung ist die höhere Geschwindigkeit. Sind p und q ungefähr gleich lang, so sind beide nur halb so lang wie pq, damit werden alle Multiplikationen ca. viermal so schnell. Da jetzt statt einmal zweimal potenziert werden muss, geht ein Teil des Geschwindigkeitsgewinns wieder verloren[4], es bleibt aber immer noch ein Faktor 2. Allerdings lassen sich nun auch die Potenzen deutlich schneller berechnen, denn statt $m_p = c^d \bmod p$ darf man nach dem Satz von Fermat auch $m_p = c^{d \bmod p-1} \bmod p$ (Analoges für m_q) rechnen. Das ursprüngliche d war so groß wie pq, $d_p = d \bmod p$ ist aber nur halb so lang. Damit lässt sich noch einmal ein Faktor 2 gewinnen, sodass sich sagen lässt, dass sich RSA-Entschlüsselung mit dem chinesischen Restsatz ca. um den Faktor 4 beschleunigen lässt. Der aufwendigste Teil im chinesischen Restsatz lässt sich überdies schon während der Schlüsselerzeugung vorausberechnen, die Primzahlen p und q ändern sich ja nicht mehr. Es ergibt sich das folgende Verfahren:

Man erzeuge das RSA-Schlüsselpaar (n, e), (p, q, d) wie gehabt. Weiterhin berechne man

$$d_p := d \bmod p - 1 \text{ und}$$
$$d_q := d \bmod q - 1$$

sowie mit dem erweiterten Euklidschen Algorithmus Zahlen x, y, so dass $px + qy = 1$ gilt. Bei der Entschlüsselung berechnet man aus m_p und m_q mit dem chinesischen Restsatz

$$m := m_p qy + m_q px \bmod pq.$$

Auf den zweiten Blick erkennt man, dass es reicht, sich die Zahl y zu merken[5], denn es ist $px + qy = 1$ und daher kann man px durch $1 - qy$ ersetzen. Der Overhead durch den chinesischen Restsatz ist nur eine Multiplikation und zwei Additionen. Diese RSA-Variante ist insbesondere im Standard RSA-PKCS#1 beschrieben.[6]

[4] Die Berechnungen modulo p und modulo q lassen sich parallel zueinander durchführen. Steht also mehr als ein Prozessorkern zur Verfügung, ist noch einmal ein Faktor 2 an Geschwindigkeit zu gewinnen.

[5] Die Zahl y wird in RFC 8017 als `qInv` bezeichnet. Es handelt sich dabei ja um den Kehrwert von q modulo p.

[6] RFC 8017 – https://datatracker.ietf.org/doc/html/rfc8017

Algorithmus 1.19: RSA-CRS

Modifizierte Schlüsselerzeugung:
Man erzeuge das RSA-Schlüsselpaar (n, e), (p, q, d) wie gehabt. Weiterhin berechne man $d_p := d \bmod p-1$ und $d_q := d \bmod q-1$ sowie mit dem erweiterten Euklidschen Algorithmus Zahlen x, y, so dass $px + qy = 1$ gilt. Der erweiterte Private Key ist (p, q, d_p, d_q, y).
Modifizierte Entschlüsselung:
Man berechne

$$\begin{aligned} m_p &:= c^{d_p} \bmod p, \\ m_q &:= c^{d_q} \bmod q, \\ h &:= (m_p - m_q)\, y \bmod p, \\ m &:= m_q + qh \bmod pq. \end{aligned}$$

1.3 Angriffe auf RSA bei falscher Wahl der Schlüssel

Der letzte Abschnitt dieses Kapitels zeigt, dass bei der Schlüsselerzeugung darauf geachtet werden muss, dass alle Parameter zufällig und von der richtigen Bitlänge gewählt werden müssen. Wieners Attacke zeigt, dass die Bitlänge des Exponenten im Private Key ausreichend groß sein muss, die Fermat-Faktorisierung zerlegt RSA-Moduln in ihre Primfaktoren, wenn sich diese zu wenig unterscheiden. Beide Probleme treten nicht auf, wenn RSA-Schlüssel wie beschrieben erzeugt werden. Dennoch tauchen immer wieder RSA-Schlüssel auf, die auf einen dieser Angriffe anfällig sind.

1.3.1 Wieners Attacke über Kettenbrüche

In diesem Abschnitt wird sich herausstellen, dass die Berechnungen für den größten gemeinsamen Teiler auch ganz woanders auftauchen. Mit der Klappe Euklids lässt sich also auch noch eine zweite Fliege schlagen. Diese wird jetzt genauer betrachtet. Es wird hier vorerst keine neue Theorie mehr benötigt, wir stürzen uns direkt in das Vergnügen.

Beispiel 1.20

Der Astronom Christiaan Huygens untersuchte die Umlaufzeiten von Planeten. Er stellte fest, dass die Erde in 365 Tagen genau 359°45′40″31‴ überstreicht (1 Grad entspricht 60 Minuten, 1 Minute entspricht 60 Sekunden, eine Sekunde 60 Tertien), Saturn in der selben Zeit 12°13′34″18‴. In Tertien beträgt das Verhältnis

$$\rho = \frac{77708431}{2640858} \approx 29,4254484716709.$$

Ein realistisches Modell, bei welchem die Bewegungen über Zahnräder gesteuert werden, würde demnach Zahnräder mit 777.708.431 und 2.640.858 Zähnen benötigen, welche recht schwer zu fertigen wären. Huygens suchte also nach Brüchen, deren Wert nahe am Wert ρ liegt, deren Zähler und Nenner aber möglichst klein sind.

Zu diesem Zweck schrieb er

$$\frac{77708431}{2640858} = 29 + \frac{1123549}{2640858},$$

das ist die Darstellung eines unechten Bruches als gemischter Bruch. In erster Näherung ist der Wert des Bruches also 29. Der Rest, also $\frac{1123549}{2640858}$, lässt sich auch schreiben als $\frac{1}{\frac{2640858}{1123549}}$. Mit dem Nenner dieses Bruchs kann man nun das gleiche Spiel spielen.

$$= 29 + \frac{1}{\frac{2640858}{1123549}} = 29 + \frac{1}{2 + \frac{393760}{1123549}}$$

Die Zahl $29 + \frac{1}{2} = \frac{59}{2} = 29,5$ ist eine beträchtlich bessere Näherung für ρ. Wir machen weiter.

$$= 29 + \frac{1}{2 + \frac{1}{\frac{1123549}{393760}}} = 29 + \frac{1}{2 + \frac{1}{2 + \frac{336029}{393760}}}$$

Die nächste Näherung ist $29 + \frac{1}{2+\frac{1}{2}} = \frac{147}{5} = 29,4$, noch besser.

$$= 29 + \frac{1}{2 + \frac{1}{2 + \frac{1}{\frac{393760}{336029}}}} = 29 + \frac{1}{2 + \frac{1}{2 + \frac{1}{1 + \frac{57731}{336029}}}}$$

Die nächste Näherung ist $29 + \frac{1}{2+\frac{1}{2+\frac{1}{1}}} = \frac{206}{7} = 29,4286.$

Dies ist eine sehr gute Näherung für ρ, bei der Zähler und Nenner noch so klein sind, dass Zahnräder mit entsprechend vielen Zähnen gefertigt werden können.

Da die Darstellung als Bruch bei Kettenbrüchen schnell recht unübersichtlich wird und dabei

höchst redundant ist, hat sich eine Platz sparende Notation durchgesetzt.

Notation 1.21

Ist $n \in \mathbb{N}$ und sind $a_0, \ldots, a_n \in \mathbb{N}$, dann sei

$$[a_0; a_1, \ldots, a_n] := a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}}.$$

Algorithmus 1.22: Kettenbruchentwicklung für rationale Zahlen

Ist $x = \frac{p}{q} \in \mathbb{Q}$, so kann die Kettenbruchentwicklung $x = [a_0; a_1, a_2, \ldots]$ auf folgende Art berechnet werden.

1. Berechne mit dem Euklidschen Algorithmus den ggT von p und q.
2. Die Quotienten der Divisionen in den einzelnen Schritten sind genau die Werte $a_0, a_1, \ldots$.

Beispiel 1.23

Der erweiterte Euklidsche Algorithmus für 77708431 und 2640858 ergibt die folgende Tabelle.

```
> from si import extended_gcd
> extended_gcd( 77708431, 2640858, verbose=1 )
                        77708431          2640858
        77708431               1                0
         2640858               0                1         29
         1123549               1              -29          2
          393760              -2               59          2
          336029               5             -147          1
           57731              -7              206          5
           47374              40            -1177          1
           10357             -47             1383          4
            5946             228            -6709          1
            4411            -275             8092          1
            1535             503           -14801          2
```

1341	-1281	37694	1
194	1784	-52495	6
177	-11985	352664	1
17	13769	-405159	10
7	-149675	4404254	2
3	313119	-9213667	2
1	-775913	22831588	3

Als Kettenbruch ist also

$$\frac{77708431}{2640858} = [29; 2, 2, 1, 5, 1, 4, 1, 1, 2, 1, 6, 1, 10, 2, 2, 3].$$

Kettenbrüche lassen sich nicht nur für Brüche, sondern für beliebige reelle Zahlen entwickeln.[7] In gewisser Hinsicht handelt es sich dabei um die besten Näherungen, die man erhalten kann. Exakter wird dies durch den nächsten Satz ausgedrückt.

Satz 1.24

Es seien $x \in \mathbb{R}$ und $[a_0; a_1, a_2, \ldots]$ die Kettenbruchentwicklung von x. Weiterhin seien r und s ganze Zahlen. Dann gilt:

$$\text{Ist } \left|x - \frac{r}{s}\right| < \frac{1}{2s^2}, \text{ dann gibt es ein } n \in \mathbb{N}, \text{ sodass } \frac{r}{s} = [a_0; a_1, \ldots, a_n].$$

Wir arbeiten weiter an Beispiel 1.23.

Beispiel 1.25

Bricht man die Kettenbruchentwicklung $[29; 2, 2, 1, 5, 1, 4, 1, 1, 2, 1, 6, 1, 10, 2, 2, 3]$ schon beim Glied a_n ab, so erhält man Näherungen an die eigentliche Zahl x.

[7] Auf Kettenbruchentwicklungen für beliebige reelle Zahlen wird hier nicht eingegangen, da diese für das Weitere nicht benötigt werden.

n	$\frac{r}{s}$	$x - \frac{r}{s}$	$\frac{1}{2s^2}$	$\left\lvert x - \frac{r}{s}\right\rvert < \frac{1}{2s^2}$
0	29	$4{,}3 \cdot 10^{-1}$	$5{,}0 \cdot 10^{-1}$	✓
1	$\frac{59}{2}$	$-7{,}5 \cdot 10^{-2}$	$1{,}3 \cdot 10^{-1}$	✓
2	$\frac{147}{5}$	$2{,}5 \cdot 10^{-2}$	$2{,}0 \cdot 10^{-2}$	
3	$\frac{206}{7}$	$-3{,}1 \cdot 10^{-3}$	$1{,}0 \cdot 10^{-2}$	✓
4	$\frac{1177}{40}$	$4{,}5 \cdot 10^{-4}$	$3{,}1 \cdot 10^{-4}$	
5	$\frac{1383}{47}$	$-8{,}3 \cdot 10^{-5}$	$2{,}3 \cdot 10^{-4}$	✓
6	$\frac{6709}{228}$	$9{,}9 \cdot 10^{-6}$	$9{,}6 \cdot 10^{-6}$	
7	$\frac{8092}{275}$	$-6{,}1 \cdot 10^{-6}$	$6{,}6 \cdot 10^{-6}$	✓
8	$\frac{14801}{503}$	$1{,}2 \cdot 10^{-6}$	$2{,}0 \cdot 10^{-6}$	✓
9	$\frac{37694}{1281}$	$-4{,}0 \cdot 10^{-7}$	$3{,}0 \cdot 10^{-7}$	
10	$\frac{52495}{1784}$	$4{,}1 \cdot 10^{-8}$	$1{,}6 \cdot 10^{-7}$	✓
11	$\frac{352664}{11985}$	$-5{,}6 \cdot 10^{-9}$	$3{,}5 \cdot 10^{-9}$	
12	$\frac{405159}{13769}$	$4{,}7 \cdot 10^{-10}$	$2{,}6 \cdot 10^{-9}$	✓
13	$\frac{4404254}{149675}$	$-1{,}8 \cdot 10^{-11}$	$2{,}2 \cdot 10^{-11}$	✓
14	$\frac{9213667}{313119}$	$3{,}6 \cdot 10^{-12}$	$5{,}1 \cdot 10^{-12}$	✓
15	$\frac{22831588}{775913}$	$-4{,}9 \cdot 10^{-13}$	$8{,}3 \cdot 10^{-13}$	✓
16	$\frac{77708431}{2640858}$	0	$7{,}2 \cdot 10^{-14}$	✓

Zwei Dinge sind an diesem Beispiel zu erkennen. Zum Einen über- und unterschätzen die so erhaltenen Näherungen abwechselnd den Wert x, die erste Näherung ist für positive Zahlen stets zu klein. Zum Anderen haben nicht alle Näherungen die in Satz 1.24 angegebene Eigenschaft. Satz 1.24 sagt, dass es außer den markierten Brüchen keine weiteren Brüche mit dieser Eigenschaft mehr gibt. In dieser Hinsicht liefert die Kettenbruchzerlegung alle guten Näherungen, aber nicht ausschließlich gute Näherungen.

Nun werden sich Kettenbrüche als überraschend wirkungsvolles Hilfsmittel bei einer Attacke auf das RSA-Verschlüsselungsverfahren herausstellen.

Satz 1.26: Wiener

Seien $p, q \in \mathbb{P}$ mit $q < p < 2q$. Es sei $n := pq$ und $d, e \in \mathbb{N}$ seien so gewählt, dass $ed = 1 \pmod{\varphi(n)}$. Ist $d < \frac{1}{3}\sqrt[4]{n}$, dann lässt sich d aus dem RSA Public Key (n, e) einfach berechnen.

Der Trick liegt darin, zu erkennen, dass bei den gegebenen Parametern der geheime Exponent als einer der Nenner in der Kettenbruchentwicklung des Bruchs e/n auftaucht (und e und n sind öffentlich bekannt). Genauer: Sei k jene positive ganze Zahl, für die

$$ed - 1 = k\varphi(n).$$

(So eine Zahl gibt es, denn $ed = 1 \pmod{\varphi(n)}$.) Dann ist k/d ein Bruch, so dass

$$\left|\frac{e}{n} - \frac{k}{d}\right| < \frac{1}{2d^2}. \tag{1.1}$$

(Details dazu finden sich in Anhang A.2.) Nach Satz 1.24 taucht der Bruch k/d also in der Kettenbruchentwicklung von e/n auf. Diese ist einfach zu berechnen. Aus der Herleitung geht auch hervor, dass k/d stets größer ist als e/n, es reicht also, jede zweite Näherung zu betrachten, weil Näherungen ja abwechselnd über- und unterschätzen.

Aus diesen Beobachtungen lässt sich folgender Angriff machen.

Algorithmus 1.27: Wiener-Attacke

Mit den folgenden Schritten lassen sich aus einem Public Key (n, e) die Primfaktoren von n berechnen, wenn d ausreichend klein ist.

1. Setze $i := 1$.
2. Berechne die Kettenbruchentwicklung $K/D = [a_0; a_1, \ldots, a_i]$ des Bruchs e/n.
3. Ist D gerade, setze $i := i + 2$. Weiter bei Schritt 2.
   (Das gesuchte d ist stets ungerade, denn es besitzt einen Kehrwert modulo $(p-1)(q-1)$ und diese Zahl ist gerade.)
4. Berechne $\Phi := (e \cdot D - 1)/K$.
5. Ist Φ keine ganze Zahl, setze $i := i + 2$. Weiter bei Schritt 2.
   (Es ist $e \cdot D - 1 = k\varphi(n)$, also $\Phi = \varphi(n)$. Das muss eine ganze Zahl sein.)
6. Berechne die beiden Lösungen der Gleichung

   $$x^2 - (n - \Phi + 1)x + n = 0.$$

   Sind die beiden Lösungen ganzzahlig, so handelt es sich um die beiden Primfaktoren von n.
   (Sind n und $\Phi = \varphi(n)$ bekannt, lassen sich die Primfaktoren von n berechnen (vgl. Abschnitt 1.1.2). Diese sind wieder ganze Zahlen.)
7. Andernfalls, setze $i := i + 2$. Weiter bei Schritt 2.

Auch wenn die Kettenbruchentwicklung schnell berechnet ist, ist es zunächst mühsam, einen Kettenbruch wie $[0; 1, 1, 1, 15, 1, 1, 1, 3, 2]$ in den Bruch $\frac{807}{1223}$ umzurechnen. Da hilft es auf den ersten Blick auch wenig, wenn man den Kettenbruch $[0; 1, 1, 1, 15, 1, 1, 1] = \frac{97}{147}$ zuvor schon ausgerechnet hat. Mit viel Nachdenken lassen sich aber ganz brauchbare Formeln finden, die es

erlauben, aus den letzten beiden Kettenbrüchen den nächsten schnell zu berechnen.

Lemma 1.28

Es sei $[a_0; a_1, a_2, \ldots]$ ein Kettenbruch. Für jedes $n \in \mathbb{N}$ seien $p_n, q_n \in \mathbb{N}$ wie folgt gewählt.

$$\begin{aligned} p_0 &:= a_0, & q_0 &:= 1, \\ p_1 &:= a_0 \cdot a_1 + 1, & q_1 &:= a_1, \\ p_n &:= a_n \cdot p_{n-1} + p_{n-2}, & q_n &:= a_n \cdot q_{n-1} + q_{n-2} \quad (n \geq 2). \end{aligned}$$

Dann gilt für jedes $n \in \mathbb{N}$: $[a_0; a_1, a_2, \ldots, a_n] = \frac{p_n}{q_n}$.

Das folgende Beispiel zeigt, wie sich damit einfach Näherungen berechnen lassen.

Beispiel 1.29

Für den Kettenbruch $[4; 3, 2, 5, 2, 1, 3]$ ergibt sich

$$\begin{aligned} p_0 &= 4, & q_0 &= 1, \\ p_1 &= 4 \cdot 3 + 1 = 13, & q_1 &= 3, \\ p_2 &= 2 \cdot 13 + 4 = 30, & q_2 &= 2 \cdot 3 + 1 = 7, \\ p_3 &= 5 \cdot 30 + 13 = 163, & q_3 &= 5 \cdot 7 + 3 = 38 \text{ usw.} \end{aligned}$$

Die ersten Näherungen sind somit die Brüche

$$\frac{4}{1}, \frac{13}{3}, \frac{30}{7} \text{ und } \frac{163}{38}.$$

Den Abschnitt beschließt ein Beispiel mit etwas größeren Zahlen.

Beispiel 1.30

Zum Brechen des RSA Public Key

$$(n, e) = (19452881344027252501, 11591841614497619999)$$

werden zunächst mit dem Euklidschen Algorithmus (nur die erste und die letzte Spalte müssen berechnet werden) die Kettenbruchdarstellungen ermittelt.

11591841614497619999	
19452881344027252501	0
11591841614497619999	1
7861039729529632502	1
3730801884967987497	2
399435959593657508	9
135878248625069925	2
127679462343517658	1
8198786281552267	15
4697668120233653	1
3501118161318614	1
1196549958915039	2
1108018243488536	1
⋮	⋮

Eine davon wird sich gleich als ausreichend für den Angriff herausstellen.

i	$[a_0; a_1, \ldots, a_i]$	K/D	Φ	(p, q)
1	$[0; 1]$	$1/1$ $(D = 1)$		
3	$[0; 1, 1, 2]$	$3/5$	19319736024162699998	$p, q \notin \mathbb{Z}$
5	$[0; 1, 1, 2, 9, 2]$	$59/99$	$\Phi \notin \mathbb{Z}$	
7	$[0; 1, 1, 2, 9, 2, 1, 15]$	$1364/2289$	$\Phi \notin \mathbb{Z}$	
9	$[0; 1, 1, 2, 9, 2, 1, 15, 1, 1]$	$2815/4724$ (D ger.)		
11	$[0; 1, 1, 2, 9, 2, 1, 15, 1, 1, 2, 1]$	$9896/16607$	19452881335081040352	(5218623757, 3727588393)

1.3.2 Fermat-Faktorisierung

Die für RSA verwendeten Primzahlen p und q sind idealerweise von gleicher Bitlänge, um den Aufwand für das Faktorisieren zu maximieren. Allerdings heißt das nicht, dass p und q nahe beieinander liegen. Die Differenz $p - q$ zweier 1500 Bit langer Zahlen ist im Mittel eine 1498 Bit lange Zahl.[8]

Ganz im Gegenteil stellt es ein Problem dar, wenn p und q zu nahe beieinander liegen. Wir sehen uns in diesem Abschnitt die Methode der Fermat-Faktorisierung an, die in solchen Fällen sehr gut funktioniert. Der Einfachheit halber sollen nur RSA-Moduln n von der Form $n := p \cdot q$ mit $p, q \in \mathbb{P}$ faktorisiert werden.

Wenn sich Zahlen $a, b \in \mathbb{N}$ finden lassen, so dass $n = a^2 - b^2$, dann lässt sich n einfach faktorisieren, denn dann ist $n = a^2 - b^2 = (a + b) \cdot (a - b)$. Ist $a - b \neq 1$, dann sind $p = a + b$ und $q = a - b$ die Primfaktoren von n.

[8] Eine der beiden Zahlen p und q ist größer, wir nehmen hier an, dass p die größere der beiden ist.

Um passende a und b zu finden, könnte man bspw. verschiedene Werte für a und b probieren. Damit diese Suche ein wenig zielgerichtet erfolgt, überlegen wir uns Folgendes:

1. Das lineare Gleichungssystem

$$p = a + b$$
$$q = a - b$$

   lässt sich nach a und b auflösen und man erhält:

$$a = \frac{p+q}{2}$$
$$b = \frac{p-q}{2}$$

   Liegen also p und q nahe beieinander, dann ist b klein.

2. Ist $n = a^2 - b^2$, dann ist $b^2 = a^2 - n$. Damit b klein ist, muss a^2 ein wenig größer als n sein, also a ein wenig größer als $\sqrt{n}$. Wir können also mit $a := \lceil \sqrt{n} \rceil$ beginnen und dann immer größere a probieren.

Interessanterweise muss man oft nicht viele Werte für a probieren, manchmal funktioniert schon der erste, denn:

Angenommen, $p - q < 2\sqrt[4]{n}$. Dann ist

$$a - \sqrt{n} = \frac{(a-\sqrt{n})(a+\sqrt{n})}{a+\sqrt{n}} = \frac{a^2-n}{a+\sqrt{n}} =$$
$$= \frac{b^2}{a+\sqrt{n}} < \frac{b^2}{2\sqrt{n}} = \frac{(p-q)^2/4}{2\sqrt{n}} = \frac{(p-q)^2}{8\sqrt{n}} < \frac{4\sqrt{n}}{8\sqrt{n}} = 1/2.$$

Es unterscheidet sich dann also a von $\sqrt{n}$ maximal um $1/2$. In diesem Fall ist also der Startwert $a = \lceil \sqrt{n} \rceil$ bereits der richtige. Für ein 3000 Bit langes n wäre die Zahl $\sqrt[4]{n}$ etwa 750 Bit lang, der Unterschied zwischen p und q ist also riesig, im Vergleich zu p und q aber doch zu klein.

Lassen sich echte RSA-Moduln mit dieser Methode faktorisieren? Eine Antwort gibt z. B. CVE-2022-26320[9]: Für die in einer Druckerserie verwendeten TLS-Zertifikate wurden solche anfälligen RSA-Schlüssel erzeugt.

[9] https://www.cvedetails.com/cve/CVE-2022-26320/

Rückblick 1.2

In diesem Kapitel hast du dir die wichtigsten Grundlagen zum Rechnen mit Restklassen wieder ins Gedächtnis gerufen. Du weißt, wie man Euklids Algorithmus zum Berechnen eines größten gemeinsamen Teilers oder von Kehrwerten von Restklassen benutzt. Du kannst mit dem chinesischen Restsatz Restklassengleichungssysteme effizient lösen. Du weißt, wie man RSA mit dem chinesischen Restsatz beschleunigen kann. Du kannst die Performanceeinbußen aufgrund größerer Schlüssellängen für das RSA-Verfahren abschätzen. Dir ist die Gefahr von zu kurzen Private Keys bewusst und du kannst diese Gefahr anhand der Wiener-Attacke erklären. Du weißt, dass die Primfaktoren von RSA-Moduln am besten zufällig gewählt werden, u.a. damit Fermat-Faktorisierung nicht effizient funktioniert.

Kapitel 2

Auf diskreten Logarithmen basierende Verfahren

Ziele 2.1

In diesem Kapitel lernst du,

- unter welchen Bedingungen das DH-Verfahren sicher ist.
- wie sich das DH-Verfahren verallgemeinern lässt. Du lernst die Basics der dafür nötigen Theorie der Gruppen.
- unter welchen Bedingungen diskrete Logarithmen einfach zu berechnen sind.
- wie sich diskrete Logarithmen mit dem Baby-Step-Giant-Step-Algorithmus, Pohlig-Hellman-Algorithmus oder dem Index-Calculus-Algorithmus berechnen lassen.
- wie sich digitale Signaturen auf Basis des DLP erzeugen lassen und worauf dabei zu achten ist.
- wie sich Schlüssellängen verschiedener Verfahren miteinander vergleichen lassen.

2.1 Diffie-Hellman-Key-Exchange

Unter welchen Voraussetzungen (an p und g) das Diskrete-Logarithmen-Problem (DLP) und verwandte Probleme schwierig sind, wird uns noch in den nächsten Abschnitten beschäftigen. Für das Erste bleibt dieser Punkt außer Acht. Es wird angenommen, dass p und g stets so gewählt sind, dass diese Probleme schwierig sind.

Wir tasten uns vorsichtig an die Materie heran. Das erste Verfahren, das wir betrachten, ist das bereits aus den „Anwendungen der Kryptographie“ und den „Grundlagen der Kryptographie“

bekannte Diffie-Hellman Key Agreement. Mit so einem System können zwei Personen einen gemeinsamen Schlüssel für ein symmetrisches Verschlüsselungsverfahren berechnen.

Algorithmus 2.1: Diffie-Hellman-Schlüsselaustausch nach [DH76]

Setup: Alice und Bob einigen sich auf eine Primzahl p und auf ein Element g in $\mathbb{Z}_p^*$ mit der Ordnung ω. Diese Parameter (Domain Parameter) sind öffentlich.

Key Agreement:

Alice		Bob
$\alpha \overset{R}{\leftarrow} \mathbb{Z}_\omega$	$\xrightarrow{A \leftarrow g^\alpha \bmod p}$	
	$\xleftarrow{B \leftarrow g^\beta \bmod p}$	$\beta \overset{R}{\leftarrow} \mathbb{Z}_\omega$
$K \leftarrow B^\alpha \bmod p$		$K \leftarrow A^\beta \bmod p$

Dieses Verfahren ist unter dem Namen „Ephemeral Diffie Hellman Key Agreement“ (engl.: ephemeral = flüchtig, kurzlebig) bekannt.[1] Die Werte A und B werden als ephemeral Public Key von Alice bzw. Bob bezeichnet, α und β sind die ephemeral Private Keys. Die Parameter g und p heißen Domain Parameter.

Alice und Bob haben in Algorithmus 2.1 das selbe K berechnet, denn $B^\alpha = (g^\beta)^\alpha = g^{\alpha\beta} \pmod p$ und $A^\beta = (g^\alpha)^\beta = g^{\alpha\beta} \pmod p$. Eine Angreiferin Eve kennt die Domain Parameter p und g, die ephemeral Public Keys A und B, nicht aber die ephemeral Private Keys α und β. Daraus den Schlüssel $K = g^{\alpha\beta} \pmod p$ zu berechnen, ist das sogenannte Computational Diffie-Hellman Problem (CDH). Für CPA-Sicherheit ist aber noch mehr erforderlich. Im Sicherheitsspiel zur CPA-Sicherheit wird davon ausgegangen, dass der Schlüssel K zufällig ist. Dies ist hier nicht mehr der Fall, er muss ja zu den Nachrichten A und B im DH-Protokoll passen. Die Schwierigkeit des Decisional Diffie-Hellman Problem (DDH) bedeutet aber gerade, dass der Schlüssel K für Zusehende beim Protokoll (passiv Angreifende) nicht von einem zufälligen Schlüssel zu unterscheiden ist.

Im folgenden Abschnitt werden wir erkennen, dass sich die Restklassen modulo p durch andere Strukturen ersetzen lassen, für die das Berechnen diskreter Logarithmen wesentlich schwieriger ist; so schwierig, dass 256 Bit lange Schlüssel das selbe Sicherheitsniveau garantieren können wie 3072 Bit lange Primzahlen im herkömmlichen Diffie-Hellman-Verfahren. In Abschnitt 2.5.3 werden wir sehen, wie bspw. mit dem Index-Calculus-Algorithmus diskrete Logarithmen zumindest für kleinere Primzahlen berechnet werden können.

[1] vgl. [Res99].

2.2 Gruppen

Beim Diffie-Hellman-Schlüsselaustausch und den Varianten davon wurde bislang stets mit Restklassen modulo p gerechnet. Dabei werden ausschließlich die Multiplikation und iterierte Multiplikation (also Potenzieren) verwendet, Addition und Subtraktion werden nicht benötigt.

Als das zentrale Sicherheitsmoment ergeben sich in den Verfahren das DLP bzw. das DDH-Problem. In diesem Abschnitt beschäftigen wir uns mit Alternativen zu $\mathbb{Z}_p^*$, wo diese grundlegenden Operationen genauso möglich sind, womit beispielsweise Diffie-Hellman-Schlüsselaustausch ebenfalls realisiert werden kann, wo aber möglicherweise das DLP bzw. das CDH/DDH-Problem schwieriger sind als in $\mathbb{Z}_p^*$.

Beim Lesen der folgenden Definition denke man bei $\mathbb{G}$ an $\mathbb{Z}_p^*$, bei $\circ$ an die Multiplikation modulo p, bei $\widehat{\ }$ an den Kehrwert modulo p und bei $\bot$ an die Restklasse $[1]_p$.

Definition 2.2

Eine Gruppe ist ein Quadrupel $(\mathbb{G}, \circ, \bot, \widehat{\ })$, wobei $\mathbb{G}$ irgendeine Menge ist, $\circ$ eine Funktion von $\mathbb{G} \times \mathbb{G}$ nach $\mathbb{G}$, $\widehat{\ }$ eine Funktion von $\mathbb{G}$ nach $\mathbb{G}$ und $\bot \in \mathbb{G}$, und die folgenden Gesetze erfüllt sind:

1. Für alle $a, b, c \in \mathbb{G}$ gilt: $(a \circ b) \circ c = a \circ (b \circ c)$.
2. Für jedes $a \in \mathbb{G}$ gilt: $a \circ \widehat{a} = \bot$ und $\widehat{a} \circ a = \bot$.
3. Für jedes $a \in \mathbb{G}$ gilt: $\bot \circ a = a$ und $a \circ \bot = a$.

Gilt außerdem $a \circ b = b \circ a$ für alle $a, b \in \mathbb{G}$, so heißt $(\mathbb{G}, \circ, \bot, \widehat{\ })$ abelsche Gruppe. Die Funktion $\circ$ wird auch als Verknüpfung oder Gruppenoperation bezeichnet. Das Element $\widehat{a}$ heißt inverses Element von a. Das Element $\bot$ der Gruppe wird ihr neutrales Element genannt.

Wir werden uns ausschließlich mit abelschen Gruppen beschäftigen. Wenn klar ist, was mit $\circ$, $\widehat{\ }$ und $\bot$ gemeint ist, schreiben wir statt $(\mathbb{G}, \circ, \bot, \widehat{\ })$ einfach $\mathbb{G}$.

Die folgenden Beispiele sollen zeigen, dass es eine Vielzahl von Möglichkeiten gibt, wie Gruppen entstehen. Alles, was wir auf den folgenden Seiten über Gruppen lernen, wird in all diesen Fällen gleichermaßen anwendbar sein.

Beispiel 2.3

Wir kennen schon eine ganze Reihe von abelschen Gruppen.

- $(\mathbb{Z}, +, 0, -)$ ist eine abelsche Gruppe.

- $(\mathbb{R}, +, 0, -)$ ist eine abelsche Gruppe.
- $(\mathbb{Z}_n, +, 0, -)$ ist eine abelsche Gruppe, wir schreiben gern einfach $\mathbb{Z}_n$.
- $((\mathbb{Z}_n)^m, +, \vec{0}, -)$ (Vektoren mit m Koordinaten, gerechnet wird modulo n) ist eine abelsche Gruppe.
- $(\mathbb{R} \setminus \{0\}, \cdot, 1, ^{-1})$ ist eine abelsche Gruppe.
- $(\mathbb{Z}_p^*, \cdot, 1, ^{-1})$ ist eine abelsche Gruppe, wir schreiben dafür gerne $\mathbb{Z}_p^*$.
- $(\mathbb{Z}_n^*, \cdot, 1, ^{-1})$ (vgl. Definition 1.3) ist eine abelsche Gruppe, wir schreiben dafür gerne $\mathbb{Z}_n^*$. Diese Gruppe ist im Zusammenhang mit RSA von großem Interesse.
- Sind $p \in \mathbb{P}$ und $n \in \mathbb{N}$, so ist $(\mathrm{GF}(p^n) \setminus \{0\}, \cdot, 1, ^{-1})$ eine abelsche Gruppe, genannt $\mathrm{GF}(p^n)^*$. Diese Gruppen werden in Kapitel 5 auftauchen.
- Elliptische Kurven sind Gruppen, diese werden in Kapitel 3 auftauchen.

Auch nicht-abelsche Gruppen kennen wir.

- Es sei M die Menge aller reellen invertierbaren 2×2-Matrizen. Dann ist

$$(M, \cdot, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, ^{-1})$$

  eine nicht abelsche Gruppe.

In Definition 2.2 wird verlangt, dass man in einer Gruppe Elemente miteinander verknüpfen kann, dass so eine Verküpfung wieder rückgängig gemacht werden kann (durch Invertieren) und dass es ein neutrales Element gibt.

Etwas sonderbar mutet zunächst die Forderung 1 (Assoziativitätsgesetz) an. Einer der Gründe, warum diese Eigenschaft so wichtig ist, dass man sie von jeder Gruppe fordert, ist, dass nur mit dieser Eigenschaft Square and Multiply funktioniert. Da erwartet man sich beispielsweise, dass sich $x^4 = ((x \circ x) \circ x) \circ x$ auch als $(x^2)^2 = (x \circ x) \circ (x \circ x)$ berechnen lässt (was sich mit einer Verknüpfung weniger erledigen lässt). Dies klappt, denn

$$x^4 = \underbrace{((x \circ x) \circ x) \circ x}_{(a \circ b) \circ c} = \underbrace{(x \circ x) \circ (x \circ x)}_{a \circ (b \circ c)} = \left(x^2\right)^2$$

Dieser Trick lässt sich jedoch nur anwenden, wenn die Klammern auch anders gesetzt werden dürfen, ohne das Ergebnis zu beeinflussen. Umgekehrt reicht diese Eigenschaft aber aus, dass alle möglichen Potenzen beliebig berechnet werden können.

In der LVA „Grundlagen der Kryptographie“ ist der Begriff der Ordnung einer Restklasse modulo p aufgetaucht. Das war damals sozusagen nur der Spezialfall der Gruppe $\mathbb{Z}_p^*$. Jetzt können

wir diesen Begriff ganz allgemein für Gruppen benutzen.

Definition 2.4

Besitzt eine Gruppe unendlich viele Elemente (wie z.B. $(\mathbb{Z}, +, 0, -)$), so nennen wir sie eine unendliche Gruppe. Andernfalls heißt die Gruppe endlich. Ist $(\mathbb{G}, \circ, \bot, \widehat{\ })$ eine endliche Gruppe, so bezeichnen wir mit $|\mathbb{G}|$ die Anzahl der Elemente der Gruppe und nennen $|\mathbb{G}|$ die Ordnung von $\mathbb{G}$.

Ist g ein Element der Gruppe $(\mathbb{G}, \circ, \bot, \widehat{\ })$ und $\alpha \in \mathbb{N}$, so schreiben wir

$$g^{\alpha} \text{ statt } \underbrace{g \circ g \circ \cdots \circ g}_{\alpha \text{ mal}},$$
$$g^{-\alpha} \text{ statt } \underbrace{\widehat{g} \circ \widehat{g} \circ \cdots \circ \widehat{g}}_{\alpha \text{ mal}} \quad (\text{insbes. } \widehat{g} = g^{-1}) \text{ und definieren}$$
$$g^0 := \bot.$$

Die kleinste natürliche Zahl ω, so dass $g^{\omega} = \bot$, heißt (sofern es so eine Zahl gibt) die Ordnung von g und wird mit $\mathrm{ord}(g)$ bezeichnet. Gibt es keine solche Zahl, so sagen wir, die Ordnung von g ist unendlich.

Satz 2.5

Ist $(\mathbb{G}, \circ, \bot, \widehat{\ })$ eine Gruppe und $g \in \mathbb{G}$ und sind $\alpha, \beta \in \mathbb{Z}$, dann gelten die bekannten Potenzrechenregeln

$$\widehat{g^{\alpha}} = \widehat{g}^{\alpha} = g^{-\alpha}$$
$$g^{\alpha} \circ g^{\beta} = g^{\alpha+\beta}$$
$$(g^{\alpha})^{\beta} = g^{\alpha \cdot \beta}$$

Ist g ein Element der Ordnung ω, so gilt darüber hinaus

$$g^{\alpha} = g^{\alpha \bmod \omega}.$$

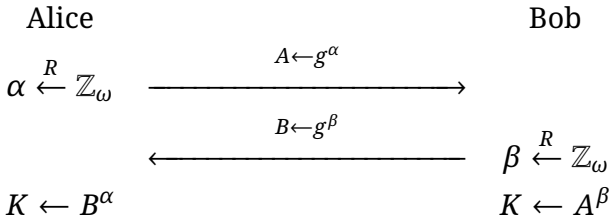
2.3 Diffie-Hellman mit Gruppen

Ein DH-Schlüsselaustausch lässt sich mit jeder Gruppe anstatt mit $\mathbb{Z}_p^*$ durchführen.

Algorithmus 2.6: Diffie-Hellman-Schlüsselaustausch in einer Gruppe $\mathbb{G}$

Setup: Alice und Bob einigen sich auf eine Gruppe $\mathbb{G}$ und auf ein Element $g \in \mathbb{G}$ mit der Ordnung ω. Diese Parameter (Domain Parameter) sind öffentlich.

Key Agreement:



Das Diffie-Hellman-Problem und das diskrete Logarithmusproblem können ganz allgemein für Gruppen definiert werden. Damit ergibt sich eine größere Auswahl an Parametern für kryptographische Verfahren. Besonders in Kapitel 3 werden wir diesen Aspekt vertiefen.

Definition 2.7: DLP, CDH, DDH

Es seien $\mathbb{G}$ eine Gruppe, $g \in \mathbb{G}$ ein Element der Ordnung ω und $\alpha, \beta \in \mathbb{Z}_\omega$. Weiterhin seien $A := g^\alpha$ und $B := g^\beta$.
Das diskrete Logarithmusproblem (DLP) lautet:

> **Gegeben:** $\mathbb{G}, g, A$
> **Gesucht:** α

Die Zahl α wird diskreter Logarithmus von A zur Basis g (in $\mathbb{G}$) genannt.
Das Computational Diffie-Hellman Problem (CDH-Problem) lautet:

> **Gegeben:** $\mathbb{G}, g, A, B$
> **Gesucht:** $g^{\alpha\beta}$.

Das Decisional Diffie-Hellman Problem (DDH-Problem) lautet:
Es seien $g_0 := g^{\alpha\beta}$, $g_1 \xleftarrow{R} \mathbb{G}$ und $b \xleftarrow{R} \{0, 1\}$.

> **Gegeben:** $\mathbb{G}, g, A, B, g_b$
> **Gesucht:** b.

Ganz klar, wer das DLP löst, kann auch das CDH-Problem lösen, wer das CDH-Problem löst, kann das DDH-Problem lösen. Ob das jeweils auch umgekehrt so ist, weiß niemand. Alle drei Probleme gelten bislang – wie das Faktorisierungsproblem – als praktisch nicht lösbar (zumindest für bestimmte Gruppen). Die Sicherheit eines Großteils der Verfahren in diesem Kapitel beruht

auf dem DDH-Problem.

Beispiel 2.8

1. Wählt man als Gruppe in Algorithmus 2.6 $(\mathbb{Z}_p^*, \cdot, 1, ^{-1})$, so erhält man das bekannte DH-Verfahren. Die DL-, CDH- und DDH-Probleme gelten – zumindest für geeignete Parameter – als praktisch nicht lösbar. Damit das DLP jedoch ausreichend schwierig ist, müssen sehr große Primzahlen p verwendet werden. Außerdem werden auch Elemente g mit sehr großer Ordnung ω benötigt. In Abschnitt 2.5 beschäftigen wir uns noch eingehender mit der Berechnung von diskreten Logarithmen.

2. Wählt man als Gruppe $(\mathbb{Z}_n, +, 0, -)$, so lassen sich diskrete Logarithmen ganz einfach berechnen. In diesem Fall lautet das Problem:

   Gegeben: $n, g, A = \underbrace{(g + \cdots + g)}_{\alpha\text{-mal}} \bmod n$

   Gesucht: α

   Offenbar ist $A = \alpha \cdot g \bmod n$. Mit dem erweiterten Euklidschen Algorithmus lässt sich der Kehrwert von g modulo n berechnen und damit $\alpha = A \cdot g^{-1} \bmod n$.

   Wie schwierig das DLP ist, hängt also maßgeblich von der gewählten Gruppe ab.

3. Weitere interessante Gruppen – die elliptischen Kurven – werden wir in Kapitel 3 kennenlernen. Für diese Gruppen stellt sich das DLP als noch etwas schwieriger heraus als für $\mathbb{Z}_p^*$. Als Konsequenz können dort bei gleicher Sicherheit kleinere Parameter verwendet werden, was die Verfahren bedeutend performanter macht.

2.4 Mehr zur Ordnung

Es sei wiederum $\mathbb{G}$ eine Gruppe. Sind $g, h, h' \in \mathbb{G}$ und ist $g \circ h = g \circ h'$, so ist $h = h'$; dies ist einfach zu erkennen, man braucht nur die erste Gleichung (links und rechts) mit $\widehat{g}$ zu verknüpfen.

$$\begin{aligned} g \circ h &= g \circ h' \\ \underbrace{\widehat{g} \circ g}_{=\perp} \circ h &= \underbrace{\widehat{g} \circ g}_{=\perp} \circ h' \\ h &= h' \end{aligned}$$

Dies bedeutet aber, dass aus zwei verschiedenen Gruppenelementen h, h' durch Verknüpfung mit g wiederum verschiedene Gruppenelemente werden.

Ist $\mathbb{G}$ eine endliche abelsche Gruppe mit den Elementen $g_1, g_2, \ldots, g_n$ und ist $g \in \mathbb{G}$, so sind auch $g \circ g_1, \ldots, g \circ g_n$ verschiedene Elemente – und damit alle Elemente – der Gruppe $\mathbb{G}$. Betrachtet

man nun das Ergebnis der Verknüpfung aller Elemente von $\mathbb{G}$, so erhält man

$$g_1 \circ g_2 \circ \cdots \circ g_n = (g \circ g_1) \circ (g \circ g_2) \circ \cdots \circ (g \circ g_n),$$
$$\cancel{g_1 \circ g_2 \circ \ldots \circ g_n} = g^n \circ \cancel{(g_1 \circ g_2 \circ \ldots \circ g_n)}.$$

Verknüpfen mit $(g_1 \circ g_2 \circ \cdots \circ g_n)^{-1}$ ergibt

$$\bot = g^n.$$

Dieses Resultat gilt allgemeiner nicht nur in abelschen Gruppen.

Satz 2.9

Ist $\mathbb{G}$ eine endliche Gruppe, dann gilt für jedes Element $g \in \mathbb{G}$

$$g^{|\mathbb{G}|} = \bot.$$

Bemerkung 2.10

Für die Gruppen $\mathbb{Z}_n^*$ ist Satz 2.9 gerade der Satz von Fermat (Satz 1.10), wenn $n \in \mathbb{P}$ bzw. der Satz von Euler und Lagrange (Satz 1.11) für allgemeine $n \in \mathbb{N}$. (Hinweis: Vergleiche den einleitenden Beweis mit jenem für den Satz von Fermat (Satz 1.10)).

Satz 2.11

Es seien $\mathbb{G}$ eine endliche Gruppe, $g \in \mathbb{G}$ ein Element der Ordnung ω und $\alpha \in \mathbb{N}$. Dann gilt:

1. Die Elemente $g, g^2, \ldots, g^\omega$ sind alle verschieden.
2. Ist $g^\alpha = \bot$, dann gilt $\omega \mid \alpha$, und umgekehrt.
3. Die Ordnung ω von g ist ein Teiler von $|\mathbb{G}|$.

Beweis. Es seien $\mathbb{G}$ eine Gruppe und $g \in \mathbb{G}$ ein Element der Ordnung ω.

1. Angenommen zwei Potenzen g^x und g^y wären gleich und dabei wäre $0 < x < y \leq \omega$. Dann wäre

$$g^y = g^x$$
$$g^y \circ g^{-x} = g^x \circ g^{-x}$$
$$g^{y-x} = \bot.$$

Es wäre dann $y - x > 0$ ein Exponent, der kleiner ist als ω, der das neutrale Element ergibt. Die Potenzen g^x und g^y müssen also verschieden sein.

2. Weiterhin sei $\alpha \in \mathbb{N}$ eine positive ganze Zahl, für die $g^\alpha = \perp$ gilt. Seien nun q und r Quotient und Rest bei Division von α durch ω. Dann ist $\alpha = q \cdot \omega + r$ und $r < \omega$. Es gilt

$$\perp = g^\alpha = g^{q\cdot\omega+r} = g^{q\cdot\omega} \circ g^r = (g^\omega)^q \circ g^r. \tag{2.1}$$

Da ω die Ordnung von g ist, ist $g^\omega = \perp$, und somit lässt sich Gleichung (2.1) vereinfachen zu $\perp = g^r$. Da $r < \omega$ ist, kann r aber nur 0 sein, denn ω ist ja per definitionem die kleinste positive Zahl, für die $g^\omega = \perp$ gilt. Der Rest r bei der Division von α durch ω ist also stets gleich 0.

Umgekehrt: Ist $\omega \mid \alpha$, dann gibt es ein $v \in \mathbb{Z}$, so dass $\alpha = v \cdot \omega$. Dann ist $g^\alpha = g^{v\cdot\omega} = (g^\omega)^v = \perp^v = \perp$.

3. Folgt aus Satz 2.9 und 2.

□

Es kann passieren, dass die Ordnung eines Elements g einer Gruppe $\mathbb{G}$ gleich der Ordnung n der Gruppe ist. In diesem Fall sind die Potenzen $g, g^2, \ldots, g^n$ alle verschieden und somit genau die n verschiedenen Elemente von $\mathbb{G}$. Aus gutem Grund heißt g dann erzeugendes Element von $\mathbb{G}$. Die Gruppe $\mathbb{G}$ heißt in diesem Fall zyklische Gruppe.

Um zu überprüfen, ob ein Element g einer Gruppe $\mathbb{G}$ die Ordnung ω hat, ist folgender Satz nützlich, den wir (für die Gruppen $\mathbb{Z}_p^*$) schon aus der LVA „Grundlagen der Krytographie“ kennen.

Satz 2.12

Es sei $\mathbb{G}$ eine Gruppe und $g \in \mathbb{G}$. Weiterhin sei $\omega \in \mathbb{N}$. Dann gilt:

1. Ist $g^\omega \neq \perp$, dann ist $\mathrm{ord}(g) \neq \omega$.
2. Gibt es einen Primfaktor p von ω, so dass $g^{\omega/p} = \perp$, dann ist $\mathrm{ord}(g) \neq \omega$ (sondern maximal ω/p).
3. Andernfalls (wenn 1. und 2. nicht zutreffen) ist $\mathrm{ord}(g) = \omega$.

Ist die Ordnung eines Elements bekannt, so lassen sich die Ordnungen von Potenzen dieses Elements bestimmen.

Satz 2.13

Ist $g \in \mathbb{G}$ ein Element der Ordnung ω und ist $\alpha \in \mathbb{N}$, dann ist die Ordnung von g^α gleich $\frac{\omega}{\text{ggT}(\alpha,\omega)}$.

Beweis. Es sei ω die Ordnung von g, $\alpha \in \mathbb{N}$ und τ die Ordnung von g^α, also die kleinste positive ganze Zahl, so dass $(g^\alpha)^\tau = \perp$ gilt. Da $\perp = (g^\alpha)^\tau = g^{\alpha\cdot\tau}$ muss $\alpha \cdot \tau$ ein Vielfaches von ω sein. Klarerweise ist $\alpha\cdot\tau$ auch ein Vielfaches von α. Da τ möglichst klein sein soll, muss $\alpha\cdot\tau$ das kleinste gemeinsame Vielfache von α und ω sein, welches sich bekanntermaßen als $\frac{\alpha\cdot\omega}{\text{ggT}(\alpha,\omega)}$ berechnet.[2] Daher ist

$$\alpha \cdot \tau = \frac{\alpha \cdot \omega}{\text{ggT}(\alpha, \omega)} \quad \text{und somit}$$
$$\tau = \frac{\omega}{\text{ggT}(\alpha, \omega)}.$$

□

Je größer die Ordnung eines Elements, desto schwieriger wird es, diskrete Logarithmen bzgl. dieser Basis zu berechnen. Offenbar kann die Ordnung eines Elements nicht größer sein als die Ordnung der Gruppe. In vielen Gruppen gibt es aber (viele) Elemente mit der maximal möglichen Ordnung, zu diesen gehören u.a. die Gruppen $(\mathbb{Z}_p^*, \cdot, [1]_p, ^{-1})$, sie sind zyklische Gruppen.

Satz 2.14

Es sei $p \in \mathbb{P}$. Es sei d ein positiver Teiler von $p-1$. Dann gibt es genau $\varphi(d)$ Elemente der Ordnung d in $\mathbb{Z}_p^*$. Insbesondere gibt es genau $\varphi(p-1)$ erzeugende Elemente in $\mathbb{Z}_p^*$.

2.5 Verfahren zur Berechnung diskreter Logarithmen

2.5.1 Baby Step Giant Step

Das erste Verfahren zur Berechnung diskreter Logarithmen funktioniert ganz allgemein für jede Gruppe.

Es seien $\mathbb{G}$ eine Gruppe, $g \in \mathbb{G}$ ein Element der Ordnung ω, $\alpha \in \mathbb{Z}_\omega$ und $A := g^\alpha$. Diskrete Logarithmen lassen sich durch Probieren bestimmen. Beim einfachen Durchprobieren würden der Reihe nach die Potenzen $g^1, g^2, g^3, \ldots$ berechnet, bis das Ergebnis A auftaucht. Dies geschieht im Mittel nach $\omega/2$ Versuchen. Schneller ist der Baby-Step-Giant-Step-Algorithmus, der durchschnittlich $1{,}5 \cdot \sqrt{\omega}$ Gruppenoperationen braucht.

[2] Für alle ganzen Zahlen a und b gilt $a \cdot b = \text{ggT}(a, b) \cdot \text{kgV}(a, b)$. Ein Beweis dafür ist in Anhang A.3 zu finden.

Das folgende (viel zu kleine) Beispiel soll die Idee für diesen Algorithmus illustrieren.

Beispiel 2.15

Es sei $p := 389$. Dann ist $g := 5$ ein Element der Ordnung $\omega = 97$ in $\mathbb{Z}_p^*$.

$g^0 = 1$	$g^1 = 5$	$g^2 = 25$	$g^3 = 125$	$g^4 = 236$	…	$g^9 = 345$
$g^{10} = 169$	$g^{11} = 67$	$g^{12} = 335$	$g^{13} = 119$	$g^{14} = 206$	…	$g^{19} = 344$
$g^{20} = 164$	$g^{21} = 42$	$g^{22} = 210$	$g^{23} = 272$	$g^{24} = 193$	…	$g^{29} = 175$
$g^{30} = 97$	$g^{31} = 96$	$g^{32} = 91$	$g^{33} = 66$	$g^{34} = 330$		⋮
⋮					⋱	⋮
$g^{90} = 79$	…	…	…	…	…	$g^{99} = 25$

Die Ordnung ω ist kleiner als $10 \cdot 10$, alle verschiedenen Potenzen von g sind demnach in dieser Tabelle enthalten. Um den diskreten Logarithmus von 42 in $\mathbb{Z}_p^*$ zur Basis g zu berechnen, muss diese Zahl (42) nun in diesem Quadrat gesucht werden. Sie findet sich in Zeile 2 (der dritten Zeile) und Spalte 1 (der zweiten Spalte). Daher ist der dort verwendete Exponent (10·2+1). In der Folge wird nun gezeigt, dass es möglich ist, die Zeile und Spalte des gesuchten Elements (und damit den Exponenten) zu bestimmen, ohne alle Potenzen zu berechnen.

Wähle $N := \lfloor \sqrt{\omega} \rfloor + 1$. Dann ist $N^2 > \omega$. Seien q und r nun Quotient und Rest der Division α / N, also $\alpha = qN + r$. Dann ist $r < N$ und $q < N$. Nun sollen q und r gefunden werden, q entspricht der gesuchten Zeile und r der Spalte in der Tabelle aller Potenzen von g.

Man beachte, dass nun

$$\begin{aligned} A = g^{\alpha} &= g^{qN+r} \\ A &= \left(g^N\right)^q \cdot g^r \\ A\left(g^{-N}\right)^q &= g^r. \end{aligned}$$

Nun berechnet man alle Potenzen auf der rechten Seite:

$$\mathcal{B} := \{(g^r, r) \mid r \in \{0, 1, \ldots, N-1\}\}.$$

Diese Potenzen heißen Baby Steps, denn hier geht es in kleinen Schritten immer um den Faktor g weiter. Insbesondere lässt sich g^{r+1} als $g^r \cdot g$ mit nur einer Multiplikation aus g^r berechnen. Die Baby Steps sind genau die Werte in der ersten Zeile der Tabelle aller Potenzen.

Sodann prüft man für jedes $q \in \{0, 1, \ldots, N-1\}$, ob für ein $r \in \{0, 1, \ldots, N-1\}$ das Paar $(A(g^{-N})^q, r)$ in $\mathcal{B}$ vorkommt. Am besten berechnet man zunächst $f := g^{-N}$ und dann Af^q. Auch hier erhält man Af^{q+1} aus $Af^q \cdot f$ mit nur einer Multiplikation mit f. Für wachsendes q steigt hier das Ergebnis um den Faktor $f = g^{-N}$, darum spricht man hier von Giant Steps (ein Giant Step

entspricht N Baby Steps). Die Giant Steps sind nun genau die Kehrwerte der ersten Spalte in der Tabelle aller Potenzen. Ist ein passendes Paar (r, q) gefunden, so gilt

$$\alpha = Nq + r.$$

Somit lässt sich α aus q und r berechnen. Dabei braucht man N Multiplikationen, um die Baby Steps $\mathcal{B}$ zu erstellen, und im Mittel $N/2$ Multiplikationen, bis der Index q gefunden ist, in Summe also $1{,}5 \cdot N$ Multiplikationen.

Bei einer Ordnung von $\approx 2^{128}$ ergibt sich ein Aufwand von 2^{127} Multiplikationen beim einfachen Durchprobieren; das ist praktisch nicht schaffbar. Der Baby-Step-Giant-Step-Algorithmus benötigt nur $1{,}5 \cdot 2^{64} < 2^{65}$ Multiplikationen; das ist hingegen berechenbar.[3]

Umgekehrt wird auch immer ein Paar (q, r) gefunden, denn jedes $\alpha < \omega$ besitzt eine eindeutige Darstellung $\alpha = Nq + r$ (Division mit Rest), wo $0 \leq r < N$. Klarerweise ist auch $q < N$, denn sonst wäre $\alpha \geq qN > N^2 > \omega$.

Praktisch stößt man auch auf das Problem, dass die Liste $\mathcal{B}$ der Baby Steps gespeichert werden muss. Bereits bei 2^{60} Einträgen mit je 16 Byte wird diese Liste 16 EB groß. Diese Datenmenge zu speichern – und dann effizient zu durchsuchen – ist schwieriger als die entsprechende Anzahl an Multiplikationen durchzuführen. Time Memory Trade-off kann dieses Ungleichgewicht ein wenig korrigieren: Wählt man N kleiner, so müssen nicht so viele Baby Steps berechnet und gespeichert werden. Dafür muss womöglich länger nach q gesucht werden, denn q kann sich nun im Bereich $\{0, 1, \ldots, \omega/N\}$ bewegen. Je kleiner N ist, desto größer ist ω/N.

2.5.2 Pohlig-Hellman

In diesem Abschnitt soll ein Beispiel die Idee des Pohlig-Hellman-Algorithmus zur Bestimmung von diskreten Logarithmen illustrieren.

Dieses Verfahren zeigt, dass für die Schwierigkeit des DLP nur der größte Primfaktor der Ordnung eines Elements ausschlaggebend ist. Aus diesem Grund werden in DLP-basierten Verfahren bevorzugt Elemente mit großer primer Ordnung verwendet.

Angenommen, das Element g der Gruppe $\mathbb{G}$ hat die Ordnung ω und $\omega = m \cdot n$, wobei $\mathrm{ggT}(m, n) = 1$. Weiterhin sei $A := g^\alpha$. Wir nehmen an, dass g und seine Ordnung ω bekannt sind, ebenfalls bekannt ist A. Gesucht wird der diskrete Logarithmus von A zur Basis g, also der Exponent α.

[3] Aus diesem Grund muss die Bitlänge der Ordnung immer zumindest das Doppelte des gewünschten Sicherheitsparameters betragen, unabhängig davon, mit welcher Gruppe gearbeitet wird. Eine Situation ganz ähnlich wie bei Hashfunktionen (dort wegen dem Geburtstagsparadoxon). Im Unterschied zur Situation bei Hashfunktionen muss aber beachtet werden, dass diskrete Logarithmen sich in vielen Gruppen effizienter als mit dem Baby-Step-Giant-Step-Algorithmus berechnen lassen; ein Beispiel (unter vielen) ist der Index-Calculus-Algorithmus für die Gruppen $\mathbb{Z}_p^*$, der in Abschnitt 2.5.3 vorgestellt wird.

Zunächst ist wegen Satz 2.13

$$\begin{aligned}\operatorname{ord}(g^m) &= \frac{\omega}{\operatorname{ggT}(m,\omega)} = \frac{mn}{m} = n < \omega \text{ und}\\ \operatorname{ord}(g^n) &= \frac{\omega}{\operatorname{ggT}(n,\omega)} = \frac{mn}{n} = m < \omega.\end{aligned}$$

Da $g^\alpha = A$, ergibt sich

$$(g^\alpha)^m = A^m \text{ und } (g^\alpha)^n = A^n$$

und weiter

$$(g^m)^\alpha = A^m \text{ und } (g^n)^\alpha = A^n.$$

Der Exponent α in der linken Gleichung lässt sich einfacher finden, denn $\operatorname{ord}(g^m)$ ist kleiner als $\operatorname{ord}(g)$, dasselbe gilt für die rechte Gleichung. Allerdings können wir α nur modulo der jeweiligen Ordnung bestimmen. Aus der linken Gleichung erhalten wir $\alpha \bmod n$, aus der rechten $\alpha \bmod m$. Da aber $\operatorname{ggT}(m,n) = 1$ ist, lässt sich daraus mit dem chinesischen Restsatz $\alpha \bmod m \cdot n$, also $\alpha \bmod \omega$ berechnen.

Beispiel 2.16

Das Element $g = 10$ in $\mathbb{Z}_{71}^*$ hat die Ordnung $\omega = 35$. In diesem Fall können wir $m = 5$ und $n = 7$ wählen. Wir suchen den diskreten Logarithmus α von $A = 38$ zur Basis g.

Anstatt der ursprünglichen Gleichung

$$10^\alpha = 38 \pmod{71} \qquad (g^\alpha = A)$$

erhalten wir mit

$$\begin{aligned}g^m &= 10^5 \bmod 71 = 32,\\ A^m &= 38^5 \bmod 71 = 20,\\ g^n &= 10^7 \bmod 71 = 5 \quad \text{und}\\ A^n &= 38^7 \bmod 71 = 54\end{aligned}$$

die neuen Gleichungen

$$\begin{aligned}32^\alpha &= 20 \pmod{71} \qquad ((g^m)^\alpha = A^m) \text{ und}\\ 5^\alpha &= 54 \pmod{71} \qquad ((g^n)^\alpha = A^n).\end{aligned}$$

Die Ordnung von 32 ist 7, die Ordnung von 5 ist 5. Aus der ersten Gleichung ergibt sich (mit Baby-Step-Giant-Step oder einfachem Durchprobieren) $\alpha = 6 \pmod 7$. Aus der zweiten Gleichung ergibt sich $\alpha = 3 \pmod 5$. Mit dem chinesischen Restsatz erhalten wir daraus $\alpha = 13 \pmod{35}$.

Eine einfache Möglichkeit, zu verhindern, dass die Pohlig-Hellman-Strategie erfolgreich ist, ist sicherzustellen, dass die Ordnung von g eine Primzahl ist. Dann kann diese nicht in ein Produkt kleinerer Zahlen zerlegt werden. In vielen Verfahren wird deshalb bei der Erzeugung der Domain Parameter gefordert, dass für das Element g ein Element mit großer primer Ordnung ω gewählt wird.[4]

2.5.3 Der Index-Calculus-Algorithmus

Wir sehen uns abschließend eine Methode an, um diskrete Logarithmen in $\mathbb{Z}_p^*$ zu berechnen, die sich zu Nutze macht, dass es in $\mathbb{Z}$ eine eindeutige Primfaktorzerlegung gibt: den Index-Calculus-Algorithmus. Diese Methode funktioniert – anders als die bisherigen – nicht für alle Gruppen, sondern nur für die Gruppen $\mathbb{Z}_p^*$. Sie ist aber wesentlich schneller als die bisher beschriebenen Verfahren, zeigt also, dass das DLP möglicherweise in anderen Gruppen schwieriger sein könnte als in den Gruppen $\mathbb{Z}_p^*$.

Es seien $p \in \mathbb{P}$ und $g \in \mathbb{Z}_p^*$ ein Element mit primer Ordnung ω. Wir möchten diskrete Logarithmen zur Basis g modulo p berechnen.

Zunächst wählen wir eine Menge $\mathcal{F}$ von Primzahlen, die sogenannte Faktorbasis. Diese Primzahlen müssen Potenzen von g sein. Dies lässt sich leicht überprüfen, indem man prüft, ob ihre Ordnung gleich ω ist.

Eine Zahl z nennen wir $\mathcal{F}$-glatt, falls alle Primfaktoren von z in $\mathcal{F}$ liegen.

Wir bestimmen zunächst[5] die diskreten Logarithmen aller Primzahlen in der Faktorbasis, also für jedes $q \in \mathcal{F}$ ein α_q mit $g^{\alpha_q} \bmod p = q$. Um dann den diskreten Logarithmus α von $A \in \mathbb{Z}_p^*$ zur Basis g zu berechnen, suchen wir einen Exponenten γ, so dass $A \cdot g^\gamma$ $\mathcal{F}$-glatt ist. Dann ist

$$g^\alpha \cdot g^\gamma = \prod_{q \in \mathcal{F}} q^{e_q} = \prod_{q \in \mathcal{F}} g^{\alpha_q \cdot e_q} \pmod p,$$
$$g^{\alpha+\gamma} = g^{\sum_{q \in \mathcal{F}} \alpha_q \cdot e_q} \pmod p.$$

[4] In dem Fall, dass ω eine Potenz p^s einer Primzahl ist, ist eine Zerlegung $\omega = m \cdot n$ mit $\mathrm{ggT}(m, n) = 1$ nicht möglich. Für diesen Fall fanden Pohlig und Hellman aber ebenfalls eine Methode, das Problem von p^s auf p zu reduzieren. Details finden sich in [PH78]. Es wird hier nicht näher auf diesen Fall eingegangen.

[5] Das wird gleich ein paar Zeilen weiter unten erledigt.

Die Basis der Potenzen links und rechts sind gleich, es reicht also die Exponenten (modulo der Ordnung von g) zu vergleichen.

$$\alpha + \gamma = \sum_{q \in \mathcal{F}} \alpha_q \cdot e_q \pmod{\omega}$$
$$\alpha = -\gamma + \sum_{q \in \mathcal{F}} \alpha_q \cdot e_q \pmod{\omega}.$$

Benötigt werden also: Die diskreten Logarithmen α_q und ein $\mathcal{F}$-glattes $A \cdot g^\gamma$ samt dessen Primfaktorzerlegung. Die Primfaktorzerlegung einer $\mathcal{F}$-glatten Zahl ist einfach, denn es können ja nur Primfaktoren aus $\mathcal{F}$ vorkommen.

Zurück zur Bestimmung der diskreten Logarithmen der Faktorbasiselemente. Dazu berechnet man g^ζ für viele Zahlen ζ und merkt sich diejenigen ζ, für die $g^\zeta \bmod p$ $\mathcal{F}$-glatt ist. Hat man viele solche Zahlen gefunden, kann man die diskreten Logarithmen der Faktorbasiselemente berechnen.

Beispiel 2.17

Dieses Beispiel illustriert die Methode für $p = 2027$, $g = 1000$ und die Faktorbasis $\mathcal{F} = \{3, 13, 17, 19, 31\}$. Die Ordnung von g ist $\omega = 1013$. Mit etwas Probieren finden wir z. B.

ζ	g^ζ	in $\mathcal{F}$
738	171	$3^2 \cdot 19$
474	1581	$3 \cdot 17 \cdot 31$
666	867	$3 \cdot 17^2$
336	1521	$3^2 \cdot 13^2$
168	39	$3 \cdot 13$
265	1989	$3^2 \cdot 13 \cdot 17$

Wir bestimmen nun Zahlen $\alpha_3, \alpha_{13}, \alpha_{17}, \alpha_{19}, \alpha_{31}$, so dass

$$3 = g^{\alpha_3},\ 13 = g^{\alpha_{13}},\ 17 = g^{\alpha_{17}}, 19 = g^{\alpha_{19}}, 31 = g^{\alpha_{31}} \pmod{2027}.$$

Die Primfaktorzerlegungen von oben ergeben das Gleichungssystem (modulo 2027)

$$g^{738} = 3^2 \cdot 19 = (g^{\alpha_3})^2 \cdot (g^{\alpha_{19}}) = g^{2\alpha_3 + \alpha_{19}}$$
$$g^{474} = 3 \cdot 17 \cdot 31 = (g^{\alpha_3}) \cdot (g^{\alpha_{17}}) \cdot (g^{\alpha_{31}}) = g^{\alpha_3 + \alpha_{17} + \alpha_{31}}$$
$$g^{666} = 3 \cdot 17^2 = (g^{\alpha_3}) \cdot (g^{\alpha_{17}})^2 = g^{\alpha_3 + 2\alpha_{17}}$$
$$g^{336} = 3^2 \cdot 13^2 = (g^{\alpha_3})^2 \cdot (g^{\alpha_{13}})^2 = g^{2\alpha_3 + 2\alpha_{13}}$$
$$g^{168} = 3 \cdot 13 = (g^{\alpha_3}) \cdot (g^{\alpha_{13}}) = g^{\alpha_3 + \alpha_{13}}$$
$$g^{265} = 3^2 \cdot 13 \cdot 17 = (g^{\alpha_3})^2 \cdot (g^{\alpha_{13}}) \cdot (g^{\alpha_{17}}) = g^{\alpha_3 + \alpha_{13} + \alpha_{17}}$$

Wir betrachten die Exponenten und es ergibt sich ein lineares Gleichungssystem (modulo $\omega = 1013$):

$$
\begin{aligned}
738 &= 2\alpha_3 + \alpha_{19} \\
474 &= \alpha_3 + \alpha_{17} + \alpha_{31} \\
666 &= \alpha_3 + 2\alpha_{17} \\
336 &= 2\alpha_3 + 2\alpha_{13} \\
168 &= \alpha_3 + \alpha_{13} \\
265 &= \alpha_3 + \alpha_{13} + \alpha_{17}
\end{aligned}
$$

Das Gleichungssystem lässt sich modulo 1013 genau so lösen wie über $\mathbb{R}$, denn auch $\mathbb{Z}_{1013}$ ist ein Körper. Das Modul `galois` erlaubt das Rechnen mit entsprechenden Matrizen.

```
> from galois import GF
> k = GF(1013)
> m = k( [[2,0,0,1,0,738],
          [1,0,1,0,1,474],
          [1,0,2,0,0,666],
          [2,2,0,0,0,336],
          [1,1,0,0,0,168],
          [2,1,1,0,0,265]])
> r = m.row_reduce(); r
GF([[  1,   0,   0,   0,   0, 541],
    [  0,   1,   0,   0,   0, 640],
    [  0,   0,   1,   0,   0, 569],
    [  0,   0,   0,   1,   0, 669],
    [  0,   0,   0,   0,   1, 377],
    [  0,   0,   0,   0,   0,   0]], order=1013)
> (alpha3,alpha13,alpha17,alpha19,alpha31) = r[:-1,-1].tolist()
> pow( 1000, alpha19, 2027 )
19
```

Nun können wir beliebige diskrete Logarithmen berechnen. Um den diskreten Logarithmus von 1469 zur Basis g zu berechnen, suchen wir (durch Probieren) nach einer Zahl γ, so

dass $A \cdot g^y$ $\mathcal{F}$-glatt ist. Wir finden (modulo 2027)

$$\begin{aligned} 1469 \cdot 1000^{696} &= 3^2 \cdot 19 \\ 1469 \cdot 1000^{696} &= 1000^{2 \cdot 541 + 669} \\ 1469 &= 1000^{1055} = 1000^{42} \end{aligned}$$

Der diskrete Logarithmus von 1469 zur Basis 1000 modulo 2027 ist also 42.

Dass es mit dem Index-Calculus-Algorithmus eine Methode gibt, diskrete Logarithmen modulo p zu berechnen, äußert sich nicht zuletzt in den Schlüssellängen, also in der Größe der Primzahl p. Reines Durchprobieren ließe sich mit Primzahlen von 128 Bit Länge bereits erfolgreich verhindern. Geschicktes Probieren (Baby-Step-Giant-Step) lässt sich mit 256 Bit langen Primzahlen verhindern. Tatsächlich empfiehlt sich heute die Verwendung von zumindest 3072 Bit langen Primzahlen, um auch den Index-Calculus-Algorithmus zu aufwendig werden zu lassen.

2.6 DSA-Signaturen

In diesem Abschnitt werfen wir einen genaueren Blick auf das bereits bekannte Signaturverfahren DSA.Mit dem Inkrafttreten der neuen Standard-Version FIPS 186-5 (im Februar 2023) dürfen DSA-Signaturen nur mehr geprüft, aber nicht mehr erstellt werden. In Kapitel 3 werden wir uns ansehen, wie andere Gruppen (statt $\mathbb{Z}_p^*$) eingesetzt werden können. Dabei ändert sich nicht viel am Signaturverfahren selbst, weshalb DSA-Signaturen hier auch noch einmal genauer betrachtet werden.

Algorithmus 2.18: DSA

Setup: Als Hashfunktion wird eine Hashfunktion H aus der SHA-x-Familie verwendet. Eine L Bit große Primzahl p wird vereinbart, so dass $p-1$ einen N Bit großen Primfaktor ω besitzt, weiterhin ein Element g der Ordnung ω in der Gruppe $\mathbb{Z}_p^*$.

Schlüsselerzeugung: Alice wählt zufällig eine Zahl $\alpha \in \mathbb{Z}_\omega$. Sie berechnet $A := g^\alpha \bmod p$ und veröffentlicht ihren Public Key A. Den Private Key α hält sie geheim.

Signieren: Um zu signieren, wählt Alice zufällig eine Zahl $k \in \mathbb{Z}_\omega$. Dann berechnet sie die Signatur (r, s) der Nachricht m als

$$\begin{aligned} r &:= (g^k \bmod p) \bmod \omega, \\ s &:= k^{-1}(H(m) + \alpha r) \bmod \omega. \end{aligned}$$

Verifizieren: Will Bob die Signatur überprüfen, so führt er die folgenden Schritte durch:

1. Er prüft: Ist $1 \leq r < \omega$ und $1 \leq s < \omega$?
2. Er berechnet $x := s^{-1} \cdot H(m) \bmod \omega$ und $y := s^{-1} \cdot r \bmod \omega$.
3. Er prüft: Ist $r = (g^x \cdot A^y \bmod p) \bmod \omega$?

Eine gültige Unterschrift besteht alle Tests:

1. Für r und s wurde modulo ω gerechnet, r und s sind also kleiner als ω.

   Die Werte r und/oder s könnten (theoretisch) gleich 0 sein. Im Fall, dass $s = 0$, würde in Schritt 2 ein Fehler beim Berechnen von s^{-1} auftreten, das Verifikationsverfahren also nicht funktionieren. Im Fall, dass $r = 0$, würde sich $y = 0$ ergeben und damit die Verifikation vom Public Key A unabhängig werden: ein Sicherheitsproblem. Um diese Probleme zu vermeiden, wird in diesen Fällen die Signatur gleich ungültig. Praktisch tritt dieser Fall nicht auf, weil die Wahrscheinlichkeit, dass r oder s den Wert 0 annehmen $1/\omega$, also vernachlässigbar klein ist.

3.

$$\begin{aligned} g^x \cdot A^y &= g^{s^{-1} \cdot H(m)} \cdot (g^\alpha)^{s^{-1} r} \\ &= g^{s^{-1}(H(m)+\alpha r)} \\ &= g^k \pmod p. \end{aligned}$$

2.6.1 Performance

Die Wahl eines L Bit langen p mit großem Primfaktor ω von $p - 1$ verhindert Index-Calculus-Angriffe (vgl. Anhang 2.5.3). Die Wahl eines N Bit langen ω verhindert das Berechnen von diskreten Logarithmen mittels Durchprobieren (z. B. Baby-Step-Giant-Step). Die Primalität von ω verhindert schließlich das Berechnen von diskreten Logarithmen mit der Pohlig-Hellman-Methode.

Im Standard erlaubte Werte für (L, N) sind u.a. $(3072, 256)$ für 128 Bit Sicherheit und $(15360, 512)$ für 256 Bit Sicherheit. Entsprechende Hashfunktionen sind dann zu verwenden: SHA-256 für 128 und SHA-512 für 256 Bit Sicherheit. Details lassen sich dem öffentlich zugänglichen Standarddokument FIPS 186-4 [KSD13] entnehmen, zusätzliche Empfehlungen, wie die Auswahl einer geeigneten Hashfunktion, dem ebenfalls öffentlichen Dokument NIST SP 800-57 [BBB+20]. Dem Stand der Technik entsprechen die Parameter $(3072, 256)$.

Private Keys sind nun N bit lang, Public Keys L bit. Signaturen nur $2N$ Bit lang, also (bei gleichem Sicherheitsniveau) bedeutend kürzer als RSA-Signaturen. Der wesentliche Aufwand beim Signieren entsteht durch einmaliges Potenzieren mit einem N bit langen Exponenten. Im Vergleich zu RSA ist DSA damit wesentlich effizienter beim Signieren. Beim Prüfen einer Signatur

tauchen ebenfalls nur mehr N bit lange Exponenten auf. Dank extrem kurzer Exponenten ist RSA beim Prüfen von Signaturen dennoch bedeutend schneller als DSA.

2.6.2 Attacken auf DSA-Signaturen

Nonce Reuse

Es ist nicht ratsam, sich Arbeit zu sparen, indem man beim Signieren jedes Mal die selbe Nonce k verwendet. Werden zwei verschiedene Nachrichten m_1 und m_2 mit dem selben Wert für k signiert, so lässt sich aus den beiden Signaturen[6] (r, s_1) und (r, s_2) der Private Key α berechnen:

Es ist dann (modulo ω)

$$\begin{aligned} s_1 - s_2 &= k^{-1}\big(H(m_1) + \alpha r\big) - k^{-1}\big(H(m_2) + \alpha r\big) \\ &= k^{-1}\big(H(m_1) + \cancel{\alpha r} - H(m_2) - \cancel{\alpha r}\big) \\ &= k^{-1}\big(H(m_1) - H(m_2)\big). \\ k &= (s_1 - s_2)^{-1}\big(H(m_1) - H(m_2)\big) \pmod{\omega}. \end{aligned}$$

Kennt man aber k, so lässt sich α einfach berechnen.

$$\begin{aligned} s_1 &= k^{-1}\big(H(m_1) + \alpha r\big) \\ ks_1 &= H(m_1) + \alpha r \\ \alpha r &= ks_1 - H(m_1) \\ \alpha &= r^{-1}\big(ks_1 - H(m_1)\big) \pmod{\omega}. \end{aligned}$$

Ein bekannter Fall von Nonce Reuse ist bei einigen Bitcoin-Wallet-Implementierungen aufgetreten, wo die Autorisierung von Transaktionen durch ECDSA-Signaturen[7] geschieht.[8] Dieses Problem tritt auch auf, wenn der für die Erzeugung der Nonce k zuständige (P)RNG nicht korrekt arbeitet und sich so Wiederholungen ergeben. Neben diesen sehr offensichtlich fehlerhaften Implementierungen wurden auch weitere Probleme entdeckt, wenn schwache PRNGS eingesetzt werden, um sehr große Mengen von Signaturen zu erstellen. In Blockchain-Szenarien, wo schnell Millionen von Signaturen zusammenkommen, sind auch diese Probleme praktisch relevant.[9]

Abschließend soll hier aber festgehalten werden, dass all diese Probleme bei einem standardkonform implementierten DSA/ECDSA-Verfahren nicht auftreten.

[6] Beachte, dass sich für das gleiche k auch zweimal der selbe Wert r ergibt. Nonce Reuse ist also auch sehr einfach zu erkennen.

[7] Das DSA-Verfahren lässt sich mit geringfügigen Adaptierungen auch mit elliptischen Kurven durchführen. Dies führt zu kürzeren Schlüsseln und höherer Geschwindigkeit bei gleichbleibender Sicherheit. Wir werden in Kapitel 3 auf elliptische Kurven genauer eingehen.

[8] Eine einfach lesbare Zusammenfassung findet sich auf `https://ishaana.com/blog/nonce_reuse/`.

[9] Einen aktuellen Überblick zum „State of the Art“ der Correlated-Nonce-Attacken findet man bspw. in `https://eprint.iacr.org/2023/305`.

2.7 Schnorr-Signaturen

Am Ende dieses Kapitels wird ein Signaturverfahren von Schnorr vorgestellt, das zwar praktisch in dieser Form nicht zum Signieren eingesetzt wird;[10] die Idee, auf diese Art Signaturen zu erstellen, wird aber beim quantensicheren Crystals-Dilithium-Verfahren in Abschnitt 5.5 wieder auftauchen.

Algorithmus 2.19: Schnorr-Signatur

Setup: Als Domain-Parameter werden eine Hashfunktion H, eine Gruppe $\mathbb{G}$ und ein Element $g \in \mathbb{G}$ der Ordnung ω vorbereitet.

Schlüsselerzeugung: Alice wählt zufällig eine Zahl $\alpha \in \mathbb{Z}_\omega$. Sie berechnet $A := g^\alpha$ und veröffentlicht ihren Public Key A. Den Private Key α hält sie geheim.

Signieren: Um zu signieren, wählt Alice zufällig eine Zahl $k \in \mathbb{Z}_\omega$ und berechnet

$$\begin{aligned} r &:= g^k, \\ c &:= H(r, m) \bmod \omega, \\ s &:= k + c\alpha \pmod{\omega}. \end{aligned}$$

Die Signatur ist dann das Paar (c, s).

Verifizieren: Will Bob die Signatur überprüfen, so führt er die folgenden Schritte durch:

1. Er berechnet $r := g^s \circ A^{-c}$ und
2. prüft, ob $c = H(r, m) \bmod \omega$.

Wir bemerken, dass die Überprüfung für eine korrekt erstellte Signatur erfolgreich ist. Es ergibt sich in Schritt 1 das richtige r, denn

$$g^s \circ A^{-c} = g^{k+c\alpha} \circ g^{-c\alpha} = g^{k+\cancel{c\alpha}-\cancel{c\alpha}} = g^k = r.$$

[10] Ein wichtiger Grund dafür, dass Schnorr-Signaturen heute praktisch nicht eingesetzt werden, ist, dass sie von Schnorr zunächst patentiert wurden. Als DSA-Standard kamen sie damit nicht mehr infrage. Bei genauerem Vergleich von DSA- mit Schnorr-Signaturen zeigen sich allerdings viele Ähnlichkeiten, die auch zu Rechtsstreitigkeiten geführt haben. Inzwischen ist das Patent auf Schnorr-Signaturen ausgelaufen und Schnorr-Signaturen erhalten nun mehr Aufmerksamkeit.

Rückblick 2.2

Du hast dich in diesem Kapitel mit dem DLP (diskreten Logarithmenproblem) und dem DHP (Diffie-Hellman-Problem) auseinandergesetzt. Du kannst beide Probleme erklären und weißt, wie Domainparameter gewählt werden müssen, damit diese Probleme ausreichend schwierig sind. Du bist vertraut mit den wichtigsten Rechenregeln zum Potenzrechnen mit Restklassen, insbesondere mit dem Satz von Fermat. Du verstehst jetzt, dass Restklassen im DH-Verfahren und anderen DLP-basierten Verfahren durch andere Objekte ersetzt werden können, die eine Gruppe bilden. Du kannst mit den Begriffen Ordnung, neutrales Element und inverses Element einer Gruppe etwas anfangen. Du hast verschiedene Verfahren zur Berechnung diskreter Logarithmen, wie den Baby-Step-Giant-Step-Algorithmus, das Pohlig-Hellman-Verfahren und den Index-Calculus-Algorithmus durchgeführt und weißt, wie du Domainparameter zu wählen hast, um Angriffe mit diesen Verfahren ausschließen zu können. Du kannst mit DSA ein auf dem DLP basierendes Signaturverfahren beschreiben. Mit Schnorr-Signaturen kennst du ein weiteres solches Verfahren, das du auch gleich für beliebige Gruppen beschreiben kannst.

Kapitel 3

Elliptische Kurven

Ziele 3.1

In diesem Kapitel lernst du,

- was elliptische Kurven sind und wie man damit rechnet.
- warum elliptische Kurven effizientere kryptographische Verfahren ermöglichen.
- wie projektive Koordinaten eingesetzt werden können, um die Performance weiter zu steigern, und was Montgomery Curves sind.
- wie elliptische Kurven zum Schlüsselaustausch (ECDH) und zum Signieren (ECDSA, Ed25519) eingesetzt werden.
- abschließend, welche Schlüssellängen für verschiedene Public-Key-Verfahren empfohlen werden, um ein bestimmtes Sicherheitsniveau zu erreichen.

In über 40 Jahren RSA und DH haben sich das Faktorisierungsproblem und das DLP/CDH/DDH wie in den Jahrhunderten davor nicht vollständig effizient lösen lassen. Allerdings werden immer mehr und bessere Methoden entwickelt, so dass es immer schwieriger wird, sichere Parameter für diese Verfahren zu finden.

Beispiel 3.1

Für das RSA-Verfahren hat man zur Zeit wenigstens 1536 Bit lange starke Primzahlen zu wählen. Glücklicherweise geht das, weil auch das Finden (und Testen) von Primzahlen immer einfacher wird. Es bleibt jedoch das Problem, dass mit immer größeren Zahlen gerechnet werden muss, und das möglichst schnell.

Beispiel 3.2

Für DLP-verwandte Verfahren, bei denen als Gruppe $\mathbb{Z}_p^*$ verwendet wird, muss p wenigstens 3072 Bit lang sein. Grund dafür ist, dass zum Berechnen von diskreten Logarithmen spezielle Methoden – wie z. B. der Index-Calculus-Algorithmus – verwendet werden können. Auch hier muss man mit sehr großen Zahlen potenzieren – viel Aufwand.

Das DLP lässt sich jedoch, wie wir gesehen haben, für beliebige Gruppen formulieren. Wir studieren in diesem Kapitel eine ganz andere Art von Gruppen: elliptische Kurven.

3.1 Elliptische Kurven über $\mathbb{R}$

Zum Einstieg betrachten wir elliptische Kurven über dem Körper $\mathbb{R}$. Diese werden in der Kryptographie nicht verwendet, erleichtern uns aber den Einstieg, weil sie uns eine bildliche Vorstellung der Gruppenoperation geben.

Definition 3.3

Es seien $a, b \in \mathbb{R}$ so, dass $4a^3 + 27b^2 \neq 0$. Dann ist

$$\mathcal{E} := \{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b\} \cup \{\infty\}$$

eine elliptische Kurve über $\mathbb{R}$.

Der Einfachheit halber schreiben wir statt der elliptischen Kurve nur die Gleichung. Das Symbol ∞ bekommt später noch eine Bedeutung, im Moment dürfen wir es ignorieren.

Abbildung 3.1 zeigt die Graphen verschiedener elliptischer Kurven über $\mathbb{R}$.

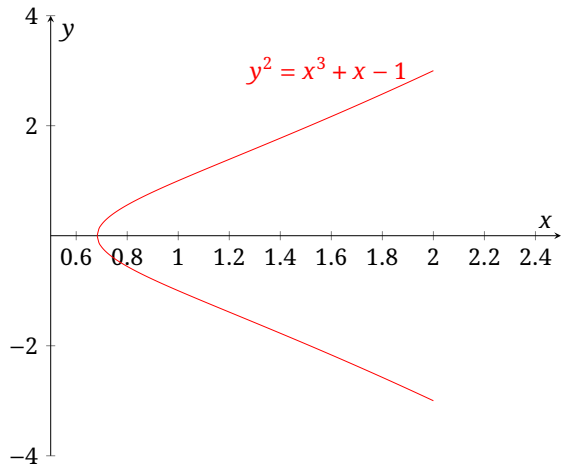
Aus $\mathcal{E}$ lässt sich eine Gruppe machen. Dazu brauchen wir eine Verknüpfung (Addition), die aus zwei Punkten P und Q auf der elliptischen Kurve wieder einen Punkt $(P+Q)$ auf der Kurve macht.[1] Die Idee für so eine Verknüpfung ist recht einfach und ist in Abbildung 3.2a illustriert: Man lege durch die beiden Punkte eine Gerade. Diese schneidet die Kurve in einem dritten Punkt $(P * Q)$, spiegelt man diesen an der x-Achse, so erhält man erneut einen Punkt $(P+Q)$ auf der Kurve.[2] Will

[1] Nicht nur das, es wird auch noch ein neutrales Element und zu jedem Element ein inverses Element benötigt. Darum kümmern wir uns dann auch noch.

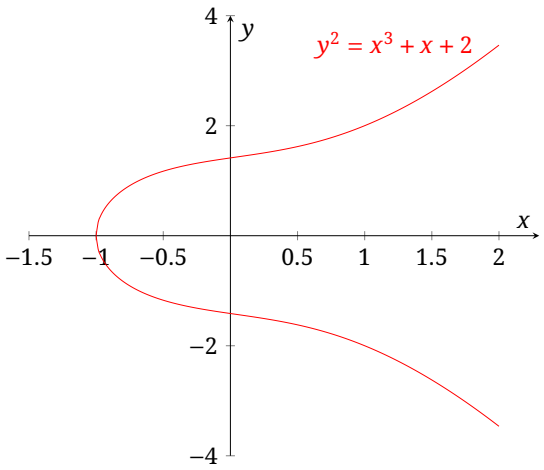
[2] Das Spiegeln an der x-Achse zum Abschluss scheint auf den ersten Blick überflüssig. Lässt man diesen Schritt jedoch weg, so erhält man eine recht sonderbare Verknüpfung, die das Assoziativitätsgesetz

$$(P * Q) * R = P * (Q * R)$$

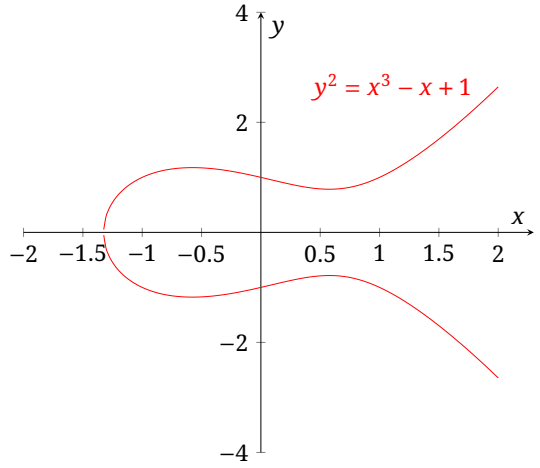
nicht erfüllt, sodass sich also keine Gruppe aus der elliptischen Kurve machen lässt.



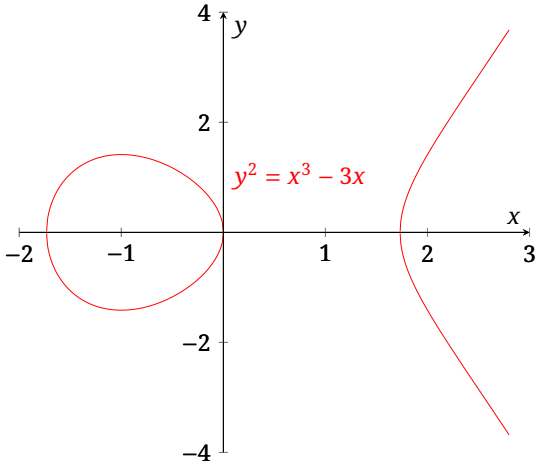
(a) $a = 1, b = -1$



(b) $a = 1, b = 2$

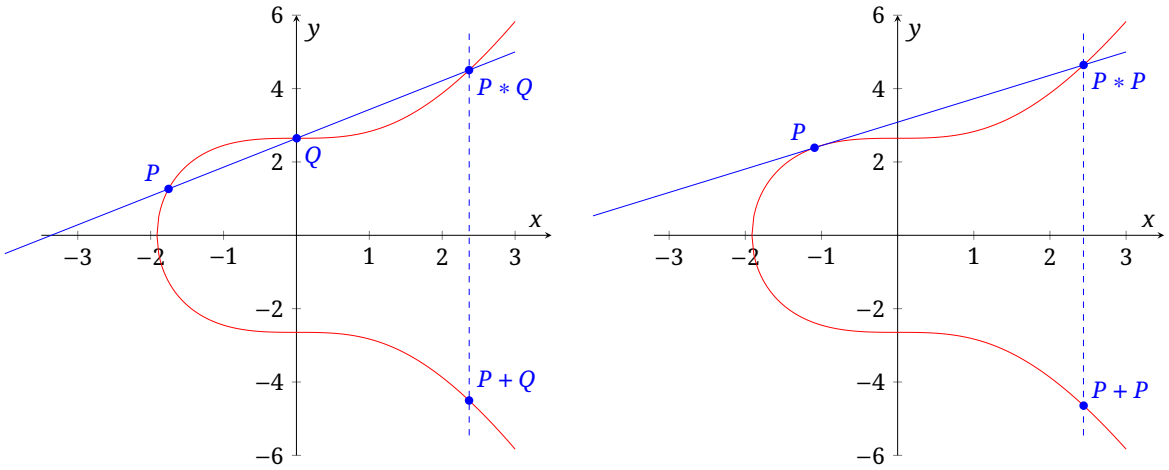


(c) $a = -1, b = 1$



(d) $a = -3, b = 0$

Abbildung 3.1: Elliptische Kurven $y^2 = x^3 + ax + b$ für verschiedene Parameter (a, b)



(a) Addition $P + Q$ (b) Verdopplung $P + P$

Abbildung 3.2: Addition von Punkten und Punktverdopplung auf einer elliptischen Kurve

man einen Punkt zu sich selbst addieren, muss man etwas anders vorgehen (vgl. Abbildung 3.2b): Im Punkt wird die Tangente an die Kurve gelegt und ein weiterer Schnittpunkt ($P * P$) mit der Kurve gesucht, dieser schließlich an der x-Achse gespiegelt, um $P + P$ zu erhalten.

Um schließlich wirklich eine Gruppe zu erhalten, fehlt noch ein neutrales Element. Dieses muss „dazuerfunden“ werden. Es wird üblicherweise (auch wenn das verwirrend ist) mit dem Symbol ∞ bezeichnet, ein Punkt, der keine Koordinaten besitzt. Dann legen wir die Regeln für die Verknüpfung wie folgt fest:

Definition 3.4

Es sei $\mathcal{E}$ eine elliptische Kurve. Wir definieren für alle $P, Q \in \mathcal{E}$:

1. $-\infty := \infty$ und $P + \infty := \infty + P := P$. (Damit wird ∞ zum neutralen Element.)
2. Ist $P = (x, y)$, dann sei $-P := (x, -y)$.
3. Ist $Q = -P$, dann sei $P + Q := \infty$.
4. $P + P := -(P * P)$, wobei $P * P$ der andere Schnittpunkt der Tangente an $\mathcal{E}$ in P ist (siehe Abb. 3.2b).
5. Ist $P \neq \pm Q$, dann sei $P + Q := -(P * Q)$, wobei $P * Q$ der dritte Schnittpunkt der Geraden durch P und Q mit $\mathcal{E}$ ist (Abb. 3.2a).

Es bietet sich an, das aufwendige Aufstellen der Geraden, das Schneiden mit der Kurve und das Spiegeln einmal ganz allgemein durchzurechnen, denn es ergeben sich dabei kurze Formeln, die man sogar auswendig lernen kann. In Anhang A.4 wird dies durchgeführt.

Satz 3.5

Es seien $\mathcal{E}: \; y^2 = x^3 + ax + b$ eine elliptische Kurve über $\mathbb{R}$ und $P = (x_1, y_1)$ und $Q = (x_2, y_2)$ zwei Punkte auf $\mathcal{E}$. Dann lassen sich die Koordinaten (x_3, y_3) von $R := P + Q$ nach folgenden Formeln berechnen.

- Falls $P = -Q$, dann ist $R = \infty$.
- Falls $P \neq Q$, dann ist

$$\begin{aligned} x_3 &= k^2 - x_1 - x_2, \\ y_3 &= -y_1 + k(x_1 - x_3), \\ \text{wobei } k &:= \frac{y_2 - y_1}{x_2 - x_1}. \end{aligned} \tag{3.1}$$

- Falls $P = Q$, dann ist

$$\begin{aligned} x_3 &= k^2 - 2x_1, \\ y_3 &= -y_1 + k(x_1 - x_3), \\ \text{wobei } k &:= \frac{3x_1^2 + a}{2y_1}. \end{aligned} \tag{3.2}$$

Ist $P = \infty$, dann ist $P + Q := Q$. Ist $Q = \infty$, dann ist $P + Q := P$.

Die sich ergebenden Formeln zeigen, dass für die Berechnungen lediglich die Grundrechenoperationen $(+, -, \cdot, /)$ benötigt werden. Neben den reellen Zahlen lassen sich diese Formeln auch benutzen, wenn mit Restklassen modulo einer Primzahl gearbeitet wird.

Mit den Elementen der Mengen $\mathbb{R}$ und $\mathbb{Z}_p$ lässt sich addieren, subtrahieren, multiplizieren und dividieren, als weitere Rechenregel gibt es das Ausmultiplizieren. Auch die rationalen Zahlen $\mathbb{Q}$ „funktionieren“. Wenn all das mit den Elementen einer Menge möglich ist, spricht man von einem Körper.[3]

[3] Vergleiche die folgende Definition mit der Definition eines Körpers in der LVA „Diskrete Mathematik und lineare Algebra“. Die beiden Definitionen sind nicht (wort-)gleich, aber äquivalent.

Definition 3.6

Ein Körper ist ein Septupel $(K, +, -, 0, \cdot, ^{-1}, 1)$, wobei K eine Menge ist und die folgenden Gesetze erfüllt sind:

1. $(K, +, 0, -)$ ist eine abelsche Gruppe.
2. $(K \setminus \{0\}, \cdot, 1, ^{-1})$ ist eine abelsche Gruppe.
3. Für alle $a, b, c \in K$ gilt: $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$.

Eine weitere Klasse von Körpern wird in Kapitel 5 auftauchen.

Es bietet sich also an, elliptische Kurven nicht nur über $\mathbb{R}$, sondern über beliebigen Körpern – bspw. über $\mathbb{Z}_p$ – zu betrachten. Der folgende Satz ist bereits so formuliert, dass (fast) jeder Körper verwendet werden kann.

Satz 3.7

Es sei K ein Körper, in dem $1 + 1 \neq 0$ und $1 + 1 + 1 \neq 0$. Weiterhin seien $a, b \in K$ so, dass $4a^3 + 27b^2 \neq 0$ und $\mathcal{E} : y^2 = x^3 + ax + b$ eine elliptische Kurve über K und $P = (x_1, y_1)$ und $Q = (x_2, y_2)$ zwei Punkte auf $\mathcal{E}$. Dann lassen sich die Koordinaten (x_3, y_3) von $R := P + Q$ nach den Formeln in Satz 3.5 berechnen.

In den Formeln in Satz 3.5 taucht eine 2 im Nenner auf. In Körpern, wo $2 = 0$ gilt, führt das zu Divisionen durch Null. Bei der Punktverdopplung erfolgt eine Multiplikation mit 3. Diese führt ebenfalls zu Problemen (anderer Art), wenn in einem Körper $3 = 0$ gilt. Elliptische Kurven, für deren Parameter $4a^3 + 27b^2 = 0$ gilt heißen singulär, mit solchen Kurven erhält man kein schwieriges DLP, wir betrachten sie daher nicht weiter.

Daher wurden diese Fälle in Satz 3.7 einfach ausgeschlossen und werden auch in der Folge nicht betrachtet.

Eine nicht ganz einfach zu beweisende Tatsache ist, dass die Punktaddition die Punkte auf einer elliptischen Kurve zu einer Gruppe macht.

Satz 3.8

Ist $\mathcal{E}$ eine elliptische Kurve, dann ist $(\mathcal{E}, +, \infty, -)$ eine abelsche Gruppe.

Der Beweis, dass jetzt das Assoziativgesetz gilt, ist alles andere als einfach, und viel zu lang für uns. Wir begnügen uns mit einem Blick auf Abbildung 3.3, die illustriert, dass $(P + Q) + R = P + (Q + R)$ gilt.

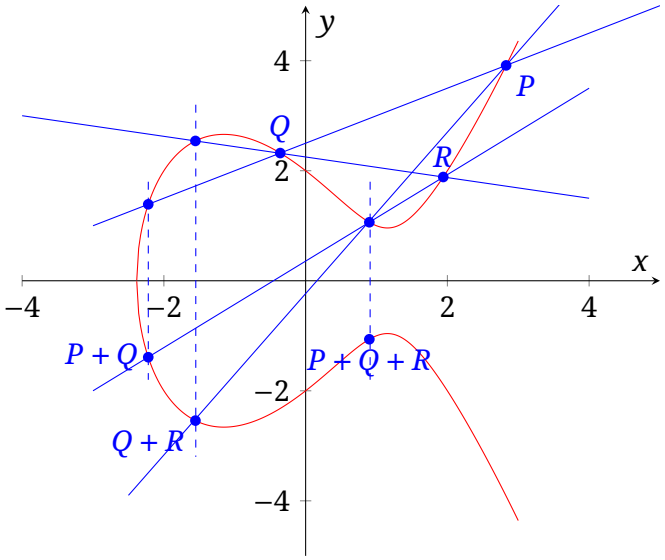


Abbildung 3.3: Die Addition von Punkten auf elliptischen Kurven ist assoziativ.

Für kryptografische Anwendungen ist (man erinnere sich an DH) das Potenzieren in Gruppen von Interesse. Im Zusammenhang mit elliptischen Kurven ist dies das mehrfache Addieren eines Punktes zu sich selbst. Naheliegend ist, hier $\alpha \cdot P := \underbrace{P + P + \cdots + P}_{\alpha \text{ mal}}$ zu definieren, genauer:

Definition 3.9: Punktmultiplikation auf elliptischen Kurven

Es seien $\mathcal{E}$ eine elliptische Kurve, $P \in \mathcal{E}$ und $\alpha \in \mathbb{Z}$. Dann sei

$$\alpha \cdot P := \begin{cases} \infty & \text{, falls } \alpha = 0 \\ \underbrace{P + P + \cdots + P}_{\alpha \text{ mal}} & \text{, falls } \alpha > 0 \text{ und} \\ (-\alpha) \cdot (-P) & \text{, falls } \alpha < 0. \end{cases}$$

Effizient lassen sich Punktmultiplikationen durch Anwenden der Square-and-Multiply-Idee durchführen. Das Quadrieren entspricht nun einer Punktverdopplung, das Multiplizieren einer Punktaddition, man könnte also von einer Double-and-Add-Methode sprechen.

3.2 Elliptische Kurven modulo p

Wir haben die Additionsformeln gleich für elliptische Kurven über (fast) beliebigen Körpern hergeleitet. Wir sehen uns jetzt elliptische Kurven über den endlichen Körpern $\mathbb{Z}_p$ genauer an.

3.2.1 Der Satz von Hasse

Modulo p gibt es nur p verschiedene x- und y-Koordinaten für Punkte, also maximal $p^2 + 1$ verschiedene Punkte. Es könnte aber auch sein, dass nur ganz wenige Punkte auf einer gegebenen Kurve liegen. Um elliptische Kurven kryptographisch einsetzen zu können, muss das DLP schwierig zu lösen sein. Dazu werden Elemente mit großer Ordnung benötigt. Da die Ordnung eines Elements stets Teiler der Gruppenordnung ist, brauchen wir also elliptische Kurven mit vielen Punkten. Der Satz von Hasse gibt eine beruhigende Antwort auf die Frage „Wie viele Punkte liegen auf einer elliptischen Kurve über $\mathbb{Z}_p$?“

Satz 3.10: Satz von Hasse

Es sei n die Anzahl der Punkte auf einer elliptischen Kurve modulo p. Dann ist

$$(p+1) - 2\sqrt{p} \leq n \leq (p+1) + 2\sqrt{p}.$$

(D.h. die Anzahl der Punkte auf der elliptischen Kurve ist von der selben Größenordnung wie p, denn $\sqrt{p}$ hat nur halb so viele Stellen wie p.)

Somit ist eine elliptische Kurve modulo p eine Gruppe, deren Ordnung in etwa so groß wie p ist. Es ist keine Formel bekannt, mit der sich die Ordnung dieser Gruppe einfach berechnen ließe. Es gibt allerdings effiziente Algorithmen für die Bestimmung der Ordnung einer elliptischen Kurve, so z. B. den Algorithmus von Schoof [Sch85] und Varianten davon. Um sichere Kryptografie mit elliptischen Kurven zu treiben, sollte sichergestellt sein, dass die Ordnung der Gruppe einen großen Primfaktor besitzt. Wenn man eine elliptische Kurve wählt, wird man also nicht um das (etwas mühsame) Berechnen der Ordnung herumkommen, bzw. sich einer Kurve bedienen, deren Ordnung bereits berechnet worden ist.

Beispiel 3.11

Untersuchen wir die elliptische Kurve

$$y^2 = x^3 + 4x + 4 \pmod{29}. \tag{3.3}$$

Dazu bestimmen wir für jedes $x \in \mathbb{Z}_{29}$ den Wert der rechten Seite der Kurvengleichung (3.3).

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	4	9	20	14	26	4	12	27	26	15	0	16	11	20	20
x	15	16	17	18	19	20	21	22	**23**	24	25	26	27	28	
	17	17	26	21	8	22	11	10	**25**	4	11	23	17	28	

Nun berechnen wir für alle $y \in \mathbb{Z}_{29}$ den Wert der linken Seite der Kurvengleichung (3.3).

y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	1	4	9	16	25	7	20	6	23	13	5	28	24	22
y		28	27	26	25	**24**	23	22	21	20	19	18	17	16	15
		1	4	9	16	**25**	7	20	6	23	13	5	28	24	22

Damit lässt sich nun einfach erkennen, dass die folgenden Punkte Elemente der elliptischen Kurve sind.

$$(0,2),(0,27),(1,3),(1,26),(2,7),(2,22),(5,2),(5,27),(10,0),$$
$$(11,4),(11,25),(13,7),(13,22),(14,7),(14,22),(20,14),(20,15),$$
$$(23,5),(\mathbf{23},\mathbf{24}),(24,2),(24,27),(26,9),(26,20),(28,12),(28,17),\infty$$

Die Ordnung der Gruppe ist 26. Dies ist innerhalb der Hasse-Grenzen $p+1\pm 2\sqrt{p}$, die hier eine Ordnung zwischen 20 und 40 ergeben.

Berechnet man für den Punkt $P=(10,0)$ den Punkt $2\cdot P$ (also $P+P$), so erhält man mit den bekannten Formeln $2\cdot P=\infty$. Daher ist die Ordnung von P gleich 2. Dies ist der einzige Punkt der Ordnung 2. Punkte auf dieser elliptischen Kurve haben die Ordnungen 1, 2, 13 oder 26, denn dies sind die einzigen Teiler der Gruppenordnung.

Der Punkt $Q=(2,7)$ hat die Ordnung 26. Um dies zu verifizieren, muss nach Satz 2.12 ausgeschlossen werden, dass $13\cdot Q=\infty$ und dass $2\cdot Q=\infty$. Dazu berechnet man $2\cdot Q=(5,2)\neq\infty$ und $13\cdot Q=8Q+4Q+Q=(11,25)+(23,5)+(2,7)=(10,0)\neq\infty$. Somit ist Q erzeugendes Element der Gruppe. Der diskrete Logarithmus von $(10,0)$ zur Basis Q ist 13.

Das Modul `si` enthält Funktionen zum Rechnen mit elliptischen Kurven.

```
> import si
> from si import EC, Point
> e = EC( 4, 4, 29 )
> e.order()
26
> e.list_of_points()
[Point( EC( 4, 4, 29 ), None ),
Point( EC( 4, 4, 29 ), ( 0, 27 ) ), Point( EC( 4, 4, 29 ), ( 0, 2 ) ),
Point( EC( 4, 4, 29 ), ( 1, 26 ) ), Point( EC( 4, 4, 29 ), ( 1, 3 ) ),
[...]
Point( EC( 4, 4, 29 ), ( 28, 12 ) ), Point( EC( 4, 4, 29 ), ( 28, 17 ) )]
> p = Point( e, (10,0) )
> p.double()
Point( EC( 4, 4, 29 ), None )
> 2*p
```

```
Point( EC( 4, 4, 29 ), None )
> q = Point( e, (2,7) )
> q.inverse()
Point( EC( 4, 4, 29 ), ( 2, 22 ) )
> -q
Point( EC( 4, 4, 29 ), ( 2, 22 ) )
> q.double()
Point( EC( 4, 4, 29 ), ( 5, 2 ) )
> 13*q
Point( EC( 4, 4, 29 ), ( 10, 0 ) )
> q.mult( 13, verbose=1 )
         computing 13 * ( 2, 7 )
         adding ...
         doubling ...
         doubling ...
         adding ...
         doubling ...
         adding ... Point( EC( 4, 4, 29 ), ( 10, 0 ) )
> p+q
Point( EC( 4, 4, 29 ), ( 1, 3 ) )
> p.add( q, verbose=2 )
    ----------------------------------
    adding ( 10, 0 ) and ( 2, 7 ):
      k = (0-7) / (10-2)
        = 22 / 8
        = 22 * 11
        = 10
     x3 = 10**2 - 10 - 2
        = 1
     y3 = -0 + 10*(10-1)
        = -0  + 10*9
        = 3
    ( 10, 0 ) + ( 2, 7 ) = ( 1, 3 )
    ==================================
Point( EC( 4, 4, 29 ), ( 1, 3 ) )
```

3.3 Kryptografische Verfahren mit elliptischen Kurven

3.3.1 ECDH

Exemplarisch soll hier noch einmal gezeigt werden, wie ein DH-Schlüsselaustausch mit einer elliptischen Kurve über $\mathbb{Z}_p$ abläuft.

Algorithmus 3.12: Ephemeral ECDH

Setup: Alice und Bob einigen sich auf eine Primzahl p, eine elliptische Kurve $\mathcal{E} : y^2 = x^3 + ax + b$ modulo p und einen Punkt G mit primer Ordnung ω auf $\mathcal{E}$. Als Domain Parameter werden $\mathcal{E}$, G, p und ω veröffentlicht.

Key Agreement:

Alice		Bob
$\alpha \xleftarrow{R} \mathbb{Z}_\omega$	$\xrightarrow{A \leftarrow \alpha \cdot G}$	
	$\xleftarrow{B \leftarrow \beta \cdot G}$	$\beta \xleftarrow{R} \mathbb{Z}_\omega$
$K \leftarrow \alpha \cdot B$		$K \leftarrow \beta \cdot A$

3.3.2 ECDSA

In FIPS 186 werden auch Signaturen mit elliptischen Kurven standardisiert. Wie nicht anders zu erwarten handelt es sich dabei um die konsequente Übersetzung des DSA auf elliptische Kurven.

Im folgenden Algorithmus bezeichnet $P_{x\text{-Koord}}$ die x-Koordinate des Punktes P.

Algorithmus 3.13: ECDSA

Setup: Hier werden als Domain Parameter eine zumindest 256 Bit lange Primzahl p und eine elliptische Kurve $\mathcal{E}$ modulo p vereinbart, deren Ordnung einen großen Primfaktor ω besitzt, weiterhin ein Element G der Ordnung ω auf $\mathcal{E}$. Als Hashfunktion wird eine Hashfunktion H aus der SHA-x-Familie verwendet.

Schlüsselerzeugung: Alice wählt zufällig eine Zahl $\alpha \in \mathbb{Z}_\omega$. Sie berechnet $A := \alpha \cdot G$ und veröffentlicht ihren Public Key A. Den Private Key α hält sie geheim.

Signieren: Um zu signieren, wählt Alice zufällig eine Zahl $k \in \mathbb{Z}_\omega$. Dann berechnet sie die Signatur (r, s) der Nachricht m als

$$r := (k \cdot G)_{x\text{-Koord}} \bmod \omega,$$
$$s := k^{-1}\big(H(m) + \alpha r\big) \bmod \omega.$$

Verifizieren: Will Bob die Signatur überprüfen, so führt er die folgenden Schritte durch:

1. Er prüft: Ist $1 \leq r < \omega$ und $1 \leq s < \omega$?
2. Er berechnet $x := s^{-1} \cdot H(m) \bmod \omega$ und $y := s^{-1} \cdot r \bmod \omega$.
3. Er prüft: Ist $r = (x \cdot G + y \cdot A)_{x\text{-Koord}} \bmod \omega$?

Eine Kurve mit geeigneter Ordnung zu finden, ist nicht ganz einfach. Im NIST-Standard FIPS 186 findet sich eine Liste geeigneter Kurven, auf die in der Regel zurückgegriffen wird.

Ed25519

Für besonders effiziente Signaturen mit elliptischen Kurven wird gerne das Verfahren Ed25519 eingesetzt, das hier ganz kurz dargestellt werden soll.

Als Gruppe für die Berechnungen wird hier die elliptische Kurve mit der Gleichung

$$-x^2 + y^2 = 1 + dx^2y^2 \pmod{p},$$

mit den Parametern $p = 2^{255} - 19$ (daher der Name) und $d = -\frac{121665}{121666}$ verwendet. Diese Kurve, eine sogenannte Edwards-Kurve, ist eng verwandt mit Curve25519, die in Abschnitt 3.5.1 vorgestellt wird. Details zu dieser Verwandtschaft finden sich in [BBJ+08].

Die Form der elliptischen Kurve ist etwas anders als wir es gewöhnt sind. Die Formeln für $P+Q$ und $-P$ sehen auch etwas anders aus. Als sehr angenehm stellt sich heraus, dass das neutrale Element keine besondere Form (∞), sondern ganz normale Koordinaten hat. Es können auch die selben Formeln für Punktaddition und Punktverdopplung verwendet werden.

In FIPS 186-5 sind auch Ed25519 und das damit verwandte Verfahren Ed448 standardisiert.

Satz 3.14

Es sei $p = 2^{255} - 19$ und $d = -\frac{121665}{121666} \pmod{p}$. Weiterhin sei

$$\mathcal{E} := \left\{(x, y) \in (\mathbb{Z}_p)^2 \,\middle|\, -x^2 + y^2 = 1 + dx^2y^2\right\}.$$

Definiert man für $P = (x_1, y_1) \in \mathcal{E}$ und $Q = (x_2, y_2) \in \mathcal{E}$

$$\begin{aligned} \perp &:= (0, 1), \\ -P &:= (-x_1, y_1) \text{ und} \\ P + Q &:= (x_3, y_3), \text{ mit} \\ t &:= dx_1x_2y_1y_2, \\ x_3 &:= \frac{x_1y_2 + x_2y_1}{1 + t} \qquad \text{und} \\ y_3 &:= \frac{y_1y_2 + x_1x_2}{1 - t}, \end{aligned}$$

so ist $(\mathcal{E}, +, \perp, -)$ eine abelsche Gruppe und die Ordnung des Punkts G mit y-Koordinate $4/5 \pmod{p}$ ist die Primzahl

$$\omega = 2^{252} + 27742317777372353535851937790883648493.$$

Die Ordnung der Kurve $\mathcal{E}$ ist 8ω.

Als Signaturverfahren ist Ed25519 dem ECDSA-Verfahren sehr ähnlich, Details sind [BDL+12] zu entnehmen. Zwei Modifikationen wurden vorgenommen, um die Sicherheit zu erhöhen.

Keine zufälligen Parameter. Für ECDSA-Signaturen wird ein zufälliger Wert k gewählt, um

$$r := (k \cdot G)_x \bmod \omega$$

zu berechnen. Bereits in Abschnitt 2.6.2 haben wir gesehen, dass Probleme entstehen, wenn k nicht (pseudo-)zufällig gewählt wird (Nonce Reuse). Bei Ed25519 wird deshalb k als Hashwert über einen Teil des Private Key und die zu signierende Nachricht berechnet. Wird die selbe Nachricht mit dem selben Private Key ein zweites Mal signiert, so ergibt sich exakt die gleiche Signatur.

Hashwert hängt nicht nur von der Nachricht ab. ECDSA-Signaturen berechnen den zweiten Teil der Signatur nach der Formel

$$s := k^{-1}\big(H(m) + \alpha r\big) \bmod \omega.$$

Bei Ed25519 hängt der Hashwert nicht allein von m ab, sondern zusätzlich vom Public Key und dem zuvor berechneten Wert r (und damit auch vom Private Key). Durch diese Veränderung reicht ein Brechen der (starken) Kollisionsresistenz der Hashfunktion nicht mehr,

um Signaturen zu fälschen. Diese Konstruktion ist dem Schnorr-Signaturverfahren „abgeschaut“. Schließlich wird der Wert s ebenfalls ähnlich wie bei Schnorr-Signaturen als

$$s := k + \alpha H(r, A, m) \mod \omega$$

berechnet. Die Prüfung einer Signatur erfolgt dann auch analog zu Schnorr-Signaturen.

3.4 Vergleich von Schlüssellängen

Abbildung 3.4 zeigt einen Vergleich von Schlüssellängen für verschiedene Verfahren bei vergleichbarer Sicherheit. In Klammern sind Beispiele für konkrete Verfahren angegeben. Schlüssellängen, die einen Aufwand unter 2^{128} für Angreifende bedeuten, werden heute als unsicher erachtet.

Aufwand	Blockchiffre	MAC	Hashfunktion	RSA/DH	EC
2^{128}	128 (AES-128)	128 (AES-128-CMAC)	256 (SHA-256)	3072	256
2^{192}	192 (AES-192)	192	384	7680	384
2^{256}	256 (AES-256)	256 (SHA-256-HMAC)	512 (SHA-512)	15360	512

Abbildung 3.4: Vergleich von Schlüssellängen

3.5 Effizienz elliptischer Kurven

Die kryptographischen Verfahren, bei denen elliptische Kurven eingesetzt werden, sind jene, deren Sicherheit auf dem DDH-Problem beruht. Es soll hier verglichen werden, ob die Punktmultiplikation für elliptische Kurven schneller möglich ist als das modulare Potenzieren für die bekannten Gruppen $\mathbb{Z}_p^*$.[4]

Als Beispiel wird eine Schlüssellänge gewählt, die der Sicherheit einer AES-Verschlüsselung mit 128-Bit-Schlüssel entspricht.

Für die Gruppe $\mathbb{Z}_p^*$ bedeutet dies, dass eine Primzahl mit 3072 Bit Länge verwendet werden muss. Um den Aufwand im Rahmen zu halten, kann aber zumindest ein Basiselement für die Domain Parameter gewählt werden, dessen Ordnung ω nur 256 Bit lang ist. Für elliptische Kurven ist derzeit der Baby-Step-Giant-Step-Algorithmus der beste bekannte Algorithmus, um diskrete Logarithmen zu berechnen. Daher reicht es hier, wenn die Ordnung der Gruppe 256 Bit lang ist. Nach dem Satz von Hasse bedeutet dies, dass eine elliptische Kurve modulo p mit einer 256 Bit langen Primzahl p verwendet werden kann.

[4] Die Berechnungen in diesem Abschnitt sind nicht exakt, sondern sollen einen Eindruck vermitteln, warum elliptische Kurven einen Effizienzvorteil bringen.

Für das Berechnen einer Potenz in $\mathbb{Z}_p^*$ müssen, da der zufällig gewählte Exponent 256 Bit lang ist, im Mittel $1,5 \cdot 256 = 384$ modulare Multiplikationen mit 3072 Bit langen Zahlen durchgeführt werden. Für das Berechnen eines Vielfachen eines Punktes sind im Mittel ebenfalls $1,5 \cdot 256 = 384$ Punktadditionen erforderlich. Es reicht also, den Aufwand für eine Punktaddition mit dem einer Multiplikation in $\mathbb{Z}_p^*$ zu vergleichen.

Jede Punktaddition besteht aus einer Reihe von Multiplikationen, Additionen, Subtraktionen und einer Division. Geht man davon aus, dass eine Division (mit dem erweiterten Euklidschen Algorithmus) etwa 10–12 mal so aufwendig wie eine Multiplikation ist, so entspricht eine Punktaddition in etwa 15 Multiplikationen von 256 Bit langen Zahlen. Bei den elliptischen Kurven sind die Zahlen allerdings um einen Faktor von $3072/256 = 12$ kürzer. Dies bedeutet, dass jede Multiplikation um einen Faktor $12^2 = 144$ schneller ist.

Insgesamt ist eine Lösung mit elliptischen Kurven hier also um einen Faktor von $144/15 = 9{,}6$ schneller.

Für noch größere Schlüssellängen verschiebt sich das Verhältnis weiter zugunsten elliptischer Kurven. Um 256 Bit Sicherheit zu bekommen, sind 15360 Bit[5] lange Primzahlen für $\mathbb{Z}_p^*$ erforderlich, für elliptische Kurven aber nur 512 Bit lange. Hier ergibt sich (mit den selben Überlegungen) bereits ein Faktor 60 zugunsten der elliptischen Kurven.

3.5.1 Performancesteigerung

In diesem Abschnitt werden ein paar Möglichkeiten gestreift, die Performance weiter zu erhöhen. Details können weiterführender Literatur entnommen werden.

Punktkompression

Punkte auf elliptischen Kurven können für die Übertragung oder Speicherung komprimiert werden. Ist $P = (x, y)$ ein Punkt auf einer elliptischen Kurve $\mathcal{E}$ über $\mathbb{Z}_p$, dann ist auch $-P = (x, -y) \in \mathcal{E}$. Dies sind die einzigen Punkte auf $\mathcal{E}$ mit x-Koordinate x. Ist x bekannt, lässt sich $\pm y$ bestimmen.[6] Welche der beiden Lösungen $\pm y$ die richtige y-Koordinate ist, muss noch gespeichert werden. Ist $y < p/2$, dann ist $-y = p - y > p/2$, und umgekehrt. Es reicht also, die Information zu speichern, ob $y < p/2$, dafür reicht ein Bit.

Umgekehrt bedeutet dies auch, dass die y-Koordinate eines Punktes auf einer elliptischen Kurve fast ausschließlich redundante Information trägt. Sie wird daher z. B. nicht verwendet, um Schlüsselmaterial daraus zu gewinnen. Die x-Koordinate ist im Gegenteil sehr wenig redundant, denn nach dem Satz von Hasse ist ca. jede zweite Zahl potenziell x-Koordinate eines Punkts einer elliptischen Kurve.

[5] Vergleiche Abbildung 3.4

[6] Im Anhang in Abschnitt A.7 finden sich Details zum Berechnen von Quadratwurzeln.

Signed Double-&-Add

Anders als bei der Gruppe $\mathbb{Z}_p^*$ ist bei elliptischen Kurven der Aufwand für die Berechnung eines inversen Elements vernachlässigbar klein, es muss ja nur das Vorzeichen der y-Koordinate des Punktes geändert werden. Damit lässt sich nun z.B. $62 \cdot P$ auch als $64 \cdot P - 2 \cdot P$ mit 6 Punktverdopplungen und 1 Punktaddition berechnet werden. Klassisches Double-&-Add braucht 5 Punktverdopplungen und 4 Punktadditionen, um $62 \cdot P = 32 \cdot P + 16 \cdot P + 8 \cdot P + 4 \cdot P + 2 \cdot P$ zu berechnen.

Projektive Koordinaten

Den meisten Aufwand bei Punktadditionen machen die Divisionen. Eine Möglichkeit, diese Divisionen (weitestgehend) zu vermeiden, ist die Verwendung von projektiven Koordinaten. Dabei werden Punkte nicht in der Form (x, y) gespeichert, sondern in der Form $(X : Y : Z)$. Ein Punkt $(X : Y : Z)$ mit $Z \neq 0$ steht dann stellvertretend für den Punkt $(X/Z, Y/Z)$. Die Koordinaten werden also sozusagen als Brüche (mit dem gleichen Nenner Z) gespeichert. Umgekehrt ist die Darstellung eines Punktes (x, y) in projektiven Koordinaten $(X : Y : Z)$ nicht eindeutig, denn $(\lambda X : \lambda Y : \lambda Z)$ ist ebenfalls Repräsentant für (x, y) für jedes $\lambda \neq 0$. Der Punkt ∞ wird in projektiven Koordinaten als $(0 : Y : 0)$ mit beliebigem $Y \neq 0$, am einfachsten als $(0 : 1 : 0)$, dargestellt. Dies hat auch zur Folge, dass das neutrale Element keiner Sonderbehandlung bedarf, alle Punkte haben jetzt die selbe Darstellung.

Das Umwandeln von normalen Koordinaten[7] in projektive ist einfach:

$$\begin{aligned}(x, y) &\rightarrow (x : y : 1)\\ \infty &\rightarrow (0 : 1 : 0).\end{aligned}$$

Umgekehrt ist die Berechnung eines Kehrwerts erforderlich:

$$\begin{aligned}(X : Y : Z) &\rightarrow (XZ^{-1}, YZ^{-1})\\ (0 : Y : 0) &\rightarrow \infty.\end{aligned}$$

Für die Summe zweier Punkte $P := (X_1 : Y_1 : Z_1)$ und $Q := (X_2 : Y_2 : Z_2)$ in projektiven Koordinaten erhält man Formeln, die ohne aufwendige Divisionen auskommen. Als Beispiel ist in Abbildung 3.5 die Formel für die Addition zweier Punkte P und Q (im Fall $P \neq \pm Q$) dargestellt.[8]

Damit lassen sich Punkte mit zwölf Multiplikationen und zweimaligem Quadrieren, aber ohne eine Division addieren. Für die Punktverdopplung lassen sich analog Formeln bestimmen.

Das Ergebnis liegt nun in projektiven Koordinaten vor und somit kann sofort damit (z. B. im Rahmen einer Punktmultiplikation mittels Double-and-Add) weitergerechnet werden. Das Endergebnis muss schließlich in affine Koordinaten umgerechnet werden (projektive Koordinaten

[7] Normale Koordinaten werden in der Literatur oft auch als affine Koordinaten bezeichnet.

[8] Eine kurze Herleitung ist in Anhang A.5 zu finden.

$$
\begin{aligned}
u &:= Y_2Z_1 - Y_1Z_2,\\
v &:= X_2Z_1 - X_1Z_2,\\
w &:= u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2,\\
P+Q &:= (vw \ : \ u(v^2X_1Z_2 - w) - v^3Y_1Z_2 \ : \ v^3Z_1Z_2).
\end{aligned}
$$

Abbildung 3.5: Addition der Punkte $P := (X_1 : Y_1 : Z_1)$ und $Q := (X_2 : Y_2 : Z_2)$ in projektiven Koordinaten.

könnten etwas über die durchgeführten Berechnungen verraten). Dazu ist abschließend die Berechnung eines Kehrwerts erforderlich. Man beachte aber, dass hier nur eine Kehrwertberechnung pro Punktmultiplikation erforderlich ist und nicht mehr für jede Punktaddition.

Montgomery Curves

Montgomery Curves haben eine etwas andere Kurvengleichung:

$$By^2 = x^3 + Ax^2 + x \pmod{p}$$

Formeln für die Addition von Punkten lassen sich analog zu vorher herleiten. Speziell für die Punktmultiplikation ergeben sich unter Verwendung von projektiven Koordinaten sehr effiziente Formeln. Insbesondere können in diesen Formeln die y-Koordinaten der Punkte ignoriert werden.

Sind $(X_1 : \_ : Z_1)$ die projektiven Koordinaten[9] von P, $(X_n : \_ : Z_n)$ die projektiven Koordinaten des Punkts $n \cdot P$ und $(X_{n+1} : \_ : Z_{n+1})$ die projektiven Koordinaten des Punkts $(n+1) \cdot P$, dann lassen sich die projektiven Koordinaten $(X_{2n+1} : \_ : Z_{2n+1})$ und $(X_{2n} : \_ : Z_{2n})$ der Punkte $(2n+1) \cdot P$ und $(2n) \cdot P$ daraus wie folgt berechnen:

$$
\begin{aligned}
X_{2n+1} &= Z_1\big((X_{n+1} - Z_{n+1})(X_n + Z_n) + (X_{n+1} + Z_{n+1})(X_n - Z_n)\big)^2,\\
Z_{2n+1} &= X_1\big((X_{n+1} - Z_{n+1})(X_n + Z_n) - (X_{n+1} + Z_{n+1})(X_n - Z_n)\big)^2.
\end{aligned}
$$

[9] Die projektiven Y-Koordinaten werden in den Formeln überhaupt nicht benötigt. Wie bereits zuvor besprochen, sind die Y-Koordinaten auch für das Endergebnis nicht von Interesse, da sie sehr redundant sind. Sie werden daher hier einfach weggelassen.

$$
\begin{aligned}
X_{2n} &= p_n \cdot q_n, \\
Z_{2n} &= (p_n - q_n)\big(q_n + \frac{A+2}{4}(p_n - q_n)\big), \text{ wobei} \\
p_n &= (X_n + Z_n)^2, \\
q_n &= (X_n - Z_n)^2.
\end{aligned}
$$

Nutzt man zum Berechnen von Vielfachen eines Punktes die unter dem Namen „Montgomery Ladder“ bekannte Methode, so werden stets nur die beiden angegebenen Formeln benötigt. Details dazu finden sich im Anhang in Abschnitt A.6 und in [Mon87].

Beliebt sind Montgomery Curves auch deswegen, weil die Dauer der Punktmultiplikation $\alpha \cdot G$ mit der Montgomery Ladder nur von der Bitlänge von α, aber nicht von den konkreten Bits von α abhängt. Wird ganz normal via Double-and-Add multipliziert, entsteht aber so eine Abhängigkeit, denn in Abhängigkeit von den Bits von α werden Add-Schritte durchgeführt oder nicht. Außerdem führen die Unterschiede in den Formeln für Punktaddition und Punktverdopplung zu unterschiedlichen Berechnungsdauern. Solche Abhängigkeiten erlauben Seitenkanalangriffe (in diesem Fall Timing Attacks).

Curve25519 ist eine Montgomery-Kurve, für welche die Berechnungen besonders effizient implementiert werden können. Dafür werden als Parameter

$$
\begin{aligned}
p &= 2^{255} - 19 \qquad \text{(daher der Name)} \\
A &= 486662 \qquad \text{und} \\
B &= 1
\end{aligned}
$$

gewählt. Als Basispunkt für ECDH dient der Punkt $G = (9 : \_ : 1)$ mit der primen Ordnung

$$\omega = 2^{252} + 27742317777372353535851937790883648493.$$

Die Primzahl $p = 2^{255} - 19$ bietet für das Rechnen einige Vorteile, z. B. ist $2^{256} = 38 \pmod{p}$, womit sich die Reste modulo p für große Zahlen recht einfach (und damit schnell) berechnen lassen. Details finden sich in [Ber06]. Leider ist die Ordnung der Kurve nicht ω, sondern $8 \cdot \omega$. Es gibt auf dieser Kurve also auch Punkte der Ordnung 1, 2, 4 und 8, die in Bezug auf das DLP problematisch sind. In Protokollen ist es daher ggf. erforderlich, zu testen, ob solche Punkte auftauchen. Glücklicherweise ist der Aufwand für so einen Test nicht sehr hoch (3 Punktverdopplungen).

Rückblick 3.2

Du hast in diesem Kapitel elliptische Kurven kennen gelernt, die aktuell als Basis für effiziente Public-Key-Verfahren verwendet werden. Du kannst für elliptische Kurven über den reellen Zahlen geometrisch erklären, wie Punktoperationen (Addition, Verdopplung) funktionieren. Du kennst praktische Formeln für die Punktoperationen. Du weißt, dass elliptische Kurven Gruppen ergeben, die für kryptographische Verfahren verwendet werden können, die auf dem DLP oder DHP basieren. Du kannst erklären, weshalb die auf elliptischen Kurven basierenden Verfahren effizienter sind. Du kennst weiterhin Methoden, um das Arbeiten mit elliptischen Kurven noch schneller zu machen. Insbesondere kannst du mit projektiven Koordinaten arbeiten und du hast gelernt, wie Punktmultiplikationen mit Montgomery Curves (bspw. Curve25519) noch effizienter durchgeführt werden können. Du kennst auf elliptischen Kurven basierende Verfahren wie ECDH, ECDSA und Ed25519. Du kannst einem vorgegebenen Sicherheitsniveau geeignete Schlüssellängen für alle in dieser LVA behandelten Verfahren zuordnen. Darüber hinaus bist du in der Lage, die verschiedenen Verfahren in Hinblick auf Sicherheit, Schlüssellängen und Performance miteinander zu vergleichen und das am besten geeignete Verfahren für eine konkrete Anwendung auszuwählen.

Kapitel 4

Hashfunktionen

Ziele 4.1

In diesem Kapitel lernst du,

- wie sich mit Merkle Trees die Integrität von großen Datenmengen in manchen Fällen einfacher überprüfen lässt.
- wie mit Hashfunktionen One-Time-Signaturverfahren wie das Lamport-Verfahren oder Winternitz-Signaturen gebaut werden können.
- wie aus One-Time-Signaturverfahren Signaturverfahren mit Zustand wie das XMSS-Verfahren erzeugt werden können.
- wie weiter – unter Verwendung von Few-Time-Signaturverfahren – auch zustandslose Signaturverfahren auf Basis von Hashfunktionen wie das Post-Quantum-Signaturverfahren SPHINCS+ entstehen.

4.1 Merkle Trees

Eine wichtige Anwendung von Hashfunktionen sind die von Ralph Merkle erfundenen und nach ihm benannten Merkle Trees oder Hash Trees. Merkle verwendete diese ursprünglich für ein Signaturverfahren[1], das Post-Quantum-Signaturverfahren SPHINCS+ verwendet ebenfalls Merkle Trees. Merkle Trees haben sich aber auch in vielen anderen Anwendungen als nützlich erwiesen.

Die Idee von Merkle Trees ist, für eine sehr große Anzahl an Datensätzen einen Hashwert zu berechnen, so dass damit die Integrität jedes einzelnen Datensatzes mit wenig Aufwand überprüft werden kann. Verwendet werden dazu binäre Bäume, d. h. jeder Knoten im Baum, der kein

[1] Merkle hat dieses Verfahren patentiert, aus diesem Grund hat sich das Interesse für dieses Signaturverfahren stark in Grenzen gehalten.

Blatt ist, hat genau zwei Kinder. Der Einfachheit halber gehen wir hier davon aus, dass die Anzahl der Datensätze eine Potenz von 2 ist. In diesem Fall lassen sich Merkle Trees besonders schön bauen.[2]

4.1.1 Erstellen eines Merkle Tree

Um zu $N = 2^n$ Datensätzen $d_0, \ldots, d_{N-1}$ einen Merkle Tree zu erstellen, geht man wie folgt vor (vergleiche Abbildung 4.1):

Schritt 1: Blätter des Merkle Tree berechnen: Zunächst wird jeder Datensatz d_i mit einer Hashfunktion h gehasht. Den Hashwert bezeichnen wir mit h_i (wobei wir für i die n-stellige Binärdarstellung verwenden). Diese Hashwerte sind die Blätter des Merkle Tree.

Schritt 2, 3, ...: Innere Knoten des Merkle Tree berechnen: Von den Blättern hin zur Wurzel des Baums werden nun die Knoten des Merkle Tree berechnet. Der Wert an einem inneren Knoten ergibt sich dabei als der Hashwert über die Konkatenation der Werte der beiden Kinder des Knotens.

Letzter Schritt: Merkle Root berechnen: Der Wert an der Wurzel des Baums – Merkle Root genannt – ergibt sich genau so wie die Werte der inneren Knoten.

4.1.2 Verfizieren eines Datensatzes

Sind alle verarbeiteten Datensätze bekannt, lässt sich auf die selbe Art und Weise die Merkle Root wieder berechnen. Jede Änderung eines oder mehrerer Datensätze ändert den Wert der Merkle Root.

Einfach einen Hashwert über alle Hashwerte zu berechnen hätte den selben Effekt und wäre viel einfacher. Interessant wird die Angelegenheit aber, sobald ein einzelner Datensatz überprüft werden soll. In diesem Fall kann – anstatt alle Datensätze (oder deren Hashwerte) – zu verwenden, auch gerade so viel Information geliefert werden, wie nötig ist, um zur Merkle Root hochzuhashen. Abbildung 4.2 zeigt, welche Informationen nötig sind, um den Datensatz d_3 zu überprüfen. Die drei Hashwerte h_{010}, h_{00} und h_1 reichen aus. Sie bilden den sogenannten Authentication Path.[3]

Es ist hier einfach zu erkennen, dass für $N = 2^n$ nicht alle 2^n Blätter des Baums benötigt werden, sondern nur n Hashwerte für den Authentication Path – einer auf jeder Ebene des Baums. Diese n Hashwerte zusammen mit dem Datensatz können also gegen die Merkle Root verglichen werden.

[2] Im Fall von anderen Anzahlen muss man ein wenig aufpassen, es ist dann nicht alles ganz so einfach hinzuschreiben, die Grundidee funktioniert aber weiterhin.

[3] Es ist noch zusätzlich Information nötig, damit der Datensatz mit den Werten des Authentication Path auch in der richtigen Reihenfolge zusammengesetzt wird.

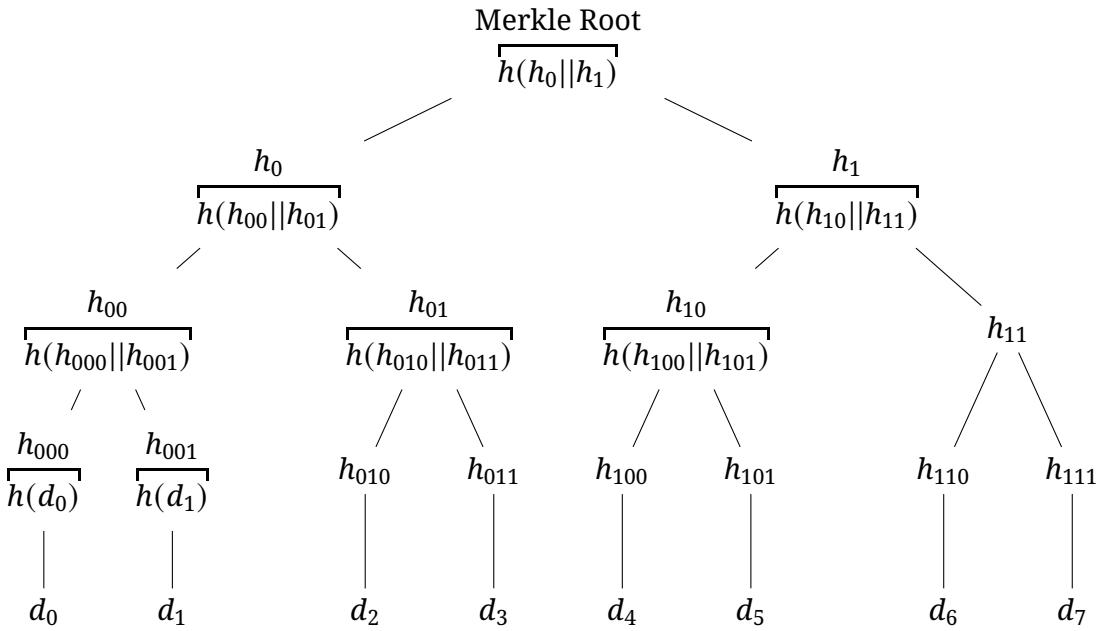


Abbildung 4.1: Ein Merkle-Tree für acht Datensätze $d_0, \ldots, d_7$.

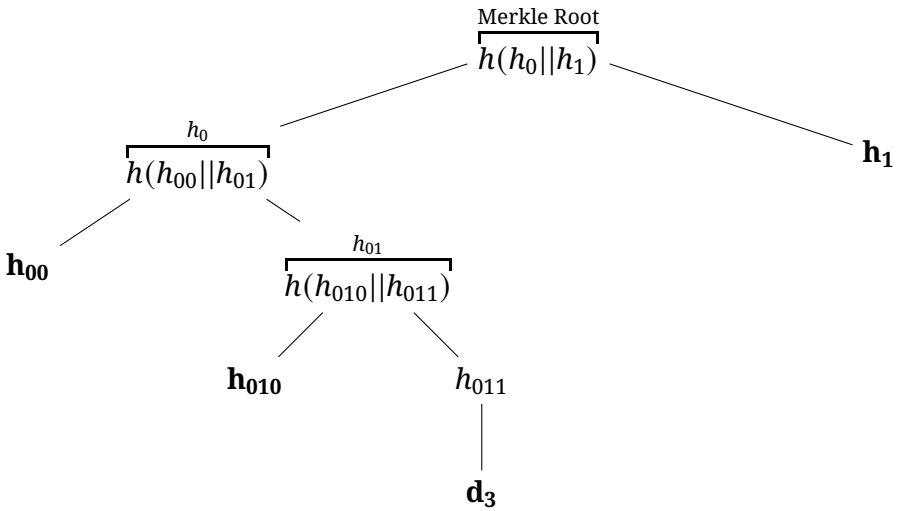


Abbildung 4.2: Verifikation des Datensatzes d_3 mit h_{010}, h_{00}, h_1 als Authentication Path.

Eine praktische Anwendung von Merkle Trees zur Integritätssicherung ist in Filesharing-Systemen zu finden. Dort werden (große) Dateien in gleich große Stücke aufgeteilt, die dann als einzelne Teile auch aus verschiedenen Quellen geladen und dann wieder zusammengebaut werden können. Einzelne Teile können durch Fehler bei der Übertragung oder vorsätzlich durch Filesharer verändert werden. Um derartige Veränderungen von einzelnen Teilen erkennen zu können, können Merkle Trees eingesetzt werden. Jeder Teil wird zu einem Blatt im Merkle Tree, die Merkle Root dient als Prüfsumme für die gesamte Datei, die zentral an einer vertrauenswürdigen Stelle heruntergeladen werden kann. Mit einem Teil wird nun auch sein Authentication Path heruntergeladen und dessen Integrität kann damit über die Merkle Root auch sofort überprüft werden. In Bittorent wurden ursprünglich in einem Torrent Hashwerte für alle Teile (Chunks) gespeichert. In Bittorrent Version 2 wird nur mehr die Merkle Root gespeichert.

4.2 One-Time Signatures

4.2.1 Lamport-Signaturen

Lamport stellte im Jahr 1979 ein Signaturverfahren vor, für das nur eine (kollisionsresistente) Hashfunktion benötigt wird. Bei diesem Verfahren kann mit einem Private Key nur einmal eine Signatur erstellt werden, es handelt sich um ein sog. One-Time-Signaturverfahren. Weiterhin lässt sich mit diesem Verfahren zunächst nur ein Bit signieren.

Schlüsselerzeugung

Wähle eine Hashfunktion h mit 256 Bit Outputlänge. Der Private Key Pr besteht aus zwei zufällig gewählten Bitfolgen s_0, s_1 der Länge 256. Der dazugehörige Public Key besteht aus den Hashwerten $p_0 := h(s_0)$ und $p_1 := h(s_1)$. Private Keys und Public Keys sind also je 512 Bit lang.

Signieren

Mit diesem Verfahren kann nur ein einzelnes Bit b signiert werden. Für $b = 0$ ist die Signatur s_0, für $b = 1$ ist die Signatur s_1, oder kürzer:

$$\mathrm{Sign}_{Pr}(b) = s_b.$$

Verifizieren

Um zu prüfen, ob die Signatur gültig ist, braucht nur ihr Hashwert berechnet zu werden. Für das Bit $b = 0$ sollte sich p_0, für das Bit $b = 1$ sollte sich p_1 ergeben.

Für längere Nachrichten geht man genauso vor, es wird Bit für Bit signiert, für jedes Bit ist jedoch ein weiteres Schlüsselpaar erforderlich. Um bspw. 256 Bit lange Hashwerte zu signieren,

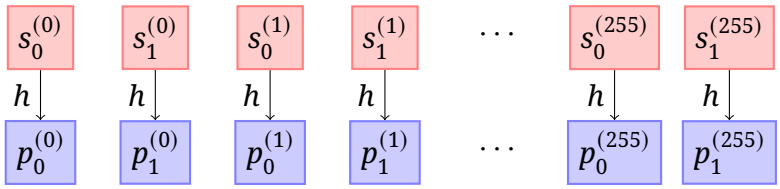


Abbildung 4.3: Berechnung des Public Keys $((p_0^{(0)}, p_1^{(0)}), \ldots, (p_0^{(255)}, p_1^{(255)}))$ aus dem Private Key $((s_0^{(0)}, s_1^{(0)}), \ldots, (s_0^{(255)}, s_1^{(255)}))$ im Lamport-Verfahren.

werden 256 mal 2×256 Bits benötigt (vgl. Abb. 4.3). Private Keys und Public Keys werden in diesem Fall 16 KiB groß.

Die Signatur zu einer 256 Bit langen Nachricht $b_0, \ldots, b_{255}$ ist dann

$$\mathrm{Sign}_{Pr}(b_0, \ldots, b_{255}) = s_{b_0}^{(0)}, s_{b_1}^{(1)}, \ldots, s_{b_{255}}^{(255)}.$$

Signaturen sind also 8 KiB groß.

Das Verfahren ist sehr schnell: Signieren macht überhaupt keinen Rechenaufwand, zum Verfizieren sind 256 Hashberechnungen erforderlich. Die Einschränkung, dass ein Schlüssel nur einmal verwendet werden kann, ist aber für viele Anwendungen ein Ausschlussgrund.

4.2.2 Winternitz One-Time Signatures (WOTS)

Von Winternitz stammt eine Idee, wie Schlüssel und Signaturen im Lamport-Verfahren verkleinert werden können.

Um auszudrücken, dass ein Wert s k-mal hintereinander gehasht wird, schreiben wir hier

$$h^k(s) := \underbrace{h(h(\ldots h(s)))}_{k\text{-mal}}.$$

Schlüsselerzeugung

In diesem Verfahren werden 67 256-Bit-Werte $s_0, \ldots, s_{66}$ als Private Key benötigt. Private Keys sind damit nur etwas über 2 KiB lang. Wiederum werden Public Keys durch Hashen aus Private Keys gewonnen, allerdings werden die Private Keys hier 15-mal hintereinander gehasht. Es ist also $p_i := h^{15}(s_i)$ für $i = 0, \ldots, 66$ (vgl. Abb. 4.4). Public Keys sind ebenfalls etwas über 2 KiB lang.

Signieren

Zum Signieren einer 256 Bit langen Nachricht m wird diese in 64 4-Bit-Stücke $m_0, \ldots, m_{63}$ zerlegt. Jeder dieser Werte wird als eine Zahl zwischen 0 und 15 interpretiert. Dann wird der Wert $c := 64 \cdot 15 - (m_0 + m_1 + \cdots + m_{63})$ berechnet. Diese Zahl liegt zwischen 0 und $64 \cdot 15$ und kann binär mit 12 Bits dargestellt werden. Diese zwölf Bits werden wieder in drei 4-Bit-Stücke c_0, c_1, c_2 zerlegt,

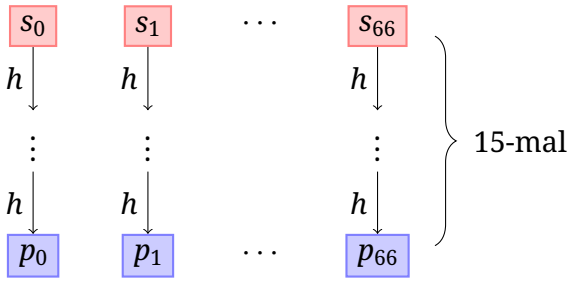


Abbildung 4.4: Berechnung des Public Keys $(p_0, p_1, \ldots, p_{66})$ aus dem Private Key $(s_0, s_1, \ldots, s_{66})$ im Winternitz-Verfahren.

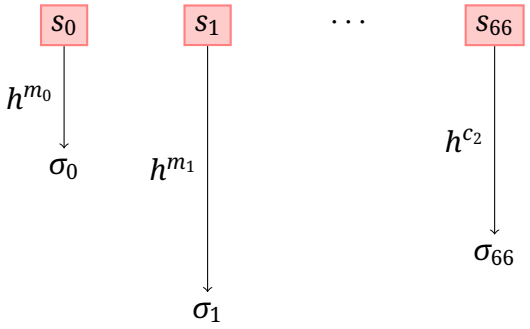


Abbildung 4.5: Signieren

die auch wieder als Zahlen zwischen 0 und 15 interpretiert werden. Insgesamt erhält man so 67 Werte zwischen 0 und 15.

Die Signatur ist nun (vgl. Abbildung 4.5)

$$\begin{aligned}\mathrm{Sign}_{Pr}(m) &:= (\sigma_0, \sigma_1, \ldots, \sigma_{66}) \\ &= (h^{m_0}(s_0), h^{m_1}(s_1), \ldots, h^{m_{63}}(s_{63}), h^{c_0}(s_{64}), h^{c_1}(s_{65}), h^{c_2}(s_{66}))).\end{aligned}$$

Signaturen sind somit ebenfalls etwas über 2 KiB lang.

Verfizieren

Zur Verifikation einer Signatur werden $m_0, \ldots, m_{63}$ und c_0, c_1, c_2 wie beim Signieren berechnet. Der erste Wert s_0 wurde beim Signieren m_0-mal gehasht und ergab den Wert σ_0 in der Signatur. Dieses σ_0 wird nun noch weitere $(15 - m_0)$-mal gehasht. Das Ergebnis wird mit p_0 verglichen (vgl. Abbildung 4.6). Für die übrigen Blöcke wird analog verfahren.

Die zusätzlichen Bits c_0, c_1, c_2 dienen hier als Prüfbits. Sieht man nämlich eine Signatur z. B. für eine Nachricht mit einem Block m_0, musste für diese Signatur s_0 genau m_0-mal gehasht werden. Es ist dann ganz einfach, eine Signatur für eine Nachricht zu erstellen, wo m_0 um 1 größer ist. Dazu muss man den m_0-mal gehashten Wert aus der Signatur nur noch einmal hashen. Der Wert c hilft gegen diesen Angriff. Wird nämlich einer der Werte m_i größer, dann auch deren Summe und damit wird c kleiner. Daher wird einer der Werte c_0, c_1, c_2 kleiner. Dort kann der passende Hashwert nicht mehr einfach berechnet werden.

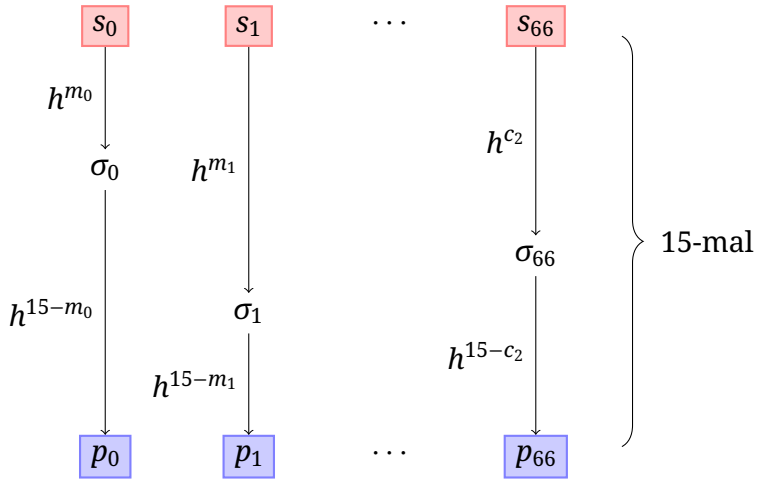


Abbildung 4.6: Verifikation der Signatur $(\sigma_0, \ldots, \sigma_{66})$.

4.3 Stateful Signatures

Merkle Trees können nun eingesetzt werden, um aus One-Time-Signaturverfahren gewöhnliche Signaturverfahren zu machen, indem eine große Anzahl an Schlüsselpaaren für One-Time-Signaturen vorbereitet und mit einer Merkle Root gesichert wird.

4.3.1 Merkle Signatures

Im Merkle-Signaturverfahren wird das Lamportverfahren mit einem Merkle Tree mit $N = 2^{32}$ Blättern kombiniert.

Schlüsselerzeugung

Es werden 2^{32} Lamport-Schlüsselpaare erzeugt. Die Lamport Private Keys werden als Signaturschlüssel verwendet. Die Lamport Public Keys sind die Datensätze für den Merkle Tree. Die Merkle Root ist der Public Key, der zum Verifizieren von Signaturen verwendet wird.

Anstatt die Private Keys zufällig zu erzeugen, werden sie mit einem PRNG aus einem zufällig gewählten Seed generiert. So können sie auch nach der Erzeugung der Merkle Root wieder gelöscht und für das Signieren aus dem Seed mit dem PRNG neu berechnet werden.

Signieren

Für das Signieren muss sichergestellt werden, dass keiner der Lamport Private Keys ein zweites Mal verwendet wird. Eine Möglichkeit ist, die Private Keys der Reihe nach zu verwenden. Dann braucht bloß ein Zähler als State gespeichert zu werden.

Das Signieren läuft wie beim Lamport-Verfahren ab. Zusätzlich wird der für die Verifikation zu verwendende Lamport Public Key an die Signatur angehängt, denn für die Verifikation liegt

nur die Merkle Root vor. Zusätzlich wird für diesen Datensatz im Merkle Tree auch noch der Authentication Path an die Signatur angehängt.

Verifizieren

Für die Verifikation wird zunächst die Lamport-Signatur mit dem mitgesendeten Lamport Public Key geprüft. Dann wird mit diesem Public Key und dem Authentication Path gegen die Merkle Root geprüft, die den Signatur-Public-Key bildet.

4.3.2 Weitere Verbesserungen

Das Merkle-Signaturverfahren wurde mehrfach erweitert und verbessert.

XMSS

Im XMSS-Verfahren (eXtended Merkle Signature Scheme, RFC 8391) [BDH11] konnte die Länge von Signaturen weiter verkleinert werden. Als One-Time-Signaturverfahren wird eine verbesserte Version des Winternitz-Verfahrens, das sog. WOTS+-Verfahren als One-Time-Signaturverfahren eingesetzt; das Verfahren wird so modifiziert, dass anstatt der Kollisionsresistenz der Hashfunktion nur deren schwache Kollisionsresistenz benötigt wird.

XMSS^{MT}

Die Multi-Tree-Idee von Goldreich erlaubt es, mit kleineren Merkle Trees zu arbeiten. Anstatt bspw. einen Baum der Tiefe 32 zu verwenden, für dessen Erstellung sofort alle 2^{32} Blätter verwendet und insgesamt $2^{33} - 1$ Hashwerte berechnet werden müssen, werden Bäume der Tiefe 16 verwendet. Von diesen Bäumen werden zu jedem Zeitpunkt nur zwei benötigt, was die Aufwände stark reduziert.

Die Idee von Goldreich ist, zunächst einen Merkle Tree der Tiefe 16 für 2^{16} One-Time Public Keys zu erstellen. Dann werden die ersten 2^{16} eigentlichen One-Time-Schlüsselpaare erstellt und für die 2^{16} Public Keys wie im Merkle-Verfahren die Merkle Root berechnet. Diese Merkle Root wird mit dem ersten Schlüssel aus dem ersten Baum signiert.

Beim Signieren wird wie im Merkle-Verfahren vorgegangen. Als Signatur werden die One-Time-Signatur der zu signierenden Daten, der Authentication Path und die Merkle Root des verwendeten (unteren) Trees, sowie die One-Time-Signatur über dessen (untere) Merkle Root und der Authentication Path für den (oberen) Merkle Tree verwendet.

Die Prüfung solcher Signaturen läuft wie die Prüfung einer Zertifikatskette ab. Zunächst kann überprüft werden, dass die Merkle Root korrekt ist. Dann kann mit dieser Merkle Root die Signatur über die Daten (wie im Merkle-Verfahren) überprüft werden.

Der große Vorteil liegt darin, dass anstatt mit einem großen Merkle Tree mit zwei wesentlich kleineren Merkle Trees gearbeitet werden kann. Die kleineren Bäume haben um den Faktor $2^{16} = 65536$ weniger Blätter, was alle Berechnungen wesentlich schneller macht.

Sobald 2^{16} Signaturen erstellt worden sind, ist der erste (untere) Baum aufgebraucht. Nun wird der zweite Public Key aus dem oberen Baum verwendet, um die Merkle Root eines zweiten (unteren) Baums zu signieren.

Insgesamt stehen $2^{16} \cdot 2^{16}$, also wieder 2^{32}, Signaturschlüsselpaare zu Verfügung.

Diese Idee kann auch mit mehr als zwei Ebenen von Bäumen umgesetzt werden. Das XMSS^{MT} (eXtended Merkle Signature Scheme with Multiple Trees, RFC 8391) verwendet bspw. vier Ebenen jeweils mit Bäumen der Tiefe 20. Damit werden 2^{80} Signaturen möglich, im Grunde also eine unbegrenzte Menge an Signaturen. Dennoch muss stets nur mit vier Bäumen mit jeweils 2^{20} Blättern gearbeitet werden.

HORS und HORST

HORS (Hash to Obtain a Random Subset) ist ein Few-Time-Signaturverfahren. Der selbe Private Key kann mehr als einmal, aber nicht beliebig oft verwendet werden.

Private Keys sind in diesem Fall 2^{16} 256-Bit-Werte $s_0, \ldots, s_{65535}$, also 2 MiB groß. Public Keys $p_0, \ldots, p_{65535}$ ergeben sich (wie im Lamport-Verfahren) durch einmaliges Hashen dieser 256-Bit-Werte, auch sie sind 2 MiB groß.

Um einen 256 Bit langen (Hash-)Wert m zu signieren, wird er in 32 16-Bit-Blöcke $m_0, \ldots, m_{31}$ zerlegt. Diese werden als Zahlen zwischen 0 und 65535 interpretiert.

Die Signatur von m ist dann

$$\mathrm{Sign}_{Pr}(m) = (s_{m_0}, \ldots, s_{m_{31}}).$$

Signaturen sind somit 1 KiB groß.

Auch dieses Signaturverfahren wird verwendet, um damit Hashwerte zu signieren. Ein Problem mit der Unfälschbarkeit ergibt sich, wenn ein Schlüssel zum Signieren von zwei Hashwerten verwendet wird, bei denen gleiche 16-Bit-Blöcke vorkommen. Die Parameter sind hier so gewählt, dass dies bei weniger als 128 Nachrichten mit einer Wahrscheinlichkeit von unter 2^{-128} passiert. Solange also wenige Werte signiert werden, treten praktisch keine Kollisionen auf (Few-Time Signature).

Das HORS-Verfahren kann zum HORST-Verfahren (HORS with Trees) erweitert werden, indem ein Merkle Tree der Tiefe 16 für die 2^{16} Public Keys verwendet wird.

SPHINCS+

Das SPHINCS+-Verfahren (Stateless Practical Hash-based Incredibly Nice Cryptographic Signatures; [BHH+14]) ist zudem stateless, braucht also keinen Zustand zu speichern, was es in der praktischen Anwendung flexibler macht.

Dazu wird anstatt eines One-Time-Signaturverfahrens zum Signieren einer Nachricht, das HORST-Few-Time-Signaturverfahren eingesetzt. Welcher Schlüssel zum Signieren einer Nachricht verwendet wird, wird nicht in einem State festgehalten, stattdessen wird er zufällig ausgewählt. Somit kommen Schlüssel u.U. auch öfter zum Einsatz, was aber bei dem Few-Time-Signaturverfahren kein Problem mehr darstellt.

Die vielen Public Keys werden wieder in Merkle Trees gesammelt. Mit der Idee von Goldreich wird wieder eine Kaskade von Merkle Trees mit WOTS+-Signaturen über Merkle Roots verwendet.

SPHINCS+ gehört zu den ersten Post-Quantum-Signaturverfahren, die 2022 vom NIST zur Standardisierung ausgewählt wurden.[4]

Rückblick 4.2

In diesem letzten Kapitel hast du dich mit weiterführenden Anwendungen von kryptographischen Hashfunktionen beschäftigt. Du weißt, wie man mit Merkle Trees (Hash-Bäumen) arbeitet und wie solche Bäume verwendet werden können, um effizient die Integrität großer Datenmengen zu sichern. Du kannst die Einschränkungen von One-Time-Signaturen und Few-Time-Signaturen erklären und weißt, wie Merkle Trees verwendet werden können, um aus solchen Verfahren vollständige Signaturverfahren gemacht werden können. Du kannst stateful und stateless Signaturverfahren unterscheiden. Mit dem SPHINCS+-Verfahren kennst du ein Post-Quantum-Signaturverfahren, dessen Sicherheit allein auf der Kollisionsresistenz einer Hashfunktion beruht.

[4] S. https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022

Kapitel 5

Kryptographische Verfahren auf Basis von Polynomen

Ziele 5.1

In diesem Kapitel lernst du,

- wie sich mit Polynomen Restklassen bilden lassen.
- wie man so wieder endliche Körper – die sog. Galoisfelder – erhält.
- wie verschiedene Galoisfelder in der Blockchiffre AES und im Galois Counter Mode (GCM) zum Einsatz kommen.
- wie mit Restklassen von Polynomen Vektoren und Matrizen gebildet werden und wie diese im Post-Quantum-Signaturverfahren CRYSTALS-Dilithium und im Schlüsselaustauschverfahren CRYSTALS-Kyber zur Anwendung kommen.

In diesem Kapitel geht es aus mathematischer Sicht um die sogenannten Galoisfelder[1] oder endlichen Körper. Ein wichtiger und bekannter Vertreter ist die Menge $\mathbb{Z}_p$ der Restklassen modulo einer Primzahl p. Es ist dies ein sogenannter Primkörper. In diesem Kapitel tauchen weitere wichtige endliche Körper auf.

5.1 Polynome

Polynome, im Speziellen Polynome über den Körpern $\mathbb{R}$ und $\mathbb{Z}_p$, waren bereits Thema in der LVA „Diskrete Mathematik und lineare Algebra“ (Kapitel 6). Von den Grundrechenarten (Addition, Subtraktion, Multiplikation, Division mit Rest, Euklidscher Algorithmus, größter gemeinsamer

[1] sprich: Galoah, nicht wie die Zigaretten

Teiler) war dort bereits die Rede. In diesem Abschnitt werden ein paar weitere Erkenntnisse über diese Polynome ergänzt.

Für das Folgende soll K ein beliebiger Körper sein. Wie in $\mathbb{Z}$ gibt es auch in $K[x]$ Elemente, die sich in Produkte zerlegen lassen und andere, die dies nicht erlauben.

Definition 5.1

Ein Polynom $f \in K[x]$ heißt reduzibel, wenn es Polynome g und h gibt, so dass $p = g \cdot h$ und die Grade der Polynome g und h kleiner sind als der Grad von f. Andernfalls heißt f irreduzibel.

Die irreduziblen Polynome sind sozusagen die Primzahlen in $K[x]$. Wie in $\mathbb{Z}$ ist auch in $K[x]$ eine eindeutige Primfaktorzerlegung möglich.

5.2 Endliche Körper

In diesem Abschnitt überlegen wir, ob nicht auch die Sache mit den Restklassen in $K[x]$ funktioniert. Wir definieren (diesmal für $f, g, h \in K[x]$):

$$g = h \pmod{f} \text{ genau dann, wenn } f \mid g - h \tag{5.1}$$

Wirklich stellt sich heraus, dass der Satz über das Rechnen mit Restklassen mutatis mutandis wieder gilt. Für die Menge der Restklassen modulo f schreiben wir $K[x]/(f)$. Bevor es weitergeht, sehen wir uns das einmal an. Wir nehmen uns dazu ein letztes Mal $\mathbb{R}[x]$ vor.

Beispiel 5.2

Das Polynom $f := x^2 + 1$ ist irreduzibel in $\mathbb{R}[x]$. Wir rechnen in $\mathbb{R}[x]/(x^2+1)$.

Addition $[(4+3x)]_f + [(3-x)]_f = [4+3x+3-x]_f = [7+2x]_f$.

Multiplikation $[(4+3x)]_f \cdot [(3-x)]_f = [(4+3x)(3-x)]_f = [12+5x-3x^2]_f$. Hier passiert etwas: wir können den Rest bei Division durch f berechnen. Also

$$\begin{array}{lll} -3x^2+5x+12 & : & x^2+1 = -3 \\ -3x^2 \qquad -3 & & \\ \qquad\quad 5x+15 & \text{Rest} & \end{array}$$

Also ist $[(4+3x)]_f \cdot [(3-x)]_f = [5x+15]_f$.

Division $\frac{[4+3x]_f}{[3-x]_f} = ?$.

Auch keine Überraschung: erweiterter Euklidscher Algorithmus wie in $\mathbb{Z}_p$.

	x^2+1	$3-x$	
x^2+1	1	0	
$-x+3$	0	1	$-x-3$
10	**1**	$\mathbf{3+x}$	$-\frac{1}{10}x+\frac{3}{10}$
0			

Gut, der ggT ist 1 und $[3-x]_f^{-1} = [\frac{1}{10}(3+x)]_f$. Also ist

$$\frac{[4+3x]_f}{[3-x]_f} = [(4+3x)]_p \cdot [\frac{1}{10}(3+x)]_f = \ldots = [\frac{9}{10}+\frac{13}{10}x]_f.$$

Das Berechnen der Reste ist etwas mühsam, es geht auch bequemer, wie, das wissen wir schon längst. Schaut man genauer hin, so bemerkt man, dass wir hier $\mathbb{C}$, den Körper der komplexen Zahlen, konstruiert haben. Wir brauchen statt x lediglich i zu schreiben. Rechnet man modulo f, so ist $x^2+1 = 0 \pmod f$, oder mit i: $i^2+1 = 0 \pmod f$, bzw. $i^2 = -1 \pmod f$. Jetzt ist alles klar.

Man kann es auch so sehen: wir haben einen neuen Körper konstruiert, indem wir zu $\mathbb{R}$ einfach ein neues Element i hinzugefügt haben. Man schreibt daher auch oft $\mathbb{R}(i)$ für diesen Körper und nennt $\mathbb{R}(i)$ Erweiterungskörper von $\mathbb{R}$. In $\mathbb{R}(i)$ ist f nicht mehr irreduzibel, denn nun ist $x^2+1 = (x+i)(x-i)$. Also „zerfällt“ f jetzt in die Linearfaktoren $(x+i)$ und $(x-i)$, weswegen der Körper $\mathbb{R}(i)$ auch Zerfällungskörper von f genannt wird. In aller Kürze zusammengefasst haben wir

$$\mathbb{C} \triangleq \mathbb{R}(i) \triangleq \mathbb{R}[x]/(x^2+1).$$

Satz 5.3

Ist K ein Körper und f ein irreduzibles Polynom in $K[x]$, dann ist $K[x]/(f)$ ein Körper.

Das Gleiche versuchen wir jetzt mit dem Körper $K := \mathbb{Z}_2$.

Beispiel 5.4

Das Polynom $f = x^2+x+1$ ist irreduzibel in $\mathbb{Z}_2[x]$. Wie vorher rechnen wir am bequemsten, indem wir eine fiktive Lösung α der Gleichung $x^2+x+1 = 0$ zu $\mathbb{Z}_2$ hinzufügen, wir erhalten den Körper $\mathbb{Z}_2(\alpha) = \mathbb{Z}_2[x]/(x^2+x+1)$. Sehen wir uns diesen Körper an.

Wir verzichten gleich auf die Schreibweise als Restklassen und verwenden α. Die Elemente von $\mathbb{Z}_2(\alpha)$ sind:

$$0, 1, \alpha, \alpha + 1.$$

$\mathbb{Z}_2(\alpha)$ ist ein Körper mit 4 Elementen. Höhere Potenzen von α treten nicht auf, denn α ist eine Lösung von $x^2 + x + 1 = 0$, und daher ist $\alpha^2 = \alpha + 1$ (man beachte die Ähnlichkeit zu $\mathbb{C}$). In $\mathbb{Z}_2(\alpha)$ rechnet man wie folgt:

$+$	0	1	α	$\alpha+1$
0	0	1	α	$\alpha+1$
1	1	0	$\alpha+1$	α
α	α	$\alpha+1$	0	1
$\alpha+1$	$\alpha+1$	α	1	0

$\cdot$	0	1	α	$\alpha+1$
0	0	0	0	0
1	0	1	α	$\alpha+1$
α	0	α	$\alpha+1$	1
$\alpha+1$	0	$\alpha+1$	1	α

In einem dritten Beispiel soll auch gezeigt werden, wie in Python mit endlichen Körpern gearbeitet werden kann.

Beispiel 5.5

In diesem Beispiel wählen wir $K = \mathbb{Z}_5$ und $f = x^3 + 3x + 2$. Wir erhalten den Körper $\mathbb{Z}_5(\alpha) = \mathbb{Z}_5[x]/(x^3 + 3x + 2)$. In diesem Körper gilt $\alpha^3 = 2\alpha + 3$.

Jetzt sind alle Elemente von der Form

$$a + b\alpha + c\alpha^2, \text{ wobei } a, b, c \in \mathbb{Z}_5,$$

denn höhere Potenzen von α werden wir wieder los. So ist

$$\begin{aligned}
\alpha^3 &= 2\alpha + 3 \\
\alpha^4 &= \alpha \cdot \alpha^3 = \alpha(2\alpha + 3) = \\
&= 2\alpha^2 + 3\alpha \\
\alpha^5 &= \alpha^2 \cdot \alpha^3 = \alpha^2(2\alpha + 3) = \\
&= 2\alpha^3 + 3\alpha^2 = 2(2\alpha + 3) + 3\alpha^2 = \\
&= 3\alpha^2 + 4\alpha + 6 \text{ usw.}
\end{aligned}$$

Diesmal ergeben sich $5 \cdot 5 \cdot 5 = 125$ verschiedene Elemente.

Das Modul `galois` (`pip install galois`) stellt eine Klasse für endliche Körper und deren Elemente zur Verfügung.

```
> from galois import GF
> k = GF( 5**3, irreducible_poly="x^3+3x+2", repr="poly" )
```

```
> g = k("x^2+3x+1"); g
GF(α^2 + 3α + 1, order=5^3)
> g**2 * ( g**2 + 2*g )
GF(α + 2, order=5^3)
> g**(-1)
GF(α^2 + 2, order=5^3)
```

Auf die beschriebene Art kann man aus einem Körper $\mathbb{Z}_p$ einen Körper $\mathbb{Z}_p[x]/(f)$ basteln. Ist f ein irreduzibles Polynom vom Grad n, so hat der Körper $\mathbb{Z}_p[x]/(f)$ genau p^n Elemente. Wir haben also Körper mit p^n Elementen gefunden. Dazu brauchen wir aber irreduzible Polynome. Es drängt sich die Frage auf, ob es in jedem $\mathbb{Z}_p[x]$ irreduzible Polynome von jedem beliebigen Grad n gibt.

Satz 5.6

Für jedes $p \in \mathbb{P}$ und jedes $n \in \mathbb{N}$ gibt es ein irreduzibles Polynom $f \in \mathbb{Z}_p[x]$ vom Grad n und somit einen endlichen Körper mit p^n Elementen. Wir nennen diesen Körper $\mathrm{GF}(p^n) := \mathbb{Z}_p[x]/(f)$, Galoisfeld mit p^n Elementen. Das sind alle endlichen Körper.

Besonders von Interesse in der Kryptographie sind die endlichen Körper $\mathrm{GF}(2^n)$. Bei der Blockchiffre AES garantieren sie einige wichtige Sicherheitseigenschaften (s. Abschnitt 5.3).

5.3 AES

AES setzt in seinem Design stark auf Operationen im endlichen Körper

$$\mathrm{GF}(2^8) = \mathbb{Z}_2[x]/(x^8 + x^4 + x^3 + x + 1) = \mathbb{Z}_2(\alpha),$$

wobei $\alpha^8 = \alpha^4 + \alpha^3 + \alpha + 1$ ist.

Ein Byte $b_7 b_6 \ldots b_0$ im AES State Array lässt sich auch verstehen als das Element $b_7\alpha^7 + b_6\alpha^6 + \ldots + b_0 \in \mathbb{Z}_2(\alpha)$. In dieser Interpretation ist dann die Summe zweier Bytes (in $\mathrm{GF}(2^8)$) nichts anderes als das bitweise XOR der Bytes. Die Multiplikation lässt sich allerdings auf Bitebene nicht mehr einfach erklären. Dies ist ein wichtiger Punkt, der die Analyse des Verfahrens schwierig und AES gegen viele Arten von Angriffen resistent macht.

An zwei Stellen setzt AES auf den endlichen Körper $\mathrm{GF}(2^8)$: einmal beim Mischen der Spalten (MixColumn) und einmal für den Aufbau der S-Box (SubBytes).

Abbildung 5.1 zeigt die Struktur des Algorithmus.

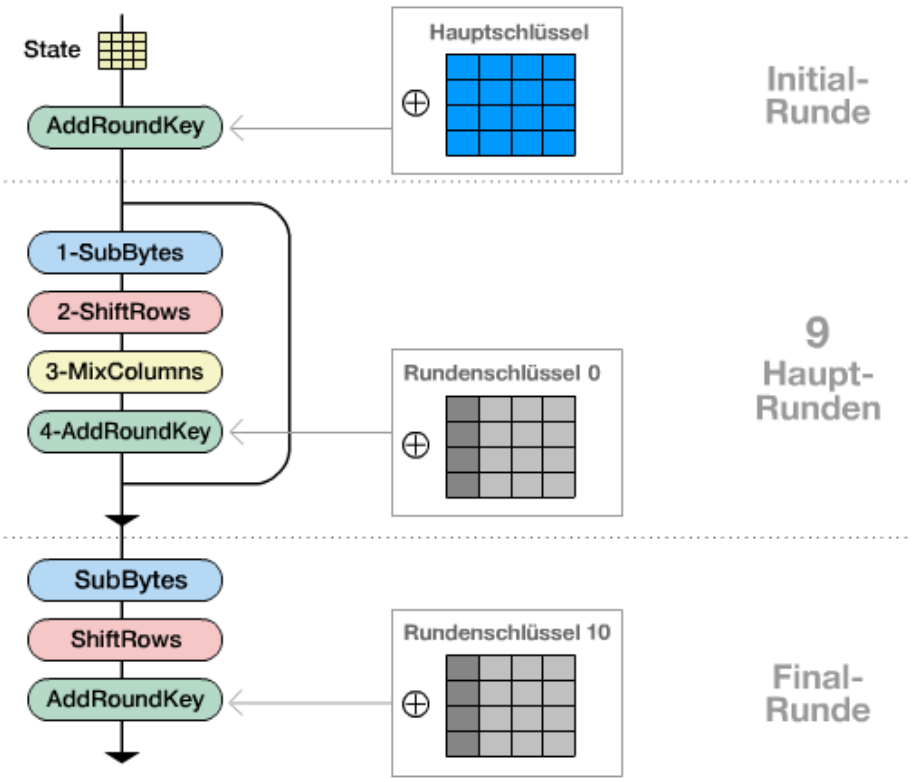


Abbildung 5.1: AES im Überblick.[2]

5.3.1 MixColumns

Im MixColumns-Schritt wird das State Array als eine 4×4-Matrix S mit Elementen aus $\mathrm{GF}(2^8)$ als Einträgen interpretiert und gemäß

$$S \leftarrow \begin{pmatrix} \alpha & \alpha+1 & 1 & 1 \\ 1 & \alpha & \alpha+1 & 1 \\ 1 & 1 & \alpha & \alpha+1 \\ \alpha+1 & 1 & 1 & \alpha \end{pmatrix} \cdot S$$

upgedatet. Die Multiplikation und Addition sind hier wieder im Körper $\mathrm{GF}(2^8)$ gemeint. Durch diese Kontruktion als Matrixmultiplikation lässt sich mathematisch einfach beweisen, dass Änderungen in wenigen Einträgen der State-Matrix S durch den MixColumns-Schritt stets zu vielen Änderungen in der State-Matrix führen. Diese Eigenschaft wird üblicherweise als Diffusion bezeichnet, sie sorgt dafür, dass schwerer vom Chiffrat auf den Klartext geschlossen werden kann.

[2] Quelle: formaestudio.com/rijndaelinspector/archivos/Rijndael_Animation_v4_eng-html5.html

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Abbildung 5.2: Die AES S-Box. 0x7c wird zu 0x10.

5.3.2 SubBytes

Die S-Box von AES ist in Abbildung 5.2 zu sehen. Diese Tabelle wird verwendet, um byteweise die Einträge im State Array zu verändern.[3]

Die Werte in Abbildung 5.2 sind nicht willkürlich gewählt. Aus einem Byte $b_7 b_6 \dots b_0$ wird in zwei Schritten ein Byte $s_7 s_6 \dots s_0$. Dabei wird zunächst das Byte $b_7 b_6 \dots b_0$ als Element von $\mathrm{GF}(2^8)$ interpretiert und dort sein Kehrwert berechnet, also

$$(b_7\alpha^7 + \ldots + b_0)^{-1} = y_7\alpha^7 + \ldots + y_0 \text{ in } \mathrm{GF}(2^8).$$

Dann wird das Ergebnis als ein 8-dimensionaler Vektor über dem Körper $\mathbb{Z}_2$ interpretiert und dort der affinen Transformation[4]

$$\begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

[3] Der Vollständigkeit halber sei ergänzt, dass die S-Box auch bei der Berechnung der Rundenschlüssel verwendet wird.

[4] Eine affine Transformation ist eine Multiplikation mit einer Matrix und anschließendes Addieren eines fixen Vektors.

unterworfen. Das Ergebnis wird dann wieder als ein Byte $s_7 s_6 \dots s_0$ gelesen.

Praktisch wird in aller Regel natürlich die Tabelle zum Nachschlagen verwendet, weil dies viel schneller geht.[5] Dennoch hat diese Konstruktion der S-Box ihr Gutes: Da man sie durch einfache Formeln beschreiben kann, lassen sich bestimmte Eigenschaften einfacher beweisen. So lässt sich beispielsweise nachweisen, dass diese S-Box optimal gegen differentielle Kryptanalyse schützt.

Beispiel 5.7

Hier rechnen wir den Wert für das Byte `7c` in der S-Box kurz nach. Zunächst wird im Körper $\mathbb{Z}_2[x]/(x^8 + x^4 + x^3 + x + 1)$ gerechnet.

```
> from galois import GF
> k = GF( 2**8, irreducible_poly="x^8+x^4+x^3+x+1" )
> b = k(0x7c)
> b**(-1)
GF(161, order=2^8)
> bin(161)
'0b10100001'
```

Das Ergebnis fassen wir nun als Vektor über $\mathbb{Z}_2$ auf und achten darauf, dass die Bits von rechts nach links gelesen werden müssen.

```
> z2 = GF(2)
> y = z2( [1,0,0,0,0,1,0,1] )
> m = z2([[1,0,0,0,1,1,1,1],
          [1,1,0,0,0,1,1,1],
          [1,1,1,0,0,0,1,1],
          [1,1,1,1,0,0,0,1],
          [1,1,1,1,1,0,0,0],
          [0,1,1,1,1,1,0,0],
          [0,0,1,1,1,1,1,0],
          [0,0,0,1,1,1,1,1]])
> c = z2( [1,1,0,0,0,1,1,0] )
> s = m@y + c; s
GF([0, 0, 0, 0, 1, 0, 0, 0], order=2)
```

Das Ergebnis (wieder von rechts nach links gelesen) ist `0b00010000` bzw. `0x10`.

[5] Mehr noch: Zumeist wird die MixColumns-Operation gleich noch in die SubBytes-Operation integriert. Das führt zu größeren Tabellen, aber deutlich höherer Geschwindigkeit.

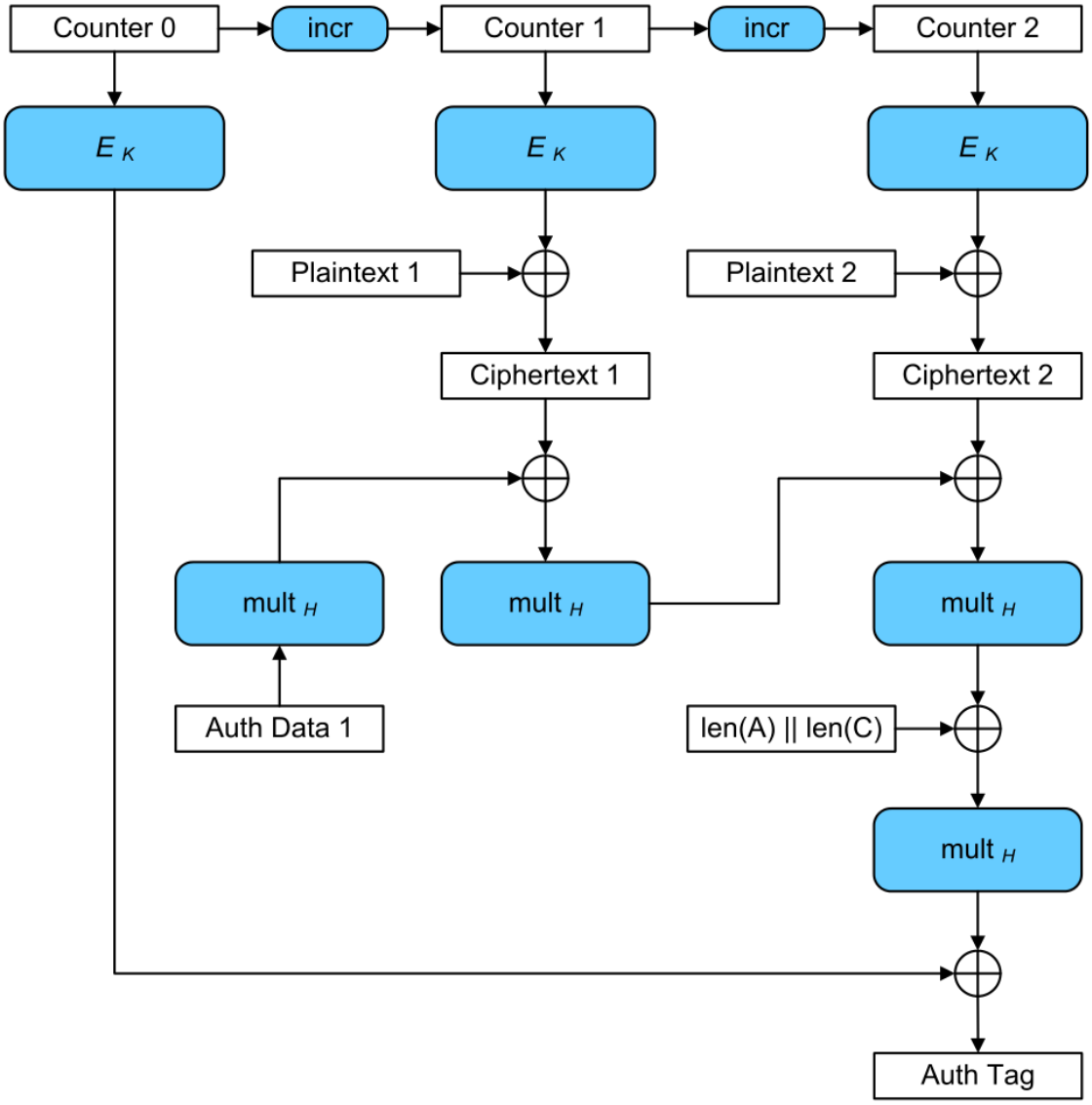


Abbildung 5.3: Der Galois Counter Mode. Quelle: Wikimedia Commons (NIST)

5.4 Der Galois Counter Mode (GCM)

In den Grundlagen der Kryptographie wurde bereits der Galois Counter Mode als ein Mode of Operation angeführt, der Authenticated Encryption und damit CCA-sichere Verschlüsselung erlaubt. Dieser Mode soll an dieser Stelle kurz genauer betrachtet werden.

Abbildung 5.3 zeigt den grundsätzlichen Aufbau des GCM. Es lässt sich erkennen, dass es sich zunächst um einen einfachen Counter Mode handelt. Zusätzlich wird aber über verschlüsselten Daten ein Authentication Tag berechnet, der von allen verschlüsselten Blöcken und zusätzlichen Daten (Authentication Data) abhängt. Die Berechnung dieses Tags wird über eine Multiplikation im Körper $\mathrm{GF}(2^{128})$ – in Abb. 5.3 als „mult_H“ dargestellt – durchgeführt.

Im Körper $\mathrm{GF}(2^{128})$ wird modulo $f := x^{128} + x^7 + x^2 + x + 1$ gerechnet, es werden also nach der

Regel $\alpha^{128} = \alpha^7 + \alpha^2 + \alpha + 1$ höhere Potenzen eliminiert. Die Bits werden wieder als Koeffizienten eines Polynoms verstanden. Die Operation mult_H multipliziert in $\text{GF}(2^{128})$ mit $H := E_K(0)$, d. h. der Wert, der für H verwendet werden soll, ergibt sich als die Verschlüsselung eines Blocks aus lauter 0-Bits mit dem aktuellen Schlüssel. Die XOR-Verknüpfungen mit den Chiffratblöcken sind Additionen in $\text{GF}(2^{128})$. Für die Berechnung des Tags sind damit nur sehr schnelle Multiplikationen erforderlich, was die Berechnung deutlich effizienter macht als die Berechnung eines HMAC oder CMAC. Details entnimmt man am besten dem Standard [Dwo07].

5.5 Post-Quantum-Public-Key-Verschlüsselung und Post-Quantum-Signaturen

5.5.1 Allgemeines

Als Ausklang in diesem Kapitel betrachten wir zwei der Verfahren, die 2022 vom NIST als erste in den Bereichen Key Encapsulation Mechanisms (KEM) und digitale Signaturen zur Standardisierung ausgewählt wurden.[6]

In beiden Verfahren wird mit Restklassen von Polynomen über $\mathbb{Z}_q$ ($q \in \mathbb{P}$) modulo dem Polynom $x^{256} + 1$, also in $\mathcal{R} := \mathbb{Z}_q[x]/(x^{256} + 1)$, gerechnet. Dieses Polynom ist nicht irreduzibel, Divisionen sind aber in diesem Fall auch gar nicht erforderlich.

Aus diesen Restklassen werden Vektoren ($\mathcal{R}^n$) und Matrizen ($\mathcal{R}^{m \times n}$) gebildet. Als Besonderheit werden Restklassen modulo q nicht als ganze Zahlen zwischen 0 und $q-1$ angeschrieben, sondern als ganze Zahlen zwischen $-q/2$ und $q/2$. Ist $p := a_0 + a_1 x + \cdots + a_{255} x^{255}$ ein Element von $\mathcal{R}$, dann wird dessen Norm definiert als der Betrag des Koeffizienten a_i mit dem größten Betrag. Bildet man einen Vektor solcher Polynome, so wird dessen Norm als die größte auftauchende Norm in den Koordinaten des Vektors definiert.

5.5.2 Das Problem

Die CRYSTALS-Verfahren beziehen ihre Sicherheit – wie alle Public-Key-Verfahren – aus der Schwierigkeit eines mathematischen Problems. In diesem Fall ist es das folgende:[7]

Gegeben sei eine Matrix M. Wählt man zwei Vektoren α und ε mit kleiner Norm (also kurze Vektoren), so lässt sich einfach $a := M \cdot \alpha + \varepsilon$ berechnen. Umgekehrt ist es aber sehr schwierig, kurze Vektoren α und ε zu finden, so dass $a = M \cdot \alpha + \varepsilon$, wenn M und a gegeben sind.

Beachte: Irgendwelche Vektoren $\hat{\alpha}$ und $\hat{\varepsilon}$ zu finden, so dass $a = M \cdot \hat{\alpha} + \hat{\varepsilon}$ gilt, ist sehr einfach: man wähle irgendeinen kurzen Vektor $\hat{\alpha}$ und berechne $\hat{\varepsilon} := a - M \cdot \hat{\alpha}$. Nur ist der Vektor $\hat{\varepsilon}$ dann

[6] https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022

[7] Genau genommen geht es um das sogenannte Module-Learning-With-Errors-Problem (MLWE). Eine recht zugängliche Einführung dazu findet sich bspw. auf https://mark-schultz.github.io/nist-standard-out/.

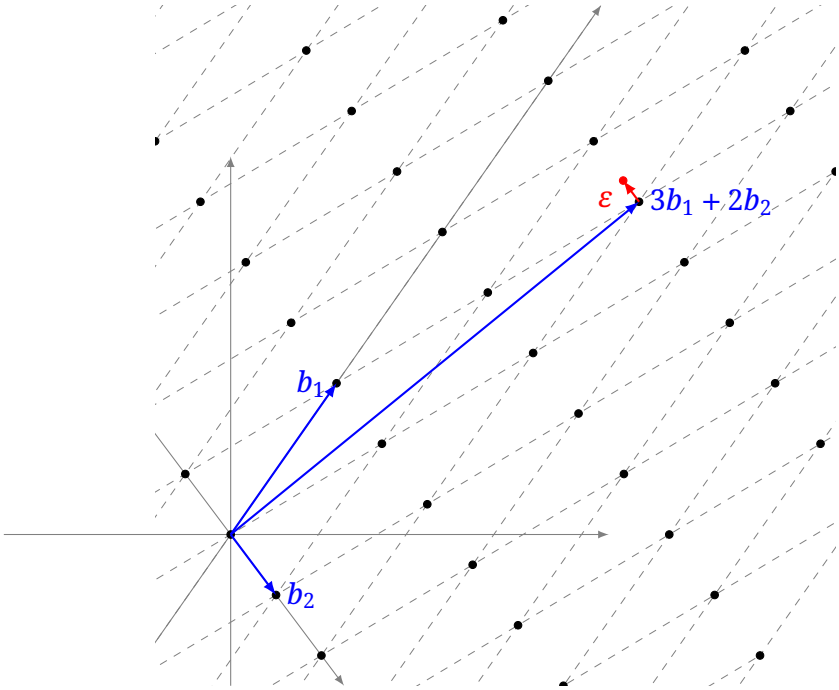


Abbildung 5.4: Das von den Vektoren b_1 und b_2 erzeugte Gitter und ein Punkt nah an einem Gitterpunkt

nicht kurz. Die Aufgabe, zwei **kurze** Vektoren α und ε zu finden, führt auf ein sogenanntes Gitterproblem. Diese Art von Problemen scheint selbst für Quantencomputer schwierig zu lösen zu sein. Darüber hinaus führt die Einschränkung, dass α und ε kurze Vektoren sind, (bei geeigneter Wahl der Parameter) dazu, dass das Gleichungssystem $a = M \cdot \alpha + \varepsilon$ eine eindeutige Lösung besitzt.

Abbildung 5.4 zeigt zur Veranschaulichung ein einfaches Gitter. Fasst man beiden Vektoren b_1 und b_2 zu einer Matrix $M := \begin{pmatrix} | & | \\ b_1 & b_2 \\ | & | \end{pmatrix}$ zusammen, so ergibt sich bspw. mit $\alpha := \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ $M \cdot \alpha = 3b_1 + 2b_2$, ein Punkt auf dem Gitter. Ist ε ein kurzer Vektor, dann ist $M \cdot \alpha + \varepsilon$ ein Punkt der knapp neben einem Gitterpunkt liegt. Das Closest Vector Probem (CVP) beschreibt die Schwierigkeit, den nächstgelegenen Gitterpunkt zu diesem Punkt (also den Vektor α) zu finden.

In den folgenden Abschnitten werden für kurze Vektoren griechische Buchstaben verwendet, damit diese einfacher zu erkennen sind.

5.5.3 Digitale Signatur: CRYSTALS-Dilithium

Das CRYSTALS-Dilithium-Verfahren verwendet die grundsätzliche Idee der Schnorr-Signaturen (vgl. Algorithmus 2.19). Das MLWE-Problem (vgl. Abschnitt 5.5.2) ersetzt hier das DLP. In dieser Beschreibung werden die für mittlere Sicherheit (128 Bit) empfohlenen Parameter verwendet. Einige technische Details, insbesondere wenn deren Zweck die Performanceoptimierung ist, wer-

den in dieser Beschreibung ausgelassen, um die zugrundeliegende Idee klarer darzustellen.

Algorithmus 5.8: CRYSTALS-Dilithium

In diesem Verfahren ist $q := 8380417$ und $\mathcal{R} := \mathbb{Z}_q[x]/(x^{256}+1)$.

Schlüsselerzeugung: Alice wählt eine zufällige Matrix $M \in \mathcal{R}^{6\times 5}$. Weiterhin wählt sie zwei Vektoren $\alpha \in \mathcal{R}^5$ und $\varepsilon \in \mathcal{R}^6$ zufällig, deren Norm höchstens 4 ist, d. h. alle auftretenden Koeffizienten der Polynome sind zwischen -4 und 4. Schließlich berechnet sie

$$a := M \cdot \alpha + \varepsilon.$$

Der Public Key ist dann (M, a). Der dazugehörige Private Key ist α.

Signieren: Alice möchte eine Nachricht m signieren (typischerweise ist das der Hashwert einer Nachricht). Sie wählt einen Vektor $k \in \mathcal{R}^5$ zufällig, dessen Norm höchstens 2^{19} ist. Alice berechnet nun $r := \text{high}(M \cdot k)$, die höchstwertigen Bits aller Koordinaten des Vektors $M \cdot k$ (direkt als Bitfolge interpretiert). Nun werden r und m zusammen gehasht, das Ergebnis wird kodiert als ein Polynom $\zeta \in \mathcal{R}$, das genau 49 Koeffizienten hat, die den Wert 1 oder -1 haben und dessen restliche Koeffizienten 0 sind. (Es wird hier nicht darauf eingegangen, wie dies genau geschieht.) Schließlich wird $s := k + \zeta \cdot \alpha$ berechnet.

Kompakter also:

$$\begin{aligned} r &:= \text{high}(M \cdot k), \\ \zeta &:= H(r, m), \\ s &:= k + \zeta \cdot \alpha. \end{aligned}$$

Die Signatur ist (ζ, s).

Verifizieren: Bob prüft, dass die Norm von s nicht zu groß ist und berechnet dann $r := \text{high}(M \cdot s - \zeta \cdot a)$. Damit wird $H(r, m)$ berechnet und abschließend mit ζ verglichen.

Tatsächlich erhält man beim Verifizieren dasselbe a wie beim Signieren, denn

$$M \cdot s - \zeta \cdot a = M \cdot (k + \zeta \cdot \alpha) - \zeta \cdot (M \cdot \alpha + \varepsilon) = M \cdot k + \cancel{\zeta \cdot M \cdot \alpha} - \cancel{\zeta \cdot M \cdot \alpha} - \zeta \cdot \varepsilon = M \cdot k - \zeta \cdot \varepsilon.$$

Da sowohl in ζ als auch in ε nur kleine Koeffizienten vorkommen, beeinflussen diese die höchstwertigen Bits nicht. Daher ist $r = \text{high}(M \cdot s - \zeta \cdot a) = \text{high}(M \cdot k - \zeta\varepsilon) = \text{high}(M \cdot k)$.

5.5.4 KEM: CRYSTALS-Kyber

Der Key Encapsulation Mechanism (KEM) CRYSTALS-Kyber soll symmetrische (256 Bit lange) Schlüssel für den Transport verschlüsseln. Im Folgenden wird die Variante KYBER768 für mittlere Sicherheit (128 Bit) beschrieben.

Algorithmus 5.9: CRYSTALS-Kyber

In diesem Verfahren ist $q := 3329$ und $\mathcal{R} := \mathbb{Z}_q[x]/(x^{256}+1)$.

Schlüsselerzeugung: Alice wählt eine zufällige Matrix $M \in \mathcal{R}^{3\times 3}$. Weiterhin wählt sie zwei Vektoren $\alpha \in \mathcal{R}^3$ und $\varepsilon \in \mathcal{R}^3$ zufällig, deren Norm höchstens 2 ist, d. h. alle auftretenden Koeffizienten der Polynome sind zwischen -2 und 2. Schließlich berechnet sie

$$a := M \cdot \alpha + \varepsilon.$$

Der Public Key ist dann (M, a). Der dazugehörige Private Key ist α.

Verschlüsseln: Um einen 256 Bit langen Schlüssel zu verschlüsseln, wird dieser zunächst als jenes Element $\kappa \in \mathcal{R}$ dargestellt, nämlich als jenes Polynom, dessen Koeffizienten die Schlüsselbits sind. Daraus erhält man $k := \lfloor q/2 \rfloor \cdot \kappa$. (Die Multiplikation von k mit $\lfloor q/2 \rfloor$ führt dazu, dass die Koeffizienten dieses Polynoms entweder den (betragsmäßig) kleinsten Wert 0 oder oder größten Wert $\lfloor q/2 \rfloor = 1664$ haben. Das hat zur Folge, dass kleine Fehler korrigiert werden können.)

Bob wählt Vektoren $\beta, \zeta \in \mathcal{R}^3$ sowie $\gamma \in \mathcal{R}$ zufällig, deren Norm höchstens 2 ist. Er berechnet nun

$$u := M^\intercal \cdot \zeta + \beta,$$
$$v := a^\intercal \cdot \zeta + k + \gamma.$$

Es ergibt sich als Chiffrat das Paar (u, v).

Entschlüsseln: Alice berechnet

$$k' := v - \alpha^\intercal \cdot u.$$

Beim Entschlüsseln ergibt sich die ursprüngliche Nachricht, denn

$$\begin{aligned}
k' &= v - \alpha^\intercal \cdot u \\
&= a^\intercal \cdot \zeta + k + \gamma - \alpha^\intercal \cdot (M^\intercal \cdot \zeta + \beta) && \text{(Einsetzen von } u \text{ und } v\text{)} \\
&= a^\intercal \cdot \zeta + k + \gamma - \alpha^\intercal \cdot M^\intercal \cdot \zeta - \alpha^\intercal \cdot \beta && \text{(Ausmultiplizieren)} \\
&= a^\intercal \cdot \zeta + k + \gamma - (M \cdot \alpha)^\intercal \cdot \zeta - \alpha^\intercal \cdot \beta && (\alpha^\intercal M^\intercal = (M\alpha)^\intercal) \\
&= a^\intercal \cdot \zeta + k + \gamma - (a - \varepsilon)^\intercal \cdot \zeta - \alpha^\intercal \cdot \beta && (a = M\alpha + \varepsilon) \\
&= \cancel{a^\intercal \cdot \zeta} + k + \gamma - \cancel{a^\intercal \cdot \zeta} + \varepsilon^\intercal \cdot \zeta - \alpha^\intercal \cdot \beta && \text{(Ausmultiplizieren)} \\
&= \lfloor q/2 \rfloor \cdot \kappa + \underbrace{\gamma + \varepsilon^\intercal \cdot \zeta - \alpha^\intercal \cdot \beta}_{\text{nur kleine Koeffizienten}} .
\end{aligned}$$

In den Ausdrücken γ, $\varepsilon^\intercal \cdot \zeta$ und $\alpha^\intercal \cdot \beta$ kommen nur kleine Koeffizienten vor. Werden diese Werte zum Polynom $\lfloor q/2 \rfloor \cdot \kappa$ addiert oder von diesem subtrahiert, können diese kleinen „Fehler“ sehr einfach korrigiert werden, um die Schlüsselbits zu erhalten.

Rückblick 5.2

In diesem Kapitel hast du dich mit den Galois-Feldern, den endlichen Körpern, beschäftigt. Du weißt, wie man Restklassen von Polynomen bildet und wie man mit diesen Restklassen rechnet. Insbesondere den (erweiterten) Euklidschen Algorithmus für Polynome beherrschst du. Du hast einen Eindruck davon bekommen, wie und wo endliche Körper im AES und für die Erstellung des Authentication Tag im Galois Counter Mode (GCM) eingesetzt werden. Mit den Post-Quantum-Verfahren CRYSTALS-Dilithium für digitale Signaturen und CRYSTALS-Kyber für Key Agreement kannst du zwei Public-Key-Verfahren aus dem Bereich der gitterbasierten Kryptographie beschreiben, die auch gegenüber Angriffen mit Quantencomputern Sicherheit bieten.

Anhang A

Details

A.1 Chinesischer Restatz

Hier der Beweis für den chinesischen Restsatz (für zwei Gleichungen).

Seien n_1 und n_2 ganze Zahlen mit $\text{ggT}(n_1, n_2) = 1$. Weiterhin seien $n := n_1 n_2$ und x und y ganze Zahlen, so dass $n_1 x + n_2 y = 1$. Schließlich seien $z_1, z_2 \in \mathbb{Z}$. Es ist zu beweisen, dass $z := z_1 n_2 y + z_2 n_1 x$ eine Lösung des Restklassengleichungssystems

$$\begin{aligned} z &= z_1 \pmod{n_1} \\ z &= z_2 \pmod{n_2} \end{aligned}$$

ist und dass jede Lösung dieses Restklassengleichungssystems modulo n gleich z ist.

Um zu beweisen, dass das angegebene z eine Lösung ist, braucht nur überprüft zu werden, ob es die Gleichungen erfüllt.

Nun ist

$$\begin{aligned} z \bmod n_1 &= (z_1 n_2 y + z_2 n_1 x) \bmod n_1 \\ &= ((z_1 n_2 y \bmod n_1) + \cancel{(z_2 n_1 x \bmod n_1)}) \bmod n_1 \\ &= (z_1 n_2 y \bmod n_1) + 0. \end{aligned}$$

Da $n_1 x + n_2 y = 1$ ist, ist $n_2 y = 1 - n_1 x$, also

$$\begin{aligned} z \bmod n_1 &= (z_1(1 - \cancel{n_1 x}) \bmod n_1) \\ &= z_1 \pmod{n_1} \end{aligned}$$

Analog erhält man $z \bmod n_2 = z_2$.

Angenommen, es ist z' eine weitere Lösung. Dann ist $z' - z_1$ ein Vielfaches von n_1 und auch $z - z_1$ ist ein Vielfaches von n_1. Daher ist die Differenz $(z' - \cancel{z_1}) - (z - \cancel{z_1}) = z' - z$ ein Vielfaches von n_1. Analog erhält man, dass $z' - z$ ist ein Vielfaches von n_2 ist. Da $\text{ggT}(n_1, n_2) = 1$ ist, ist $z' - z$ auch Vielfaches von $n = n_1 n_2$. Dies bedeutet, dass $z' = z \pmod{n}$. Somit gibt es modulo n nur eine Lösung.

A.2 Satz von Wiener

Für alle, die genau wissen wollen, warum Ungleichung (1.1) gilt:

Da $p < 2q$, ist $p^2 < 2pq = 2n$, also $p < \sqrt{2n}$. Weiterhin ist $q < p < \sqrt{2n}$. Daneben ist

$$n - \varphi(n) = p + q - 1 < p + q < 2q + q = 3q < 3\sqrt{2n}$$

und

$$k = \frac{ed-1}{\varphi(n)} < \frac{ed}{\varphi(n)} < d,$$

denn $e < \varphi(n)$. Schließlich folgt aus $d < \frac{1}{3}\sqrt[4]{n}$, dass $9d^2 < \sqrt{n}$. Somit ist

$$\begin{aligned}
\left|\frac{e}{n} - \frac{k}{d}\right| &= \left|\frac{k}{d} - \frac{e}{n}\right| = \left|\frac{kn - ed}{nd}\right| \\
&= \left|\frac{kn - (1 + k\varphi(n))}{nd}\right| = \left|\frac{k(n - \varphi(n)) - 1}{nd}\right| \\
&= \frac{k(n-\varphi(n)) - 1}{nd} < \frac{k(n-\varphi(n))}{nd} < \frac{d(n-\varphi(n))}{nd} \\
&= \frac{n-\varphi(n)}{n} < \frac{3\sqrt{2n}}{n} = \frac{3\sqrt{2}}{\sqrt{n}} < \frac{3\sqrt{2}}{9d^2} = \frac{\sqrt{2}}{3d^2} < \frac{1}{2d^2}.
\end{aligned}$$

A.3 Kleinste gemeinsame Vielfache und größte gemeinsame Teiler

Kleinste gemeinsame Vielfache (kgV) werden in diesem Text nur an einer Stelle verwendet. Tatsächlich lassen sie sich praktisch immer durch größte gemeinsame Teiler (ggT) ausdrücken. Das zugrundeliegende Resultat dafür ist das folgende.

Satz A.1

Es seien $a, b \in \mathbb{Z}$. Dann gilt

$$a \cdot b = \mathrm{ggT}(a, b) \cdot \mathrm{kgV}(a, b).$$

Beweis. Zunächst halten wir fest, dass für alle $x, y \in \mathbb{Z}$ gilt:

$$\min(x, y) + \max(x, y) = x + y.$$

Es seien nun $p_1, p_2, \ldots, p_n$ alle verschiedenen Primfaktoren, die in a und/oder b vorkommen. Weiterhin seien $a = \prod_{i=1}^{n} p_i^{\alpha_i}$ und $\prod_{i=1}^{n} p_i^{\beta_i}$ die Primfaktorzerlegungen von a und b. (In diesen

Zerlegungen können manche Primfaktoren auch gar nicht (also mit Exponent 0) vorkommen.) Dann ist:

$$a \cdot b = \prod_{i=1}^{n} p_i^{\alpha_i} \cdot \prod_{i=1}^{n} p_i^{\beta_i} = \prod_{i=1}^{n} p_i^{\alpha_i+\beta_i}.$$

Andererseits ist

$$\mathrm{ggT}(a,b) = \prod_{i=1}^{n} p_i^{\min(\alpha_i,\beta_i)} \quad \text{und}$$
$$\mathrm{kgV}(a,b) = \prod_{i=1}^{n} p_i^{\max(\alpha_i,\beta_i)}.$$

Daher ist

$$\begin{aligned}\mathrm{ggT}(a,b) \cdot \mathrm{kgV}(a,b) &= \prod_{i=1}^{n} p_i^{\min(\alpha_i,\beta_i)} \cdot \prod_{i=1}^{n} p_i^{\max(\alpha_i,\beta_i)} \\ &= \prod_{i=1}^{n} p_i^{\min(\alpha_i,\beta_i)+\max(\alpha_i,\beta_i)} \\ &= \prod_{i=1}^{n} p_i^{\alpha_i+\beta_i}.\end{aligned}$$

□

A.4 Punktaddition auf elliptischen Kurven

Wir wählen $\mathcal{E}: \; y^2 = x^3 + ax + b$. Die Punkte $P = (x_1, y_1)$ und $Q = (x_2, y_2)$ sind Punkte auf $\mathcal{E}$.

Wir legen durch P und Q eine Gerade

$$g: \; y = kx + d. \tag{A.1}$$

Deren Steigung k lässt sich schnell berechnen als

$$k = \frac{y_2 - y_1}{x_2 - x_1}.$$

Wir schneiden g mit $\mathcal{E}$:

$$\begin{aligned}(kx + d)^2 &= x^3 + ax + b \\ k^2x^2 + 2dkx + d^2 &= x^3 + ax + b \\ x^3 - k^2x^2 + (a - 2dk)x + (b - d^2) &= 0 \end{aligned} \tag{A.2}$$

Es ergibt sich eine Gleichung dritten Grades, deren Lösungen die x-Koordinaten der Schnittpunkte der Geraden mit der Kurve sind. Einerseits hat so eine Gleichung dritten Grades maximal drei

Lösungen[1], andererseits kennen wir bereits zwei davon (x_1 und x_2). Angenommen x_3 ist die dritte Lösung. Dann ist nach einem Satz von Vietá

$$x^3 - k^2x^2 + (a - 2dk)x + (b - d^2) = (x - x_1)(x - x_2)(x - x_3).$$

Nach dem Ausmultiplizieren und Sortieren auf der rechten Seite heißt das

$$x^3 - k^2x^2 + (a - 2dk)x + (b - d^2) = x^3 - (x_1 + x_2 + x_3)x^2 + (x_1x_2 + x_1x_3 + x_2x_3)x - x_1x_2x_3.$$

Ein Koeffizientenvergleich ergibt nun, dass $k^2 = x_1 + x_2 + x_3$ sein muss, also

$$x_3 = k^2 - x_1 - x_2.$$

Nachdem zwei Lösungen bekannt sind (die x-Koordinaten der Punkte P und Q), ist die dritte also ganz einfach zu berechnen. Die y-Koordinate erhalten wir durch Einsetzen in Gleichung (A.1), das Spiegeln ändert noch das Vorzeichen der y-Koordinate

$$y_3 = -(kx_3 + d) = -kx_3 - y_1 + kx_1 = -y_1 + k(x_1 - x_3).$$

Der Achsenabschnitt d ergibt sich dabei aus Gleichung (A.1) als

$$d = y_1 - kx_1.$$

Ganz ähnlich läuft die Addition eines Punkts $P = (x_1, y_1)$ zu sich selbst (Punktverdopplung) auf der Kurve $\mathcal{E}: \; y^2 = x^3 + ax + b$. Wir legen in P die Tangente an $\mathcal{E}$. Die Steigung der Tangente in P ist gerade die Steigung von $\mathcal{E}$ in P. Diese lässt sich mit implizitem Differenzieren berechnen:

$$\begin{aligned} 2y \cdot y' &= 3x^2 + a \\ y' &= \frac{3x^2 + a}{2y} \\ k = y'(x = x_1, y = y_1) &= \frac{3x_1^2 + a}{2y_1} \end{aligned}$$

Beim Schneiden der Tangente mit der Kurve landen wir wieder bei Gleichung (A.2). Auch hier sind bereits zwei Lösungen der Gleichung bekannt, denn im Fall eines Berührpunkts in P taucht x_1 zweimal als Lösung auf. Es ergibt sich also

$$\begin{aligned} x_3 &= k^2 - 2x_1 \text{ und} \\ y_3 &= -y_1 + k(x_1 - x_3). \end{aligned}$$

[1] Vgl. Kapitel 5.

A.5 Addition in projektiven Koordinaten

Die Summe zweier Punkte $P = (X_1, Y_1, Z_1)$ und $Q = (X_2, Y_2, Z_2)$ in projektiven Koordinaten lässt sich wie folgt berechnen: In affinen Koordinaten ist $P = (x_1, y_1) = (X_1/Z_1, Y_1/Z_1)$ und $Q = (x_2, y_2) = (X_2/Z_2, Y_2/Z_2)$. Wir nehmen nun an, dass $P \neq \pm Q$. Dann lässt sich die Summe nach der bekannten Formel berechnen.

$$
\begin{aligned}
k &= \frac{y_2 - y_1}{x_2 - x_1} \\
&= \frac{Y_2/Z_2 - Y_1/Z_1}{X_2/Z_2 - X_1/Z_1} = \frac{\frac{Y_2Z_1 - Y_1Z_2}{\cancel{Z_1Z_2}}}{\frac{X_2Z_1 - X_1Z_2}{\cancel{Z_1Z_2}}} = \frac{\overbrace{Y_2Z_1 - Y_1Z_2}^{=:u}}{\underbrace{X_2Z_1 - X_1Z_2}_{=:v}} = \frac{u}{v}
\end{aligned}
$$

Durch einfaches Umformen ergibt sich auch

$$
\begin{aligned}
X_2Z_1 &= v + X_1Z_2, \text{ bzw.} \\
X_1Z_2 + X_2Z_1 &= v + 2X_1Z_2.
\end{aligned}
$$

Die affinen Koordinaten X_3 und y_3 lassen sich nun berechnen.

$$
\begin{aligned}
x_3 &= k^2 - x_1 - x_2 = (u/v)^2 - X_1/Z_1 - X_2/Z_2 \\
&= \frac{u^2Z_1Z_2 - v^2X_1Z_2 - v^2X_2Z_1}{v^2Z_1Z_2} = \frac{u^2Z_1Z_2 - v^2(X_1Z_2 + X_2Z_1)}{v^2Z_1Z_2} \\
&= \frac{\overbrace{u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2}^{=:w}}{v^2Z_1Z_2} = \frac{w}{v^2Z_1Z_2}.
\end{aligned}
$$

Ganz ähnlich ergibt sich auch für y_3 eine Darstellung als Bruch

$$
\begin{aligned}
y_3 &= -y_1 + k(x_1 - x_3) \\
&= -Y_1/Z_1 + u/v \cdot (X_1/Z_1 - w/(v^2Z_1Z_2)) \\
&= -\frac{Y_1}{Z_1} + \frac{u}{v} \cdot \frac{X_1v^2Z_2 - w}{v^2Z_1Z_2} \\
&= \frac{-Y_1v^3Z_2 + u(X_1v^2Z_2 - w)}{v^3Z_1Z_2}.
\end{aligned}
$$

Wählt man für x_3 und y_3 den gleichen Nenner $v^3Z_1Z_2$, so erhält man

$$
\begin{aligned}
x_3 &= \frac{vw}{v^3Z_1Z_2}, \\
y_3 &= \frac{u(v^2X_1Z_2 - w) - v^3Y_1Z_2}{v^3Z_1Z_2}.
\end{aligned}
$$

Gegeben: G, $\alpha = (b_0, b_1, \ldots, b_m)_2$
Gesucht: $A := \alpha \cdot G$

$A \leftarrow \infty$
$B \leftarrow G$
for $i = 0, \ldots, m$
    if $b_i = 0$ then
        $B \leftarrow A + B$
        $A \leftarrow 2A$
    else
        $A \leftarrow A + B$
        $B \leftarrow 2B$
return A

Abbildung A.6.1: Montgomery Ladder zur Punktmultiplikation

A.6 Die Montgomery Ladder

Hier wird eine Methode zum Potenzieren in einer Gruppe vorgestellt. Da diese Methode besonders bei elliptischen Kurven (Montgomery Curves) eingesetzt wird, wird die Beschreibung für die Punktmultiplikation auf elliptischen Kurven gegeben.

Es sei im Folgenden $\alpha \in \mathbb{N}$ und G ein Punkt auf einer elliptischen Kurve. Gesucht ist der Punkt $\alpha \cdot G$.

Abbildung A.6.1 zeigt den Algorithmus.

Wesentlich für die Anwendung bei Montgomery Curves ist, dass zu jedem Zeitpunkt in diesem Algorithmus sowohl A als auch B Vielfache des Punktes G sind. Genauer: Stets gibt es eine nichtnegative ganze Zahl n, so dass $A = n \cdot G$ und $B = (n+1) \cdot G$.

Am Beginn ist dies klar ($n = 0$). Ist zu irgendeinem Zeitpunkt $A = n \cdot G$ und $B = (n+1) \cdot G$, dann betrachten wir die Änderungen in den beiden if-Zweigen.

Im ersten if-Zweig wird

- A zu $2 \cdot A = 2n \cdot G$ und
- B zu $A + B = n \cdot G + (n+1) \cdot G = (2n+1) \cdot G$.

Im zweiten if-Zweig

- A zu $A + B = n \cdot G + (n+1) \cdot G = (2n+1) \cdot G$ und
- B zu $2 \cdot B = 2(n+1) \cdot G = (2n+2) \cdot G$.

In beiden Fällen sind A und B wieder Vielfache von G und der Koeffizient vor B ist um 1 größer als jener von A.

A.7 Quadratwurzeln in endlichen Körpern

Es sei $p \in \mathbb{P}$. Um zu bestimmen, ob a ein Quadrat in $\mathbb{Z}_p$ ist, ob man also die Quadratwurzel daraus ziehen kann, kann man sich des Euler-Kriteriums bedienen.

Es seien $p \in \mathbb{P} \setminus \{2\}$ und $a \in \mathbb{Z}_p$. Dann ist

$$a^{\frac{p-1}{2}} \pmod{p} = \begin{cases} 0, & \text{wenn } a = 0 \pmod{p}, \\ 1, & \text{wenn } a \text{ ein Quadrat in } \mathbb{Z}_p \text{ ist, und} \\ -1, & \text{wenn } a \text{ kein Quadrat in } \mathbb{Z}_p \text{ ist.} \end{cases}$$

In dem Fall, dass $p \bmod 4 = 3$ ist, lassen sich Quadratwurzeln auch schnell berechnen. Es sind dann

$$\pm a^{\frac{p+1}{4}} \pmod{p}$$

die Quadratwurzeln von a in $\mathbb{Z}_p$, denn

$$\left(\pm a^{\frac{p+1}{4}}\right)^2 = a^{\frac{p+1}{2}} = a^{\frac{p-1}{2}+1} = a \pmod{p}.$$

Ist $p \bmod 4 \neq 3$, so ist das Berechnen von Quadratwurzeln etwas komplizierter. Mit dem Algorithmus von Tonelli-Shanks können dann dennoch Quadratwurzeln effizient berechnet werden. In [Coh93] findet sich eine Beschreibung dieses Algorithmus.

Anhang B

Faktorisieren

Wir haben gesehen, dass das RSA-Verfahren gebrochen werden kann, wenn man in der Lage ist, eine große Zahl n, die das Produkt zweier großer Primzahlen p und q ist, zu faktorisieren.

Wir betrachten im Folgenden zwei Methoden zur Faktorisierung großer Zahlen, die Pollard-$(p-1)$-Methode und das quadratische Sieb.

B.1 Die Pollard-Methode

Mit dieser Methode ist es möglich, einen Primfaktor p einer Zahl n zu finden, wenn in der Primfaktorzerlegung von $p-1$ nur kleine Primzahlpotenzen vorkommen.

Angenommen, die Zahl n habe einen Primfaktor p, so dass alle Primzahlpotenzen, die in der Primfaktorzerlegung von $p-1$ vorkommen, kleiner oder gleich B sind, wobei B eine nicht allzu große Zahl ist.[1] Wir bilden zunächst das kleinste gemeinsame Vielfache E aller Zahlen, die kleiner oder gleich B sind. Somit ist E sicher ein Vielfaches von $p-1$ (warum?), also $E = v(p-1)$ für ein passendes $v \in \mathbb{Z}$.

Nach dem kleinen Satz von Fermat (Satz 1.10) gilt:

$$\begin{aligned} a^{p-1} &= 1 \pmod{p}, \\ \text{also} \quad a^E = a^{v(p-1)} &= (a^{p-1})^v = 1^v = 1 \pmod{p} \\ \text{d.h.} \quad p &\mid a^E - 1 \end{aligned}$$

Also ist $a^E - 1$ ein Vielfaches von p. Falls $a^E - 1$ kein Vielfaches von n ist, dann ist $\mathrm{ggT}(a^E - 1, n)$ ein echter Teiler von n.

Wir brauchen also nur $a^E - 1$ modulo n zu berechnen und dann $\mathrm{ggT}(a^E - 1, n)$. (Warum reicht es hier, $a^E - 1$ modulo n zu berechnen?) Die Parameter a und B sind hier frei wählbar, a kann zufällig gewählt werden, B muss groß genug sein.

[1] Natürlich kennen wir p nicht. Wir vertrauen blind darauf, dass $p-1$ die gewünschte Eigenschaft besitzt. Ist dies nicht der Fall, werden wir keinen Erfolg haben.

Hat man bei dieser Methode keinen Erfolg ($a^E - 1$ ist ein Vielfaches von n), dann wählt man ein anderes a. Haben mehrere a nicht funktioniert, muss man wahrscheinlich ein größeres B wählen.

Beispiel B.1

Wir faktorisieren $n = 1241143$ und wählen $B = 13$. Dann ist $E = 2^3 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13$. Für $a = 2$ erhalten wir $2^E - 1 \bmod n = 861525$. Beachte, dass $a^E \bmod n$ hier einfach als

$$(((((((((a^2)^2)^2)^3)^3)^5)^7)^{11})^{13} \bmod n$$

berechnen. Für jede einzelne Potenz geht das recht schnell. Weiter ergibt sich $\mathrm{ggT}(861525, 1241143) = 547$. Schon haben wir einen Teiler von n gefunden. Tatsächlich ist $547 - 1 = 2 \cdot 3 \cdot 7 \cdot 13$.

B.2 Das quadratische Sieb

Die Pollard-$(p-1)$-Methode ist für nicht allzu große Zahlen eine einfache Faktorisierungsmethode, die allerdings schnell an ihre Grenzen gelangt. Die folgende Methode ist noch besser, dafür brauchen wir aber noch ein bisschen mehr Mathematik. Dem quadratischen Sieb liegt folgende Beobachtung zu Grunde: Sind x und y ganze Zahlen mit

$$x^2 = y^2 \pmod{n} \quad \text{und} \tag{B.1}$$

$$x \neq \pm y \pmod{n}, \tag{B.2}$$

dann ist $1 < \mathrm{ggT}(n, x - y) < n$, denn

$$n \mid x^2 - y^2 = (x + y)(x - y)$$
$$n \nmid x + y,\ n \nmid x - y.$$

Dies ist nur möglich, wenn $n = uv$ und $u \mid x + y$, $v \mid x - y$.

Nun geht es nur noch darum, die Zahlen x und y zu finden.

Sei $m := \lfloor \sqrt{n} \rfloor$ und $f(x) := (x + m)^2 - n$.

Bemerkung: Ist x klein, so ist auch $f(x)$ klein.

Beispiel B.2

Wir beginnen bescheiden und faktorisieren $n = 3277$. Dann ist $m = 57$. Zunächst berechnen wir die Werte

$$\begin{aligned} f(-2) &= 55^2 - 3277 = -252 = (-1) \cdot 2^2 \cdot 3^2 \cdot 7 \\ f(-1) &= 56^2 - 3277 = -141 = (-1) \cdot 3 \cdot 47 \\ f(0) &= 57^2 - 3277 = -28 = (-1) \cdot 2^2 \cdot 7 \\ f(1) &= 58^2 - 3277 = 87 = 3 \cdot 29 \\ f(2) &= 59^2 - 3277 = 204 = 2^2 \cdot 3 \cdot 17 \end{aligned}$$

Daraus lesen wir (unter anderem)

$$\begin{aligned} 55^2 &= (-1) \cdot 2^2 \cdot 3^2 \cdot 7 \pmod{3277} \quad \text{und} \\ 57^2 &= (-1) \cdot 2^2 \cdot 7 \pmod{3277} \end{aligned}$$

Multipliziert man die beiden Gleichungen, so erhält man

$$\begin{aligned} 55^2 \cdot 57^2 &= (-1) \cdot 2^2 \cdot 3^2 \cdot 7 \cdot (-1) \cdot 2^2 \cdot 7 \pmod{3277} \quad \text{bzw.} \\ (55 \cdot 57)^2 &= (-1)^2 \cdot 2^4 \cdot 3^2 \cdot 7^2 = (2^2 \cdot 3 \cdot 7)^2 \pmod{3277} \end{aligned}$$

Wir wählen also $x = 55 \cdot 57$ und $y = 2^2 \cdot 3 \cdot 7 = 84$. Tatsächlich sind (B.1) und (B.2) erfüllt. Wir berechnen $x - y = 3051$ und $\mathrm{ggT}(3051, 3277) = 113$, ein Teiler von n ist gefunden.

So einfach geht es nicht immer, hier haben wir Glück gehabt. Im Allgemeinen muss man ein bisschen mehr tun. Auf der linken Seite steht immer ein Quadrat, wenn man Gleichungen multipliziert. Es stellt sich die Frage, welche Gleichungen (möglicherweise mehr als zwei) multipliziert werden müssen, damit sich auch auf der rechten Seite ein Quadrat ergibt. Manchmal kann man raten, meistens jedoch muss man rechnen, wir setzen unser Beispiel fort.

Beispiel B.3: Fortsetzung

Die rechten Seiten sind

$$(-1) \cdot 2^2 \cdot 3^2 \cdot 7,\ (-1) \cdot 4 \cdot 47,\ (-1) \cdot 2^2 \cdot 7,\ 3 \cdot 29 \text{ und } 2^2 \cdot 3 \cdot 17.$$

Wir definieren nun $\lambda_i := 1$, wenn die i-te Gleichung verwendet werden soll und sonst $\lambda_i := 0$.

Multipliziert man nun die Gleichungen, so erhält man auf der rechten Seite

$$\Big((-1)\cdot 2^2\cdot 3^2\cdot 7\Big)^{\lambda_1}\cdot((-1)\cdot 3\cdot 47)^{\lambda_2}\cdot\Big((-1)\cdot 2^2\cdot 7\Big)^{\lambda_3}\cdot(3\cdot 29)^{\lambda_4}\cdot\Big(2^2\cdot 3\cdot 17\Big)^{\lambda_5}$$

Wir sortieren das Ergebnis nach den Basen und erhalten

$$(-1)^{\lambda_1+\lambda_2+\lambda_3}\cdot 2^{2\lambda_1+2\lambda_3+2\lambda_5}\cdot 3^{2\lambda_1+\lambda_2+\lambda_4+\lambda_5}\cdot 7^{\lambda_1+\lambda_3}\cdot 17^{\lambda_5}\cdot 29^{\lambda_4}\cdot 47^{\lambda_2}$$

Damit ein Quadrat herauskommt, müssen alle Exponenten gerade sein, d. h.

$$\begin{aligned}
\lambda_1+\lambda_2+\lambda_3 &= 0 \pmod 2\\
2\lambda_1+2\lambda_3+2\lambda_5 &= 0 \pmod 2\\
2\lambda_1+\lambda_2+\lambda_4+\lambda_5 &= 0 \pmod 2\\
\lambda_1+\lambda_3 &= 0 \pmod 2\\
\lambda_5 &= 0 \pmod 2\\
\lambda_4 &= 0 \pmod 2\\
\lambda_2 &= 0 \pmod 2
\end{aligned}$$

Es ergibt sich ein lineares Gleichungssystem über dem Körper $\mathbb{Z}_2$, welches sich wie gewohnt lösen lässt. Besitzt das lineare Gleichungssystem neben dem Nullvektor eine weitere Lösung, so wissen wir, welche Gleichungen multipliziert werden müssen. In diesem Fall ergibt sich $\lambda_1=\lambda_3=1$ und $\lambda_2=\lambda_4=\lambda_5=0$. Den Rest kennen wir ja schon.

B.3 Faktorisieren mit elliptischen Kurven

Auch elliptische Kurven lassen sich verwenden, um große Zahlen zu faktorisieren. Wir sehen uns eine Methode von Lenstra an. Diese Methode beruht darauf, dass man Vielfache von Punkten auf einer elliptischen Kurve über $\mathbb{Z}_n$ berechnet. Da $\mathbb{Z}_n$ kein Körper ist, geht das mit einer gewissen Wahrscheinlichkeit (bei einer Division) schief. Diese Wahrscheinlichkeit ist bei passenden Vielfachen sehr groß.

Algorithmus B.4: Lenstra

Die Zahl n ist zu faktorisieren.

1. Wähle (zufällig) eine elliptische Kurve $\mathcal{E}:\ y^2=x^3+ax+b$ über $\mathbb{Z}_n$ und einen Punkt $P=(x,y)$ auf $\mathcal{E}$.

2. Wähle zwei natürliche Zahlen B und C.

3. Setze
$$k := \prod_{p \leq B,\ p \in \mathbb{P}} p^{\alpha_p},$$
wobei α_p die größte ganze Zahl ist, für die $p^{\alpha_p} \leq C$ ist.

4. Berechne $k \cdot P$. Dabei muss modulo n invertiert werden (Nenner). Das funktioniert nicht, wenn der Nenner nicht relativ prim zu n ist. In diesem Fall berechnet der erweiterte Euklidsche Algorithmus einen Teiler von n.

5. Ist alles gut gegangen, wähle neue $\mathcal{E}$ und P.

Beispiel B.5

Will man eine 10-stellige Zahl n faktorisieren, wählt man etwa $B = 20$ und $C = 100700$. Dann ergibt sich $k = 2^{16} \cdot 3^{10} \cdot 5^7 \cdot 7^5 \cdot 11^4 \cdot 13^4 \cdot 17^4 \cdot 19^3$. Wie bei der Pollard-Methode kann auch hier $k \cdot P$ als eine Folge von Punktmultiplikationen berechnet werden.

Diese Faktorisierungsmethode ist in etwa so schnell wie das quadratische Sieb. Sie ist aber schneller, wenn n einen sehr kleinen Primfaktor besitzt. Außerdem haben wir gesehen, dass das quadratische Sieb einen sehr großen Speicherbedarf hat, es müssen ja riesige Gleichungssysteme gelöst werden. Dieses Problem hat diese Methode nicht. Schließlich kann man hier mehr Parameter wählen (B, C und $\mathcal{E}$), natürlich kann man dann mit verschiedenen Parametern parallel an der Faktorisierung arbeiten.

Die Faktorisierungsmethode mit elliptischen Kurven hat gezeigt, dass das Faktorisierungsproblem noch lange nicht vollständig untersucht ist. Die relativ einfache Methode führt zu einem sehr schnellen Faktorisierungsalgorithmus (lediglich der Beweis, dass diese Methode wirklich eine schnelle Faktorisierungsmethode ist, ist schwierig).

Literaturverzeichnis

[Aum18] Jean-Philippe Aumasson. *Serious Cryptography*. No Starch Press, 2018.

[BBB+20] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for Key Management – Part 1: General. *NIST Special Publication 800-57 (Rev. 5)*, 2020.

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In *Proceedings of the Cryptology in Africa 1st International Conference on Progress in Cryptology*, AFRICACRYPT'08, pages 389–405, Berlin, Heidelberg, 2008. Springer-Verlag.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. Xmss - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 117–129, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77–89, 2012.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BHH+14] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. *IACR Cryptology ePrint Archive*, 2014:795, 2014.

[Bun92] Peter Bundschuh. *Einführung in die Zahlentheorie*. Springer-Verlag, Berlin, 2nd edition, 1992.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, Berlin, 1993.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. *NIST Special Publication 800-38D*, 2007.

[Gal12] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

[Kah67] David Kahn. *The Codebreakers*. Macmillan Publishing Company, 1967.

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 3rd edition, 2020.

[Knu77] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1977.

[Kob94] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer, 2nd edition, 1994.

[KSD13] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. Digital Signature Standard (DSS), 2013.

[Mon87] P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.

[Paa16] Christof Paar. *Kryptographie verständlich*. Springer, 2016.

[PH78] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[Res99] E. Rescorla. Diffie-Hellman Key Agreement Method. *RFC 2631*, 1999.

[Sch85] R. Schoof. Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p. *Math. Comp.*, 44:483–494, 1985.

[Sin00] Simon Singh. *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet*. dtv, 2000.

[Ste09] William Stein. *Elementary Number Theory: Primes, Congruences, and Secrets*. Springer, 2009.

[TW06] Wade Trappe and Lawrence Washington. *Introduction to Cryptography with Coding Theory*. Pearson Education, 2nd edition, 2006.