und Mobiltelefonen, oder auf Servern, die viele 100 oder 1000 asymmetrische Operationen pro Sekunde berechnen müssen. In den Kap. 7, 8 und 9 führen wir u.a. Techniken zur effizienten Implementierung asymmetrischer Algorithmen ein.

6.3 Grundlagen der Zahlentheorie für asymmetrische Algorithmen

Nachfolgend werden wir einige zahlentheoretische Grundlagen einführen, die für die asymmetrische Kryptografie notwendig sind. Es werden der euklidische Algorithmus vorgestellt sowie die eulersche Phi-Funktion, der kleine fermatsche Satz und der Satz von Euler. Alle sind wichtig für asymmetrische Algorithmen, insbesondere für das Verständnis des RSA-Kryptosystems.

6.3.1 Der euklidische Algorithmus

Wir starten mit dem Problem der Berechnung des *größten gemeinsamen Teilers (ggT)*. Der ggT zweier positiver ganzer Zahlen r_0 und r_1 wird als

$$\text{ggT}(r_0, r_1)$$

bezeichnet und ist die größte positive ganze Zahl, die sowohl r_0 als auch r_1 teilt. Beispielsweise ist $\text{ggT}(21, 9) = 3$. Für kleine Zahlen ist der ggT einfach durch Faktorisierung der beiden Zahlen und Finden des größten gemeinsamen Faktors zu berechnen.

Beispiel 6.1 Seien $r_0 = 84$ und $r_1 = 30$. Die Faktorisierung liefert

$$\begin{aligned} r_0 &= 84 = 2 \cdot 2 \cdot 3 \cdot 7 \\ r_1 &= 30 = 2 \cdot 3 \cdot 5 \end{aligned}$$

Der ggT ist das Produkt aller gemeinsamen Primfaktoren:

$$2 \cdot 3 = 6 = \text{ggT}(30, 84)$$

Für derart große Zahlen, wie sie bei asymmetrischen Verfahren verwendet werden, ist eine Faktorisierung jedoch nicht möglich, sodass ein effizienteres Verfahren, der euklidische Algorithmus, für die ggT-Berechnungen verwendet wird. Der Algorithmus basiert auf der Beobachtung, dass

$$\text{ggT}(r_0, r_1) = \text{ggT}(r_0 - r_1, r_1),$$

wobei wir annehmen, dass $r_0 > r_1$ und beide Zahlen positiv sind. Diese Eigenschaft kann leicht bewiesen werden: Sei $\text{ggT}(r_0, r_1) = g$. Da g sowohl r_0 als auch r_1 teilt, können wir

$r_0 = g \cdot x$ und $r_1 = g \cdot y$ schreiben, mit $x > y$ sowie x und y teilerfremd, d. h. sie besitzen keinen gemeinsamen Faktor. Damit ist einfach zu zeigen, dass $(x - y)$ und y ebenfalls teilerfremd sind. Daraus folgt:

$$\text{ggT}(r_0 - r_1, r_1) = \text{ggT}(g \cdot (x - y), g \cdot y) = g.$$

Wir verfizieren diese Eigenschaft mit den Zahlen des vorherigen Beispiels:

Beispiel 6.2 Seien wieder $r_0 = 84$ und $r_1 = 30$ gegeben. Wir schauen nun auf den ggT von $(r_0 - r_1)$ und r_1:

$$\begin{aligned} r_0 - r_1 &= 54 = 2 \cdot 3 \cdot 3 \cdot 3 \\ r_1 &= 30 = 2 \cdot 3 \cdot 5 \end{aligned}$$

Der größte gemeinsame Faktor ist $2 \cdot 3 = 6 = \text{ggT}(30, 54) = \text{ggT}(30, 84)$.

Dieser Prozess kann auch iterativ angewendet werden:

$$\text{ggT}(r_0, r_1) = \text{ggT}(r_0 - r_1, r_1) = \text{ggT}(r_0 - 2r_1, r_1) = \cdots = \text{ggT}(r_0 - m\ r_1, r_1),$$

solange $(r_0 - m\ r_1) > 0$ gilt. Der Algorithmus verwendet die geringste Anzahl an Schritten, wenn wir den größtmöglichen Wert für m wählen. Dies ist der Fall für:

$$\text{ggT}(r_0, r_1) = \text{ggT}(r_0 \bmod r_1, r_1)$$

Da der erste Term $(r_0 \bmod r_1)$ kleiner als der zweite Term r_1 ist, vertauschen wir üblicherweise die beiden Terme:

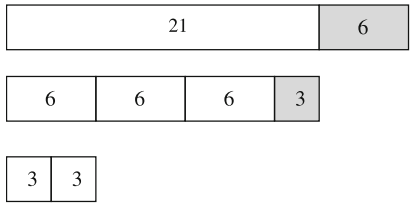
$$\text{ggT}(r_0, r_1) = \text{ggT}(r_1, r_0 \bmod r_1)$$

Eine wesentliche Beobachtung bei diesem Vorgehen ist, dass wir die Aufgabenstellung, den ggT zu finden, auf das Problem **reduzieren** können, den ggT zweier kleinerer Zahlen zu finden. Dieser Prozess kann nun rekursiv so lange wiederholt werden, bis wir schließlich $\text{ggT}(r_l, 0) = r_l$ erhalten. Da jeder Schritt der Rekursion den ggT des vorherigen Schritts erhält, ist der endgültige ggT auch der ggT des ursprünglichen Problems, d. h.:

$$\text{ggT}(r_0, r_1) = \cdots = \text{ggT}(r_l, 0) = r_l$$

Bevor wir den euklidischen Algorithmus weiter diskutieren, schauen wir uns einige Beispiele für die ggT-Berechnung an.

Beispiel 6.3 Seien $r_0 = 27$ und $r_1 = 21$. Abb. 6.6 gibt uns ein Gefühl für den Algorithmus, indem gezeigt wird, wie die Längen der Parameter bei jeder Iteration kleiner



ggT(27, 21) = ggT(1 · 21+6, 21) = ggT(21, 6)

ggT(21, 6) = ggT(3 · 6+3, 6) = ggT(6, 3)

ggT(6, 3) = ggT(2 · 3+0, 3) = ggT(3, 0) = 3

ggT (27, 21) = ggT (21, 6) = ggT (6, 3) = ggT (3, 0) = 3

Abb. 6.6 Beispiel für den euklidischen Algorithmus für die Eingabewerte $r_0 = 27$ und $r_1 = 21$

werden. Die grauen Bereiche in der Iteration sind die neuen Reste $r_2 = 6$ (erste Iteration) und $r_3 = 3$ (zweite Iteration), die jeweils die Eingabe für die nächste Iteration bilden. Man beachte, dass sich in der letzten Iteration der Rest $r_4 = 0$ ergibt, was gleichzeitig das Ende des Algorithmus anzeigt.

Es ist ebenfalls hilfreich, sich den euklidischen Algorithmus mit etwas größeren Zahlen anzuschauen, wie in Beispiel 6.4 zu sehen.

Beispiel 6.4 Seien $r_0 = 973$ und $r_1 = 301$. Der ggT wird wie folgt berechnet:

$$\begin{aligned} 973 &= 3 \cdot 301 + 70 & \text{ggT}(973, 301) &= \text{ggT}(301, 70) \\ 301 &= 4 \cdot 70 + 21 & \text{ggT}(301, 70) &= \text{ggT}(70, 21) \\ 70 &= 3 \cdot 21 + 7 & \text{ggT}(70, 21) &= \text{ggT}(21, 7) \\ 21 &= 3 \cdot 7 + 0 & \text{ggT}(21, 7) &= \text{ggT}(7, 0) = 7 \end{aligned}$$

Nun sollten wir eine Vorstellung von der Idee hinter dem euklidischen Algorithmus haben und widmen uns einer formaleren Beschreibung des Algorithmus.

Euklidischer Algorithmus

Eingang: Positive ganze Zahlen r_0 und r_1 mit $r_0 > r_1$

Ausgang: $\text{ggT}(r_0, r_1)$

Initialisierung: $i = 1$

Algorithmus:

1. DO
   1.1 $i = i + 1$
   1.2 $r_i = r_{i-2} \bmod r_{i-1}$
   WHILE $r_i \neq 0$
2. RETURN
   $\text{ggT}(r_0, r_1) = r_{i-1}$

Man beachte, dass der Algorithmus dann terminiert, wenn der neu berechnete Rest den Wert $r_l = 0$ hat. Der Rest aus der vorangegangenen Iteration, d. h. r_{l-1}, ist der ggT des ursprünglichen Problems.

Der euklidische Algorithmus ist sehr effizient, selbst mit sehr großen Zahlen, wie sie üblicherweise in der asymmetrischen Kryptografie verwendet werden. Die Anzahl der Iterationen entspricht in etwa der Anzahl der Ziffern der Eingangswerte. Dies bedeutet beispielsweise, dass die Anzahl an Iterationen eines ggT von 1024 Bit großen Zahlen 1024 mal einer Konstante ist. Algorithmen mit einigen tausend Iterationen können auf modernen PC sehr schnell berechnet werden, was den euklidischen Algorithmus sehr effizient in der Praxis macht.

6.3.2 Der erweiterte euklidische Algorithmus

Bisher haben wir gesehen, dass das Finden des ggT zweier natürlicher Zahlen r_0 und r_1 rekursiv durch Reduktion der Operanden durchgeführt werden kann. Die Hauptanwendung des euklidischen Algorithmus ist jedoch nicht die Berechnung des ggT. Eine Erweiterung des Algorithmus erlaubt es uns, die Inverse modulo einer natürlichen Zahl zu berechnen, was von großer Bedeutung in der asymmetrischen Kryptografie ist. Neben dem ggT berechnet der *erweiterte euklidische Algorithmus* (EEA) eine Linearkombination der Form:

$$\text{ggT}(r_0, r_1) = s \cdot r_0 + t \cdot r_1,$$

wobei s und t ganzzahlige Koeffizienten sind. Diese Gleichung wird häufig als *diophantische Gleichung* bezeichnet.

Die Frage lautet nun: Wie berechnen wir die beiden Koeffizienten s und t? Die Idee hinter dem EEA ist, dass wir den normalen euklidischen Algorithmus ausführen, aber zusätzlich den aktuellen Rest r_i in jeder Iteration als Linearkombination der Form

$$r_i = s_i r_0 + t_i r_1 \tag{6.1}$$

darstellen.

Wenn uns dies gelingt, enden wir in der letzten Iteration mit der Gleichung:

$$r_l = \text{ggT}(r_0, r_1) = s_l r_0 + t_l r_1 = s r_0 + t r_1$$

Dies bedeutet, dass der letzte Koeffizient s_l der gesuchte Koeffizient s aus (6.1) ist und ebenso $t_l = t$. Schauen wir uns ein Beispiel an.

Beispiel 6.5 Wir betrachten den erweiterten euklidischen Algorithmus mit den gleichen Werten wie im vorherigen Beispiel, $r_0 = 973$ und $r_1 = 301$. Auf der linken Seite führen wir den normalen euklidischen Algorithmus aus, d. h. wir berechnen die neuen Reste $r_2, r_3, \ldots$ Zusätzlich berechnen wir in jeder Iteration den ganzzahligen Quotienten q_{i-1}. Auf der rechten Seite berechnen wir die Koeffizienten s_i und t_i, sodass $r_i = s_i r_0 + t_i r_1$.

Die Koeffizienten der Linearkombination, die sich in jeder Iteration ergibt, sind jeweils in eckigen Klammern dargestellt.

i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = [s_i]r_0 + [t_i]r_1$
2	$973 = 3 \cdot 301 + 70$	$70 = [1]r_0 + [-3]r_1$
3	$301 = 4 \cdot 70 + 21$	$21 = 301 - 4 \cdot 70$ $= r_1 - 4(1r_0 - 3\, r_1)$ $= [-4]r_0 + [13]r_1$
4	$70 = 3 \cdot 21 + 7$	$7 = 70 - 3 \cdot 21$ $= (1r_0 - 3r_1) - 3(-4r_0 + 13r_1)$ $= [13]r_0 + [-42]r_1$
	$21 = 3 \cdot 7 + 0$	

Der Algorithmus berechnet die drei Parameter $\text{ggT}(973, 301) = 7$, $s = 13$ und $t = -42$. Die Korrektheit des Ergebnisses kann wie folgt verifiziert werden:

$$\text{ggT}(973, 301) = 7 = [13]\ 973 + [-42]\ 301 = 12.649 - 12.642$$

Es ist hilfreich, sich die Umformungen in der rechten Spalte des obigen Beispiels anzuschauen. Man achte besonders darauf, dass die Linearkombination auf der rechten Seite immer mithilfe der *vorherigen* Linearkombinationen konstruiert wird. Wir leiten nun eine rekursive Formel für die Berechnung der Werte s_i und r_i in jeder Iteration her. Wir nehmen an, wir befinden uns in der Iteration mit dem Index i. In den beiden vorherigen Iterationen wurden die folgenden Werte berechnet:

$$r_{i-2} = [s_{i-2}]r_0 + [t_{i-2}]r_1 \tag{6.2}$$
$$r_{i-1} = [s_{i-1}]r_0 + [t_{i-1}]r_1 \tag{6.3}$$

In der aktuellen Iteration i berechnen wir zuerst den Quotienten q_{i-1} und den neuen Rest r_i aus r_{i-1} und r_{i-2}:

$$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$$

Diese Gleichung kann umgeformt werden zu:

$$r_i = r_{i-2} - q_{i-1} \cdot r_{i-1} \tag{6.4}$$

Erinnern wir uns an unser Ziel, den aktuellen Rest r_i als Linearkombination aus r_0 und r_1 darzustellen, wie in (6.1) zu sehen. Um dies zu erreichen, ist der wesentliche Schritt wie folgt: In (6.4) ersetzen wir einfach r_{i-2} durch (6.2) und r_{i-1} durch (6.3):

$$r_i = (s_{i-2}r_0 + t_{i-2}r_1) - q_{i-1}(s_{i-1}r_0 + t_{i-1}r_1)$$

Wenn wir nun die Terme umordnen, bekommen wir das gewünschte Resultat:

$$r_i = [s_{i-2} - q_{i-1}s_{i-1}]r_0 + [t_{i-2} - q_{i-1}t_{i-1}]r_1 \tag{6.5}$$
$$r_i = [s_i]r_0 + [t_i]r_1$$

Gleichung (6.5) gibt uns direkt die rekursive Formel für die Berechnung von s_i und t_i: $s_i = s_{i-2} - q_{i-1}s_{i-1}$ und $t_i = t_{i-2} - q_{i-1}t_{i-1}$. Diese Rekursionen gelten für alle Indizes $i \geq 2$. Wie bei jeder rekursiven Formel benötigen wir Startwerte für s_0, s_1, t_0, t_1. Wie wir in Aufgabe (6.13) herleiten werden, sind die Startwerte gegeben durch $s_0 = 1$, $s_1 = 0$, $t_0 = 0$, $t_1 = 1$.

Erweiterter euklidischer Algorithmus (EEA)

Eingang: Positive ganze Zahl r_0 und r_1 mit $r_0 > r_1$

Ausgang: $\text{ggT}(r_0, r_1)$ sowie s und t mit $\text{ggT}(r_0, r_1) = s \cdot r_0 + t \cdot r_1$

Initialisierung: $s_0 = 1 \quad t_0 = 0$

$s_1 = 0 \quad t_1 = 1$

$i = 1$

Algorithmus:

1. DO
   - 1.1 $i = i + 1$
   - 1.2 $r_i = r_{i-2} \bmod r_{i-1}$
   - 1.3 $q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
   - 1.4 $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
   - 1.5 $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$

   WHILE $r_i \neq 0$
2. RETURN

   $\text{ggT}(r_0, r_1) = r_{i-1}$

   $s = s_{i-1}$

   $t = t_{i-1}$

Die Hauptanwendung des EEA in der asymmetrischen Kryptografie ist, wie bereits oben erwähnt, die Berechnung der modularen Inversen. Wir haben dieses Problem bereits im Zusammenhang mit der affinen Chiffre in Kap. 1 kennengelernt. Für die affine Chiffre mussten wir die Inverse des Schlüsselwerts a modulo 26 berechnen. Mit dem euklidischen Algorithmus können wir genau diese Berechnung ausführen, auch für sehr große Zahlen. Nehmen wir an, wir suchen die Inverse von r_1 mod r_0, wobei $r_1 < r_0$. Die Inverse existiert nur dann, wenn $\text{ggT}(r_0, r_1) = 1$ (vgl. Abschn. 1.4.2). Durch Anwendung des EEA erhalten wir $s \cdot r_0 + t \cdot r_1 = 1 = \text{ggT}(r_0, r_1)$. Wenn wir die Gleichung nun modulo r_0 nehmen, ergibt sich:

$$\begin{aligned} s \cdot r_0 + t \cdot r_1 &= 1 \\ s \cdot 0 + t \cdot r_1 &\equiv 1 \bmod r_0 \\ r_1 \cdot t &\equiv 1 \bmod r_0 \end{aligned} \tag{6.6}$$

Gleichung (6.6) enthält genau die Definition der Inversen von r_1, d. h. t selbst ist die Inverse von r_1:

$$t = r_1^{-1} \bmod r_0$$

Wenn wir eine Inverse $a^{-1} \bmod m$ berechnen wollen, wenden wir daher den EEA mit den Eingaben m und a an. Der Parameter t, der von dem Algorithmus berechnet wird, ist die Inverse. Es folgt ein Beispiel hierzu.

Beispiel 6.6 Unser Ziel ist die Berechnung von $12^{-1} \bmod 67$. Die Werte 12 und 67 sind teilerfremd, d. h. $\text{ggT}(67, 12) = 1$. Der EEA berechnet die Koeffizienten s und t in der Gleichung $\text{ggT}(67, 12) = 1 = s \cdot 67 + t \cdot 12$. Mit den Startwerten $r_0 = 67$ und $r_1 = 12$ führt der Algorithmus die folgenden Iterationen aus:

i	q_{i-1}	r_i	s_i	t_i
2	5	7	1	−5
3	1	5	−1	6
4	1	2	2	−11
5	2	1	−5	**28**

Dies gibt uns die Linearkombination

$$-5 \cdot 67 + 28 \cdot 12 = 1$$

Wie oben gezeigt folgt hieraus unmittelbar die Inverse von 12 als

$$12^{-1} \equiv 28 \mod 67$$

Dieses Ergebnis kann einfach verifiziert werden als

$$28 \cdot 12 = 336 \equiv 1 \mod 67$$

Der Koeffizient s wird für die Inverse nicht benötigt und wird daher häufig gar nicht erst berechnet. Man beachte auch, dass das Resultat des Algorithmus für t einen negativen Wert annehmen kann. Dennoch ist das Ergebnis korrekt. Um eine positive Inverse zu bekommen, müssen wir nur $t = t + r_0$ berechnen, was eine gültige Operation ist, da $t \equiv t + r_0 \bmod r_0$.

Der Vollständigkeit halber skizzieren wir, wie der EEA auch für die Berechnungen von multiplikativen Inversen in endlichen Körpern $GF(2^m)$ verwendet werden kann. In der modernen Kryptografie ist dies beispielsweise relevant für die Herleitung der AES-S-Boxen und für Kryptoverfahren basierend auf elliptische Kurven. Der EEA kann vollständig analog für Polynome anstelle für ganze Zahlen verwendet werden. Wenn wir eine Inverse für das Element $A(x)$ in einem endlichen Körper $GF(2^m)$ berechnen wollen, sind die Eingaben des Algorithmus das Element $A(x)$ und das irreduzible Polynom $P(x)$. Der EEA berechnet sowohl die Hilfspolynome $s(x)$ und $t(x)$ als auch den größten gemeinsamen Teiler $\text{ggT}(P(x), A(x))$:

$$s(x)P(x) + t(x)A(x) = \text{ggT}(P(x), A(x)) = 1$$

Man beachte, dass der ggT immer gleich 1 ist, da $P(x)$ irreduzibel ist. Wenn wir in der obigen Gleichung beide Seiten modulo $P(x)$ reduzieren, sieht man direkt, dass das Hilfspolynom $t(x)$ die Inverse von $A(x)$ ist:

$$\begin{aligned} s(x)\,0 + t(x)\,A(x) &\equiv 1 \bmod P(x) \\ t(x) &\equiv A^{-1}(x) \bmod P(x) \end{aligned}$$

An dieser Stelle zeigen wir den Algorithmus anhand eines Beispiels in dem kleinen Körper $GF(2^3)$.

Beispiel 6.7 Gesucht ist die Inverse von $A(x) = x^2$ im endlichen Körper $GF(2^3)$ mit $P(x) = x^3 + x + 1$. Die Startwerte für das Polynom $t(x)$ sind: $t_0(x) = 0$, $t_1(x) = 1$.

Iteration	$r_{i-2}(x) = [q_{i-1}(x)]\, r_{i-1}(x) + [r_i(x)]$	$t_i(x)$
2	$x^3 + x + 1 = [x]\, x^2 + [x + 1]$	$t_2 = t_0 - q_1 t_1 = 0 - x\,1 \equiv x$
3	$x^2 = [x]\,(x + 1) + [x]$	$t_3 = t_1 - q_2 t_2 = 1 - x\,(x) \equiv 1 + x^2$
4	$x + 1 = [1]\, x + [1]$	$t_4 = t_2 - q_3 t_3 = x - 1\,(1 + x^2) \equiv 1 + x + x^2$
5	$x = [x]\,1 + [0]$	Abbruch, da $r_5 = 0$

Man beachte, dass die Koeffizienten des Polynoms in $GF(2)$ berechnet werden. Da Addition und Subtraktion dieselbe Operation sind, können wir immer einen negativen Koeffizienten (wie z. B. $-x$) durch einen positiven Koeffizienten austauschen. Der neue Quotient und der neue Rest, die in jeder Iteration berechnet werden, sind in eckigen Klammern dargestellt. Die Polynome $t_i(x)$ werden nach der gleichen rekursiven Formel berechnet, die wir auch für den EEA mit ganzen Zahlen früher in diesem Abschnitt benutzt haben.

Der EEA bricht ab, wenn der Rest den Wert 0 hat, was in der fünften Iteration der Fall ist. Nun ist die Inverse gegeben durch den letzten Wert $t_i(x)$, der berechnet wurde, d. h. $t_4(x)$:

$$A^{-1}(x) = t(x) = t_4(x) = x^2 + x + 1$$

Wir verfizieren nun, ob $t(x)$ tatsächlich die Inverse von x^2 ist, wobei wir die Eigenschaften $x^3 \equiv x + 1 \bmod P(x)$ und $x^4 \equiv x^2 + x \bmod P(x)$ nutzen:

$$\begin{aligned} t_4(x) \cdot x^2 &= x^4 + x^3 + x^2 \\ &\equiv (x^2 + x) + (x + 1) + x^2 \bmod P(x) \\ &\equiv 1 \bmod P(x) \end{aligned}$$

In jeder Iteration des EEA wird Polynomdivision verwendet (im Beispiel oben nicht gezeigt), um den neuen Quotienten $q_{i-1}(x)$ und den neuen Rest $r_i(x)$ zu berechnen. Die Inversen in Tab. 4.2 aus Kap. 4 wurden mit dem erweiterten euklidischen Algorithmus berechnet.

6.3.3 Die eulersche Phi-Funktion

Wir führen nun ein weiteres hilfreiches Werkzeug für asymmetrische Kryptosysteme, insbesondere RSA, ein. Wir betrachten den Ring $\mathbb{Z}_m$, d. h. die Menge der ganzen Zahlen $\{0, 1, \ldots, m-1\}$. Uns interessiert die (momentan noch etwas merkwürdig anmutende) Fragestellung, *wie viele* Zahlen in dieser Menge teilerfremd zu m sind. Diese Anzahl ist durch die *eulersche Phi-Funktion* gegeben, die wie folgt definiert ist:

Definition 6.2 (Eulersche Phi-Funktion)
Die Anzahl der ganzen Zahlen in $\mathbb{Z}_m$, die teilerfremd zu m sind, wird mit $\Phi(m)$ bezeichnet.

Es folgen zwei Beispiele, in denen die eulersche Phi-Funktion durch einfaches Abzählen aller ganzen Zahlen in $\mathbb{Z}_m$, die teilerfremd zu m sind, berechnet wird.

Beispiel 6.8 Sei $m = 6$. Der dazugehörige Ring ist $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$.

$$\begin{aligned}
\text{ggT}(0, 6) &= 6 \\
\text{ggT}(1, 6) &= 1 \quad \star \\
\text{ggT}(2, 6) &= 2 \\
\text{ggT}(3, 6) &= 3 \\
\text{ggT}(4, 6) &= 2 \\
\text{ggT}(5, 6) &= 1 \quad \star
\end{aligned}$$

Da es zwei Zahlen in der Menge gibt, die teilerfremd zu 6 sind, nämlich 1 und 5, nimmt die Phi-Funktion den Wert 2 an, d. h. $\Phi(6) = 2$.

Hier ist ein weiteres Beispiel:

Beispiel 6.9 Sei $m = 5$. Der dazugehörige Ring ist $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$.

$$\begin{aligned}
\text{ggT}(0, 5) &= 5 \\
\text{ggT}(1, 5) &= 1 \quad \star \\
\text{ggT}(2, 5) &= 1 \quad \star \\
\text{ggT}(3, 5) &= 1 \quad \star \\
\text{ggT}(4, 5) &= 1 \quad \star
\end{aligned}$$

Dieses Mal haben wir vier Zahlen, die zu 5 teilerfremd sind, und daher $\Phi(5) = 4$.

Aus den oben stehenden Beispielen kann man ahnen, dass die Berechnung der eulerschen Phi-Funktion für große Zahlen m sehr langsam ist, wenn man für jedes Element eine ggT-Berechnung ausführen muss. Tatsächlich ist die Berechnung der eulerschen Phi-Funktion über diesen naheliegenden Ansatz unmöglich für die großen Zahlen, die in der asymmetrischen Kryptografie benötigt werden. Die Funktion kann allerdings sehr einfach berechnet werden, wenn die Faktorisierung von m bekannt ist, wie der folgende Satz zeigt.

Satz 6.1
m habe folgende kanonische Faktorisierung

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n},$$

wobei p_i unterschiedliche Primzahlen und e_i natürliche Zahlen sind. Dann gilt

$$\Phi(m) = \prod_{i=1}^{n} (p_i^{e_i} - p_i^{e_i - 1})$$

Da der Wert von n, d. h. die Anzahl der unterschiedlichen Primfaktoren, auch für große Zahlen m recht klein ist, ist das Produkt $\prod$ einfach zu berechnen. Schauen wir uns ein Beispiel an, in dem wir die eulersche Phi-Funktion berechnen:

Beispiel 6.10 Sei $m = 240$. Die Faktorisierung von 240 in der kanonischen Form ist

$$m = 240 = 16 \cdot 15 = 2^4 \cdot 3 \cdot 5 = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3}$$

Es gibt drei unterschiedliche Primfaktoren, d. h. $n = 3$. Der Wert der eulerschen Phi-Funktion ergibt sich damit als:

$$\Phi(m) = (2^4 - 2^3)(3^1 - 3^0)(5^1 - 5^0) = 8 \cdot 2 \cdot 4 = 64.$$

Das Ergebnis bedeutet, dass 64 ganze Zahlen in dem Bereich $\{0, 1, \ldots, 239\}$ teilerfremd zu $m = 240$ sind. Die alternative Methode, alle 240 Elemente mithilfe von ggT-Berechnungen zu überprüfen, wäre schon für die vergleichsweise kleine Zahl von 240 wesentlich langsamer gewesen.

Es ist wichtig zu betonen, dass wir die Faktorisierung von m kennen müssen, um die eulersche Phi-Funktion auf diese schnelle Art und Weise berechnen zu können. Wie wir im nächsten Kapitel sehen werden, ist diese Eigenschaft der Kern des asymmetrischen RSA-Verfahrens: Wenn wir die Faktorisierung einer bestimmten Zahl kennen, kann die eulersche Phi-Funktion effizient berechnet und das Chiffrat entschlüsselt werden. Ist die Faktorisierung hingegen nicht bekannt, kann die Phi-Funktion nicht berechnet werden und eine Dechiffrierung ist nicht möglich.

6.3.4 Der kleine fermatsche Satz und der Satz von Euler

Nachfolgend werden zwei weitere für die asymmetrische Kryptografie nützliche Sätze eingeführt. Wir beginnen mit dem *kleinen fermatschen Satz*[1]. Der Satz ist hilfreich für Primzahltests und für viele andere Aspekte der asymmetrischen Kryptografie. Der Satz gibt uns ein überraschend erscheinendes Ergebnis, wenn wir modulo einer ganzen Zahl exponentieren.

Satz 6.2 (Kleiner fermatscher Satz)
Seien a eine ganze Zahl und p eine Primzahl, dann gilt:

$$a^p \equiv a \pmod{p}$$

Da in endlichen Körpern $GF(p)$ modulo p gerechnet wird, gilt der Satz für alle ganzen Zahlen a, die Elemente eines endlichen Körpers $GF(p)$ sind. Der Satz kann auch in folgender Form geschrieben werden:

$$a^{p-1} \equiv 1 \pmod{p},$$

was oft nützlich in der Kryptografie ist. Eine Anwendung ist die Berechnung der Inversen in einem endlichen Körper. Wir können die Gleichung umschreiben als $a \cdot a^{p-2} \equiv 1 \pmod{p}$. Dies ist genau die Definition der multiplikativen Inversen und es folgt ein Weg für die Berechnung der Inversen einer ganzen Zahl a modulo einer Primzahl:

$$a^{-1} \equiv a^{p-2} \pmod{p} \tag{6.7}$$

Man beachte, dass diese Inversenberechnung nur möglich ist, wenn p eine Primzahl ist. Schauen wir uns ein Beispiel an:

Beispiel 6.11 Seien $p = 7$ und $a = 2$. Wir können die Inverse von a wie folgt berechnen:

$$a^{p-2} = 2^5 = 32 \equiv 4 \bmod 7$$

Das Ergebnis ist einfach zu überprüfen: $2 \cdot 4 \equiv 1 \bmod 7$.

Die Exponentiation in (6.7) durchzuführen ist üblicherweise langsamer, als den erweiterten euklidischen Algorithmus für die Bestimmung der Inversen zu nutzen. Dennoch

[1] Dieser sollte nicht mit dem großen fermatschen Satz (auch nur fermatscher Satz genannt) verwechselt werden, einem der berühmtesten zahlentheoretischen Probleme, der in den 1990er-Jahren nach 350 Jahren bewiesen wurde.

gibt es Situationen, in denen die Verwendung von Fermats kleinem Satz vorteilhaft ist, wie z. B. auf Smartcards oder anderen Geräten, die ohnehin einen Hardwarebeschleuniger für eine schnelle Exponentiation haben. Dies ist nicht unüblich, da die meisten asymmetrischen Algorithmen Exponentiation benötigen, wie wir in den folgenden Kapiteln sehen werden.

Eine Verallgemeinerung des kleinen fermatschen Satzes auf beliebige ganzzahlige Moduln, d. h. Moduln, die nicht prim sind, ist *der Satz von Euler*.

Satz 6.3 (Satz von Euler)
Seien a und m ganze Zahlen mit $\text{ggT}(a, m) = 1$. Dann gilt:

$$a^{\Phi(m)} \equiv 1 \pmod{m}$$

Da man hier modulo m rechnet, ist der Satz anwendbar auf Ringe ganzer Zahlen $\mathbb{Z}_m$. Wir zeigen nun ein Beispiel für Eulers Satz mit kleinen Werten.

Beispiel 6.12 Seien $m = 12$ und $a = 5$. Zuerst berechnen wir die eulersche Phi-Funktion von m:

$$\Phi(12) = \Phi(2^2 \cdot 3) = (2^2 - 2^1)(3^1 - 3^0) = (4 - 2)(3 - 1) = 4.$$

Nun können wir den Satz von Euler verifizieren:

$$5^{\Phi(12)} = 5^4 = 25^2 = 625 \equiv 1 \bmod 12.$$

Es ist einfach zu zeigen, dass der kleine fermatsche Satz ein Spezialfall des Satzes von Euler ist. Wenn p eine Primzahl ist, gilt $\Phi(p) = (p^1 - p^0) = p - 1$. Wenn wir diesen Wert in den Satz von Euler einsetzen, erhalten wir $a^{\Phi(p)} = a^{p-1} \equiv 1 \pmod{p}$, was genau den kleinen fermatschen Satz ergibt.

6.4 Diskussion und Literaturempfehlungen

Asymmetrische Kryptografie im Allgemeinen Asymmetrische Kryptografie wurde in der richtungsweisenden Veröffentlichung von Whitfield Diffie und Martin Hellman [3] eingeführt. Unabhängig davon hat Ralph Merkle das Konzept der asymmetrischen Kryptografie erfunden, dafür allerdings einen völlig anderen asymmetrischen Algorithmus vorgeschlagen [5]. Es gibt eine Anzahl von guten Quellen zur Geschichte der asymmetrischen Kryptografie. Die Abhandlung in [2] von Diffie ist sehr zu empfehlen. Ein weiterer guter Überblick ist [6]. Eine sehr detaillierte Schilderung der Geschichte der Kryptografie mit