Kryptographie

Sommersemester 2024

Dr Dirk Frettlöh
Technische Fakultät
Universität Bielefeld

24. April 2024

Inhaltsverzeichnis

1 Einführung

Ziele dieser Veranstaltung sind die **theoretischen Grundlagen** der modernen Kryptographie, sowie die Bereitstellung der wichtigsten **Formeln** und **Algorithmen** (letztere in etwas grober Form, praktisch Pseudo-Pseudocode) Die Teilnehmer sollen so in die Lage versetzt werden, die Methoden zu verstehen und zu implementieren, sowie ihre Stärken und auch eventuelle Schwächen kenntnisreich zu beurteilen. Die konkrete Umsetzung, also das Implementieren selbst, oder gar praktische Angriffe (Stichwort: *penetration testing*) sind hier nicht Thema. 10. April

1.1 Ziele der Kryptographie

Ein zentrales Ziel in der Kryptographie ist das Übermitteln einer (verschlüsselten) Nachricht vom Absender (traditionell oft „Alice“) zum Empfänger (traditionell oft „Bob“), ohne dass jemand dazwischen, der die Nachricht abfängt (traditionell oft „Eve“) diese lesen kann. (Geben Sie mal in Ihrer Lieblingssuchmaschine in der Bildersuche „Alice Bob Eve“ ein!) Ein anderes Ziel ist gerade das unberechtigte Entschlüsseln einer verschlüsselten Nachricht. Weitere Ziele sind Authentifikation (“Ist Alice wirklich Alice?”) und Anonymität (“Alice möchte unerkannt bleiben”). Auf all diese Aspekte werden wir in dieser Vorlesung mehr oder weniger eingehen.

- Verschlüsselung (Kapitel 1,5,8)
- Entschlüsselung (Kapitel 5.1.1,5.2.1)
- Authentifikation (Kapitel 9.3)
- Anonymität (z.B. Kapitel 9.4, 9.6)

Viele der modernen Verfahren benötigen ein paar tricksige Zutaten, wie etwas Algebra und Zahlentheorie (Kapitel 2), Erzeugung von Zufallszahlen (Kapitel 4), Primzahltests (Kapitel 3) oder Hashfunktionen (Kapitel 7).

Alle vorgestellten Anwendungen erfordern eine Absprache zwischen Alice und Bob, eine Festlegung eines Verfahrens. Dieses nennen wir im Folgenden oft auch **Protokoll**. Viele Protokolle erfordern einen geheimen Schlüssel zum Ver- und Entschlüsseln der Nachricht. Klassisch (bis vor 50 Jahren) wurde ein und derselbe Schlüssel zum Ver- und Entschlüsseln benutzt. Solche Verfahren heißen **symmetrische Verfahren**. Verfahren, bei denen zum Verschlüsseln ein anderer Schlüssel benötigt wird als zum Entschlüsseln heißen **asymmetrische Verfahren**. Es ist manchmal hilfreich, das zu formalisieren. Daher ein paar Vereinbarungen für die gesamte Dauer der Veranstaltung:

- m (message) ist immer die eigentliche Nachricht. Falls es Text ist, ist diese immer in Kleinbuchstaben gesetzt (z.B. m =hallo)
- k (key) ist meistens ein Schlüssel. Bei asymmetrischen Verfahren heißen die (Teil-) Schlüssel oft e (encode) zum Verschlüsseln und d (decode) zum Entschlüsseln.
- Für das Verschlüsseln der Nachricht m mit dem Schlüssel k in einen Geheimtext c (ciphertext) schreiben wir oft $f(k, m)$. (Es ist also $c = f(k, m)$.)

- Für das Entschlüsseln des Geheimtexts c mit dem Schlüssel k schreiben wir oft $f^*(k,c)$. (Es ist also $m = f^*(k,c)$.)

Beispiel 1.1 (Cäsarcode)**.** Der Cäsarcode (nach dem römischen Kaiser) ist ein symmetrisches Verfahren. Er codiert Buchstaben für Buchstaben. Setzen wir a=0, b=1, ... , z=25, dann ist also $m \in \{0, 1, \ldots, 25\}$. Wir wählen $1 \leq k \leq 25$ und

$$f(k,m) = m + k \mod 26 \quad \text{und} \quad f^*(k,c) = c - k \mod 26$$

Also ist z.B. f(9,techfak) = CNLQOJT, und f^*(9,CNLQOJT) = techfak. (Nachrechnen bzw -zählen! Z.B. t = 19, $19 + 9 = 28 = 2 \mod 26$ = C usw)

Beispiel 1.2 (Vigenèrecode)**.** Dies ist eine Weiterentwicklung des Cäsarcodes aus dem 17.(?) Jhd. und auch ein symmetrisches Verfahren. Der Schlüssel ist ein Wort $k = (k_1, k_2, \ldots, k_\ell)$, z.B. $k = KEY$. Wir verschlüsseln nun $m = (m_1, m_2, \ldots, m_n)$ (mit $m_i \in \{0, 1, \ldots, 25\}$) als

$$f(k,m) = (m_1{+}k_1 \bmod 26, m_2{+}k_2 \bmod 26, \ldots, m_\ell{+}k_\ell \bmod 26, m_{\ell+1}{+}k_1 \bmod 26, m_{\ell+2}{+}k_2 \bmod 26, ...)$$

So ist etwa mit $k = KEY$ dann $f(k$,techfak) = DIARJYU (Nachrechnen! Z.B. t = 19, K = 10, also $19 + 10 = 29 = 3 \mod 26$ = D, dann e = 4, E = 4, also $4 + 4 = 8 \mod 26$ = I, dann c = 2, Y = 24, also $2 + 24 \equiv 26 \equiv 0$ = A usw)

Hier haben wir schon ein einfaches grundlegendes Prinzip benutzt: Wir übersetzen die Nachricht in Zahlen. Das ist einfach, darauf gehen wir nicht näher ein. Z.B. kann ein Text in UTF8 geschrieben werden, und damit als Hexadezimalzahl oder Binärzahl dargestellt werden. Im Folgenden gehen wir immer davon aus, dass das (a) einfach und (b) bereits passiert ist.

1.2 Was heißt „einen Code knacken“?

In der Vergangenheit gab es Codes, die *geheime* (und besonders komplizierte) Verfahren benutzten. Eine kurze Darstellung findet sich hier:
`https://www.math.uni-bielefeld.de/~frettloe/teach/panorama17.html` (Videos 16, 17 und 18 bzw Folien 16 und 17). Eine ausführlichere und sehr lesenswerte Darstellung findet in dem Buch *Geheime Botschaften* von Simon Singh.

Heute ist das Prinzip immer ein faires: Ein Verfahren gilt nur als sicher, wenn Eve das Verfahren bekannt ist, nur der Schlüssel nicht, und sie dennoch praktisch (dazu gleich mehr) keine Chance hat, Nachrichten zu entschlüsseln, bzw. genauer: den Schlüssel k bzw. e zu ermitteln (Warum? Siehe unten). Dabei unterscheidet man immer noch vier Szenarien:

- **Ciphertext only attack**: Eve kennt nur einen oder mehrere Geheimtexte.
- **Known plaintext attack**: Eve kennt ein oder mehrere Klartext-Geheimtext-Paare.
- **Chosen plaintext attack**: Eve kann f nutzen (kennt aber nicht den Schlüssel k bzw e). Eve kann z.B. den Text aaaa....aaa verschlüsseln und ihre Schlüsse ziehen.
- **Chosen ciphertext attack**: Eve kann f^* nutzen (kennt aber nicht den Schlüssel k bzw d). Eve kann z.B. den Text aaaa....aaa entschlüsseln und ihre Schlüsse ziehen.

Ein Verfahren gilt nur als sicher, falls es allen vier Szenarien widersteht. Jedoch:

1.3 Was heißt „sicher“?

„Definition“ Ein Verfahren gilt als effizient und sicher, falls für alle m das $f(e, m)$ einfach zu berechnen ist; $f^*(d, c)$ ebenso, falls man das d kennt; und $f^*(d, c)$ soll für fast alle c schwer zu berechnen sein, falls man das d nicht kennt.

Diese Definition erklärt die Begriffe „effizient“ und „sicher“ mittels anderer Begriffe wie „leicht“ und „schwer“ und „fast alle“. Zum Glück stellt die theoretische Informatik genaue Erklärungen der letzteren bereit.

Fast alle heißt für einen endlichen Wertebereich $X = \{0, 1, \ldots, N\}$ (also $m \in X$ bzw $c \in X$): der Anteil der Ausnahmen geht gegen 0 für $N \to \infty$.

Analog heißt für einen unendlichen Wertebereich „fast alle“, dass der Anteil der Ausnahmen bezüglich aller Werte gleich 0 ist.

Beispiel 1.3. Fast alle natürlichen Zahlen haben mehr als sechs Dezimalstellen. Denn:

$$\lim_{N\to\infty} \frac{\text{Anzahl der Zahlen} \leq N \text{ mit bis zu sechs Dezimalstellen}}{\text{Anzahl der Zahlen} \leq N} = \lim_{N\to\infty} \frac{1000000}{N} = 0.$$

Fast alle natürlichen Zahlen sind keine Primzahlen. Denn die Anzahl der Primzahlen zwischen 1 und N ist nach dem Primzahlsatz (Thm 3.1 auf Seite 14) $O(\frac{N}{\log N})$. Also ist ihr Anteil

$$\lim_{N\to\infty} \frac{\left(\frac{c\cdot N}{\log N}\right)}{N} = \lim_{N\to\infty} \frac{c}{\log N} = 0.$$

Leicht heißt in der Theorie: polynomiell berechenbar (in P). Es reicht auch: randomisiert polynomiell (RP, siehe unten “probabilistische Primzahltests”). In der Praxis heißt es: in realistischer Zeit berechenbar (Sekunden, Minuten, Tage, je nach Anwendung).

Schwer heißt in der Theorie: nicht in randomisiert polynomieller Zeit berechenbar (nicht in RP). Genauer: Für jeden polynomiellen randomisierten Algorithmus kann die Wahrscheinlichkeit des Erfolgs beliebig klein gemacht werden, indem der Wertebereich X — also N — vergrößert wird.

In der Praxis heißt schwer: nicht in vernünftiger Zeit berechenbar. (Jahre, Jahrmillionen). Der Unterschied ist für die Theorie nicht wichtig, für die Praxis schon: eine Identifikation im Internet soll in Millisekunden durchführbar sein. Eine kriegswichtige Nachricht soll dagegen auch innerhalb mehrerer Jahre nicht entschlüsselt werden können.

Leider ist die theoretische Anforderung „nicht in P“, bzw „nicht in RP“ schwierig nachzuweisen, da unbekannt ist, ob P≠NP. Für die Bedeutung von P, NP usw verweisen wir auf die Vorlesungen “Algorithmen und Datenstrukturen” bzw. “Grundlagen Theoretischer Informatik”.

Beispiel 1.4 (One time pad)**.** Ein symmetrisches Verfahren, und eines der wenigen Verfahren, von dem man beweisen kann, dass es sicher im Sinne unserer Definition ist, zumindest falls

- der Schlüssel k genau so lang ist wie die Nachricht m,
- der Schlüssel k nur einmal benutzt wird („one-time“),

- der Schlüssel k zufällig ist, sowie natürlich
- der Schlüssel k geheim ist.

Was hier „sicher“ heißt, lässt sich im Rahmen der Informationstheorie sogar noch weiter präzisieren. Das führt zu weit, aber es heißt im Wesentlichen, dass Eve aus dem Geheimtext c keinerlei Information über m bekommt (außer evtl die Länge).

One-Time-Pad Wir beschreiben das Verfahren hier für Binärstrings. Sei also $m \in \{0,1\}^n$. Dann muss auch $k \in \{0,1\}^n$ sein, und c entsteht einfach durch Addieren der Ziffern modulo 2. (Das ist dasselbe wie XOR). Also ist z.B. für $m = 0001\,1011$ und $k = 1011\,0100$

$$c = f(k,m) = 1010\,1111$$

Genauso gut könnte man etwa die Buchstaben des Alphabets durch Zahlen 0,1,2,...,25 darstellen. Das Wort m besteht dann aus den Zeichen $m_1, m_2, \ldots, m_n$, mit $m_i \in \{0,1,\ldots,25\}$. Der Schlüssel $k = k_0, k_1, \ldots, k_n$ muss dann auch n Zeichen haben, und c wird dann berechnet als $c_i = m_i + k_i \bmod 26$.

Sobald ein Klartext-Geheimtext-Paar (m,c) bekannt ist, kann man leicht den dazu genutzten Schlüssel k ermitteln (vgl Übung). Daher ist das Verfahren nur sicher, wenn man jeden Schlüssel nur einmal benutzt.

2 Mathematische Grundlagen

Es gibt nicht praktischeres als eine gute Theorie. *(Bob der Baumeister).*

17. April

In diesem Kapitel kommt viel vor, das auch in der Vorlesung „Diskrete Mathematik“ (Modul Vertiefung Mathematik für die Bioinformatik) vorkommt. Dieses sind wichtige Zutaten für die Beschreibung und Umsetzung der kryptographischen Protokolle in späteren Kapiteln. Vorausgesetzt werden die Konzepte Primzahl, Primzahlzerlegung, mod (= modulo), ggT (= größter gemeinsamer Teiler). Die Mächtigkeit einer Menge M (also Anzahl ihrer Elemente) wird mit $|M|$ bezeichnet.

Die erste Zutat für viele Verfahren ist “der Großvater aller Algorithmen”:

Erweiterter Euklidischer Algorithmus:
Berechnet zu $k,m \in \mathbb{N}$ den $\mathrm{ggT}(k,m)$ sowie Zahlen $c,d \in \mathbb{Z}$ mit $ck + dm = \mathrm{ggT}(k,m)$.
Seien $k,m \in \mathbb{N}$ gegeben $(k > m)$.

1. $a_1 := k,\ a_2 := m,\ n := 1.$
   $c_1 := 1,\ c_2 := 0,\ d_1 := 0,\ d_2 := 1.$
2. $n := n+1, \quad q_n := \max\{r \in \mathbb{N} \mid a_{n-1} - r a_n \geq 0\}$
3. $a_{n+1} := a_{n-1} - q_n a_n, \quad c_{n+1} := c_{n-1} - q_n c_n, \quad d_{n+1} := d_{n-1} - q_n d_n.$
4. Falls $a_{n+1} \neq 0$ weiter bei 2. Sonst STOP, Ausgabe ggT=a_n, sowie c_n, d_n.

Dann ist $c_n k + d_n m = \mathrm{ggT}(k,m)$.

Beispiel 2.1. Gesucht $c, d \in \mathbb{Z}$ mit $c \cdot 160 + d \cdot 7 = 1$.

n	a_n	q_n	c_n	d_n
1	160	\	1	0
2	7	22	0	1
3	6	1	1	-22
4	1	6	-1	23
5	0			

Also $(-1) \cdot 160 + 23 \cdot 7 = 1$.

Satz 2.1 (Chinesischer Restsatz). *Seien $a_1, \ldots, a_n \in \mathbb{N}_0$ und $p_1, \ldots, p_n$ paarweise teilerfremd (also $ggT(p_i, p_j) = 1$ für alle $1 \leq i < j \leq n$). Dann hat das Gleichungssytem*

$$\begin{aligned} & x \equiv a_1 \mod p_1 \\ \wedge \quad & x \equiv a_2 \mod p_2 \\ & \vdots \\ \wedge \quad & x \equiv a_n \mod p_n \end{aligned} \tag{1}$$

genau eine Lösung $x \bmod p_1 \cdot p_2 \cdots p_n$. (Alle Lösungen in $\mathbb{Z}$ sind also von der Form $x + k \cdot p_1 \cdot p_2 \cdots p_n$ für $k \in \mathbb{Z}$.)

Zur Schreibweise mit mod, die wir hier benutzen, siehe Bemerkung 2.2 auf Seite 9.

Beachte: dieser Satz liefert eine Eins-zu-Eins-Beziehung zwischen den Lösungen von (1) und den Elementen in $\{0, 1, 2, \ldots, p_1 \cdots p_n - 1\}$, denn jede der $p_1 \cdots p_n$ Möglichkeiten für $x \in \{0, 1, \ldots, p_1 \cdots p_n - 1\}$ liefert ein Gleichungssystem der Form (1) (mit $a_i = x \bmod p_i$), und jedes davon hat genau eine Lösung. (Keine zwei verschiedenen x können dasselbe Gleichungssystem liefern, da das dann ja zwei Lösungen hätte.)

Wir brauchen den chinesischen Restsatz im Folgenden oft in der Form

$$(x \equiv a \bmod p \quad \wedge \quad x \equiv a \bmod q) \Leftrightarrow x \equiv a \bmod p \cdot q$$

Der Beweis des Satzes ist nicht schwierig, siehe z.B. die englische Wikipedia. Fast der ganze Beweis steckt aber schon im Berechnen der Lösung x:

Berechnen einer Lösung: für $n = 2$ berechne eine Lösung k, m für $kp_1 + mp_2 = 1$ mit dem erweiterten euklidischen Algorithmus. Dann ist die gesuchte Lösung

$$z = a_2 k p_1 + a_1 m p_2.$$

Das ist klar, denn z.B. ist ja $kp_1 \equiv 0 \bmod p_1$ und daher $mp_2 \equiv mp_2 + 0 \equiv 1 \bmod p_1$, also ist

$$a_2 k p_1 + a_1 m p_2 \equiv a_2 \cdot 0 + a_1 \cdot 1 \equiv a_1 \mod p_1$$

Für $n > 2$ berechne analog zu oben eine Lösung z der ersten beiden Gleichungen. Das liefert ein Gleichungssytem

$$\begin{aligned} & x \equiv z \mod p_1 p_2 \\ \wedge \quad & x \equiv a_3 \mod p_3 \\ & \vdots \\ \wedge \quad & x \equiv a_n \mod p_n \end{aligned}$$

mit $n-1$ Gleichungen. Fahre fort, bis nur noch zwei Gleichungen übrig sind. Löse die dann wie oben. Das liefert genau eine Lösung $x \mod p_1 p_2 \cdots p_n$.

2.1 Modulare Arithmetik

Definition 2.1. Ein Paar $(G, \oplus)$ heißt **Gruppe**, falls G eine nichtleere Menge ist, $\oplus$ eine binäre Verknüpfung auf G (vornehm: $\oplus : G \times G \to G$, auf deutsch: für alle $a, b \in G$ soll $a \oplus b$ wieder ein Element von G sein), sowie

1. $\forall a, b, c \in G : \ (a \oplus b) \oplus c = a \oplus (b \oplus c)$ (Assoziativität)
2. $\exists e \in G : \ \forall a \in G : \ a \oplus e = a$ (neutrales Element)
3. $\forall a \in G \ \exists a^{-1} \in G : \ a \oplus a^{-1} = e$ (inverses Element)

Eine Gruppe heißt **abelsch** (oder kommutativ) falls auch gilt $\forall a, b \in G : \ a \oplus b = b \oplus a$.

Ein Tripel $(G, \oplus, \odot)$ heißt **Körper**, falls $(G, \oplus)$ und $(G \setminus \{0\}, \odot)$ abelsche Gruppen sind (dabei ist 0 das neutrale Element von $(G, \oplus)$) und außerdem

$$\forall a, b, c \in G : \ a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c) \qquad \text{(Distributivgesetz)}.$$

Ein Tripel $(G, \oplus, \odot)$ heißt **Ring**, falls $(G, \oplus)$ abelsche Gruppe ist, für $(G, \odot)$ das Assoziativgesetz gilt und außerdem

$$\forall a, b, c \in G : \ a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c) \text{ und } (b \oplus c) \odot a = (b \odot a) \oplus (c \odot a) \qquad \text{(Distributivgesetz)}.$$

Falls es ein Element $e \in G$ gibt mit $\forall g \in G : \ e \odot g = g = g \odot e$, dann heißt $(G, \oplus, \odot)$ ein **Ring mit 1**.

Beispiel 2.2. $(\mathbb{N}, +)$ ist keine Gruppe. $(\mathbb{Z}, +)$ und $(\mathbb{Q} \setminus \{0\}, \cdot)$ sind abelsche Gruppen. (Frage: Wer kennt eine nicht-abelsche Gruppe?)

$(\mathbb{Z}, +, \cdot)$ ist kein Körper, aber ein Ring. $(\mathbb{Q}, +, \cdot)$ und $(\mathbb{R}, +, \cdot)$ und $(\{0, 1\}$, XOR, AND) sind Körper. Jeder Körper ist natürlich auch ein Ring.

Wir brauchen im Folgenden ein paar konkrete Gruppen, Ringe und Körper.

1. $Z_N = (\{0, 1, \ldots, N-1\}, \oplus)$ heißt **Restklassengruppe** (von $\mathbb{Z} \bmod N$), wobei $a \oplus b = a + b \bmod N$ ist.
2. $Z_N^* = (\{a \in \{1, \ldots, N-1\} \mid \mathrm{ggT}(a, N) = 1\}, \odot)$ heißt **Einheitengruppe** von Z_N, wobei $a \odot b = a \cdot b \bmod N$ ist.
3. Sei p eine Primzahl. $\mathbb{F}_p = (\{0, 1, \ldots, p-1\}, \oplus, \odot)$ heißt **Restklassenkörper** (mod p, wobei $\oplus$ und $\odot$ wie oben).
4. Ist N keine Primzahl, dann ist $(Z_N, + \bmod N, \cdot \bmod N)$ kein Körper, sondern nur ein Ring. Der heißt **Restklassenring** (mod N).

Bemerkung 2.1. Der Einfachheit halber identifizieren wir im Folgenden auch gerne Z_N mit den Elementen von Z_N, also der Menge $\{0, 1, \ldots, N-1\}$; und Z_N^* mit den Elementen von Z_N^*. Dann müssen wir nicht immer schreiben "$g \in \{a \in \{0, 1, \ldots, N-1\} \mid \text{ggT}(a, N) = 1\}$", sondern schreiben einfach $g \in Z_N^*$.

Das Rechnen in diesen Gruppen, Ringen, Körpern sollte uns vertraut sein. Ein wichtiges Prinzip dabei ist, dass man fast alles modulo n reduzieren darf! So ist z.B.

$$17 \cdot 24 + 31 \equiv 2 \cdot 4 + 1 \equiv 8 + 1 \equiv 3 + 1 = 4 \bmod 5, \text{ oder}$$

$$2^{32} \equiv 2^{10} \cdot 2^{10} \cdot 2^{10} \cdot 2^2 \equiv 4 \cdot 4 \cdot 4 \cdot 4 \equiv 16 \cdot 16 \equiv 6 \cdot 6 \equiv 36 \equiv 6 \mod 10.$$

Obacht: Es ist aber nicht etwa $2^{17} \equiv 2^2 \bmod 5$! Dazu siehe unten den Satz von Euler-Fermat.

Bemerkung 2.2. Wir betrachten hier oft Gleichungen in Z_N oder in Z_N^*, betrachten also ganze Gleichungen modulo N. Anstatt hinter Zahl einzeln "mod N" zu schreiben (also z.B. $(2 \cdot 4) \bmod 5 + 3 \bmod 5 = 8 \bmod 5 + 3 \bmod 5 = 1$ oder $(2 \cdot 4)\%5 + 3\%5 = 8\%5 + 3\%5 = 1$), schreiben wir hier immer lieber $2 \cdot 4 + 3 \equiv 8 + 3 \equiv 1 \bmod 5$ (und meinen damit: alles in der Gleichung wird mod 5 betrachtet).

Außerdem schreiben wir statt $a, a \odot a, a \odot a \odot a, \ldots$ von nun an $a, a^2, a^3, \ldots$.

Das neutrale Element von Z_N ist natürlich 0, denn $a + 0 \equiv a \bmod N$. Das inverse Element zu a in Z_N ist natürlich $N - a$, denn $a + N - a \equiv 0 \bmod N$.

Manchmal ist es nützlich, in Z_N die Elemente als $-\lfloor \frac{N}{2} \rfloor, \ldots, -2, -1, 0, 1, 2, \ldots, \lfloor \frac{N}{2} \rfloor$ (falls N ungerade) bzw als $-\lfloor \frac{N}{2} + 1 \rfloor, \ldots, -2, -1, 0, 1, 2, \ldots, \lfloor \frac{N}{2} \rfloor$ (falls N gerade) aufzufassen.[1] Denn $-1 \equiv 4 \bmod 5$, also kann ich statt 4 auch -1 schreiben. Die Elemente von Z_5 sind also $\{0, 1, 2, 3, 4\}$, aber genausogut können wir $\{-2, 1, 0, 1, 2\}$ benutzen.

Berechnen der Inversen in Z_N^*: Das ist etwas tricksiger. Wie finde ich denn etwa das Inverse von $a = 7$ in Z_{17}^*? Also a^{-1} so dass $7 \cdot a^{-1} \equiv 1 \bmod 17$. Das geht mit dem erweiterten euklidischen Algorithmus: Bestimme c, d, so dass $cN + da = 1$ ist. (Klappt, da $\text{ggT}(a, N) = 1$.) Dann ist $d = a^{-1}$.

Es ist klar, dass das stimmt, denn $1 = cN + da \equiv da \bmod N$, also $da \equiv 1 \bmod N$.

Ein schönes Beispiel für das Zitat am Anfang dieses Kapitels ist folgendes Resultat. Wir werden das abstrakte Resultat nutzen, um konkrete Sätze über Einheitengruppen abzuleiten.

Satz 2.2 (Lagrange)**.** *Sei G eine endliche Gruppe. Für jede Untergruppe H von G gilt, dass $|H|$ ein Teiler von $|G|$ ist.*

Untergruppe heißt: eine Teilmenge H von G einer Gruppe $(G, \odot)$, so dass $(H, \odot)$ selbst wieder eine Gruppe ist. Insbesondere ist für jedes $g \in G$ die Menge $\{g, g^2, \ldots\}$ eine Untergruppe von G. Die Anzahl ihrer Elemente ist gerade die **Ordnung** von g (also das kleinste $n \geq 1$, so dass $g^n = e$). Damit erhält man als Folgerung:

Folgerung 2.3. *Ist G eine Gruppe und $|G|$ eine Primzahl, so hat jedes $g \in G \setminus \{e\}$ die Ordnung $|G|$.*

[1] Auf einem abstrakteren Level sind die Elemente von Z_N *Restklassen*. D.h. die 1 in Z_5 ist in Wirklichkeit die *Menge* $\{\ldots, -9, -4, 1, 6, 11, \ldots\}$ aller ganzen Zahlen, die gleich 1 modulo 5 sind. Für unsere Zwecke reicht die konkretere Auffassung der Elemente von Z_N als Zahlen.

Denn die möglichen Ordnungen sind nur 1 oder $|G|$, und die Ordnung 1 hat nur das neutrale Element. Mit dem Satz von Lagrange lassen sich viele andere Sätze beweisen, u.a. der Satz von Euler-Fermat.

Satz 2.4 (Euler-Fermat)**.** $\forall a \in Z_N^* : \; a^{\varphi(N)} \equiv 1 \bmod N$
Dabei ist $\varphi(N) := |\{a \in Z_N \mid \mathrm{ggT}(a, N) = 1\}|$*. (Also* $\varphi(N) = |Z_N^*|$*.)*

Das folgt direkt aus Folgerung 2.2, denn Z_N^* ist eine Gruppe mit $\varphi(N)$ Elementen, und man sieht leicht ein, dass für jedes a dann $\{a, a^2, \ldots, a^{\varphi(N)}\}$ eine Untergruppe von Z_N^* ist. Deren Ordnung muss ein Teiler von $\varphi(N)$ sein.

Das φ heißt auch **Eulersche Phi-Funktion**. Zum Berechnen der Werte kann man einfach alle Fälle mit $\mathrm{ggT}(a, N) = 1$ für $a = 1, 2, \ldots, N-1$ zählen. Oder man benutzt

$$\varphi(p^n) = (p-1)p^{n-1} \text{ für Primzahlen } p, \quad \varphi(pq) = \varphi(p)\varphi(q) \text{ für } \mathrm{ggT}(p, q) = 1.$$

Die letzte Gleichung folgt übrigens fix aus dem chinesischen Restsatz. In den für uns wichtigen Fällen werden die Formeln oben zu:

$$\varphi(p) = p - 1 \text{ für Primzahlen } p, \quad \varphi(pq) = (p-1)(q-1) \text{ für Primzahlen } p \neq q.$$

24. April

Ein wichtiger Begriff in der Gruppentheorie ist der des **Erzeugers**[2]. Ein Element a einer Gruppe $G = (M, \odot)$ heißt *Erzeuger* von G, falls $M = \{a, a^2, a^3, \ldots, a^{|G|} = e\}$.

Der Erzeuger einer Gruppe Z_N^* — falls es ihn gibt — heißt **Primitivwurzel**. In anderen Worten: ein $g \in \mathbb{Z}_N^*$ mit $\{g^n \mid n = 0, 1, \ldots, \varphi(N)\} = Z_N^*$ ist eine Primitivwurzel. Bezogen auf den Satz von Euler-Fermat heißt das also: falls es ein $g \in Z_N^*$ gibt mit $g^{\varphi(N)} \equiv 1 \bmod N$ und $g^n \not\equiv 1 \bmod N$ für alle $n = 1, 2, \ldots, \varphi(N) - 1$, dann ist dieses g eine Primitivwurzel.

Zu Primitivwurzeln gibt es sehr viele interessante und schwierige Fragen, und eine ausgefeilte und tiefe Theorie. (Für welches N hat Z_N^* Primitivwurzeln? Wenn ja, wie viele? Wie groß sind die bzgl N?)

Satz 2.5 (Gauss)**.** *Ist N eine Primzahl, so besitzt Z_N^* eine Primitivwurzel.*

Bemerkung 2.3. In der Kryptographie möchte man oft Gruppen G, so dass $|G|$ eine Primzahl ist, denn dann gibt es einen Erzeuger, siehe Satz von Lagrange (Satz 2.2). In der Tat ist dann *jedes* Element außer das neutrale ein Erzeuger von G.

2.2 Quadratische Reste

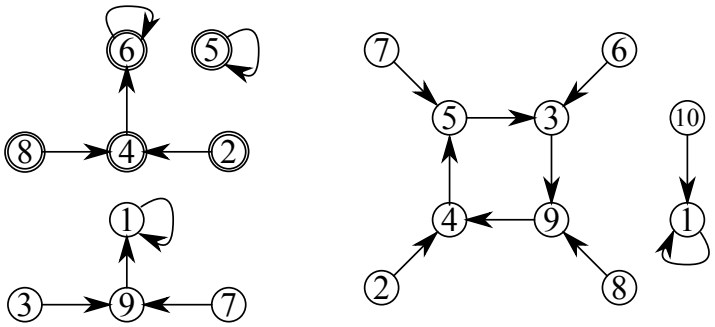
Obacht: In diesem Abschnitt rechnen wir immer im Ring $(Z_N, + \bmod N, \cdot \bmod N)$. Dann heißt z.B. a^2 immer $a \cdot a \bmod N$ usw. Wie in Bemerkung 2.1 steht dann Z_N einfach für die Menge $\{0, 1, \ldots, N-1\}$ usw.

Definition 2.2. Ein $a \in Z_N$ heißt **quadratischer Rest** (modulo N), falls es $b \in Z_N$ gibt mit $b^2 \equiv a \bmod N$. In diesem Fall heißt b auch **Quadratwurzel** von a modulo N.

[2]Es gibt auch Gruppen, die nicht einen, sondern zwei oder mehr Erzeuger haben. In Kapitel 6 sehen wir Beispiele für zwei Erzeuger, aber das führen wir hier nicht allgemein aus.

Beispiel 2.3. In Z_{18} ist 5 kein quadratischer Rest (durchprobieren!).
Dagegen ist 7 quadratischer Rest, denn $5^2 \equiv 25 \equiv 7 \bmod 18$, bzw $13^2 \equiv 169 \equiv -11 \equiv 7 \bmod 18$. Die Quadratwurzeln von 7 sind also 5 und 13.

Beispiel 2.4. Die Quadrat-Wurzel-Beziehung kann für kleine N schön visualisiert werden:



Im Bild geht ein Pfeil von Knoten a nach Knoten b, falls $a^2 \equiv b \bmod N$ ist; links für $N = 10$, rechts für $N = 11$. Elemente aus Z_N^* sind einfach umkringelt, Elemente aus $Z_N \setminus Z_N^*$ doppelt. Die 0 ist hier weggelassen, denn die ist immer quadratischer Rest, mit einziger Quadratwurzel 0.

Bemerkung 2.4. Im Allgemeinen sind beide folgende Probleme schwierig (aber nicht, falls N Primzahl):
1. Gegeben $a \in \mathbb{Z}_N$, ist a ein quadratischer Rest? (Mehr dazu im Buch von von zur Gathen.)
2. Falls ja, bestimme b mit $b^2 \equiv a \bmod N$.

Die Antwort zu 1 für $N = p$ Primzahl liefert das **Eulerkriterium:** Sei $a \not\equiv 0 \bmod p$. Falls dann

$$a^{\frac{p-1}{2}} \equiv 1 \bmod p$$

ist, dann ist a quadratischer Rest mod p. Falls a hingegen kein quadratischer Rest mod p ist, gilt $a^{\frac{p-1}{2}} \equiv -1 \bmod p$. Ein genaueres Bild liefert das folgende Resultat.

Satz 2.6. *Seien p und q ungerade Primzahlen, sei $N = pq$ und $0 \neq a$.*

1. Ist $a \in Z_p^*$ ein quadratischer Rest modulo p, dann hat a genau zwei Quadratwurzeln.

2. Ist $a \in Z_N^*$ ein quadratischer Rest modulo N, dann hat a genau vier Quadratwurzeln.

3. Ist $a \in Z_N \setminus Z_N^*, a \neq 0$ ein quadratischer Rest modulo N, dann hat a genau zwei Quadratwurzeln.

Beweis. Teil 1: Übung.

Zu Teil 2: Sei a quadratischer Rest in Z_N^*. Also gibt es $b \in Z_N$ mit $b^2 \equiv a \bmod pq$. Wir betrachten $a_1 \equiv a \bmod p$ und $a_2 \equiv a \bmod q$ und setzen $y = x^2$. Dann hat

$$y \equiv a_1 \bmod p \quad \text{und} \quad y \equiv a_2 \bmod q.$$

wegen des chinesischen Restsatzes genau eine Lösung y in $Z_{pq}^* = Z_N^*$. Allerdings gibt es nach Teil 1 zu dem a_1 genau zwei Lösungen $x_1, x_2 \in Z_p^*$ mit $y = x_i^2 \equiv a_1 \bmod p$, und genau zwei

Lösungen $x_3, x_4 \in Z_q^*$ mit $y = x_j^2 = a_2 \bmod q$. Diese insgesamt vier Möglichkeiten liefern vier verschiedene Gleichungssysteme

$$x \equiv x_i \bmod p \quad \text{und} \quad x \equiv x_j \bmod q \quad (i \in \{1,2\}, j \in \{3,4\}).$$

Nach dem chinesischen Restsatz 2.1 liefert jede davon ein $x \in Z_N^*$. Diese x sind, wegen der Bermerkung nach Satz 2.1, auch alle verschieden.

Zu Teil 3: das geht ganz analog, aber nun teilt entweder p oder q das a (denn $\mathrm{ggT}(a, pq) \neq 1$). Sagen wir ObdA p teilt a. Dann bekommen wir wie oben ein Gleichungssystem

$$x \equiv 0 \bmod p \quad \text{und} \quad x \equiv x_j \bmod q \quad (j \in \{3,4\}).$$

Analog zu oben hat das dann zwei Lösungen. □

2.2.1 *Quadratwurzeln mod n berechnen*

(2024 weggelassen.) Im Folgenden ist das Berechnen des Rechenaufwands wichtig. Wegen des fehlenden Beweises von P≠NP können wir nur relative Aussagen treffen der Art „A ist mindestens so schwer wie B“, oder “A ist genau so schwer wie B”.

Satz 2.7. *Seien p und q ungerade Primzahlen, und sei $N = pq$. Das Berechnen einer Quadratwurzel modulo N ist mindestens so schwierig wie N zu faktorisieren.*

Beweis. Angenommen wir können in Z_N^* effizient Quadratwurzeln berechnen. Dann liefert das den folgenden randomisierten polynomiellen Algorithmus zum Faktorisieren von N. (Mehr zu „randomisiert polynomiell“ in Kapitel 3.)

Wähle $b_1 \in Z_N^*$ zufällig. Berechne $a = b_1^2$. Wegen

$$(N-b)^2 \equiv N^2 - 2Nb + b^2 \equiv b^2 \mod N$$

sind die vier Quadratwurzeln von der Form $\{b_1, N-b_1, b_2, N-b_2\}$. Berechne nun eine der vier Quadratwurzeln von a. Weil b_1 zufällig war (!) gilt für das Ergebnis b_2 mit Wahrscheinlichkeit $\frac{1}{2}$, dass $b_2 \notin \{b_1, N-b_1\}$. (Falls doch: wähle ein neues $b_1 \in Z_n^*$.) Nun ist

$$(b_1+b_2)(b_1-b_2) \equiv b_1^2 - b_2^2 \equiv a - a \equiv 0 \mod N$$

(Außerdem ist $b_1 - b_2 \neq 0$, denn $b_1 \neq b_2$.) Also teilt $N = pq$ das Produkt $(b_1+b_2)(b_1-b_2)$. Daher teilt p (bzw q) den der Faktor b_1+b_2 (und q bzw p teilt den anderen Faktor b_1-b_2). Berechnen von $\mathrm{ggT}(b_1+b_2, n)$ liefert dann p (bzw q). Teilen wir N durch diese Zahl, erhalten wir den anderen Faktor.

Eine Runde dieses Algorithmus liefert mit Wahrscheinlichkeit $1-\frac{1}{2}$ die Faktorisierung. Daher liefern n Runden die Faktorisierung mit Wahrscheinlichkeit $1-\frac{1}{2^n}$. □

Genauso wie sich die Frage “ist a quadratischer Rest mod p” für Primzahlen p effizient beantworten lässt, lassen sich für eine Primzahl p effizient Quadratwurzeln aus einem quadratischen Rest mod p berechnen. In der Hälfte aller Fälle ist das besonders einfach:

Satz 2.8. *Sei p eine Primzahl mit $p \equiv 3 \bmod 4$, und sei x ein quadratischer Rest mod p. Dann ist $x^{\frac{p+1}{4}} \bmod p$ die Quadratwurzel von x mod p. Die andere Quadratwurzel ist $-(x^{\frac{p+1}{4}}) \bmod p$.*

Beweis. Wegen Satz 2.5 wissen wir, dass Z_p^* einen Erzeuger g hat. Alle Elemente von G sind also $g, g^2, g^3, g^4, \ldots$. Die quadratischen Reste sind dann genau die $g^2, g^4, \ldots$ (siehe Bemerkung 2.5 unten). Ist x quadratischer Rest, so ist also $x = g^{2k}$.

Falls $p \equiv 3 \bmod 4$, dann ist $\frac{p+1}{4}$ eine ganze Zahl. Insgesamt gilt

$$\left(x^{\frac{p+1}{4}}\right)^2 \equiv x^{\frac{p+1}{2}} \equiv x^{\frac{p-1}{2}} \cdot x \equiv (g^{2k})^{\frac{p-1}{2}} x \equiv (g^k)^{p-1} x \overset{(E.-F.)}{\equiv} 1 \cdot x \equiv x \mod p$$

Das "E.-F." heißt: wegen des Satzes von Euler Fermat (Satz 2.4). Für die andere Lösung $-x^{\frac{p+1}{4}}$ geht die Rechnung genauso (auch hier gilt "minus mal minus geich plus"). □

Für $p \equiv 1 \bmod 4$ braucht man bessere Tricks. Dazu benutzt man den Algorithmus von Tonelli-Shanks (s. wikipedia), oder den Algorithmus von Cipolla (s. wikipedia). Wegen Satz 2.7 gibt es (bisher bzw vermutlich) keinen allgemein effizienten Algorithmus für Quadratwurzeln modulo einer Nichtprimzahl N.

Im Beweis oben sahen wir einen sehr nützlichen Trick. Den wollen wir explizit aufschreiben.

Bemerkung 2.5. Ist g eine Primitivwurzel in Z_N^* (also ein Erzeuger von Z_N^*), dann sind alle quadratischen Reste in Z_N^* diese: $g^2, g^4, g^6, \ldots, g^{\varphi(N)-2}, g^{\varphi(N)} = 1$.

Das haben wir oben schon benutzt, aber man sieht es leicht ein: Einerseits ist klar, dass alle g^{2k} quadratische Reste sind (denn $g^{2k} = (g^k)^2$). Andererseits gibt es keine weiteren: denn falls $b \in Z_N^*$ ein quadratischer Rest ist, gibt es ja ein $c \in Z_N^*$, so dass $b = c^2$. Aber das c lässt sich ja darstellen als g^j (weil g ja Erzeuger von Z_N^* ist). Also ist $b = c^2 = g^{2j}$, also ist b schon in der Liste drin. (Falls $0 < 2j < \varphi(N)$ ist das klar. Falls $2j > \varphi(N)$, dann ist $0 < 2j - \varphi(N) < \varphi(N)$ und $2j - \varphi(N)$ ist gerade, also in der Liste drin.)

Damit versteht man auch das Eulerkriterium (siehe oben) ganz einfach. Falls a quadratischer Rest in Z_p^* (p Primzahl), dann ist nach der letzten Bemerkung $a = g^{2j}$, also ist

$$a^{\frac{p-1}{2}} \equiv (g^{2j})^{\frac{p-1}{2}} \equiv g^{j \cdot (p-1)} \equiv (g^{p-1})^j \overset{(E.-F.)}{\equiv} 1^j \equiv 1 \mod p.$$

Falls a kein quadratischer Rest in Z_p^* ist, dann ist es von der Form g^{2j+1}, und dann ist $a^{\frac{p-1}{2}}$ etwas anderes.

3 Primzahltests

8. Mai

Etliche der kryptographischen Verfahren unten enhalten einen Teil, wo Alice *zufällig* eine *große Primzahl* p wählt. Zu „zufällig“ siehe nächstes Kapitel. Zu „große Primzahl“ (z.B. ein p mit 512 bit) kann man sich zunächst fragen, ob es genug davon gibt. Das ist so, und das garantiert der **Primzahlsatz**. Dazu brauchen wir zwei Notationen: es sei $\pi(x)$ die Anzahl aller Primzahlen kleiner oder gleich x. (Hier sollen Primzahlen immer in $\mathbb{N}$ sein; wir wollen uns nicht mit der Frage rumärgern, ob -2 eine Primzahl ist.) Und es sei p_n die n-te Primzahl.

Satz 3.1. *Es gilt* $\pi(x) \approx \frac{x}{\ln(x)}$ *und* $p_n \approx n\ln(n)$. *Genauer gilt*

$$\frac{x}{\ln(x)}\left(1+\frac{1}{2\ln(x)}\right) < \pi(x) < \frac{x}{\ln(x)}\left(1+\frac{3}{2\ln(x)}\right) \quad \textit{falls } x \geq 59 \quad \textit{und}$$

$$n\left(\ln(n)+\ln(\ln(n))-\frac{3}{2}\right) < p_n < n\left(\ln(n)+\ln(\ln(n))-\frac{1}{2}\right) \quad \textit{falls } n \geq 20.$$

In dem Satz ist $\pi(x) \approx \frac{x}{\ln(x)}$ zu lesen als $\pi(x) = O(\frac{x}{\ln(x)})$, bzw besser als $\pi(x) = \Theta(\frac{x}{\ln(x)})$ (siehe A&D oder wikipedia).

Aus dem Satz folgt, dass eine Zahl nahe an einem gegebenen $x = 2^n$ mit Wahrscheinlichkeit $\frac{1}{\ln(2^n)} = \frac{1}{n\ln(2)}$ eine Primzahl ist. Wir brauchen also bei zufälliger Wahl also etwa $n\ln(2) = \ln(x)$ Versuche, bis wir auf eine Primzahl stoßen. (Was sich natürlich leicht verbessern lässt, wenn wir z.B. nur ungerade Zahlen ausprobieren.)

3.1 Fermat-Test

Es gibt zwar mittlerweile deterministische Primzahltests (Agarwal et al 2002: „PRIME is in P“), aber in der Praxis werden **probabilistische** Primzahltests benutzt. (Das ist ein Beispiel für einen **randomisierten** Algorithmus, vgl. auch den Beweis von Satz 2.7.) Der erste solche wurde 1974 gefunden (Solovay-Strassen 1977) und benutzt quadratische Reste. Wir zeigen hier einen anderen, der ähnlich, aber besser ist (Miller-Rabin). Dazu aber zunächst ein einführendes Beispiel.

Die Grundidee bei probabilistischen Tests (und allgemein bei randomisierten Algorithmen) ist, eine Bedingung zu prüfen, die einen Parameter a nutzt, und die eine Nichtprimzahl mit einer gewissen Wahrscheinlichkeit $1-p$ (z.B. mit $p = \frac{1}{2}$) entlarvt. Die Wahrscheinlichkeit, dass wir eine Nichtprimzahl *nicht* als solche entlarven, ist also p. Lassen wir diesen Test dann n-mal mit jeweils verschiedenen a laufen, beträgt die Wahrscheinlichkeit, dass wir eine Nichtprimzahl nicht als solche entlarven, p^n (also für $p = \frac{1}{2}$ ist's $\frac{1}{2^n}$). Also ist die Wahrscheinlichkeit, dass wir irrtümlich eine Nichtprimzahl als Primzahl einstufen, p^n.

Der allgemeine Test ist also:

Algorithmus 3.1. Probabilistischer Primzahltest:

1. Wiederhole n-mal:
   (a) Wähle a zufällig
   (b) Teste mittels a, ob N Nichtprimzahl ist. Falls ja: Stop, Ausgabe "N ist keine Primzahl".
2. Ausgabe „N ist mit Wahrscheinlichkeit $1 - p^n$ eine Primzahl".

Wir können das n also so wählen, dass wir nur mit Wahrscheinlichkeit 0,0000000000001 % falsch liegen. (Für $p = \frac{1}{2}$ reicht dazu $n = 47$.) Das reicht für praktische Zwecke aus (sogar für kryptographische). Wem das zu unsicher ist, mag sich überlegen, ob er/sie noch ein Auto oder ein Flugzeug besteigen sollte.

Ein erstes Beispiel für einen solchen Test ist der Fermat-Test. Der beruht auf Satz 2.4: ist p Primzahl, so ist für alle $a \neq 0$ doch $a^{p-1} \equiv 1 \bmod p$. Ist also $a^{N-1} \not\equiv 1 \bmod N$, kann N keine Primzahl sein.

Fermat-Test. Eingabe: eine ungerade Zahl $N > 3$.

1. Wähle zufällig $a \in \{2, 3, \ldots, N-2\}$.
2. Falls $\mathrm{ggT}(a, N) \neq 1$: Ausgabe: „$N$ ist keine Primzahl", sonst weiter:
3. Falls $a^{N-1} \neq 1 \bmod N$ Ausgabe: „N ist keine Primzahl", sonst: Ausgabe „N ist wahrscheinlich Primzahl".

Falls $\mathrm{ggT}(a, N) = k \neq 1$, dann hat N einen Teiler k mit $1 < k < N$, und der Test entlarvt N korrekt als Nichtprimzahl. Wir betrachten im Weiteren also nur noch a mit $\mathrm{ggT}(a, N) = 1$.

Eine Zahl a mit $a^{N-1} \not\equiv 1 \bmod N$ (und $\mathrm{ggT}(a, N) = 1$) heißt **Fermat-Zeuge** (dafür, dass N Nichtprimzahl ist). Falls N eine Nichtprimzahl ist, und $a^{N-1} \equiv 1 \bmod N$ ist, so heißt a **Fermat-Lügner**. Kennen wir einen Fermatzeugen für N, dann sagt uns der Fermattest, dass N garantiert eine Nichtprimzahl ist. (Es gibt keine *false positives*, höchstens *false negatives*. (Oder umgekehrt, je nachdem, was hier *positive* ist)).

Falls es einen Fermatzeugen gibt, dann gibt es viele. Denn: Sei G_N die Menge der Fermatlügner zu N. Also

$$G_N = \{a \in Z_N^* \mid a^{N-1} \equiv 1 \bmod N\}.$$

Das G_N ist eine Untergruppe von Z_N^* (Beweis: Übung). Falls N Primzahl ist, dann ist $G_N = Z_N^*$ (wegen Euler-Fermat). Falls N Nichtprimzahl ist, und es mindestens einen Zeugen dafür gibt, dann ist $G_N \neq Z_N^*$. Wegen des Satzes von Lagrange (Satz 2.2) ist $|G_N|$ ein Teiler von $|Z_N^*|$. In dem Fall kann $|G_N|$ höchstens $\frac{1}{2}|Z_N^*|$ Elemente haben. (Dazu überlege man sich: x ist ein Teiler von 100, aber $x \neq 100$. Wie groß kann x höchstens sein?) Die Zahl der Fermatlügner ist also höchstens $\frac{1}{2}|Z_N^*|$. Damit ist gezeigt:

Lemma 3.2. *Falls es mindestens einen Fermatzeugen gibt, dann gibt es sogar $\frac{1}{2}|Z_N^*|$ Stück. Der Fermattest entlarvt in diesem Fall eine Nichtprimzahl als solche mit einer Wahrscheinlichkeit von $p > \frac{1}{2}$.*

Die zweite Aussage folgt direkt aus der ersten, denn: Entweder ist $\mathrm{ggT}(a, N) \neq 1$ und N ist auf jeden Fall als Nichtprimzahl entlarvt. Oder es gilt $\mathrm{ggT}(a, N) = 1$, und in der Hälfte dieser Fälle ist N wieder als Nichtprimzahl entlarvt.

Die Frage bleibt: gibt es Nichtprimzahlen N, wo alle a mit $\mathrm{ggT}(a, N) = 1$ Fermatlügner für N sind? Die Antwort lautet: „Leider ja“. Diese Zahlen heißen **Carmichaelzahlen**. Die sind selten (man kann zeigen: alle Carmichaelzahlen sind ungerade, haben mindestens drei Primfaktoren, aber keinen doppelt), aber leider machen die den Fermat-Test untauglich: Falls wir aus Versehen eine Carmichaelzahl N wählen, wird der Fermattest liefern „N ist wahrscheinlich Primzahl“, solange wir nicht zufällig mal ein a wählen mit $\mathrm{ggT}(a, N) \neq 1$. (Letzteres ist für N mit wenigen großen Primfaktoren seeeeeehr unwahrscheinlich.) Leider wurde 1994 von Alford, Granville and Pomerance gezeigt, dass es unendlich viele Carmichaelzahlen gibt.

3.2 Miller-Rabin-Test

Ene Verbesserung des Fermattests läuft heute allgemein unter dem Namen Miller-Rabin-Test (obwohl da mehr als diese zwei Leute beteiligt waren, siehe S. 507 in von zur Gathen & Gerhard; dort heißt dieser Test „strong primality test“). Die Idee dabei ist eine Verfeinerung des Fermattests:

1. Prüfe, ob $\mathrm{ggT}(a, N) \neq 1$ (dann N keine Primzahl), sonst:
2. Prüfe, ob $a^{N-1} \neq 1 \bmod N$, (dann N keine Primzahl), sonst:
3. Prüfe, ob die 1 eine Quadratwurzel ungleich ± 1 hat, durch Berechnen von $a^{\frac{N-1}{2}}, a^{\frac{N-1}{4}}, \ldots$.

Nur, wenn bei allen drei Tests die Antwort “Nein” ist, liefert der Test “N ist wahrscheinlich Primzahl”. Ansonsten ist N als Nichtprimzahl entlarvt (vgl. Satz 2.6). Genau wie der Fermattest wird dieser Test in den allgemeinen probabilistischen Primzahltest 3.1 eingebaut. Das sieht folgendermaßen aus.

Miller-Rabin-Primzahltest. Eingabe: eine ungerade Zahl $N > 3$.

0. Sei k die größte Zahl mit $N - 1 = 2^k m$, wobei m ungerade.
1. Wähle zufällig $a \in \{2, 3, \ldots, N-2\}$.
2. Falls $\mathrm{ggT}(a, N) \neq 1$ Stop, Ausgabe „N ist Nichtprimzahl“.
3. Berechne $b_0 \equiv a^m \bmod N$. Falls $b_0 = 1$ Stop, Ausgabe „N ist wahrscheinlich Primzahl“.
4. Falls $b_0 \neq 1$: Setze $b_i := b_{i-1}^2 \bmod N$ für $i = 1, 2, \ldots, k$.
5. Falls $b_k = 1$ dann $j := \min\{0 \leq i \leq k-1 \mid b_{i+1} = 1\}$.
   Sonst (falls $b_k \neq 1$) Stop, Ausgabe „N ist Nichtprimzahl“.
6. Falls $b_j \equiv -1 \bmod N$: Ausgabe „N ist wahrscheinlich Primzahl“.
   Sonst Ausgabe „N ist Nichtprimzahl“.

In Schritt 5 wird das b_{i+1} mit dem kleinsten Index $i + 1$, das noch gleich 1 ist. Falls es ein solches gar nicht gibt, dann entlarvt bereits der Fermattest das N als Nichtprimzahl. Falls

doch: Das j wird dann auf i gesetzt. In anderen Worten: j ist der größte Index mit $b_j \neq 1$ (und $b_j + 1 = b_j^2 = 1$). In Schritt 6 prüfen wir dann, ob $b_j = -1$ ist. Falls nicht, dann hat 1 eine Quadratwurzel $\neq \pm 1 \bmod N$, also kann N keine Primzahl sein.

Beispiel 3.1. $N = 21$. Dann ist $N - 1 = 20 = 2^2 \cdot 5$. Also $k = 2, m = 5$. Wir berechnen

$$b_0 = a^5 \bmod 21, \quad b_1 = a^{10} \bmod 21, \quad b_2 = a^{20} \bmod 21.$$

Für $a = 8$ erhalten wir so $b_0 = 8, b_1 = 1, b_2 = 1$. Also ist $j = 0$, und wegen $b_0 \not\equiv -1 \bmod 21$ ist 21 als Nichtprimzahl entlarvt. (Und zwar hier, weil $8^2 = b_0^2 \equiv b_1 \equiv 1 \bmod 21$ ist, also 8 Quadratwurzel von 1 ist, und das ist unmöglich, falls N Primzahl wäre.)

Man kann zeigen:

Satz 3.3. *Falls N eine Primzahl ist, liefert der Miller-Rabin-Test „N ist wahrscheinlich Primzahl“. Falls N weder eine Primzahl noch eine Carmichaelzahl ist, liefert er „N ist Nichtprimzahl“ mit Wahrscheinlichkeit $p \geq \frac{3}{4}$. Falls N eine Carmichaelzahl ist, liefert er „N ist Nichtprimzahl“ mit Wahrscheinlichkeit $p \geq \frac{1}{2}$*

Die erste Aussage ist klar (wegen Euler-Fermat und Satz 2.6). Der zweite Teil für $p = 1/2$ ist sehr ähnlich (aber länglicher) wie der Beweis von Lemma 3.2. Zu $p = 3/4$ siehe Note 18.3 in von zur Gathen und Gerhard, das ist sehr schwierig. Der Beweis des dritten Teils ist länglich und technisch, siehe Theorem 18.6 in von zur Gathen und Gerhard. Die Grundidee ist dabei, dass das a zufällig gewählt wird. Daher gibt es ja für den Teil mit “Quadratwurzel von 1 ungleich ± 1” auf jeden Fall Zeugen, siehe wieder Satz 2.6.

Bemerkenswert ist, dass die Anzahl der Testdurchläufe, die wir brauchen, um eine vorgegebene Fehlerwahrscheinlichkeit zu erreichen, nicht von der Größe von N abhängt! Denn wir nutzen ja immer Aussagen der Art “50% der $a \in Z_N^*$ sind Zeugen”.

Bemerkung 3.1. Der erste effiziente probabilistische Primzahltest von Solovay und Strassen ist von 1977. Er funktioniert ähnlich, benutzt aber einen Zusammenhang mit quadratischen Resten. Für Details siehe z.B. das Buch von von zur Gathen.

4 Zufallszahlen auf dem Rechner

3. Mai

Zufallszahlen sind zu wichtig, um sie dem Zufall zu überlassen. *(Deutscher Lottoblock)*

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin. *(Papst Franziskus)*

Es ist einfach, Zufallszahlen selber herzustellen: Würfeln, Münzwurf, Lostrommel... Es ist aber erstaunlich schwierig, mit einem Computer, also mit einer deterministischen Maschine, echten Zufall herzustellen. (Es ist auch schwierig, sich Zufall “auszudenken” siehe `https://faculty.math.illinois.edu/~hildebr/fakerandomness/`)

Es gibt Hardwarelösungen für echten Zufall, z.B. `https://random.org`, oder USB-Sticks, die die zufälligen Spannungsunterschiede von “noisy diodes” verstärken und so echten Zufall erzeugen.

Es gibt Tricks, um (hoffentlich) zufällige Daten zu erzeugen, z.B. die Zeit zwischen zwei Tastenanschlägen messen, oder zwischen zwei Festplattenzugriffen, und davon die am wenigsten

signifikanten bits (lsb, für least significant bits) speichern. Da man so nur wenig Zufall sammeln kann, ist der Trick: wir machen aus wenig echtem Zufall deterministisch viel Zufall. Diese Tricks heißen **Pseudozufallsgeneratoren** (PRNG für Pseudo Random Number Generator).

Unter Linux wird (als Teil des Linuxkerns) in einem Pool echter Zufall gesammelt und daraus mittels eines PRNG Pseudozufall in `/dev/random` und in `/dev/urandom` abgelegt. Von dort wird z.B. beim erzeugen eines RSA-Schlüsselpaars mit `ssh-keygen` etwas Zufall ("seed") entnommen und mittels eines weiteren PRNG in viel Zufall verwandelt.

Wie aber erkennt man Zufall: Welche der folgenden Strings sind zufällig?

10101010101010101010, 11111111110000000000, 00100100001111110110, 01100010010110010110

Man kann "sieht zufällig aus" messen. Ein mögliches Maß ist die Entropie. Entropie ist ein Oberbegriff, der in vielen Feldern auftaucht (Thermodynamik, statistische Mechanik, dynamische Systeme). Meistens hat die Entropie Werte zwischen 0 und 1. Die generelle Idee ist: hohe Entropie (nahe 1) bedeutet viel Zufall, viel Information, viel Unordnung usw., niedrige Entropie (nahe 0) bedeutet wenig Zufall, wenig Information, wenig Unordnung. Um eine Idee zu bekommen zeigen wir hier zwei Begriffe: Shannon-Entropie und kombinatorische Entropie.

Shannon-Entropie: Die Shannon-Entropie misst das (Un-)Gleichgewicht der Anzahlen der vorkommenden Zeichen.
Gegeben sei eine endliche Zeichenkette (ein "Wort") $w = w_1 w_2 \cdots w_n$, wobei jedes w_i aus einem Alphabet mit b Zeichen kommt. (Für eine zehnstellige Binärzahl z.B. ist das Alphabet $\{0,1\}$, es ist also $b = 2$ und $n = 10$.) Dann ist die Shannon-Entropie eines Worts $w = w_1 w_2 \cdots w_n$ über einem Alphabet $\{0, 1, \ldots, b-1\}$ definiert als

$$H(w) = -\sum_{i=0}^{b-1} P_i \log_b(P_i), \qquad \text{wobei } P_i = \frac{1}{n}\big|\{k \mid w_k = i, k = 1, ..., n\}\big|.$$

Das P_i ist also einfach die relative Häufigkeit des Zeichens i in w. Dabei vereinbart man $0 \cdot \log_b(0) = 0$ (passend zum Grenzwert $\lim_{x\to 0} x \log_b(x)$).

Mit dieser Definition gilt für die Shannon-Entropie immer $0 \le H(w) \le 1$. Für andere Zwecke (z.B. "Information pro Zeichen in bit") gibt es andere Definitionen (siehe wikipedia: "Byte-Entropie").

Das Minus vor dem Term sorgt nur dafür, dass eine positive Zahl herauskommt (denn die Logarithmen werden aus Zahlen kleiner als 1 gezogen, sind also negativ).

Beispiel: Gegeben sei das Alphabet $\{0,1,2,3\}$ (also $b = 4$). Für $w = 0000000000000000$ ergibt sich $P_0 = \frac{16}{16} = 1$, $P_1 = P_2 = P_3 = 0$. Also

$$H(w) = -(1 \cdot \log_4 1 + 0 + 0 + 0) = 0.$$

Wie erwartet: niedrige Entropie.

Für $w' = 0123012301230123$ ergibt sich $P_0 = P_1 = P_2 = P_3 = \frac{1}{4}$, also

$$H(w') = -\Big(\frac{1}{4} \cdot \log_4 \frac{1}{4} + \frac{1}{4} \cdot \log_4 \frac{1}{4} + \frac{1}{4} \cdot \log_4 \frac{1}{4} + \frac{1}{4} \cdot \log_4 \frac{1}{4}\Big) = -\big(\log_4(\frac{1}{4})\big) = 1.$$

Wie vielleicht erwartet: hohe Entropie. Das Problem im Zusammenhang mit Zufall wird aber hier deutlich, sowohl in diesem Beispiel als auch in den beiden ersten 0-1-Beispielen oben: nicht immer bedeutet hohe Shannon-Entropie hoher Zufall. Daher:

Topologische Entropie: Die topologische Entropie misst für ein unendliches Wort den Anteil der vorkommenden Teil-Worte der Länge m zu allen möglichen Worten der Länge m. Gegeben sei eine unendliche Zeichenkette (ein "Wort") $w = w_1 w_2 \cdots$, wobei jedes w_i aus einem Alphabet mit b Zeichen kommt. Dann ist die topologische Entropie von w definiert als

$$h(w) = \lim_{m\to\infty} \frac{1}{m} \log_b p(w, m),$$

wobei $p(w, m)$ die Anzahl der verschiedenen Teilworte der Länge m in w bezeichnet.

Beispiel: Sei $w = w_1 w_2 \cdots$ ein zufälliges 0-1-Wort (also z.B. jedes w_i Ergebnis eines Münzwurfs mit einer fairen Münze). Dann sollten wir im allgemeinen erwarten, dass irgendwann jede mögliche Kombination (jedes endliche Wort) auftaucht. Es gibt 2^m verschiedene Worte der Länge m. Also ist

$$h(w) = \lim_{m\to\infty} \frac{1}{m} \log_2 2^m = \lim_{m\to\infty} \frac{m}{m} = 1.$$

Wie erwartet: maximale Entropie, also viel Zufall. Sei dagegen $w = 1010101010101\cdots$. Dann gibt es für jedes m nur zwei Worte der Länge m: entweder es fängt mit 0 an, oder mit 1. Der Rest liegt dann fest. Also ist hier

$$h(w') = \lim_{m\to\infty} \frac{1}{m} \log_2 2 = \lim_{m\to\infty} \frac{1}{m} = 0.$$

Für die topologische Entropie braucht man strenggenommen unendliche Worte. Sie kann aber auch näherungsweise für Worte der Länge n berechnet werden: Man nimmt halt nicht den Grenzwert, sondern zählt für ein gewisses sinnvolles m, z.B. $m = \sqrt{n}$. (Offenbar ist $m = n$ sinnlos, da ein Wort der Länge n nur ein einziges Teilwort der Länge n enthält, nämlich sich selbst.) Von der topologischen Entropie werden, wie erwartet, die ersten beiden Strings oben als "wenig zufällig" eingestuft, die beiden anderen als "sehr zufällig".

Ein einfacher PRNG, der in diesem Sinne viel Zufall erzeugt, ist dieser:

Linearer Kongruenzgenerator: Wähle $N \in \mathbb{N}$ (z.B. mit 64 bit, oder mit 1024 bit) sowie $x_0, s, t \in Z_N$ zufällig (etwa aus `/dev/urandom`). Berechne dann

$$x_j = s x_{j-1} + t \mod N \quad \text{für } j = 1, 2, \ldots$$

Für manche Zwecke reicht dieser PRNG völlig aus, z.B. für Primzahltests. Bezüglich der Entropie sieht das Ergebnis dieses Verfahrens zufällig aus. Aber bzgl. der Entropie sehen die Nachkommastellen von π (der dritte String oben) auch zufällig aus. Wenn nun Eve diese Zahlen belauscht: 00100100001111110110 und realisiert, dass das die Nachkommastellen von π sind (dazu reicht es diesen String zu googeln), dann kann sie die nächsten vorhersagen: ...10101000...

Genauso kann man aus vier aufeinanderfolgenden Werten $x_j, x_{j+1}, x_{j+2}, x_{j+3}$ einen guten Tipp für die Werte von m, s, t abgeben (von zur Gathen S. 454) und damit die nächsten Werte vorhersagen.

Für kryptographische Zwecke braucht daher man etwas besseres: unvorhersagbar. Das kann man quanitifizieren, indem man fordert, dass jeder polynomielle randomisierte Algorithmus nur einen kleinen Vorteil von ε gegenüber "raten" hat, und sich dieses ε beliebig klein machen

lässt. (Z.B. beim Münzwurf, also nur Werte 0 oder 1, soll für jeden solchen Algorithmus gelten, dass er nur mit Wahrscheinlichkeit $< \frac{1}{2} + \varepsilon$ richtig liegt.)

In diesem Sinne gibt es einige beweisbar gute PRNG. Beweisbar in dem Sinne: falls es einen polynomiellen probabilistischen Vorhersage-Algorithums gibt, der besser rät als $\frac{1}{2} + \varepsilon$, dann lässt sich daraus ein polynomieller probabilistischer Algorithmus ableiten, der ein als "schwer" vermutetes Problem löst (Faktorisieren großer Zahlen, diskrete Logarithmen...) Zwei gute PRNGs in diesem Sinne sind die beiden folgenden (Details zu beiden im Buch von von zur Gathen in Kapitel 11.):

Algorithmus 4.1. RSA-PRNG. Wir wählen $N = pq$ und e wie bei der RSA-Verschlüsselung und einen (echt) zufälligen Startwert $x_0 \in \{2, 3, \ldots, N-2\}$. Wir berechnen dann $x_1, x_2, \ldots$ als $x_{i+1} \equiv x_i^e \bmod N$ und geben die k kleinsten bits eines jeden x_i aus.

Algorithmus 4.2. Blum-Blum-Shub-generator. Wir wählen $N = pq$ mit zwei Primzahlen p, q so dass $p \equiv q \equiv 3 \bmod 4$. und einen (echt) zufälligen Startwert $x_0 \in \{2, 3, \ldots, N-2\}$. Wir berechnen dann $x_1, x_2, \ldots$ als $x_{i+1} \equiv x_i^2 \bmod N$ und geben jeweils das kleinste bit eines jeden x_i aus.

5 Grundlegende Public-Key-Verfahren

Can the reader say what two numbers multiplied together will produce the number 8616460799?
10. Mai *I think it unlikely that anyone but myself will ever know.* *William Stanley Jevons (1874)*

Public-Key-Verfahren sind asymmetrische Verfahren, mit zwei Schlüsseln, einem privaten Schlüssel d und einem öffentlichen Schlüssel e. Alice gibt ihren öffentlichen Schlüssel e allgemein bekannt (Homepage, Emailsignatur,...). Bob kann dann e nutzen, um Nachrichten zu verschlüsseln. Nur Alice kann dann die verschlüsselte Nachricht mittels d entschlüsseln.

Grundlage aller Public-Key-Verfahren sind **Einwegfunktionen**.

Definition 5.1. Eine Funktion $f : X \to Y$ heißt **Einwegfunktion**, falls für alle $x \in X$ das $f(x)$ leicht zu berechnen ist, und für fast alle $y \in Y$ das $f^{-1}(y)$ schwierig.

Zur Präzisierung von "leicht", "schwierig" und "fast alle" siehe Abschnitt 1.3.

Leider ist bis heute nicht bekannt, ob es echte Einwegfunktion gibt. (Falls P$\neq$NP dann lautet die Antwort "ja"). Es gibt aber einige Kandidaten, die in der Praxis als (anstatt von?) Einwegfunktionen benutzt werden:

- Multiplizieren/Faktorisieren (vgl. Blatt 1 Aufgabe 3 und Abschnitt 5.1)
- Potenzieren/Logarithmen modulo n (vgl. Abschnitt 5.2.1)
- Quadrieren/Quadratwurzel ziehen modulo n

Die ersten bekannten Public-Key Verfahren waren der Diffie-Hellman Schlüsseltausch und die Verschlüsselungverfahren RSA und El-Gamal. Die sind immer noch die in der Praxis

gebräuchlichsten (zusammen mit ihren elliptischen-Kurven-Varianten). Daher sehen wir uns die im Folgenden genauer an. Auch einige der gebräuchlichsten Authentifizierungsverfahren basieren auf diesen, ebenso fast alle gängige Krypto-Software wie https und gpg und ssh und bitcoin usw. Es gibt neben diesen dreien überhaupt nur wenige andere Public-Key Verfahren, siehe wikipedia: "Public key cryptography", und die spielen in der Praxis kaum eine Rolle. (Aber siehe AES usw).

5.1 RSA

Das erste Public-Key-Verfahren war das RSA-Verfahren, nach Rivest, Shamir, Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM* 21 (1978) 120-126. Kurioserweise ist das bereits 1973 von Clifford Cocks gefunden worden, einem Mitarbeiter des Britischen Geheimdiensts GCHQ (brit. Version der NSA). Das unterlag aber der Geheimhaltung und wurde erst 1997 publik.

RSA-Verschlüsselung: Vorab (muss nur einmal gemacht werden):

- Alice wählt zwei Primzahlen p und q (geheim).
- Alice berechnet $N = p \cdot q$ und $\varphi(N)$ als $(p-1)(q-1)$.
- Alice wählt $e \neq 1$ mit $\mathrm{ggT}(e, \varphi(N)) = 1$ und d mit $e \cdot d \equiv 1 \bmod \varphi(N)$.
- Alice gibt N und e öffentlich bekannt. $\varphi(N)$ und d sind geheim.

Dann für jede Nachricht einzeln:

- Bob kann nun die Botschaft blockweise jeweils in eine Zahl m verwandeln und m verschlüsseln als $m^e \bmod N$.
- Alice berechnet $(m^e)^d \equiv m^{ed} \equiv m \bmod N$.

Wie bei jedem Verschlüsselungsverfahren muss man sich fragen:

- Ist das Verfahren korrekt?
- Ist das Verfahren effizient? (Also Verschlüsseln; und Entschlüsseln mit Kenntnis von d)
- Ist das Verfahren sicher? (Also Entschlüsseln ohne Kenntnis von d schwierig)

Korrektheit: Warum klappt $m^{ed} \equiv m \bmod N$? Wegen des Satzes von Euler-Fermat: Ist $\mathrm{ggT}(m, N) = 1$, dann ist $m^{\varphi(N)} \equiv 1 \bmod N$. Damit gilt:

$N = pq$, p, q Primzahlen, also $\varphi(N) = (p-1)(q-1)$. Da $ed \equiv 1 \bmod \varphi(N)$ ist $ed = k\varphi(N)+1$ für ein $k \in \mathbb{N}$.

$$(m^e)^d \equiv m^{ed} \equiv m^{1+k\varphi(N)} \equiv m \cdot (m^{\varphi(N)})^k \equiv m \cdot 1^k \equiv m \mod N.$$

Dieser Beweis klappt nur für $\mathrm{ggT}(m, N) = 1$. Ist aber auch in den (wenigen!) anderen Fällen wahr: Sei also $\mathrm{ggT}(m, N) \neq 1$. Also ist dann $\mathrm{ggT}(m, N) \in \{p, q\}$. Sagen wir OBdA, dass $\mathrm{ggT}(m, N) = p$. Wegen des chinesischen Restsatzes gilt:

$$m^{ed} \equiv m \bmod pq \;\Leftrightarrow\; (m^{ed} \equiv m \bmod p \text{ und } m^{ed} \equiv m \bmod q)$$

Letzteres ist wegen $m \equiv 0 \bmod p$ und $ed \equiv 1 \bmod \varphi(N)$ äquivalent zu

$$0^{ed} \equiv 0 \bmod p \text{ und } m^{\ell(p-1)(q-1)+1} \equiv m \bmod q$$

Die erste Gleichung ist offenbar wahr. Die zweite auch, da man sie mit Euler-Fermat (für $\varphi(q) = q - 1$) schreiben kann als

$$\left(m^{q-1}\right)^{\ell(p-1)} m \equiv m \bmod q, \text{ also } 1^{\ell(p-1)} m \equiv m \bmod q$$

Also sind beide Gleichungen wahr, also auch die zu zeigende Gleichung.

Effizienz: Wie Alice hier effizient eine große Primzahl findet: Sie wählt zufällig eine Zahl und testet, ob sie eine Primzahl ist. Wie das effizient geschieht sahen wir in Kapitel 3. (Beachte: Faktorisieren ist schwierig, aber testen, ob eine Zahl prim ist, ist effizient möglich!)

Das e berechnet Alice analog: Wähle e zufällig, checke mit euklid. Alg. ob $\mathrm{ggT}(e, \varphi(N)) = 1$. Wenn nicht, wähle anderes e, solange bis es klappt.

Die Bedingung $e \cdot d = 1 \bmod \varphi(N)$ heißt ja nur, dass d das Inverse zu e in $Z^*_{\varphi(N)}$ ist. Das berechnet Alice mit dem erweiterten euklidischer Algorithmus (vgl. Aufgabe 5 auf Blatt 2).

Sicherheit: Die Sicherheit beruht auf der Annahme, dass das Faktorisieren von n schwierig ist. Heuristische Argumente machen das plausibel: Aufgabe 3 von Blatt 1, oder der Umstand, dass viele kluge Menschen es probiert haben. aber noch keinen effizienten Algorithmus zum Faktorisieren fanden.

Moderne Faktorisierungsverfahren (verteiltes Rechnen, schnelle Rechner, kluge Algorithmen) können heute 100-stellige oder 200-stellige Zahlen faktorisieren. (Weltrekord 2020[3]: Faktorisierung einer Zahl mit 829 Binärstellen, also 250 Dezimalstellen. Dazu mussten 3100 CPU-Kerne ein Jahr lang rechnen, um diese Zahl im Jahr 2019 in ihre beiden Primfaktoren zu zerlegen.) Daher empfiehlt sich bei RSA heute eine Schlüssellänge von 1024 bis 4096 bit für das N, also jeweils mehr als 512 bis 2048 bit für p und q.

Beispiel 5.1. Das Beispiel benutzt natürlich viel zu kleine Zahlen.

- Alice wählt $p = 17$ und $q = 11$ (geheim).
- Alice berechnet $N = 17 \cdot 11 = 187$ und $\varphi(N) = (p-1)(q-1) = 160$.
- Alice wählt $e = 7$ ($\mathrm{ggT}(e, \varphi(N)) = 1$, also OK), und berechnet d mit $7 \cdot d \equiv 1 \bmod 160$, also $d = 23$.
- Alice gibt $N = 187$ und $e = 7$ öffentlich bekannt.
- Bob kann nun eine Botschaft, z.B. X, in eine Zahl m verwandeln (z.B. In ASCII: $X = 88$) und verschlüsselt $88^7 \bmod 187$.
  - $88^7 = 88 \cdot 88^2 \cdot 88^4$
  - $88^2 \equiv 7744 \equiv 77 \bmod 187$
  - $88^4 \equiv 77^2 \equiv 5929 \equiv 132 \bmod 187$
  - Also $88^7 \equiv 88 \cdot 77 \cdot 132 \equiv 894432 \equiv 11 \bmod 187$
- Alice empfängt 11 und berechnet $11^{23} \equiv 11 \cdot 11^2 \cdot 11^4 \cdot 11^{16} \equiv 11 \cdot 121 \cdot 55 \cdot 154 \equiv 11273570 \equiv 88 \bmod 187$.

[3] siehe Wikipedia

5.1.1 Angriffe auf RSA

Da RSA einerseits wirklich schwierig zu knacken ist, andererseits aber in der Praxis vielfach eingesetzt wird (https, ssh, gpg...), wurden viele spezielle Szenarien ausprobiert. Ein paar einfache Ideen schildern wir in diesem Abschnitt. Mehr in von zur Gathen (Kap. 3).

Generell gilt: kein Public-Key-Verfahren erlaubt known-plaintext-Angriffe. (Darf nicht erlauben!) Denn jeder kann beliebig viele Klartext-Geheimtext-Paare erzeugen. Daher gibt es viele Szenarien für „RSA knacken“. Eve könnte etwa aus (e, N) und c

B1 die Nachricht m ermitteln,

B2 das d ermitteln,

B3 das $\varphi(N)$ ermitteln, oder

B4 einen Faktor p (oder q) von N.

Aus der Beschreibung von RSA ist leicht zu sehen, dass, wenn man B4 effizient lösen kann, dann auch B3; wenn man B3 kann, kann man B2; und wenn man B2 kann, kann man B1. (Genauer: es gibt jeweils polynomielle Reduktionen von B4 auf B3, von B3 auf B2, und von B2 auf B1). Für die umgekehrte Richtung kann man leicht sehen, dass B3 auch B4 ermöglicht.

Bemerkung 5.1. Die Faktorisierung einer Zahl $N = pq$ (p, q Primzahlen) ist genauso schwer zu berechnen wie $\varphi(N)$.

Beweis. Angenommen, wir kennen p und q, dann ist natürlich $\varphi(N) = (p-1)(q-1)$ einfach zu berechnen (Addition, Multiplikation). Umgekehrt, kennen wir den Wert von N und von $\varphi(N) = (p-1)(q-1) = pq - p - q + 1 = N - p - q + 1$, dann ist ja $p = N - \varphi(N) - q + 1$, und wegen $N = pq = (N - \varphi(N) - q + 1)q$. Also müssen wir die quadratische Gleichung

$$0 = -q^2 + (N - \varphi(N) + 1)q - N$$

lösen. Das geht effizient (p-q-Formel: Addition, Multiplikation, Quadratwurzel in $\mathbb{Z}$) □

Ebenso kann man zeigen, dass auch B2 und B3 gleich schwierig sind. Es ist eine offene Frage (und Gegenstand intensiver aktueller Forschung), ob auch B2 (und damit B3 und B4) auf B1 polynomiell reduzierbar ist oder nicht. Ob also, wenn man RSA effizient decodieren kann, man auch effizient faktorisieren kann.

Es ist klar, dass eine der Optionen B1 bis B4 heißt „RSA knacken“. Aber was ist mit folgenden Szenarien:

- Eve kann über m aussagen, ob das letzte bit 0 oder 1 ist; oder ob das Wort „hallo“ vorkommt; oder ob es deutscher Text ist.
- Eve kann aus vielen abgefangenen Botschaften (evtl mit vielen Empfängern) etwas über p oder q aussagen.
- Ein Angriff klappt nur für wenige Situationen, sagen wir 1% aller belauschten Nachrichten.

- Ein Angriff klappt nur für wenige Schlüssel (d oder e), sagen wir 1% aller Schlüssel.

Je nach Anwendung ist keines dieser Szenarien wünschenswert. Es folgen ein paar Beispiele für Angriffe auf RSA, die Eve in bestimmten Fällen mehr oder weniger Informationen liefern können. Daraus lernt man auch, wie dies vermieden werden kann.

Kleine *e*: In Teilen des deutschen Onlinebankings ("HBCI") kann man sich mit RSA authentifizieren. Dabei wird ein einheitliches $e = 2^{16} + 1$ benutzt. Das ist zunächst kein Problem. Bei noch kleineren Schlüsseln muss man aufpassen.

Angenommen, wir wählen $e = 3$. Falls Bob aus irgendeinem Grund die Botschaften $c_1 = f(e, m)$ und $c_2 = f(e, m+1)$ an Alice sendet, kann Eve folgendes tun: Wegen $c_1 \equiv m^3 \bmod N$ und $c_2 \equiv (m+1)^3 \bmod N$ ist

$$\frac{c_2 + 2c_1 - 1}{c_2 - c_1 + 2} = \frac{(m+1)^3 + 2m^3 - 1}{(m+1)^3 - m^3 + 2} = \frac{3m^3 + 3m^2 + 3m}{3m^2 + 3m + 3} = m.$$

Dies ist nur ein Spielzeugbeispiel; andere Abhängigkeiten von m liefern andere Angriffe.

Kleine *m*: Ein anderes Problem taucht auf, wenn $m^3 < N$. Denn dann ist $\sqrt[3]{m^3} = m$ (Wurzelziehen in $\mathbb{R}$ geht effizient!) Aus diesem Grund sind in realen Implementationen Schritte vorgeschaltet, die zu kurze m verhindern (Z.B. Anhängen der Länge von m an das m, oder Voransetzen von 111111. Dieser Trick heißt **Padding**).

Gemeinsame Primfaktoren: Falls zwei öffentliche Teil-Schlüssel N gemeinsame Primfaktoren haben (z.B. $N_1 = pq_1$, $N_2 = pq_2$), so liefert $\mathrm{ggT}(N_1, N_2) = p$ einen Faktor von beiden. Dividieren von N_i durch p liefert dann q_i, damit $\varphi(N_i) = (p-1)(q_i - 1)$ und damit die geheimen Schlüssel.

Interessanterweise wurde das probiert: In einem Experiment sammelte ein Forscherteam massenhaft öffentliche RSA-Schlüssel bzw N_is und berechnete paarweise alle ggTs. Dadurch wurden in einem Fall 0,2% der Schlüssel geknackt, in einem anderen Falle 0,4%. (siehe N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman: Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices, Proc. 21st USENIX Security Symposium, 2012)

Aber keine Sorge: die Autoren schreiben „However, there's no need to panic as this problem mainly affects various kinds of embedded devices". Betroffen waren keine persönlichen ssh- oder gpg-Schlüssel, sondern nur Massenware wie VoIP-Telefone, Router usw. Der Suchraum für 512-bit Primzahlen ist so gigantisch groß, dass es bei vernünftigen Schlüsselerzeugern wie `ssh-keygen` praktisch nicht zu Kollisionen kommen kann.

17. Mai

Kleine *d*: Manchmal hat der Besitzer von d nur begrenzte Rechenkapazität (etwa Authentifizierung durch RFID oder Chipkarte). Da könnte es verlocken, ein kleines d zu nehmen (und ein großes e in Kauf zu nehmen). Der **Wiener-Angriff** zeigt, dass das auf jeden Fall vermieden werden sollte. Wegen

$$de \equiv 1 \mod \varphi(N) \;\Leftrightarrow\; \exists k :\; de - k\varphi(N) = 1$$

gilt

$$\frac{e}{\varphi(N)} - \frac{k}{d} = \frac{1}{d\varphi(N)}.$$

Das $\varphi(N)$ ist in etwa so groß wie das N. Somit wird aus der letzten Gleichung

$$\frac{e}{N} - \frac{k}{d} \approx \frac{1}{dN}$$

Falls das d sehr klein ist verglichen mit N, dann steht da ein — bekannter — Bruch $\frac{e}{N}$ mit einem großen Nenner, und ein — unbekannter — Bruch $\frac{k}{d}$ mit sehr kleinem Nenner, der sehr nah an $\frac{e}{N}$ ist. Hätte man also ein Verfahren, dass zu einem gegebenen Bruch mit großem Nenner einen Bruch mit kleinem Nenner liefert, der sehr nah an dem gegebenen Bruch liegt... hm... sowas gibt es! Die Theorie der Kettenbrüche liefert:

Satz 5.1. *Sei $a > b$ und seien $c, d \in \mathbb{N}$ mit $\mathrm{ggT}(c, d) = 1$, so dass*

$$\left|\frac{b}{a} - \frac{c}{d}\right| \leq \frac{1}{4d^2}. \tag{2}$$

Dann taucht eins der Paare $(c, -d)$ oder $(-c, d)$ als (c_n, d_n) in einem der Schritte des erweiterten euklidischen Algorithmus (mit Eingabe $a_1 = a, a_2 = b$) auf.

Natürlich ist dann umgekehrt auch $\frac{a}{b} \approx \frac{d}{c}$. Zum Beweis siehe von zur Gathen. Obacht: Dieser konkrete Satz ist das Resultat, das man mathematisch beweisen kann. Er liefert aber folgende allgemeinere Heuristik:

Falls d klein ist, muss man nur den erweiterten euklidischen Algorithmus auf e und N anwenden und für die auftauchenden d_ns deren Beträge $|d_n|$ ausprobieren (denn es tauchen ja d oder $-d$ als d_n auf). Eines davon ist der korrekte private Schlüssel. (Das kann ja auch in Fällen klappen, in Gleichung (2) gar nicht erfüllt ist.) Mehr dazu auf Aufgabenblatt 7.

5.2 Diffie-Hellman-Schlüsseltausch

Die **diskrete Exponentialfunktion** zur Basis a (modulo N) ist

$$\mathrm{dexp}_a(x) := a^x \mod N$$

Das ist effizient berechenbar, z.B. durch wiederholtes Quadrieren:

$$2^{100} \equiv 2^{64} \cdot 2^{32} \cdot 2^4 \equiv (2^{32})^2 \cdot (((2^2)^2)^2)^4 \cdot 16 \equiv (2^{32})^2 \cdot ((-1)^2)^4 \cdot (-1) \equiv 1^2 \cdot 1 \cdot (-1) \equiv -1 \equiv 16 \mod 17$$

(hier haben wir benutzt, dass $16 \equiv -1 \bmod 17$), oder noch besser mittels des Satzes von Euler-Fermat: Es ist $\varphi(17) = 16$, also $a^{16} \equiv 1 \bmod 17$ und daher

$$2^{100} \equiv (2^{16})^6 \cdot 2^4 \equiv 1^6 \cdot 16 \equiv 16 \mod 17.$$

Der **diskrete Logarithmus** zur Basis a (modulo N) ist — analog zum Logarithmus in $\mathbb{R}$ — die Umkehrfunktion der Exponentialfunktion (dort, wo es sie gibt! Der dlog muss nicht unbedingt immer existieren, vgl. Übungsblatt 7). Es ist also

$$\mathrm{dlog}_a(a^k) \equiv k \equiv a^{\mathrm{dlog}_a(k)} \mod N.$$

Während die diskrete Exponentialfunktion effizient berechenbar ist, ist für den diskreten Logarithmus kein effizientes Verfahren bekannt. Ein wichtiger Algorithmus für die Praxis ist der Pollard-rho-Algorithmus. Im nächsten Abschnitt werden zwei einfachere vorgestellt, die besser sind als brute force. Der brute force Ansatz benötigt Laufzeit $O(N)$, wobei N das N in "mod N" ist.

Der Diffie-Hellman-Schlüsseltausch wurde etwa zeitgleich mit RSA entdeckt. Das Problem beim One-Time-Pad war ja u.a. das sichere Austauschen eines Schlüssels. Ziel hier ist genau das.

Als Einwegfunktion hier dient der diskrete Logarithmus vs diskretes Potenzieren. Wir beschreiben das Verfahren hier für allgemeine Gruppen G mit Erzeuger g. Konkret darf man sich für G vorstellen Z_N^* (N zusammengesetzt oder prim).

Diffie-Hellman-Schlüsseltausch:

- Alice und Bob einigen sich auf eine große Gruppe G mit Erzeuger g, die sie veröffentlichen. (Z.B. $G = Z_p^*$, p prim, $g < p$ eine Primitivwurzel)
- Alice wählt eine geheime Zahl a, Bob wählt eine geheime Zahl b (wobei $a, b \in \{2, \ldots, N-1\}$).
- Alice schickt g^a (z.B. $g^a \bmod p$) an Bob, Bob schickt g^b (z.B. $g^b \bmod p$) an Alice.
- Alice berechnet $k = (g^b)^a$, Bob $k = (g^a)^b$. (Z.B. $k \equiv (g^b)^a \bmod p$.)
- k ist der Schlüssel (z.B. als one-time-pad).

Der Algorithmus ist für $G = Z_p^*$

- **Korrekt:** denn $(g^b)^a = g^{ab} = (g^a)^b \equiv k \bmod p$.
- **Effizient:** da nur potenziert wird mod p.
- **Sicher:** Eve kennt $g^a \bmod p$ und $g^b \bmod p$. Sie braucht a oder b, also $\text{dlog}_g(g^a) \bmod p$ oder $\text{dlog}_g(g^b) \bmod p$. Dafür ist keine effiziente Methode bekannt.

Beispiel 5.2. (Natürlich sind echte Schlüssel viel höhere Zahlen.)

- Alice und Bob vereinbaren $g = 7$ und $G = Z_{11}^*$.
- Alice wählt $a = 3$, Bob $b = 6$.
- Alice schickt $7^3 = 343 = 2 \bmod 11$, also 2, an Bob. Bob schickt $7^6 = 343 \cdot 343 = 2 \cdot 2 = 4 \bmod 11$, also 4, an Alice.
- Alice berechnet $4^a = 4^3 = 64 = 9 \bmod 11$, Bob $2^b = 2^6 = 64 = 9 \bmod 11$.
- Also ist der Schlüssel $k = 9$.

5.2.1 Diskrete Logarithmen berechnen

Wie sicher das Diffie-Hellman-Verfahren ist, hängt ja davon ab, ob Eve den diskreten Logarithmus berechnen kann. Im Rest des Abschnitts schildern wir zwei der aktuell besten Ideen dazu. Der aktuelle Stand der Forschung ist sehr viel technischer, aber kaum besser.

Baby-Step-Giant-Step-Algorithmus

Da wir auch mit anderen Gruppen als Z_N bzw Z_N^* arbeiten wollen, formulieren wir den Algorithmus für allgemeine Gruppen G mit einem Erzeuger g. Für konkrete Zwecke darf man sich vorstellen dass $G = Z_N^*$ und g eine Primitivwurzel.

Baby-Step-Giant-Step-Algorithmus:
Gegeben $x \in G$ sowie ein Erzeuger g von G. Die Anzahl der Elemente von G sei mit $|G|$ bezeichnet. Finde $\mathrm{dlog}_g(x)$; also finde k mit $g^k = x$.

1. Setze $w = \lceil \sqrt{|G|} \rceil$.
2. Baby steps: Berechne die Tabelle $x, xg^1, xg^2, \ldots, xg^w$.
3. Giant steps: Berechne $g^w, g^{2w}, \ldots$, bis einer dieser Werte (nennen wir ihn g^{iw}) mit einem der Tabelleneinträge übereinstimmt (nennen wir den xg^j).
4. Gib aus $iw - j \bmod |G|$

Der Algorithmus ist korrekt, denn wir können das gesuchte k schreiben als $k = iw - j$ mit $i, j \in \{0, 1, \ldots, w\}$, mit $i \neq 0$. Also ist

$$x = g^k = g^{iw-j} = g^{iw} g^{-j} \;\Leftrightarrow\; xg^j = g^{iw} \tag{3}$$

Durch die Wahl von w und die Laufindizes im Algorithmus ist auch sichergestellt, dass $0 \leq iw - j$ ist. Falls $iw - j > |G|$ ist, geben wir $iw - j \bmod |G|$ zurück.

Praktisch erstellt man natürlich die Tabelle als geeignete Datenstruktur, z.B. als Liste von Paaren (x, xg^i), geordnet nach xg^i; oder noch besser, als Hash-Tabelle bzgl. der Hashes von xg^i.

Bemerkung 5.2. Der Baby-Step-Giant-Step-Algorithmus berechnet $\mathrm{dlog}_g(x)$ in Laufzeit $O(\sqrt{|G|} \cdot \log |G|)$ und benötigt Speicherplatz $O(\sqrt{|G|})$.
Man kann sich überlegen, dass dieser Ansatz sich nicht zu einem divide-and-conquer ausbauen lässt: das Problem lässt sich nicht weiter „halbieren“, denn ein Ansatz wie in (3) etwa mit $g^{iw^3 - jw^2 - kw - \ell}$ liefert ja nicht drei Gleichungen, sondern auch nur eine.

Beispiel 5.3. Wir nehmen $G = Z_{25}^*$. Ein Erzeuger ist $g = 2$ (ausprobieren), und $|G| = 20$ (da $\varphi(25) = 20$). Berechnen von $\mathrm{dlog}_2(17)$:

1. $w = \left\lceil \sqrt{20} \right\rceil = 5$.
2.

j	0	1	2	3	4	5
$xg^j \bmod 25$	17	9	18	11	22	19

3. $g^0 \equiv 1 \bmod 25, g^w \equiv 7 \bmod 25, g^{2w} \equiv 24 \bmod 25, g^{3w} \equiv 18 \bmod 25$. Bingo!
4. Gib aus $\mathrm{dlog}_2(17) \equiv 3 \cdot 5 - 2 \equiv 13 \bmod 20$.

Man beachte: die Potenzen werden in G berechnet, also mod 25. Aber die Ausgabe $iw - j$ mod $|G|$, also hier: mod 20. In der Tat ist $2^{13} \equiv 2^{10} \cdot 2^3 \equiv 1024 \cdot 8 \equiv (-1) \cdot 8 = 17 \bmod 25$.

Geburtstagsangriff

Grundlage ist das „Geburtstagsparadox“: wenn m Leute in einem Raum sind, wie wahrscheinlich ist es, dass darunter zwei sind, die am gleichen Tag (von den 365 Tagen im Jahr) Geburtstag haben? Die vielleicht überraschende Antwort ist: ab 23 Leuten ist die Wahrscheinlichkeit bereits etwas größer als 50%, ab 40 Leuten bereits 89%, ab 60 Leuten bereits 97%. Die allgemeine Formel für $m \leq 365$ Leute ist

$$1 - \frac{365 \cdot 364 \cdots (365 - m + 1)}{365^m}.$$

Ein noch allgemeineres Resultat ist dies:

Satz 5.2. *Wählen wir zufällig Zahlen (genauer: unabhängig und gleichverteilt) aus* $\{1, 2, \ldots, m\}$ *(mit zurücklegen), so ist die durchschnittliche Zahl der Wahlen bis zur ersten Wiederholung (gleiche Zahl wurde zweimal gezogen)* $O(\sqrt{m})$*; genauer:* $\frac{1}{2}\sqrt{2\pi m} + O(1)$.

Das liefert mit dem Ansatz

$$x = g^{j-i} \quad \Leftrightarrow \quad xg^i = g^j \tag{4}$$

folgenden Algorithmus. Die Idee ist, solange zufällig is und js zu wählen, bis (4) erfüllt ist. Wir beschreiben den Algorithmus wieder für allgemeine Gruppen G mit Erzeuger g.

Geburtstagsangriff-Algorithmus:

1. Starte mit zwei leeren Listen X, Y.
2. Wähle zufällig $i \in \{0, 1, \ldots, |G| - 1\}$ und $r \in \{0, 1\}$.
3. Falls $r = 0$ füge xg^i zu X hinzu, sonst füge g^i zu Y hinzu.
4. Falls eine Kollision $xg^i = g^j$ auftaucht, gib aus $j - i$.

Dann ist $\mathrm{dlog}_g(x) = j - i$ (also für $G = Z_n^*$: $\mathrm{dlog}_g(x) = j - i \bmod \varphi(N)$).

Bemerkung 5.3. Der Geburtstagsangriff-Algorithmus benötigt im average case Laufzeit $O(\sqrt{|G|} \log |G|)$.

Mit Satz 5.2 kann man zeigen, dass der Erwartungswert bis zur ersten Kollision $O(\sqrt{|G|})$ ist. Potenzieren mod N, Einfügen von (xg^i, i) in die geordnete Liste X bzw (g^i, i) in die geordnete Liste Y und Testen auf Kollision braucht jeweils $O(\log |G|)$.

Eine Verfeinerung des Geburtstagsangriffs ist der Pollard-rho-Algorithmus. Dazu sei hier nur auf von zur Gathen Kap 4.5 verwiesen. Der Pollard-rho-Algorithmus braucht ebenfalls Zeit $O(\sqrt{|G|} \log |G|)$, aber an Speicherplatz nur $O(1)$.

5.3 ElGamal

24. Mai

Taher Elgamal schlug 1985 eine Methode vor, wie man den Diffie-Hellman-Schlüsseltausch in ein Verschlüsselungsverfahren verwandelt. Dabei ist das a von Diffie-Hellman der private Schlüssel von Alice, g^a ist ihr öffentlicher Schlüssel, und das b wird für jede Nachricht neu erzeugt (Einmalschlüssel). Daher heißt der ab jetzt nicht mehr b, sondern r. Wir schildern

es wieder für eine allgemeine Gruppe G mit Erzeuger g. Wieder darf man sich z.B. $G = Z_p^*$ vorstellen für eine Primzahl p, und $g \in \{2, \ldots, p-2\}$ geeignet. Dann steht g^a für $g^a \bmod p$.

ElGamal-Verschlüsselung:

- Vorab: Alice wählt zufällig $a \in \{2, 3, \ldots, |G|-1\}$ (geheimer Schlüssel) und veröffentlicht G, g und g^a.
- Verschlüsseln: Bob wählt zufällig $r \in \{2, 3, \ldots, |G|-1\}$ und berechnet den Einmalschlüssel $k = (g^a)^r$. Er verschlüsselt m als $c = mk$ und sendet (c, g^r) an Alice.
- Entschlüsseln: Alice berechnet $k = (g^r)^a$, dann k^{-1} in G, und damit $m = ck^{-1}$.

Im Allgemeinen hängt die Sicherheit des Verfahrens von der Gruppe G ab. Für $G = Z_p^*$ ist der Algorithmus

- **Korrekt:** wie bei Diffie-Hellman, da $(g^r)^a \equiv (g^a)^r \bmod p$ ist und $ck^{-1} = mkk^{-1} = m$.
- **Effizient:** wie bei Diffie-Hellman: es muss nur potenziert werden mod p. Auch das Berechnen von k^{-1} geht effizient mit dem erweiterten euklidischen Algorithmus. Aber das geht hier noch einfacher: Es ist $k \equiv g^{ar} \bmod p$, und es gilt $g^{ar} g^{p-1-ar} = g^{ar+p-1-ar} \equiv g^{p-1} \equiv 1 \bmod p$. Also ist $k^{-1} \equiv g^{-ar} \equiv g^{p-1-ar} \bmod p$. Also kann Alice auch einfach Folgendes berechnen:

$$c(g^r)^{p-1-a} \equiv c(g^r)^{p-1} g^{-ar} \equiv cg^{-ar} \equiv ck^{-1} \equiv m \bmod p.$$

- **Sicher:** Wie bei Diffie-Hellman. Eve kennt g, $g^a \bmod p$ und $g^r \bmod p$. Sie braucht a oder r, also $\mathrm{dlog}_g(g^a) \bmod p$ oder $\mathrm{dlog}_g(g^r) \bmod p$. Dafür ist keine effiziente Methode bekannt.

Bemerkung 5.4. Das oben ist sozusagen das klassische ElGamal-Verfahren. Die Verschlüsselungsfunktion ist hier $f(k, m) = mk \bmod p$, die Entschlüsselungsfunktion ist $f^*(k^{-1}, c) = ck^{-1} \bmod p$. Stattdessen gibt es **ElGamal-Varianten**, die andere f verwenden, z.B. symmetrische Verfahren wie AES (s. Kap. 8)

Bemerkung 5.5. Anders als bei RSA können hier alle Nutzer dasselbe G — bzw dasselbe p — benutzen! Wegen der zufälligen Wahl von r wird derselbe Klartext (fast) immer zu zwei verschiedenen Geheimtexten verschlüsselt.

5.4 Shamirs Three-Pass-Protokoll

Um auch mal ein in der Praxis kaum benutztes Verfahren zu sehen, nehmen wir dieses, weil es nun schnell zu erklären ist. In der Praxis wird es nicht eingestetz, da es zwei offensichtliche Nachteile hat: jede Nachricht muss dreimal geschickt werden, und: es gibt gar keinen öffentlichen Schlüssel, daher bietet es keine einfache Erweiterung zu einem Authentifizierungsverfahren.

Etwa 1980 wurde das erste Three-Pass-Protokoll von Adi Shamir vorgestellt. Es hieß auch No-Key-Protokoll, da keine Schlüssel vereinbart werden müssen. Dennoch gibt es Schlüssel, und man muss sich auf eine gemeinsame Gruppe einigen.

Die Grundidee ist diese: wir stellen uns vor, die Nachricht wird in einer Kiste transportiert. Bob schließt die Kiste mit einem Vorhängeschloss ab, zu dem nur er den Schlüssel hat, und schickt sie an Alice. Alice schließt die Kiste mit einem zweiten Vorhängeschloss ab, zu dem nur sie den Schlüssel hat, und schickt sie an Bob zurück. Bob entfernt sein Schloss und schickt die Kiste wieder an Alice. Alice kann nun die Kiste öffnen und die Nachricht entnehmen. Unterwegs ist die Kiste immer verschlossen. Die Umsetzung mit kryptographischen Methoden geht folgendermaßen. Wir beschreiben das Verfahren wieder für allgemeine Gruppen.

Shamirs Three-Pass-Protokoll
Vorab: Alice und Bob einigen sich auf eine große Gruppe G mit $n = |G|$. Alice erzeugt $a, a' \in Z_n^*$ mit $aa' \equiv 1 \bmod n$, Bob erzeugt $b, b' \in Z_n^*$ mit $bb' \equiv 1 \bmod n$.
Ver- und Entschlüsseln: Bob verschlüsselt die Nachricht $m \in G$ als $c_1 = m^b$ und schickt das an Alice. Alice berechnet $c_2 = c_1^a$ und schickt das an Bob. Bob berechnet $c_3 = c_2^{b'}$ und schickt das an Alice. Alice berechnet $m = c_3^{a'}$.

Das m^b kann also heißen: $m^b \bmod p$ (in Z_p^*, dann wäre $n = p-1$), oder aber $m^b = m \odot \cdots \odot m$ in einer anderen geeigneten Gruppe. Auf jeden Fall ist wegen Euler-Fermat (Satz 2.4) bzw Lagrange (Satz 2.2) ganz ähnlich wie so oft schon

$$\begin{aligned} c_3^{a'} &= \Big(\big((m^b)^a\big)^{b'}\Big)^{a'} = m^{bab'a'} = (m^{bb'})^{aa'} = (m^{k|G|+1})^{aa'} = \big((m^k)^{|G|} \cdot m^1\big)^{aa'} \\ &= 1 \cdot m^{aa'} = (m^{\ell|G|+1}) = (m^{|G|})^\ell m^1 = m. \end{aligned}$$

Also ist das Verfahren korrekt. Die Sicherheit beruht wieder auf dem diskreten Logarithmus (in der jeweiligen Gruppe). Effizienz folgt wie bei Diffie-Hellman, ElGamal und RSA aus effizientem Potenzieren.

6 Elliptische Kurven

Oben haben wir Diffie-Hellman, ElGamal und Shamirs Three-Pass-Verfahren für beliebige Gruppen formuliert. In der Praxis werden zwei (bis drei) Gruppen benutzt:

- Z_n^* (für $n = p$ prim, bzw. für $n = pq$) mit Multiplikation mod n.
- Elliptische Kurven (dazu jetzt mehr)
- $\mathbb{F}_2[x]$ (siehe Kap. 8)

Z_p^* haben wir schon kennengelernt. Bevor wir elliptische Kurven einführen: Warum benutzen wir nicht die Restklassengruppe $(Z_n, +)$? Also die Zahlen $\{0, 1, 2, \ldots, n-1\}$ mit Addition mod$n$? Dazu überlegt man sich, was denn hier der diskrete Logarithmus ist. Hier ist

$$g^a = \underbrace{g \oplus g \oplus \cdots \oplus g}_{a \text{ mal}} \equiv \underbrace{g + g + \cdots + g}_{a \text{ mal}} \equiv a \cdot g \mod n$$

In diesem Fall ist $\oplus$ einfach Addieren modulo n. Also ist hier „g^a“ gleich $a \cdot g \bmod n$. Das heißt hier ist $\mathrm{dlog}_g(x)$ die Zahl a mit $a \cdot g \equiv x \bmod n$. Also ist $a \equiv x \cdot g^{-1} \bmod n$, wobei g^{-1} das multiplikative Inverse von g modulo n ist. Das g^{-1} ist effizient berechenbar (erweiterter euklidischer Algorithmus). Also ist hier der dlog effizient berechenbar und somit ist $(Z_n, +)$ für kryptographische Zwecke ungeeignet.

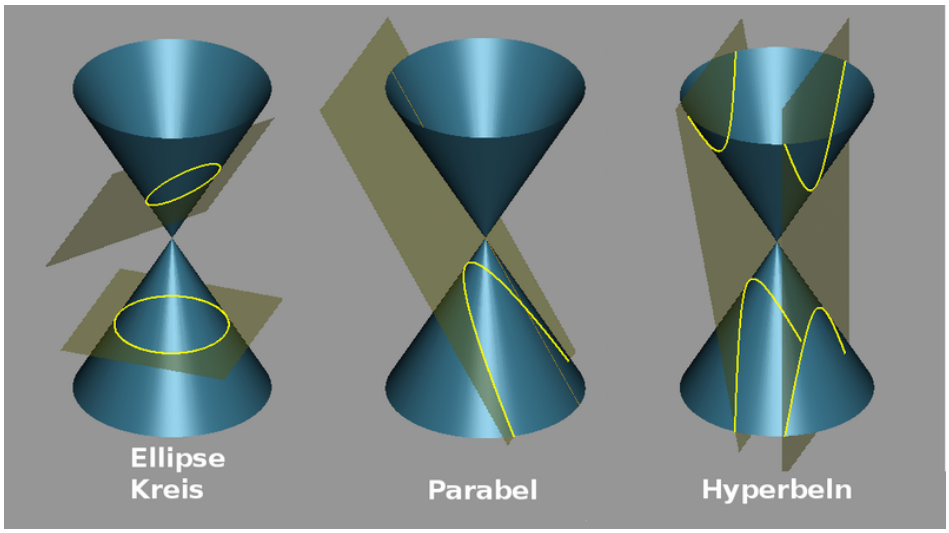


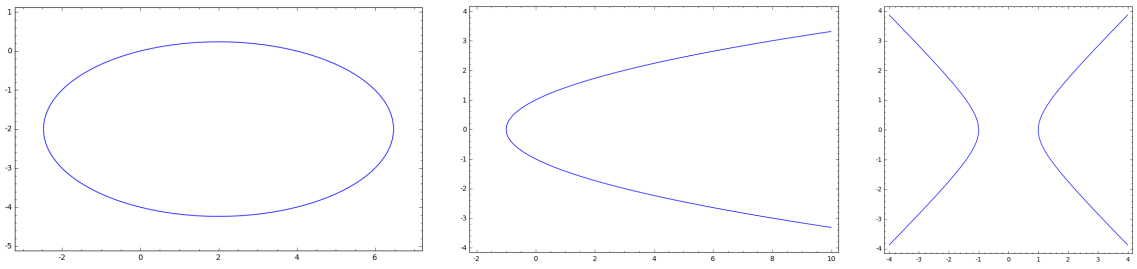
Abbildung 1: Ebene Quadriken als Schnitte einer Ebene mit einem (idealen, zweiseitigen, unendlichen) Kegel. (Bild: wikipedia)

6.1 Quadriken

Ziel ist es, elliptische Kurven über endlichen Körpern (z.B. $\mathbb{F}_7$) anzusehen. Zum Vergleich sehen wir sie uns zunächst in $\mathbb{R}^2$ an. Und zum Aufwärmen sehen wir uns zunächst **Quadriken** in $\mathbb{R}^2$ an (aka *Kegelschnitte*). Eine Quadrik in $\mathbb{R}^2$ ist die Menge aller Punkte (x, y), die eine quadratische Gleichung in den zwei Variablen x und y zu Null machen (quadratisch heißt hier: Linearkombination von $x^2, y^2, xy, x, y, 1$). Also sind z.B.

$$\{4x^2 - 4x + y^2 + 4y = 0\}, \quad \{(x, y) \mid x - y^2 + 1 = 0\}, \quad \{(x, y) \mid x^2 - y^2 - 1 = 0\}$$

Quadriken. Die drei Beispiele hier sehen so aus:



Jeweils im Bild links: eine Ellipse, Mitte: eine Parabel, rechts: eine Hyperbel. Parabeln und Ellipsen sollten bekannt sein. Hyperbeln bestehen aus zwei Kurven. Der Graph von $f : \mathbb{R} \to \mathbb{R}, f(x) = \frac{1}{x}$ ist z.B. eine Hyperbel. Eine Hyperbel hat zwei Geraden als Asymptoten. (im Fall von $f(x) = \frac{1}{x}$ sind das die x-Achse und die y-Achse.) Warum die Quadriken in $\mathbb{R}^2$ auch *Kegelschnitte* heißen, wird in Abbildung 1 deutlich.

Im Allgemeinen sind Quadriken immer eine dieser drei Sorten von Kurven (es gibt Grenzfälle: die Menge $\{(x, y) \mid x^2 - y^2 = 0\}$ besteht z.B. aus zwei sich schneidenden Geraden. Die Menge $\{(x, y) \mid x^2 + y^2 = 0\}$ besteht nur aus einem einzelnen Punkt.) Eine Quadrik Q

hat die Eigenschaft, dass jede Gerade in $\mathbb{R}^2$ mit Q genau null oder zwei Schnittpunkte hat (Wenn man richtig zählt: wenn die Gerade eine Tangente ist, dann hat zählen wir sie als "doppelten" Schnittpunkt. Für ein bis zwei weitere Fälle muss man noch einen Punkt in unendlich hinzunehmen, vgl. unten den Punkt $\mathcal{O}$. Quizfrage: welche Fälle sind das?)

6.2 Elliptische Kurven über $\mathbb{R}$

31. Mai

Eine *elliptische Kurve* in $\mathbb{R}^2$ ist eine Entsprechung einer Quadrik mit einer kubischen Gleichung (Polynom vom Grad 3; also Linearkombination von $x^3, x^2y, xy^2, y^3, x^2, xy, y^2, x, y, 1$). Es gibt viele äquivalente Definition. Hier ist eine:

Definition 6.1. Sei $\mathbb{K}$ ein Körper (hier immer: $\mathbb{K} = \mathbb{R}$, oder $\mathbb{K} = \mathbb{F}_p$ für eine Primzahl $p > 3$). Seien $a, b \in \mathbb{K}$ mit $4a^3 + 27b^2 \neq 0$. Dann ist

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

eine **elliptische Kurve** über $\mathbb{K}$.

Bemerkung 6.1. Der Punkt $\mathcal{O}$ ist ein „Punkt in unendlich". Den brauchen wir später als neutrales Element, um E zu einer Gruppe zu machen.
Werte für a und b mit $4a^3 + 27b^2 = 0$ führen zu ungeeigneten Kurven, vgl. Aufgabe 34. Diese Spezialfälle — analog zum Fall zweier Geraden als Kegelschnitt — liefern keine Gruppen. Wir müssen die daher für spätere Zwecke ausschließen.
Für $p = 2$ oder $p = 3$ gibt es analoge Definitionen: die Gleichung muss dann lauten $y^2 + xy = x^3 + ax^2 + b$, und die Bedingungen an a und b sind anders.

Beispiel 6.1. Wir geben zunächst Beispiele für elliptische Kurven in $\mathbb{R}^2$. Abb. 2 zeigt die elliptische Kurve $\{(x, y) \mid y^2 = x^3 - x\}$ über $\mathbb{R}$ (links) sowie ein paar weitere (rechts).

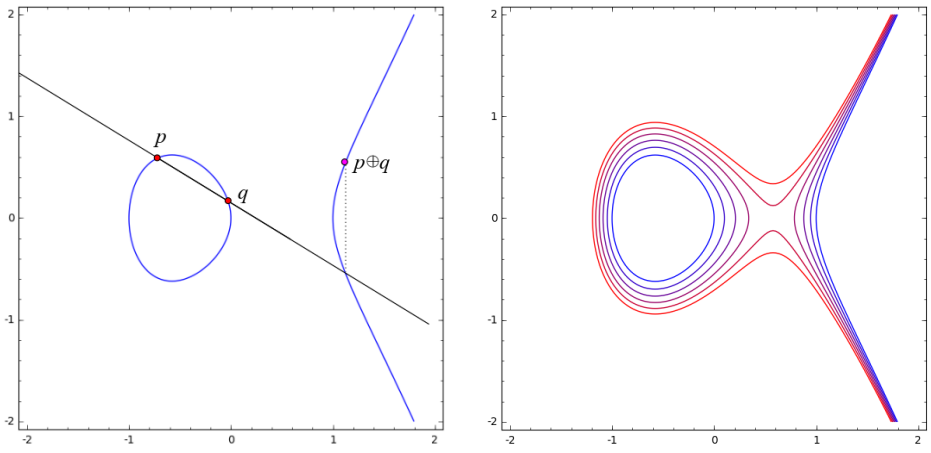
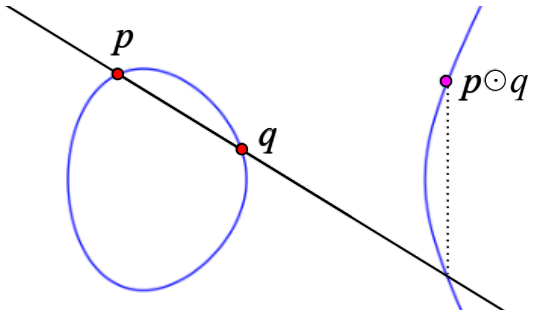


Abbildung 2: Die elliptische Kurve $y^2 = x^3 - x$ über $\mathbb{R}$ (links) sowie die Kurven $y^2 = x^3 - x + \frac{i}{10}$ für $i = 0, 1, \ldots, 5$.

Elliptische Kurven E haben die interessante Eigenschaft, dass jede Gerade in $\mathbb{R}^2$, die keine Tangente an E ist, genau ein oder drei Schnittpunkte mit E hat. Damit das aufgeht, muss

man sich einen weiteren künstlichen Punkt $\mathcal{O}$ auf der Kurve als unendlich entfernten Punkt in y-Richtung vorstellen; und zwar unendlich weit oben *und* unendlich weit unten. (Das klingt seltsam, passt aber: es gibt die *projektive Geometrie*, siehe wikipedia; auch da werden z.B. alle senkrechten Geraden als ein weiterer Punkt zu den Punkten des $\mathbb{R}^2$ hinzugenommen). Jede senkrechte Gerade trifft dann $\mathcal{O}$ (und keine andere Gerade tut das). Jede Tangente an E schneidet dann E in genau einem weiteren Punkt. (Wieder passt hier die Sichtweise, den Schnittpunkt von E mit einer Tangente an E als doppelten Schnittpunkt zu zählen.)

Dank dieser Eigenschaft können wir eine Gruppenoperation $\odot$ auf E definieren: für $p \neq q \in E$ sei ℓ die Gerade durch p und q, und (x, y) sei der (eindeutige!) dritte Schnittpunkt von ℓ mit E. Dann setze $p \odot q := (x, -y)$. Hier ein Beispiel.



Was ist $p \odot p$? Dazu betrachten wir die Tangente in p an E. Diese schneidet E in genau einem weiteren Punkt (x, y), und wir setzen $p \odot p := (x, -y)$.

Was ist $p \odot \mathcal{O}$? Sei $p = (x_p, y_p)$. Dann schneidet die senkrechte Gerade durch p die Kurve E in dem Punkt $(x_p, -y_p)$, und wir setzen $p \odot \mathcal{O} = (x_p, -(-y_p)) = (x_p, y_p) = p$. Dito für $\mathcal{O} \odot p$. Also ist $\mathcal{O}$ das neutrale Element in der Gruppe. Was ist das inverse Element zu $p = (x_p, y_p)$? Die Gerade durch $p = (x_p, y_p)$ und $p' = (x_p, -y_p)$ ist senkrecht. Also ist $p \odot p' = \mathcal{O}$, und p' ist das inverse Element zu p. Wir schreiben im Folgenden statt p' immer p^{-1}.

Bemerkung 6.2. Mit der oben erklärten Verknüpfung bildet eine elliptische Kurve über $\mathbb{K}$ eine abelsche Gruppe.

6.3 Elliptische Kurven über $\mathbb{F}_p$

Das Schöne ist, dass all das oben auch für elliptische Kurven über endlichen Körpern gilt. Und dass der diskrete Logarithmus in diesen Gruppen schwierig zu berechnen ist, während alle anderen benötigten Operationen effizient berechenbar sind.

Invertieren in einer elliptischen Kurve E ist natürlich sehr einfach, s.o. Multiplizieren ist etwas tricksig. Dazu überlegt man sich geometrisch: Gegeben zwei Punkte $p \neq q$ mit $p = (x_p, y_p)$ und $q = (x_q, y_q)$. Wir suchen $r = (x_r, y_r)$ mit $p \odot q = r$. Falls $x_p = x_q$, so ist $y_p = -y_q$ und $p \odot q = \mathcal{O}$. Falls $x_p \neq x_q$ gilt: Die Gerade ℓ durch p und q hat die Gleichung

$$\ell = \{(x, \lambda x + y_p - \lambda x_p) \mid x \in \mathbb{R}\} \text{ mit } \lambda = \frac{y_q - y_p}{x_q - x_p}$$

(Klar: λ ist die Steigung, $y_p - \lambda x_p$ der Schnitt mit der y-Achse.) Für die drei Punkte (x, y)

in $E \cap \ell$ gilt also

$$\begin{aligned}&(\lambda x + y_p - \lambda x_p)^2 = x^3 + ax + b\\ &\text{also umgeformt: } -x^3 + \lambda^2 x^2 + (2\lambda(y_p - \lambda x_p) - a)x - b + (y_p - \lambda x_p)^2 = 0\end{aligned}$$

Der Vorfaktor von x^2 in diesem Polynom ist die Summe der Nullstellen (vgl Mathe 2: Linearfaktorzerlegung). Zwei Nullstellen sind x_p und x_q. Die dritte Nullstelle ist das gesuchte x_r. Also:

$$x_r = \lambda^2 - x_p - x_q \tag{5}$$

Einsetzen in die Geradengleichung liefert $-y_r$ (!). Also ist

$$y_r = -(\lambda x_r + y_p - \lambda x_p) = \lambda(x_p - x_r) - y_p \tag{6}$$

Diese ganze Überlegung klappt auch in $\mathbb{F}_p$. Hier heißt dann der Ausdruck $\frac{y_q - y_p}{x_q - x_p}$ natürlich $(y_q - y_p) \cdot (x_q - x_p)^{-1}$.

Mit Hilfe einer Formelsammlung kann man eruieren, dass für $p \odot p$ gilt: die Tangente an $p = (x_p, y_p)$ hat die Steigung

$$\frac{\frac{dE}{dx_p}}{\frac{dE}{dy_p}} = \frac{3x_p^2 + a}{2y_p}.$$

Die Gerade ist hier also für $y_p \neq 0$:

$$\ell = \{(x, \lambda x + y_p - \lambda x_p) \mid x \in \mathbb{R}\} \text{ mit } \lambda = \frac{3x_p^2 + a}{2y_p}.$$

Wieder heißt "geteilt durch $2y_p$" in $\mathbb{F}_p$ "mal $(2y_p)^{-1}$". In E einsetzen liefert für $p \odot p = r = (x_r, y_r)$:

$$\begin{aligned}(\lambda x_r + y_p - \lambda x_p)^2 &= x_r^3 + ax_r + b, \text{ somit}\\ &\quad -x^3 + \lambda^2 x_r^2 + 2\lambda(y_p - \lambda x_p)x_r - ax_r + (y_p - \lambda x_p)^2 - b = 0\end{aligned}$$

Wieder ist der Vorfaktor von x_r^2 die Summe der drei Nullstellen. Zwei der Nullstellen sind x_p und x_p, also ist hier

$$x_r = \lambda^2 - 2x_p \quad \text{und} \quad y = \lambda(x_p - x_r) - y_p \tag{7}$$

Für $y_p = 0$ kann man sich in $\mathbb{R}$ überlegen, dass die Steigung der Tangente ∞ ist. Also ist für $y_p = 0$ immer $p \odot p = \mathcal{O}$. Zusammengefasst:

Gruppenoperation in elliptischen Kurven. Seien $p = (x_p, y_p)$ und $q = (x_q, y_q)$ Punkte in einer elliptischen Kurve. Dann ist $p \odot q$ gleich

- $\mathcal{O}$, falls $(x_p, y_p) = (x_q, -y_q)$
- $(x_r, \lambda(x_p - x_r) - y_p)$ mit $x_r = \lambda^2 - 2x_p$, $\lambda = (3x_p^2 + a)(2y_p)^{-1}$, falls $p = q$,
- $(x_r, \lambda(x_p - x_r) - y_p)$ mit $x_r = \lambda^2 - x_p - x_q$, $\lambda = (y_q - y_p) \cdot (x_q - x_p)^{-1}$ sonst.

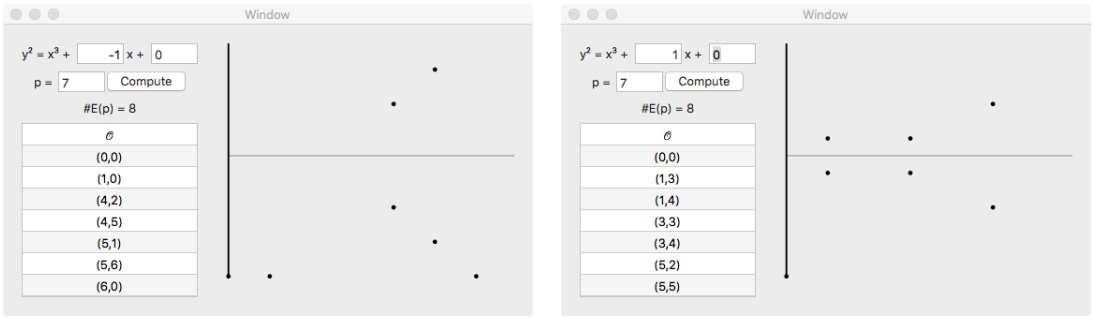


Abbildung 3: Die elliptischen Kurven E (zu $x^3 - x$, links) und E^* (zu $(x^3 + x$, rechts). Die Koordinaten muss man sich dazudenken. Links unten ist der Punkt $(0, 0)$, rechts oben der Punkt $(7, 7)$.

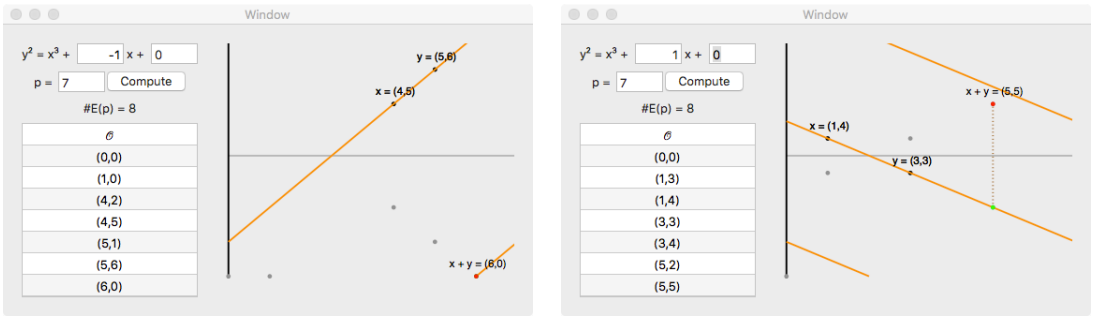


Abbildung 4: Verknüpfung in E (links) bzw E^* (rechts).

Beispiel 6.2. Wir betrachten zwei Beispiele von elliptischen Kurven über $\mathbb{F}_7$. Mit $a = -1$ und $b = 0$ gilt $4a^3 + 27b^2 \equiv -4 \equiv 3 \neq 0 \bmod 7$, also definiert $y^2 = x^3 - x$ eine elliptische Kurve E über $\mathbb{F}_7$. Sie hat die folgenden acht Punkte:

$$(0, 0), (1, 0), (4, 2), (4, 5), (5, 1), (5, 6), (6, 0) \text{ und } \mathcal{O},$$

siehe Abb. 3 (links). Ebenso liefert die Gleichung $y^2 = x^3 + x$ eine elliptische Kurve E^* mit den acht Punkten

$$(0, 0), (1, 3), (1, 4), (3, 3), (3, 4), (5, 2), (5, 5), \mathcal{O},$$

siehe Abb. 3 (rechts). Die Verknüpfung zweier Punkte kann man auch hier geometrisch veranschaulichen: $p \odot q$ ist der dritte Schnittpunkt der Geraden durch p und q mit E^*. Hier muss die „Gerade“ allerdings modulo 7 betrachtet werden. D.h., wenn die Gerade den Bereich $[0, 7[\times[0, 7[$ rechts verlässt, kommt sie links wieder rein. Vgl. Abb. 4.

Wie gehabt bezeichnen wir hier mit E sowohl die elliptische Kurve als Menge als auch die Gruppe (eigtl also E mit $\odot$). Es ist interessant, zu untersuchen, was die Struktur dieser Gruppen ist: Ist E etwa eine zyklische Gruppe? Oder aus zyklischen Gruppen zusammengesetzt? 7. Juni

Die Gruppe E^* im Beispiel oben hat z.B. den Erzeuger $g = (3, 3)$ (vgl Abb. 5 rechts). Also lassen sich alle Elemente aus E^* darstellen als g^i für ein $i \in \{1, 2, \ldots, 8\}$. Durch Ausprobieren

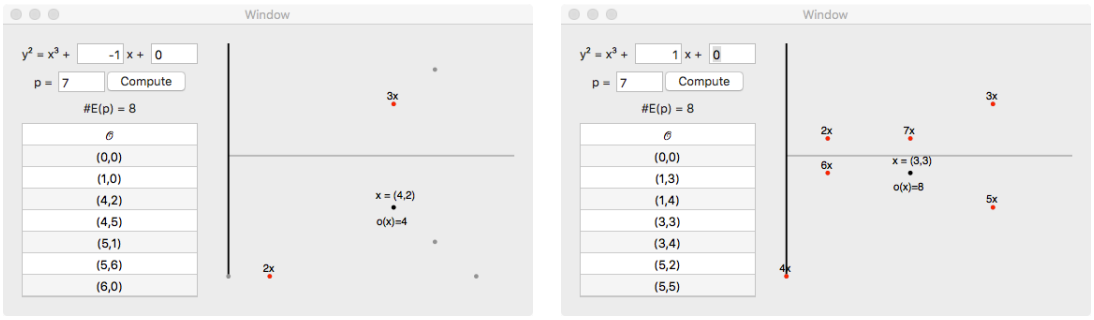


Abbildung 5: Die Gruppe E hat keinen Erzeuger: es finden sich nur Elemente der Ordnung 2 oder 4, nicht der Ordnung 8 (links). Die Gruppe E^* hat einen Erzeuger: z.B. hat $(3,3)$ die Ordnung 8 (rechts).

kann man testen, dass die Gruppe E keinen (einzelnen) Erzeuger hat. Allerdings ist E das direkte Produkt zweier zyklischer Gruppen.

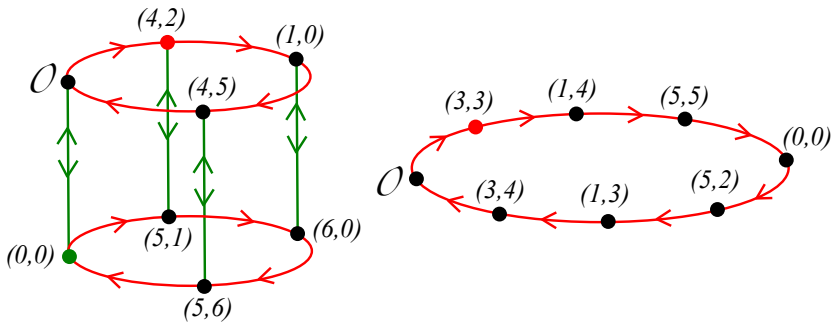
Definition 6.2. Das **direkte Produkt** $G \times H$ zweier Gruppen $(G, \otimes)$ und $(H, \odot)$ ist

$$(\{(g,h) \mid g \in G, h \in H\}, \circ) \quad \text{wobei } (g,h) \circ (g',h') = (g \otimes g', h \odot h') \tag{8}$$

Obacht: das $\times$ hat im Allg. zwei Bedeutungen: hier bezeichnet es das direkte Produkt zweier Gruppen, und im Allg. ist es auch das kartesische Produkt (so wie in $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ einmal die Gruppe $(\mathbb{R}^2, +)$ bezeichnet, und einmal die Menge $\mathbb{R}^2$).

Für das E von oben gilt: $E = Z_2 \times Z_4$. Damit hat E zwei Erzeuger, die *zusammen* E erzeugen; z.B. $g = (4,2)$ und $h = (0,0)$. D.h. also, alle Elemente von E lassen sich darstellen als $g^i \odot h^j$.

Mit den Erzeugern lassen sich die Gruppen auch durch einen sogenannten **Cayleygraphen** darstellen (links E, rechts E^*):



Ein roter Pfeil heißt dabei: einmal mit g multiplizieren, ein grüner Pfeil heißt: einmal mit h multiplizieren. Also ist das Element g^4 in E^* gleich $(0,0)$, und das Element $g^3 h$ in E ist gleich $(5,6)$ (jeweils bei $\mathcal{O}$ loslaufen).

Satz 6.1. *Eine elliptische Kurve über $\mathbb{F}_p$ ist immer entweder eine zyklische Gruppe Z_n, oder aber das direkte Produkt $Z_k \times Z_\ell$ von zwei zyklischen Gruppen, wobei ℓ ein Teiler von $\mathrm{ggT}(k, p-1)$ ist.*

Kodieren einer Nachricht in elliptischen Kurven Das Kodieren einer Nachricht in elliptischen Kurven über $\mathbb{F}_p$ ist nicht ganz so einfach wie in RSA, also in Z_n. denn: nicht jedes $m \in \{2, 3, \ldots, p-1\}$ taucht als Punkt auf der Kurve auf (weder als x- noch als y-Koordinate). Die Lösung:

Möglichkeit 1: gehashtes ElGamal. Im Verschlüsselungsschritt von ElGamal verschlüsselt man nicht $c = mk$ mit $k = g^{ar}$, sondern nutzt eine geeignete (Hash-)Funktion $h : E \to \{0,1\}^n$ und berechnet $c = m + h(k)$. Die Addition erfolgt hier in $\{0,1\}^n = (\mathbb{F}_2)^n = \mathbb{F}_2 \times \mathbb{F}_2 \times \cdots \times \mathbb{F}_2$; d.h. bitweise Addition modulo 2, also XOR (also genau wie beim One-Time-Pad, siehe S. 6).

Dabei ist jetxt $m \in (\mathbb{F}_2)^n$, also einfach ein n-Bit Wort, und m muss nicht mühsam in ein Element der elliptischen Kurve übersetzt werden. Zur Entschlüsselung brauchen wir das inverse Element $h(k)$ bzgl der Addition in $(\mathbb{F}_2)^n$. Das ist ja einfach $h(k)$ selbst (vgl One-Time-Pad: $1 + 1 = 0$)

Warum man hier nicht einfach $m + k$ nimmt, wird auf Aufgabenblatt 10 deutlich! (Das geht schon, hat aber eine entscheidende Schwäche, die man mit dem Hashen vermeidet. Mehr zu Hashfunktionen im nächsten Kapitel.)

Es gibt eine andere Methode: Koblitzkodierung. In ihrer einfachsten Form heißt das: kodiere m als (m, y_m), falls $(m, y_m) \in E$, sonst als $(m + 1, y_m)$, falls $(m + 1, y_m) \in E$, sonst als $(m + 2, y_m)$... usw bis es klappt. Genauer:

Möglichkeit 2: Koblitz-Kodierung (in einfach). Wir nutzen eine sehr hohe Primzahl p für unsere elliptische Kurve E über $\mathbb{F}_p$ mit der Gleichung $y^2 = x^3 + ax + b$ und wählen ein geeignetes d; in der Realität etwa $d = 2^8$ oder $d = 2^{10}$. Wir zerschneiden unsere Nachricht m in kleine Stücke $m_0, m_1, \ldots$, so dass $0 \leq m_i < p/d$. Dann:

1. $j = 0$
2. Solange $j < d$ ist:
   - Berechne $x = dm + j$.
   - Falls $x^3 + ax + b$ eine Quadratwurzel y in $\mathbb{F}_p$ hat: berechne y. STOP, Ausgabe (x, y).
   - Sonst $j = j + 1$, weitermachen.

Das Dekodieren einer solchen Koblitz-kodierten Nachricht (x_j, y) ist denkbar einfach: Da $dm \leq x < d(m + 1)$ ist, ist $m = \lfloor \frac{x}{d} \rfloor$.

Quadratwurzeln in $\mathbb{F}_p$ berechnen geht effizient, falls p eine Primzahl ist, siehe Kapitel 2.2. Dennoch ist dies ein probabilistischer Algorithmus, im Prinzip kann es passieren, dass keine der getesteten Zahlen x_j eine Quadratwurzel in $\mathbb{F}_p$ hat. Es gibt klügere Varianten des obigen, die garantiert klappen. Bei denen braucht man aber zum effizienten Berechnen wiederum andere probabilistische Algorithmen.

Bemerkung 6.3. Für Diffie-Hellman oder ElGamal brauchen wir einen Erzeuger g. Nicht jede elliptische Kurve hat einen einzelnen Erzeuger, s.o. Falls aber $|G|$ eine Primzahl ist, ist jedes Element außer das neutrale ein Erzeuger (Lagrange!)

Um eine Gruppe G zu finden, so dass $|G|$ eine große Primzahl ist, wählt man eine Primzahl p mit der gewünschten Bit-Anzahl (z.B. 128 oder 256). Dann berechnet man für zufällige $a, b \in Z_p$ die Größe d der elliptischen Kurve über Z_p mit $y^2 = x^3 + ax + b$. (Das ist schwierig,

siehe Kap 5.7 in von zur Gathen). Falls einer der Werte d, $d/2$, $d/3$ oder $d/4$ eine Primzahl q ist, nimmt man dieses E und betrachtet die entsprechende Untergruppe. Den Erzeuger g findet man dann, indem man x zufällig wählt und die Quadratwurzel von $x^3 + ax + b$ in Z_q zieht (das geht effizient mit Satz 2.8 bzw mit Cipollas Algorithmus, s. Kap 2.2). Falls es aufgeht, hat man seinen Erzeuger g gefunden. Damit hat man alle Zutaten: (p, a, b, q, g).

Die amerikanische Behörde NIST (die sowas wie die amerikanische Entsprechung von DIN-Normen bestimmt) und die europäische Arbeitsgruppe ECC-Brainpool (Firmen, Unis, Bundesamt für Sicherheit in der Informationstechnik) stellen geeignete (p, a, b, q, g) zur Verfügung, vgl. `https://en.wikipedia.org/wiki/Elliptic-curve_cryptography#Implementation`

Bemerkung 6.4. Zum Berechnen von dlog_g in einer elliptischen Kurve E sind nur ineffiziente Methoden bekannt, siehe Abschnitt 5.2.1. Ein Weltrekord (Stand 2013) wurde aufgestellt 2009 von Bos, Kaihara, Kleinjung, Lenstra und Montgomery: sie benutzten 200 Play-Station 3 für ihr Programm und ließen sie 6 Monate rechnen. Das waren 2^{63} Rechenoperationen. Sie zogen den diskreten Logarithmus aus einer zufälligen Zahl (mehr oder weniger: die x-Koordinate sind die ersten 34 Nachkommastellen von π) in einer elliptischen Kurve E über Z_p, wobei p und $|E|$ 112 Bit haben (also $p \approx 2^{112}$). Das wurde mittlerweile verbessert auf 113 bit bzw 117 bit, siehe `https://en.wikipedia.org/wiki/Discrete_logarithm_records`.

Problem	Weltrekord (Stand Mai 2020)
Faktorisieren von $n = pq$	829 Bit = 250 Dezimalstellen
dlog in Z_N^*	795 Bit = 240 Dezimalstellen
dlog in ellipt. Kurven	117 Bit = 35 Dezimalstellen

Tabelle 1: Die Weltrekorde für schwere Probleme in verschiedenen endlichen Gruppen.

Die Zahlen für diskrete Logarithmen in elliptischen Kurven sind also deutlich kleiner als im Faktorisierungsproblem und für diskrete Logarithmen in Z_N^*. (Vgl. auch die RSA-Challenge in Kapitel 5.1.) Elliptische Kurven erlauben also kleinere Schlüssellängen: anders als bei RSA kann man hier 196-bit-Schlüssel als sicher betrachten.

Falls man sich fragt, was der Weltrekord ist für "Quadratwurzel mod N finden" (vgl. Abschnitt 2.2, das ist ja auch ein schwieriges Problem): laut meinen Recherchen gibt es da nichts. Vermutlich liegt das daran, dass die beste Methode zum Finden einer Quadratwurzel mod $N = pq$ die ist, erst N zu faktorisieren, dann wie in Abschnitt 2.2 geschildert zuerst effizient die Quadratwurzeln mod p und mod q zu berechnen und daraus effizient die Quadratwurzeln mod N. Also ist der Aufwand im Wesentlichen derselbe.

Elliptische Kurven eignen sich für die verschiedensten Protokolle: neben ElGamal-Verschlüsselung und Diffie-Hellman-Schlüsseltausch (s.o.) auch ElGamal-Signatur, verschiedene Authentifizierungsmethoden (Schnoor, Okamoto). Eine weitere stellen wir in Kapitel 9 vor.

7 Hashfunktionen

14. Juni

Eine weitere wichtige Zutat für komplexere kryptographische Protokolle sind Hashfunktionen. Hashfunktionen kommen in vielen Bereichen und Anwendungen vor. Zwei einfache Beispiele:

Prüfsummen: Eine große gepackte Datei `file.tar.gz` wird heruntergeladen und entpackt. Dabei können eventuell einzelne bits falsch gelesen oder geschrieben werden. Um das zu überprüfen wäre eine Möglichkeit, die heruntergeladene und entpackte Datei mit der Originaldatei Zeichen für Zeichen (byte für byte) zu vergleichen. Dazu müssen viele Daten übertragen werden, so viele, dass das Packen der Datei seinen Sinn verliert. Eleganter ist es, von jeder der beiden Dateien eine Art Fingerabdruck zu nehmen und zu vergleichen, ob diese identisch sind. Wenn ja, dann ist es sehr wahrscheinlich (oder fast sicher?), dass auch die Dateien identisch sind.

Passwörter und Salting: Melden sich in einem System viele Nutzer per Nutzernamen und Passwort an, so muss ja (naiv gedacht) das System alle Paare Nutzername-Passwort kennen. Das würde dem Missbrauch Tür und Tor öffnen. Erste Lösungsidee: gespeichert wird nur der Nutzername und der *Hashwert* (also der Fingerabdruck) des Passworts. Die Idee dabei ist, dass ich aus dem "Fingerabdruck" allein nichts über das Passwort ermitteln kann. Gibt ein Nutzer sein Passwort ein, wird der Hashwert dessen mit dem gespeicherten Hashwert verglichen. Stimmen die überein, wird die Passworteingabe als korrekt akzeptiert.

Eine mittlerweile übliche Verbesserung ist das **Salting**: Es wird für jedes Passwort p ein Zufallsstring s gewählt. Gespeichert wird der Hashwert des Paares (p, s) zusammen mit s selbst. Die Verifizierung erfolgt analog zu oben. Damit wird nun vermieden, dass es auffällt, falls mehrere Nutzer dasselbe Passwort verwenden.

Eine weitere Anwendung sind Signaturen (s. Kap. 9.3). Als Fingerabdruck dienen sogenannte **Hashfunktionen**.

Für die Prüfsummen wäre ein naiver Ansatz: wähle eine große Zahl q, betrachte die beiden Dateien als (riesige) Binärzahlen m und m', berechne $m \bmod q$ und $m' \bmod q$ und vergleiche diese Zahlen. Sind sie verschieden, sind die Dateien garantiert auch verschieden. Sind beide Zahlen gleich, so ist die Wahrscheinlichkeit hoch, dass auch beide Dateien gleich sind. Dieser Ansatz ist schon für obige Anwendung brauchbar, aber für kryptographische Zwecke brauchen wir mal wieder etwas Besseres.

Um präzise formulieren zu können brauchen wir im Folgenden ein paar Begriffe.

Definition 7.1. Eine **Hashfunktion** ist eine Funktion h, die eine Zahl m beliebiger Länge auf eine Zahl $h(m)$ fester Länge abbildet.
Eine **Kompressionsfunktion** ist eine Funktion f, die eine Zahl m der Länge ℓ auf eine Zahl $f(m)$ der Länge $k < \ell$ abbildet.
Ist der Wert von $h(m)$ gegeben, so heißt jedes (!) m' mit $h(m') = h(m)$ **Urbild** von $h(m)$. (Also ist insbesondere m ein Urbild von $h(m)$).

Kryptographische Hashfunktionen h müssen dabei weitere Eigenschaften haben, um brauchbar zu sein. Sie sollen

1. Effizient berechenbar sein.
2. Es soll schwierig sein, ein Paar m, m' zu finden mit $h(m') = h(m)$. (Das h heißt dann **stark kollisionsresistent**. Das Paar m, m' heißt dann **Kollision**).
3. Für fast alle gegebenen m soll es schwierig sein, ein m' zu finden mit $h(m') = h(m)$ (Das h heißt dann **schwach kollisionsresistent**).

4. Für fast alle gegebenen Werte $h(m)$ (m ist hier unbekannt!) soll es schwer sein, ein m' zu finden mit $h(m') = h(m)$ (h heißt dann **urbildresistent**).

Man überlege sich, dass die drei Resistenz-Begriffe eine Hierarchie definieren, falls 1 erfüllt ist: Wenn Eve 4 kann ($h(m)$ ist vorgegeben, aber m ist unbekannt), kann sie auch 3 und 2. Wenn Eve nur 3 kann ($h(m)$ ist vorgegeben und m ist bekannt), kann sie auch 2, aber nicht unbedingt 4. Bei 2 darf Eve beliebige m, m' ausprobieren. (Die $h(m)$ dazu kann Eve wegen 1 einfach berechnen.) Wenn wir also Hashfunktionen mit Eigenschaft 2 finden, sind wir sehr zufrieden.

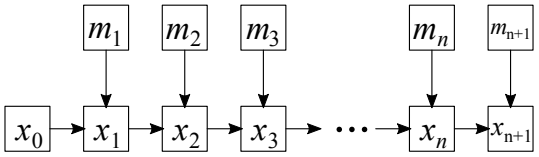
Ganz allgemein wird zur Konstruktion von Hashfunktionen ganz oft das Merkle-Damgård-Schema benutzt.

Merkle-Damgård-Schema. Wähle eine Kompressionsfunktion f, die Zahlen der Länge $\ell+k$ auf Zahlen der Länge k abbildet; sowie einen Startwert x_0 der Länge k.

1. Teile m in Blöcke $m_1, m_2, \ldots, m_n$ der Länge ℓ. Hänge $m_{n+1} = n$ an (Length Padding).
2. Berechne $x_i = f(x_{i-1}, m_i)$ für $i = 1, 2, \ldots, n+1$. Ausgabe $h(m) = x_{n+1}$.

Wie immer gehen wir hier stillschweigend davon aus, das wir die m_i mit Nullen auffüllen, falls es mit der Länge nicht hinhaut.

Manchmal wird noch ein letzter Schritt angehängt und $g(x_{n+1})$ ausgegeben (für eine als sinnvoll erachtete Funktion g). Oft ist $k = \ell$. Das Merkle-Damgård-Schema im Bild:



Beispiel 7.1. Wir betrachten das Spielzeugbeispiel der Hashfunktion h, die aus der Anwendung der Merkle-Damgård-Schemas auf die Kompressionsfunktion

$$f : Z_{100} \times Z_{100} \to Z_{100}, \quad f(x, y) = x + 7y \bmod 100$$

resultiert. Wir kodieren Buchstaben wieder als zweistellige Zahlen mit $a = 00, b = 01, \ldots, l = 11, m = 12, n = 13, \ldots, z = 25$. Wir wählen als Startwert $x_0 = 16$. Dann berechnet sich für $m =$alice der Hashwert so:

Zunächst Länge des Wortes, also 5, anhängen: alice $\to$ alicef $\to$ (0,11,8,2,4,5). Dann

$$\begin{array}{ccccccccccccc} & & 0 & & 11 & & 8 & & 2 & & 4 & & 5 \\ & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ 16 & \to & 16 & \to & 93 & \to & 49 & \to & 63 & \to & 91 & \to & 26 \end{array}$$

Also ist $h(m) = 26$. In wie weit erfüllt dieses f die obigen Anforderungen? Es ist leicht zu berechnen, check. Ist es kollisionsresistent? Nein: Probieren liefert

$$h(\text{alice}) = 26, \quad h(\text{blice}) = 33, \quad h(\text{clice}) = 40, \quad h(\text{dlice}) = 47, \ldots$$

$$\text{und } h(\text{akice}) = 19, \quad h(\text{ajice}) = 12, \quad h(\text{aiice}) = 5, \ldots$$

Also ist es leicht, Kollisionen zu "alice" zu finden, etwa bkice, cjice, ... oder auch aljbe. Somit ist h nicht schwach kollisionsresistent, also erst recht nicht stark kollisionsresistent. (Obwohl hier die Zahlen natürlich mal wieder viel zu klein sind, um realistisch zu sein: das war zu einfach.)

Bessere (und schlechtere) Hashfunktionen werden auf dem Übungsblatt 11 betrachtet.

Können wir vielleicht eine gute Hashfunktion mit diskreten Logarithmen finden? Dazu sei p eine hohe Primzahl und g ein Erzeuger von Z_p^*.

Versuch 1: Sei $h : Z_p \to Z_p^*, h(m) = g^m \bmod p$. Das ist leicht zu berechnen (check) und stark kollisionsresistent, denn: es ist sogar kollisions*frei*, also ist es gar keine Hashfunktion. OK, man könnte den Definitionsbereich erweitern, so dass $m > p$ sein darf. Aber dann ist es einfach, eine Kollision zu finden: $h(m+p-1) = h(m)$, denn $g^{m+p-1} \equiv g^m \cdot g^{p-1} \equiv g^m \cdot 1 \equiv g^m \bmod p$ (Euler-Fermat strikes again; bzw Lagrange.)

Versuch 2: Sei $h : Z_p \times Z_p \to Z_p^*, h(m_1, m_2) = g^{m_1+m_2} \bmod p$. Das ist wieder leicht zu berechnen (check), und es bildet p^2 Werte auf p Werte ab. (h könnte also auch als Kompressionsfunktion dienen.) Dennoch ist es einfach, eine Kollision zu finden:

$$h(m_1 + i, m_2 - i) = g^{m_1+i+m_2-i} = g^{m_1+m_2} = h(m_1, m_2) \mod p$$

Versuch 3: Wir wählen einen weiteren Erzeuger a von Z_p^* und setzen

$$h_a : Z_p \times Z_p \to Z_p^*,\ h(m_1, m_2) = g^{m_1} a^{m_2} \mod p$$

Das liefert in der Tat eine gute Hash- bzw Kompressionsfunktion. Man kann zeigen, dass hier eine Kollision zu finden im Allgemeinen genau so schwer ist wie diskrete Logarithmen zu berechnen. Es gibt eine Ausnahme: $\varphi(p) = p - 1$ ist gerade, also ist $\frac{p-1}{2} \in \mathbb{Z}$, und dann ist wegen Euler-Fermat

$$g^{m_1+\frac{p-1}{2}} a^{m_2+\frac{p-1}{2}} \equiv g^{m_1}(-1)a^{m_2}(-1) \equiv g^{m_1} a^{m_2} \mod p.$$

Das liefert zu jedem (m_1, m_2) aber nur genau eine Kollision. Damit kann man entweder leben, oder aber man kann den Definitionsbereich von h verkleinern auf $\{1, 2, \ldots, \frac{p-1}{2}\} \times \{1, 2, \ldots, \frac{p-1}{2}\}$. Dieses Beispiel könnte man also in. ein Merkle-Damgård-Schema einbauen und würde eine geeignete Hashfunktion erhalten. In der realen Welt nutzt man aber andere Hashfunktionen.

Bemerkung 7.1. Das Geburtstagsparadoxon (Satz 5.2) sagt uns, dass es im Schnitt etwa $\sqrt{p}$ Versuche braucht, um eine Kollision zu finden, wenn p die Anzahl der möglichen Hashwerte ist. Das liefert hier das Maß für geeignete Schlüssellängen: Wenn p nun d bit hat, also $p \approx 2^d$, dann müssen $2^{d/2}$ Versuche jenseits des Machbaren sein. So ist etwa $p \approx 2^{40}$ zu klein, denn $2^{20} = (2^{10})^2 \approx 1\,000\,000$ Versuche sind für einen heutigen Rechner leicht möglich. Jede Methode, die deutlich weniger als $2^{d/2}$ Operationen braucht (etwa um eine Kollision zu finden), wäre ein erfolgreicher Angriff.

In der Praxis benutzte krytopgraphische Hashfunktionen sind von anderer Natur als das Beispiel oben. In den letzten Jahrzehnten aktuell und wichtig waren die folgenden (Familien von) Hashfunktionen:

- MD5 (Message-Digest Algorithm 5, 1991) Erzeugt einen 128-bit Hashwert. Es wurden über die Jahre mehr und mehr Tricks zum Finden von Kollisionen gefunden, so dass MD5 heute nicht mehr geeignet gilt für kryptographische Zwecke. Z.B. wurde MD5 für Sicherheits-Zertifikate von Webseiten benutzt. Stevens, Sotirov, Appelbaum, Lenstra, Molnar, Ostvik und Weger (2009) konnten ein solches Zertifikat erfolgreich fälschen. (Also zu einem gültigen Hashwert eines echten Zertifikates ein zweites Urbild finden, das von allen Webbrowsern als gültiges Zertifikat anerkannt wurde.)

- SHA-1 (Security Hash Algorithm 1, 1995) produziert einen 160-bit Hashwert. Seit 2005 wurden mehr und mehr Angriffe auf SHA-1 bekannt, die die Zahl der Operationen zum Finden einer Kollision von 2^{80} auf 2^{69}, bzw 2^{61} senkt. Mit genug Rechnern rückt das damit ins Machbare (Schätzung: 2 Mio Euro Rechnerpower). Seit 2010 raten daher viele Organisationen zu SHA-2- oder SHA-3-basierten Standards. Seit 2017 akzeptieren die verbreitetsten Browser (von Google, Microsoft, Apple und Mozilla) keine SHA-1 basierten Zertifikate mehr. Einem Team vom CWI Amsterdam (Forschungsinstitut für Mathe und Informatik) und google gelang es 2017, zwei verschiedene pdf-Dateien mit demselben SHA-1-Hashwert zu konstruieren.

- SHA-2 (entworfen von der NSA, veröffentlicht vom NIST 2001 bis 2004) ist eine Familie von Hashfunktionen, die 224, 256, 384 or 512 bits ausspucken. Dabei sind SHA-256 und SHA-512 die eigentlichen Hashfunktionen, und SHA-224, SHA-384, SHA-512/224, SHA-512/256 schneiden einfach die Ausgaben von SHA-256 (für SHA-224) bzw von SHA-512 (für die anderen drei) auf die entsprechende Bitzahl ab.

Alle Hashfunktionen oben benutzen ein (sehr komplexes) Merkle-Damgård-Schema. Im Folgenden wird SHA-2 etwas genauer vorgestellt, wobei wir die hässlichsten Detail weglassen.

SHA-256 berechnet einen Hashwert eines 512-bit-Worts und arbeitet intern mit 32-bit-Worten ($512 = 16{\cdot}32$). Falls das Wort kürzer ist wird es aufgefüllt (eine 1 anhängen, dann Nullen, dann die Länge als 64bit Wort; so dass die Gesamt-Bitlänge nun ein Vielfaches von 512 ist). Für jedes der 512-bit Worte wird folgendes getan: Zerlege es in 16 Worte mit je 32 bit. Aus diesen Eingabeworten $m_1, m_2, \ldots, m_{16}$ werden zunächst 48 weitere Worte $m_{17}, \ldots, m_{64}$ berechnet. Eine Runde i verarbeitet acht Worte A, B, C, D, E, F, G, H plus das m_i zu acht weiteren Worten $A', B', C', D', E', F', G', H'$. (Eine Runde entspricht der einmaligen Anwendung der Kompressionsfunktion: $f(A, B, C, D, E, F, G, H, m_i) = (A', B', C', D', E', F', G', H')$.) Diese sind dann die neue Eingabe A, B, C, D, E, F, G, H für Runde $i + 1$. In Runde 1 haben die Worte A, B, C, D, E, F, G, H vorgegebene konstante Werte (und zwar die ersten 32 Nachkommastellen der Quadratwurzeln der ersten acht Primzahlen in Binärschreibweise). Es gibt insgesamt 64 Runden. Das Ergebnis $A', B', C', D', E', F', G', H'$ wird als Hashwert ausgegeben. Weiterhin sind $k_1, k_2, \ldots, k_{64}$ feste Werte („Rundenschlüssel“, die ersten 32 Nachkommastellen der Kubikwurzeln der ersten 64 Primzahlen in Binärschreibweise). Was in einer Runde passiert ist im folgenden Diagramm in Abbildung 6 dargestellt. Dabei heißt

- $+$ einfach Addition mod 2^{32} (also hier nicht bitweise, sondern mit Überträgen: 011+010=101).

- $\mathrm{Ch}(E.F.G)$ heißt bitweise „if E_j then F_j else G_j“ (für $E = (E_1, \ldots, E_{32})$ usw).

- $\mathrm{Ma}(A, B, C)$ heißt bitweise „(A_i AND B_i) OR (B_i AND C_i) OR (A_i AND C_i)“.

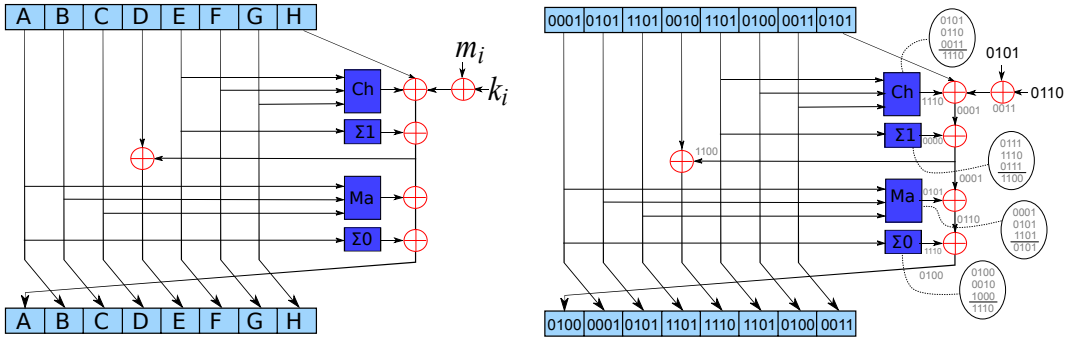


Abbildung 6: Eine Runde von SHA-2: links das allgemeine Prinzip, rechts ein Spielzeugbeispiel (mit 4 statt 32 bit pro Wort).

- ROT^k heißt: rotiere E um k Stellen zyklisch nach rechts. (Also z.B. $\mathrm{ROT}^3((1,2,3,4,5,6,7)) = (5,6,7,1,2,3,4)$).
- $\Sigma 1(A)$ heißt: $\mathrm{ROT}^2(A) \oplus \mathrm{ROT}^{13}(A) \oplus \mathrm{ROT}^{22}(A)$. $\Sigma 0(E)$ heißt: $\mathrm{ROT}^6(E) \oplus \mathrm{ROT}^{11}(E) \oplus \mathrm{ROT}^{25}(E)$.

Genaueres (Pseudocode) auf wikipedia.

Das Ergebnis ist eindrucksvoll: eine gute Hashfunktion soll ja für ganz ähnliche m sehr verschiedene Werte $h(m)$ ausgeben.

SHA256(„Franz jagt im komplett verwahrlosten Taxi quer durch Bayern“) =
d32b568cd1b96d459e7291ebf4b25d007f275c9f13149beeb782fac0716613f8

SHA256(„Frank jagt im komplett verwahrlosten Taxi quer durch Bayern“) =
78206a866dbb2bf017d8e34274aed01a8ce405b69d45db30bafa00f5eeed7d5e

Das m hier hat 60 Zeichen, also in ASCII 480 bit. Die Ausgabe hat 64 Hexadezimalziffern, also 256 bit ($(2^4)^{64} = 2^{256}$). Auch soll die Hashfunktion jede mögliche Struktur im Eingabewort verwischen. Das allereinfachste Eingabewort ist natürlich ein leerer String, und es ist

SHA256(„“) = e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

In weiser Vorausschau suchte das NIST bereits früh nach einem Nachfolger für SHA-2. Im Jahr 2007 schrieben sie einen Wettbewerb aus: bis zum 31.10.2008 konnten Vorschläge für einen neuen Hash-Standard eingereicht werden *(NIST hash function competition)*. Der Gewinner wurde 2012 gekürt und zu einem weiteren Standard gemacht: SHA-3, aka Keccak.

8 AES

Neben RSA und Elliptischen-Kurven-Verfahren ist eines der gebräuchlichsten Verschlüsselungsverfahren der *Advanced Encryption Standard*, kurz AES. Der Vorläufer DES *(Data Encryption Standard)* wurde in den frühen siebziger Jahren von IBM entwickelt. Der Vorläufer des NIST, das NBS *(National Bureau of Standards)*, hat 1977 DES als Standard für Verschlüsselung von sensiblen Dokumenten der US-Regierungen und Behörden festgelegt (sensibel, nicht geheim: für höhere Stufen der Geheimhaltung ist die NSA verantwortlich; dennoch

hat die NSA den DES-Standard mitgestaltet). Daher musste jede Software mit Verschlüsselungsfunktion, die an US-Behörden verkauft wurde, DES können. Geschäfte mit Regierungen können sehr gewinnbringend sein, somit nutzten bald viele Unternehmen DES, und so war es bald weitverbreitet.

Über die Jahre wurden viele Angriffe auf DES entwickelt. Bereits in den frühen 80er Jahren gab es Gerüchte (nie belegt), dass die NSA DES-verschlüsselte Dokumente entschlüsseln könne; ja, dass sie gar eine Hintertür in den Standard eingebaut hatte. Wahr oder nicht, 1998 stellte die EFF (*Electronic Frontier Foundation*, sowas wie ein US Chaos Computer Club, aber mit mehr Juristen) ganz offen eine Methode vor, wie man systematisch (aber aufwendig) DES-verschlüsselte Nachrichten lesen kann ("the 250 000$ DES cracker").

Als Reaktion schrieb das NIST (Nachfolger des NBS) 1997 einen Wettbewerb aus, um den Sieger zum Nachfolger von DES zu küren. In diesem Wettbewerb gab es präzise Vorgaben (128-bit Block-Verschlüsselung, Schlüssellängen von 128, 192 und 256 bit, Effizienz, usw... und es sollte plausibel gemacht werden können, dass es keine eingebaute Hintertür gibt!). Gewinner wurde in diesem gut besetzten Wettbewerb (u.a. Ron Rivest, oder Bruce Schneier, s. Literaturliste) am 2.10.2000 das System Rijndael der beiden Belgier Joan Daemen und Vincent Rijmen. Das kennen wir heute als AES. (Der Erfolg dieses Wettbewerbs motivierte gewiss auch die spätere NIST hash function competition.)

Ganz ähnlich wie bei Hashfunktionen rührt AES die Nachricht in vielen Runden so nachhaltig und gewissenhaft durch, wie es auch eine Hashfunktion macht: Fast gleiche Nachrichten werden sehr verschieden verschlüsselt, jede eventuelle Struktur in der Nachricht wird verwischt,... Es werden auch viele verschiedene Operationen benutzt, ähnlich wie bei SHA-2, und nicht eine einzige, wie bei RSA oder elliptischen Kurven. Dennoch: Algebra will strike again! Sie erlaubt eine besonders effiziente Beschreibung von AES. Wegen seiner Relevanz erläutern wir hier auch AES etwas genauer.

Bemerkung 8.1. AES benutzt Bytes, also 8-bit-Worte, auf sehr verschiedene und sehr kreative Weise. Eine Operation ist einfach bitweises Addieren, wie wir es auch schon vorher gesehen haben (z.B. bei unserer Variante des One-Time-Pads). Eine andere Weise ist die Darstellung eines 8-bit-Worts $(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$ als Polynom über $\mathbb{F}_2$: $b_7x^7+b_6x^6+\cdots+b_1x+b_0$. Die Menge aller dieser Polynome bezeichnet man mit $\mathbb{F}_2[x]$. Damit stehen uns in $\mathbb{F}_2[x]$ mindestens vier (Mix-)Operationen zur Verfügung:

- $p+q$: Das ist einfach bitweises Addieren! Wie etwa $(0,1,1,0,0,1,1,0)$ "plus" $(0,0,1,1,1,1,1,0)$: Das ist

  $$x^6+x^5+x^2+x\ +x^5+x^4+x^3+x^2+x = x^6+2x^5+x^4+x^3+2x^2+2x \equiv x^6+x^4+x^3 \mod 2,$$

  also gleich $(0,1,0,1,1,0,0,0)$.

- $p \cdot q$: Multiplizieren der Polynome. Also z.B. $(0,0,0,0,0,1,1,0) \star (0,0,0,0,1,1,0,1)$, das ist

  $$(x^2+x)\cdot(x^3+x^2+1) = x^5+2x^4+x^3+x^2+x \equiv x^5+x^3+x^2+x \mod 2,$$

  also gleich $(0,0,1,0,1,1,1,0)$. Hier haben wir offenbar ein Problem: Das Produkt zweier Polynome vom Grad 7 kann Grad 14 haben. Eine Lösung bietet die dritte Operation:

- p mod q. Ein Polynom von größerem Grad als 7, etwa $p = x^{11}+x^7+x^3+1$, kann durch ein kleineres mit Rest geteilt werden. Dazu hilft Polynomdivision: Sei etwa $q = x^7 + x^4 + x$. Dann ist

$$\begin{array}{llll} x^{11} & +x^7 & +x^3 & +1 = (x^7 + x^4 + x)(x^4 + x + 1) \; + x^4 + x^3 + x^2 + x + 1 \\ \underline{x^{11}} & \underline{+x^8} & \underline{+x^5} & \\ & x^8 & +x^7 & +x^5 + x^3 + 1 \\ & \underline{x^8} & \underline{+x^5} & \underline{+x^2} \\ & & x^7 & +x^3 + x^2 + 1 \\ & & \underline{x^7} & \underline{+x^4 + x} \\ & & & x^4 + x^3 + x^2 + x + 1 \end{array}$$

  Also ist $x^{11} + x^7 + x^3 + 1 \equiv x^4 + x^3 + x^2 + x + 1 \bmod x^7 + x^4 + x$. Das liefert auch folgende Operation:

- p^{-1}. Die Konstruktion von $\mathbb{F}_n$ aus $\mathbb{Z}$ können wir hier übertragen: wenn wir in $\mathbb{Z}$ alles modulo n rechnen ($n \in \mathbb{Z}$), dann erhalten wir $\mathbb{F}_n$. Genau so erhalten wir hier, wenn wir in $\mathbb{F}_2[x]$ alles modulo q rechnen (q in $\mathbb{F}_2[x]$, also q ein Polynom!), einen neuen Körper (bzw nur einen Ring) der mit,$\mathbb{F}_2[x]/q$ bezeichnet wird. Daher können wir auch p^{-1} berechnen — entweder, wenn $\mathbb{F}_2[x]/q$ ein Körper ist, dann geht das für alle $p \neq 0$, oder — falls $\mathbb{F}_2[x]/q$ nur ein Ring ist, kein Körper — wenn p in der Einheitengruppe von $\mathbb{F}_2[x]/q$ ist. Das geht wieder mit dem erweiterten euklidischen Algorithmus.

Bei all dem ist wichtig, dass man die Polynome als formale Ausdrücke auffasst, mit denen man wie oben gezeigt ganz konkret rechnen kann. Die Sichtweise, das Polynom als Funktion mit einer Wertetabelle aufzufassen (wie in Abschnitt 6) ist hier ungeeignet! Denn: wie viele verschiedene Polynome gäbe es dann nur?

AES. AES ist ein symmetrisches Verfahren: derselbe Schlüssel k dient zum Ver- und Entschlüsseln des Klartexts m bzw des Schlüsseltexts c. Dabei wird m in 128-bit Blöcke zerlegt. Als Schlüssellängen sind 128, 192 und 256 bit möglich. Wir beschreiben hier die Version für 128-bit-Schlüssel. Die Versionen mit den anderen Schlüssellängen unterscheiden sich u.a. dadurch, dass sie mehr Runden durchlaufen. 21. Juni

Der zu verschlüsselnde Text m wird aufgefasst als 16 Bytes $m_0, \ldots, m_{15}$ ($2^4 \cdot 2^3$ bit $= 128$ bit). Diese werden in eine 4×4-Matrix geschrieben:

$$M = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \tag{9}$$

Auf diese werden nun wiederholt vier verschiedene Operationen angewandt: AddRoundKey, SubBytes, ShiftRows, MixColumns. AES mit 128-bit-Schlüsseln durchläuft 11 Runden: eine erste Runde, die nur aus AddRoundKey besteht, dann 10 Runden mit allen vier Operationen, wobei in derletzten MixColumns weggelassen wird. Eine Übersicht ist in Abb. 7 gezeigt. Für jede Runde i wird ein eigener Schlüssel k_i benutzt, wobei k_0 der vorgegebene Schlüssel ist. Auch die k_i sind 128-bit Worte, die genau so wie m als 4×4-Matrix K geschrieben werden. Die $k_1, \ldots, k_{10}$ werden aus k_0 berechnet wie unten beschrieben. Die einzelnen Operationen funktionieren wie folgt:

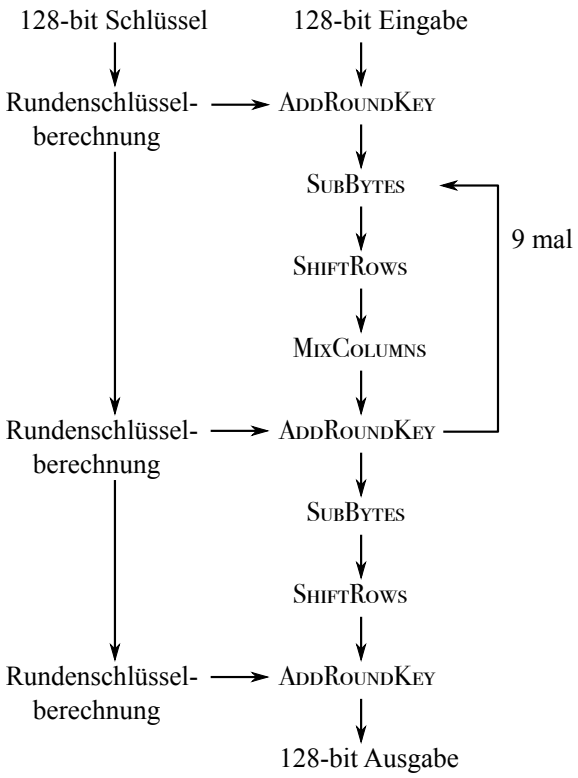


Abbildung 7: Das allgemeine Schema bei AES mit 128-bit Schlüsseln.

AddRoundKey: Für die ganze *Matrix* M berechne $M + K$. Das heißt: für jeden *Eintrag* m_j von M: Bitweises Addieren (XOR) des entsprechenden Eintrags k_j. Ist dasselbe wie Addieren der den Einträgen entsprechenden Polynome in $\mathbb{F}_2[x]$.

SubBytes: Für jeden *Eintrag* m_i: Fasse m_i als Polynom p auf (wie in Bemerkung 8.1). Setze $p_0 = x^6 + x^5 + x + 1$ und $p_1 = x^4 + x^3 + x^2 + x + 1$. Berechne

- $p' = p^{-1}$ in $F := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$.
- $p'' = p_1 \cdot p'$ in $R := \mathbb{F}_2[x]/(x^8 + 1)$.
- Ausgabe $p'' + p_0$.

Da F ein Körper ist, existiert p^{-1} für alle $p \in F \setminus \{0\}$. Falls $p = 0$, dann setzt man einfach $p^{-1} = 0$. (R ist übrigens kein Körper, nur ein Ring.)

ShiftRows: Für die ganze *Matrix*, rotiere Zeile i um i zyklisch nach links. Also:

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \longrightarrow \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_5 & m_9 & m_{13} & m_1 \\ m_{10} & m_{14} & m_2 & m_6 \\ m_{15} & m_3 & m_7 & m_{11} \end{pmatrix}$$

MixColumns: Für jede *Spalte* (b_3, b_2, b_1, b_0): Betrachte das Polynom $p = b_3y^3 + b_2y^2 + b_1y + b_0$ als Element von $F[y]$. Dann berechne $p \cdot (3y^3 + y^2 + y + 2)$ in $F[y]/(y^4 + 1)$. Das heißt:

- Berechne $(b_3y^3 + b_2y^2 + b_1y + b_0) \cdot (3y^3 + y^2 + y + 2)$ in $F[y]$ modulo $y^4 + 1$. ABER
- Zum Berechnen des Produkts $a_i \cdot b_j$ zweier Koeffizienten a_i, b_j der Polynome, oder deren Summen, fasse a_i und b_j als Polynome in $F = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$ auf und berechne das Produkt wie im ersten Schritt von SubBytes das Inverse berechnet wurde. Also in $\mathbb{F}_2$ und dann modulo $(x^8 + x^4 + x^3 + x + 1)$.

Bemerkung 8.2. Was wir hier formal tun ist sehr abgehoben: wir betrachten ein Polynom $p = b_3y^3 + b_2y^2 + b_1y + b_0$ in der Variablen y (also $p \in F[y]/(y^4 + 1)$. Aber die b_i betrachten wir als Polynome in der Variablen x (also $b_i \in F = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$). Formal ist p also ein Polynom über einem Polynomring! Das ist nicht mehr anschaulich, aber man kann es sowohl tun als auch damit rechnen.

Beispiel 8.1. Zu AddRoundKey und zu ShiftRows siehe Aufgabe 45 von Blatt 12, zu SubBytes siehe Aufgabe 46 von Blatt 12; und zu MixColumns siehe Aufgabe 47 von Blatt 12. Dennoch hier ein Beispiel:

SubBytes: Für jeden Eintrag a der Matrix müssen wir a als Polynom auffassen und zunächst das Inverse in $F = \mathbb{F}_2[x]/(x^8+x^4+x^3+x+1)$ berechnen. Sei hier ein Eintrag $(0, 0, 0, 1, 1, 0, 1, 0)$. Das entspricht $p = x^4 + x^3 + x$. Wir finden das Inverse p^{-1} in F, indem wir den erweiterten euklidischen Algorithmus auf p und $x^8 + x^4 + x^3 + x + 1$ anwenden.

$$\begin{array}{rc|c|cc} x^8 + x^4 + x^3 + x + 1 & & / & 1 & 0 \\ x^4 + x^3 + x & & x^4 + x^3 + x^2 & 0 & 1 \\ x + 1 & & x^3 + 1 & 1 & x^4 + x^3 + x^2 \\ 1 & & - & x^3 + 1 & x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1 \end{array}$$

Dazu müssen wir zwei Polynomdivisionen durchführen, nämlich: $x^8+x^4+x^3+x+1 = (x^4+x^3+x)(x^4+x^3+x^2)+x+1$ und $x^4+x^3+x = (x+1)(x^3+1)+1$. Damit ist

$$(x^4+x^3+x)(x^7+x^6+x^5+x^4+x^3+x^2+1)+(x^8+x^4+x^3+x+1)(x^3+1)=1,$$

also auch $(x^4+x^3+x)(x^7+x^6+x^5+x^4+x^3+x^2+1) \equiv 1 \bmod x^8+x^4+x^3+x+1$, also $p^{-1} = x^7+x^6+x^5+x^4+x^3+x^2+1$ in F. Dann:

$$p'p_1 = (x^7+x^6+x^5+x^4+x^3+x^2+1)(x^4+x^3+x^2+x+1) = x^{11}+x^9+x^7+x^6+x^3+x+1$$

Das nun mod x^8+1. Das geht mit einer weiteren Polynomdivision. Es geht aber auch viel einfacher: Dazu beachte man, dass wegen $x^8+1 \equiv 0 \bmod x^8+1$, also $x^8 \equiv 1 \bmod x^8+1$, gilt:

$$\begin{aligned} &b_{15}x^{15}+b_{14}x^{14}+\cdots+b_8x^8+b_7x^7+b_6x^6+\cdots b_1x+b_0 \\ \equiv&(b_{15}+b_7)x^7+(b_{14}+b_6)x^6\cdots+(b_9+b_1)x+b_8+b_0 \mod x^8+1. \end{aligned}$$

Daher ist

$$p'' = p'p_1 = x^{11}+x^9+x^7+x^6+x^3+x+1 \equiv x^7+x^6+2x^3+2x+1 \mod x^8+1,$$

also gleich x^7+x^6+1 in $\mathbb{F}_2[x]$. Dann:

$$p''+p_0 = x^7+x^6+1+x^6+x^5+x+1 \equiv x^7+x^5+x \mod 2$$

SubBytes angewandt auf $(0,0,0,1,1,0,1,0)$ liefert also $(1,0,1,0,0,0,1,0)$.

MixColumns wird auf Spalten angewandt. Bisher haben wir die m_{ij} als Polynome in F betrachtet. Genausogut können wir uns den Koeffizientenvektor als Binärzahlen vorstellen (also für x^7+x^2+1 eben 10000101 usw). Genausogut können wir das dann als zweistellige Hexadezimalzahlen schreiben. Sei also eine Spalte gleich $(A0, 80, 01, 02)$ in der Darstellung als Hexadezimalzahl. Betrachte das als Polynom $A0y^3+80y^2+01y+02$ und berechne

$$\begin{aligned} &(A0y^3+80y^2+01y+02)(03y^3+01y^2+01y+02) = A0\cdot 03y^6+(A0\cdot 01+80\cdot 03)y^5 \\ &+(A0\cdot 01+80\cdot 01+01\cdot 03)y^4+(A0\cdot 02+80\cdot 01+01\cdot 01+02\cdot 03)y^3 \\ &+(80\cdot 02+01\cdot 01+02\cdot 01)y^2+(01\cdot 02+02\cdot 01)y+02\cdot 02 \end{aligned}$$

Leider müssen wir nun die Produkte $A0\cdot 03$ usw als Produkte von Polynomen mod $x^8+x^4+x^3+x+1$ berechnen. Also ist z.B.

$$02\cdot 02 \rightsquigarrow t\cdot t = t^2 \rightsquigarrow 04, \text{ oder}$$

$$A0\cdot 02+80\cdot 01+01\cdot 01+02\cdot 03 \rightsquigarrow (x^7+x^5)x+x^7\cdot 1+1\cdot 1+x(1+x) = x^8+x^7+x^6+x^2+x+1$$

Das mod $x^8+x^4+x^3+x+1$ erfordert eine weitere Polynomdivision. Oder wir nutzen in diesem Fall einen Trick: weil $x^8+x^4+x^3+x+1 \equiv 0 \bmod x^8+x^4+x^3+x+1$ ist auch (in $\mathbb{F}_2[x]$)

$$x^8 \equiv x^4+x^3+x+1 \mod x^8+x^4+x^3+x+1.$$

Also ist

$$\begin{aligned} &x^8+x^7+x^6+x^2+x+1 \equiv x^4+x^3+x+1+x^7+x^6+x^2+x+1 \\ \equiv&x^7+x^6+x^4+x^3+x^2 \mod x^8+x^4+x^3+x+1. \end{aligned}$$

Das ist 1101 1100 bzw DC. Insgesamt erhalten wir also

$$(A0y^3+80y^2+01y+02)(03y^3+01y^2+01y+02) = FBy^6+3By^5+23y^4+DCy^3+18y^2+00y+04.$$

Das nun noch mod y^4+1. Wieder ist Letzteres wegen $y^4 \equiv 1 \bmod y^4+1$ einfach:

$$FBy^6 + 3By^5 + 23y^4 + DCy^3 + 18y^2 + 00y + 04 \equiv$$
$$DCy^3 + (FB+18)y^2 + (3B+00)y + (23+04) \equiv DCy^3 + E3y^2 + 3By + 27 \mod y^4+1$$

Addieren erfolgt wieder bitweise, z.B.: $FB+18 \rightsquigarrow (1111\,1011) \oplus (0001\,1000) = (1110\,0011) \rightsquigarrow E3$ usw. MixColumns angewandt auf $(A0, 80, 01, 02)$ liefert also $(DC, E3, 3B, 27)$.

Bemerkung 8.3. Einige der Berechnungen oben von Hand durchzuführen ist sehr aufwendig, vgl Übungsblatt 12. Fast alle sind aber sehr computerfreundlich: 28. Juni

• Erstens können und sollten Sie in `sagemath` mal den zauberhaften und wundersamen Befehl `x=PolynomialRing(GF(2),'x').gen()` ausprobieren. Danach weiß `sagemath`, dass zum Beispiel `p=x^4+x+1` und `q=x^6+x^5+1` Polynome in $\mathbb{F}_2[x]$ sind, und Sie können ganz leicht p mod q rechnen: `p%q`, oder p mal q: `p*q` usw.

Weiterhin ist z.B. mit `P=Matrix(GF(2),[[1,0,0],[0,1,1],[1,0,1]])` für `sagemath` das P nun eine Matrix über $\mathbb{F}_2$, und z.B. `P.inverse()` liefert ihre inverse Matrix über $\mathbb{F}_2$.

• Zweitens sind alles Operationen auf bits und Bytes (und ganz viel XOR, oder verschieben). Zum Beispiel ist Multiplizieren zweier Polynome in $\mathbb{F}_2[x]$ leicht bitweise zu beschreiben:

$$(b_0+b_1x+b_2x^2\cdots)(c_0+c_1x+c_2x^2\cdots) = (b_0c_0)+(b_1c_0+b_0c_1)x+(b_2c_0+b_1c_1+b_0c_2)x^2+\cdots$$

Hier ist + wieder bitweises XOR, und Mal ist zunächst wirklich malnehmen (bitweises AND mit Übertrag), bevor modulo $x^8+x^4+x^3+x+1$ reduziert wird.. Aber auch mod $x^8+x^4+x^3+x+1$ ist einfach zu beschreiben: IF (bit 8 ist da) THEN (bit 8 weglassen, Rest XOR 1B bitweise). Und: IF (bit 9 ist da) THEN (geeignet verschieben, bit 8 weglassen, Rest XOR 1B bitweise, zurückverschieben).

• Drittens werden einige Schritte per table-look-up (in einer Tabelle nachsehen) erledigt, wie etwa p^{-1} in F berechnen, oder "mal 03 in F". Dazu braucht man jeweils nur eine Tabelle mit wenigen Einträgen (Quizfrage: wieviele?).

• Viertens: einige Operationen können als Matrix-Multiplikation realisiert werden: So ist MixColumns von (a_3, a_2, a_1, a_0) dieses:

$$\begin{pmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} 02 & 01 & 01 & 03 \\ 03 & 02 & 01 & 01 \\ 01 & 03 & 02 & 01 \\ 01 & 01 & 03 & 02 \end{pmatrix} \begin{pmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix}$$

Das lässt sich ja schreiben als vier Gleichungen, also vier Zeilen Programmcode:

$$b_3 = 02\cdot a_3 + 01\cdot a_2 + 01\cdot a_1 + 03\cdot a_0$$
$$b_2 = 03\cdot a_3 + 02\cdot a_2 + 01\cdot a_1 + 01\cdot a_0$$
$$b_1 = 01\cdot a_3 + 03\cdot a_2 + 02\cdot a_1 + 01\cdot a_0$$
$$b_0 = 01\cdot a_3 + 01\cdot a_2 + 03\cdot a_1 + 02\cdot a_0$$

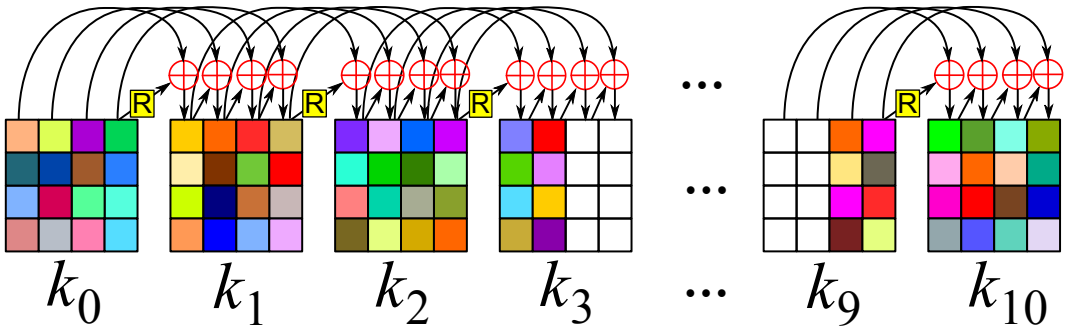


Abbildung 8: Das Schema zur Berechnung der Rundenschlüssel von AES. Die Einträge der Matrix sind Bytes, also etwa zweistellige Hexadezimalzahlen; die sind hier durch Farben dargestellt.

Obacht: "Mal 03" heißt wieder "mal 03 in F"; das wird wieder per table-lookup gemacht. (Obwohl es ja auch einfach zu implementieren wäre.)

Analog kann man Multiplikation von Polynomen modulo $x^8 + 1$ als Multiplikation von entsprechenden Matrizen darstellen,.

Es fehlt noch die Rundenschlüsselberechnung. Wir skizzieren das hier nur. Der vorgegebene Schlüssel k mit 128 bit wird genau wie m in eine 4×4-Matrix geschrieben (vgl. Gleichung (9)). Die besteht aus vier Spalten s_0, s_1, s_2, s_3, deren Einträge wieder zwei Byte groß sind (n einer Spalte also vier zweistellige Hexadezimalzahlen). Dann werden nach dem Schema in Abbildung 8 zehn weitere 4×4-Matrizen berechnet, bzw 40 weitere Spalten $s_4, s_5, \ldots, s_{43}$. Für die meisten Spalten ist $s_i = s_{i-4} \oplus s_{i-1}$ (bitweise XOR). Falls i ein Vielfaches von 4 ist, ist $s_i = s_{i-4} \oplus R(s_{i-1})$. Die Operation R benutzt SubBytes und sieht so aus:

$$R : F^4 \to F^4, \quad \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \mapsto \begin{pmatrix} \textsc{SubBytes}(b) + p_i \\ \textsc{SubBytes}(c) \\ \textsc{SubBytes}(d) \\ \textsc{SubBytes}(a) \end{pmatrix}$$

Hier ist wieder ist $F = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. Das p_i ist ein Polynom in F, und zwar ist $p_i = x^{i/4-1}$.

Bemerkung 8.4. AES gilt heute (2020) als sicher. Die US-Regierung erlaubt bzw empfiehlt, auch geheime Dokumente mit AES-192 oder AES-256 zu verschlüsseln. Ein brute-force-Angriff auf AES-128 erfordert das Durchprobieren von 2^{128} Schlüsseln. Der beste vollständige Angriff verbessert das nur um den Faktor 4 auf 2^{126}. Mehr Details auf wikipedia, oder in von zur Gathen Kapitel 6.

Bemerkung 8.5. Man könnte streiten über public-key (RSA, ElGamal) versus Systeme mit symmetrischen Schlüsseln wie AES. Die Wahrheit ist, dass sich beide prima ergänzen: Oft (z.B. bei PGP bzw GPG) wird ein public-key-Verfahren benutzt für Signatur einer Nachricht und Verschlüsseln und Senden eines Schlüssels, und der Schlüssel wird dann zum Ver- und Entschlüsseln einer Nachricht mittels eines symmetrischen Verfahrens genutzt. Bei GPG wird u.a. RSA oder ElGamal für Ersteres genutzt, und u.a. AES-128 für Letzteres. (GPG bietet

auch etliche andere Optionen an.) So nutzt man die Vorteile beider Systeme: bei Signatur oder auch sicherem Schlüsseltausch hat ein public-key-Verfahren Vorteile, AES ist effizienter zum Ver- und Entschlüsseln großer Datenmengen.

9 Anwendungen

In diesem Kapitel folgen ein paar Anwendungen, die die "höheren" Zutaten benutzen, die wir sahen. Um das Bild weiter zu bemühen: wäre Kryptographie Essenmachen, dann wären die mathematischen Grundlagen aus Kapitel 2 z.B. Mehl, Wasser, Salz und Hefe; oder passierte Tomaten, Zucker, Salz und Kräuter. Aus den ersten kann man Hefeteig machen (RSA) und aus dem zweiten Tomatensauce (Hashfunktion). Aus den "höheren" Zutaten Hefeteig, Tomatensauce und Käse kann man wiederum Pizza machen (Signatur). Pizza, Gurkensalat und Tiramisu wiederum ergeben ein leckeres Abendessen (Blockchain).

9.1 Commitment

Die Regeln von jüdischem Poker sind folgendermaßen: Alice sagt eine Zahl x. Dann sagt Bob eine Zahl y. Falls $x > y$ gewinnt Alice, falls $x < y$ gewinnt Bob (siehe Ephraim Kishon 1961). Der Nachteil liegt offenbar bei Alice: Bob kennt ihre Zahl, bevor er seine sagen muss. Lösung: Commitment (dt. Hingabe, Verpflichtung, hier: Festlegung). Das geschieht in zwei Phasen: Festlegen, Offenlegen. Beim Festlegen hinterlegt Alice eine Botschaft m. Beim Offenlegen gibt Alice m bekannt. Die geforderten Eigenschaften sind:

- Niemand kann m vorm Offenlegen lesen.
- Niemand kann m nach dem Festlegen ändern, auch Alice nicht.

Wegen des zweiten Punktes ist folgende Idee nicht gut: Alice hinterlegt die mit ihrem öffentlichen Schlüssel e_A kodierte Botschaft $c = f(e_A, m)$. Denn dann könnte Alice beim Offenlegen lügen und einen falschen privaten Schlüssel nutzen (oder bekannt geben) und somit das Ergebnis verfälschen.

Wieder helfen Hashfunktionen, bzw genauer: Sei h eine kollisionsfreie Einwegfunktion (z.B. $h : \{0, 1, \ldots, |G| - 1\} \to G$, $h(m) = g^m$ für eine geeignete Gruppe G mit Erzeuger g. Recall: kollisions*frei* soll heißen injektiv, also, dass $h(m) \neq h(m')$ für $m \neq m'$).

- Festlegung: Alice hinterlegt $c = h(m)$, das h ist öffentlich.
- Offenlegung: Alice gibt m bekannt. Jeder kann dann $c = h(m)$ checken.

Das tut's: Niemand außer Alice kann m vorher lesen, da h eine Einwegfunktion ist. Niemand kann ein falsches m' liefern mit $h(m') = h(m)$, da h kollisionsfrei ist.

9.2 Bit-Commitment

Falls m sehr kurz ist, sagen wir, $m \in \{0,1\}$ ("Nein/Ja"), gibt es bei obigem Commitment ein Problem: auf $\{0,1\}$ gibt es keine Einwegfunktion. Es gibt viele Lösungen. Eine geht so:

Wähle $f : \{0,1\} \times X \to X$ mit $X \subset \mathbb{N}$ groß genug. Das f soll eine Einwegfunktion sein, die im ersten Argument kollisionsfrei ist; d.h.: für alle $x, y \in X$ soll gelten $f(0,x) \neq f(1,y)$.

Bei der Festlegung wählt Alice ein $b \in \{0,1\}$ und ein zufälliges $r \in X$ und hinterlegt $f(b,r)$. Beim Offenlegen gibt Alice die Werte b und r bekannt.

Beispiel 9.1. Wähle $N = pq$ für zwei große Primzahlen p, q. Wähle einen quadratischen Nichtrest y modulo N (also für alle $m \in Z_N :\ m^2 \neq y$) und setze

$$f : \{0,1\} \times Z_N \to Z_N, \quad f(b,r) = y^b r^2 \mod N.$$

Das f ist eine Einwegfunktion, da es für $N = pq$ schwierig ist zu entscheiden, ob y quadratischer Rest modulo N ist (vgl. Bemerkung 2.4). Es ist nicht kollisionsfrei, da in Z_N jede Zahl vier Quadratwurzeln hat (vgl Satz 2.6). Also gibt es $r \neq s$ mit $f(b,r) = y^b r^2 \equiv y^b s^2 = f(s,r)$. Aber f ist kollisionsfrei im ersten Argument: für $b = 0$ ist $f(0,r) = y^0 r^2 = r^2 \mod N$, also quadratischer Rest. Für $b = 1$ ist $f(1,r) = y^1 r^2 = yr^2 \mod N$ kein quadratischer Rest, denn: falls ja, gäbe es m mit $yr^2 \equiv m^2 \mod N$. Falls $\mathrm{ggT}(r,N) = 1$ so existiert r^{-1} in Z_N^*, und damit liefert $m^2(r^{-1})^2 \equiv y \mod N$ einen Widerspruch. (Falls $\mathrm{ggT}(N,r) = p$, dann ist $m^2 \equiv yr^2 \mod N$ wegen des chinesischen Restsatzes äquivalent zu $m^2 = 0 \mod N$ und $m^2 \equiv yr^2 \mod q$ und der Widerspruch ergibt sich in der zweiten Gleichung ganz analog, da jetzt $\mathrm{ggT}(r,q) = 1$.)

Kritische Geister könnten hier einwenden: Falls es doch schwierig ist zu entscheiden, ob y ein quadratischer Rest modulo N ist: wie finden wir denn dann einen quadratischen Nichtrest y? Die Antwort ist: Manche quadratische Reste sind leicht zu finden. Es ist z.B. effizient machbar, zu entscheiden, ob y ein quadratischer Rest oder Nichtrest modulo einer Primzahl p ist (siehe Satz 2.8). Dann erhalten wir aus der Tatsache

$$a \text{ quadr. Rest } \mod pq \Rightarrow (a \text{ quadr. Rest } \mod p \text{ und } a \text{ quadr. Rest } \mod q)$$

durch Anwenden von Logik (Kontraposition)

$$a \text{ quadr. Nichtrest } \mod pq \Leftarrow (a \text{ quadr. Nichtrest } \mod p \text{ oder } a \text{ quadr. Nichtrest } \mod q).$$

Die Hälfte aller Zahlen in Z_p sind quadratische Nichtreste. Also brauchen wir im Schnitt zwei Versuche, um einen quadratischen Nichtrest in Z_p zu finden. Das ist nach obiger Überlegung automatisch ein quadratischen Nichtrest in Z_N. Also brauchen wir im Schnitt nur zwei Versuche.

9.3 Signaturen

5. Juli

Alle bisher besprochenen Verfahren sind anfällig für den Fall, dass sich Eve Bob gegenüber als Alice ausgibt ("Hi Bob, diese Nachricht ist von mir, Alice."). Im Alltag wird das Problem häufig durch die persönliche, handgeschriebene Unterschrift gelöst. Eine Unterschrift erfüllt idealerweise die folgenden Punkte:

- Die Unterschrift sollte fest an das unterschriebene Dokument gebunden sein (und nicht etwa auf einem Post-It).
- Es soll für mich einfach sein, eine Unterschrift anzufertigen.
- Es soll für alle einfach sein, die Echtheit der Unterschrift zu prüfen.
- Es soll für andere schwierig sein, meine Unterschrift zu fälschen. Damit erfüllt sich automatisch:
- Ich soll nicht abstreiten könne, dass ich die Unterschrift geleistet habe.

Digitale Unterschriften sollen dieselben Eigenschaften haben. Die erste naive Idee zur Lösung ist: Alice sendet an Bob die mit Bobs öffentlichem Schlüssel e_B verschlüsselte Nachricht $c = f(e_B, m)$, und außerdem dieselbe Nachricht mit ihrem privaten Schlüssel d_A verschlüsselt: $c' = f(d_A, m)$. Nun kann Bob mit seinem privaten Schlüssel d_B die Nachricht c zu m entschlüsseln. Dann kann er c' mit Alice öffentlichem Schlüssel e_A zu m' entschlüsseln. Falls $m = m'$, so muss die Absenderin im Besitz von Alice privatem Schlüssel sein, also ist die Nachricht gewiss von Alice.

Der Nachteil ist offenbar, dass nun Eve das m mitlesen kann, da ja auch sie mittels Alice öffentlichem Schlüssel e_A das c' zu m entschlüsseln kann. Aber die Lösung liegt nun nahe: Wir bauen eine Hashfunktionen h ein.

Grundprinzip Signatur:

1. Alice sendet an Bob die mit e_B verschlüsselte Nachricht $c = f(e_B, m)$, und außerdem $h(m)$ mit d_A verschlüsselt: $c' = f(d_A, h(m))$.
2. Bob entschlüsselt mit d_B die Nachricht c zu m. Er berechnet dann $h(m)$, sowie aus c' mit e_A ein m'. Falls $h(m) = m'$, so ist die Nachricht von Alice.

Das lässt sich auf viele Public-Key-Verfahren wie RSA und ElGamal anwenden.

Bemerkung 9.1. Ein wichtiger Aspekt bei Signaturen ist: Die Signatur löst das Problem der Authentifikation ("ist Alice wirklich Alice?") nur insoweit, dass nur der Besitzer des privaten Sclüssels der Absender sein kann. Das Problem, dass Eve Bob glauben lässt, der von ihr präsentierte öffentliche Schlüssel gehöre Alice, lässt sich nicht mit rein kryptographischen Methoden lösen. Das muss auch heute noch anders gelöst werden: im Falle des Austauschs verschlüsselter Nachrichten müssen sich Bob und Alice dann eben doch einmal persönlich treffen. (Siehe dazu `https://en.wikipedia.org/wiki/Man-in-the-middle_attack`).

Oder man setzt auf eine zentrale (oder dezentrale) vertrauenswürdige Stelle. Im Falle von Webseitenzertifikaten (für `https`) gibt es eine Palette von vielen nationalen und wenigen weltweiten Anbietern, kommerzielle (IdenTrust, Comodo, DigiCert) und nichtkommerzielle (Mozilla, Let's Encrypt). Bei `https` ist das natürlich sehr wichtig: hier sind Sie Bob, und ihre Onlinebankingwebseite ist Alice. Gibt eine Seite vor, dass sie ihre Bankingseite ist, prüft der Browser, ob sie das korrekte Zertifikat hat. In dem Falle passiert das Analogon einer Signatur: wenn Ihr Browser sich per `https` mit einer Webseite verbindet, fragt er sie als erste nach ihrem Zertifikat: nur wenn das wirklich auf deren Namen ausgestellt ist, und wenn es die Signatur

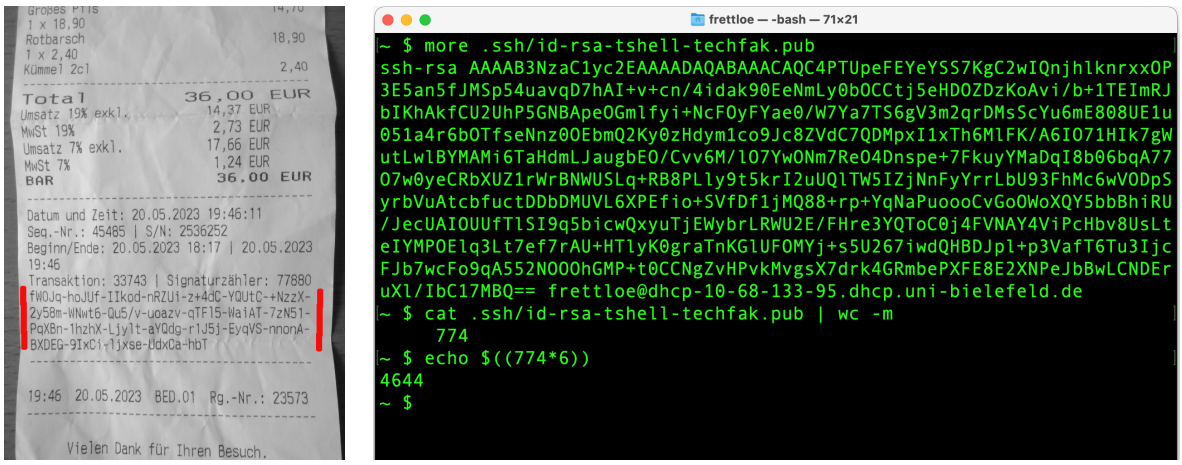


Abbildung 9: Signatur auf einer Restaurantrechnung (links) und mein (öffentlicher) RSA-Schlüssel zum Fernzugang zum Techfaknetz.

einer vertrauenwürdigen Stelle hat, verbindet sich Ihr Browser wirklich. (Auf wikipedia: siehe TLS und certificate authority.)

Im Fall von `gpg` werden die Benutzer dezentral verwaltet. Mittels der Informationen dort kriegt man schon eine gewisse Idee, wer hinter dem Schlüsselpaar steckt (Uni-Emailadresse, Foto,...) Einfach mal Namen eingeben, z.B. hier: `http://keyserver.ubuntu.com/`

Bemerkung 9.2. Signaturen tauchen also alltäglich auf. Oft sieht man sie aber nicht. In Bild 9 sind zwei sichtbare Beispiele gezeigt. Mit dem Wissen dieser Vorlesung kann man auf ein paar Fakten schließen: auf der Restaurantrechnung links ist eine Signatur mit 128 Zeichen zu sehen (das - dient offenbar als Trennzeichen). Als Zeichen kommen vor: Großbuchstaben, Kleinbuchstaben, Ziffern sowie + und /. Also insgesamt $26 + 26 + 10 + 2 = 64 = 2^6$. Es liegt also nahe, dass jedes Zeichen sechs bit kodiert. Die Signatur hat also wohl $128 \cdot 6 = 768$ bit. Weitere Details könnte man jetzt z.B. auf wikipedia recherchieren.

Das Bild rechts zeigt meinen öffentlichen RSA-Schlüsel für den Fernzugang zum Techfaknetz. Das ist der, den ich auf `https://techfak.net/remote/shell/setup` hinterlegen musste. Der private Schlüssel (nicht im Bild) muss im Verzeichnis `.ssh` unter meinem Homeverzeichnis liegen. Wie auch immer das genau funktioniert: damit kann ich mich nun von außen auf einem Techfakrechner authentifizieren, und damit darf ich mich einloggen. Und der Schlüssel hat offenbar etwas weniger als $6 \cdot 774 = 4644$ bit (vor dem Schlüssel selbst steht ja noch ssh-rsa, danach noch irgendwas wie Nutzer@Rechner). Ein guter Tipp ist also wohl, dass das ein 4096-bit-RSA-Schlüssel ist.

9.4 Blinde Signaturen

Manchmal ist es wünschenswert, dass der Unterzeichner *nicht* den Text kennt, den er unterschreibt. Z.B.: “Ich bestätige, das Text m mir am 20.6.2020 vorgelegt wurde” (ohne dass er m kennt), wobei m etwa ein Gebot bei einer Ausschreibung ist. Oder “Ich bestätige, dass

dieser Geldschein echt ist", ohne die Seriennummer zu kennen (Wahrung der Anonymität bei elektronischem Geld). Im wahren Leben könnte Alice etwa auf das zu unterschreibende Dokument einen Bogen Durchschlagpapier (Kohlepapier) legen und die beiden Blätter zusammen in einen Umschlag packen. Bob unterschreibt auf dem Briefumschlag, ohne das Dokument zu sehen. Kryptographisch geht es so:

Blinde RSA-Signatur: Es seien p, q große Primzahlen, $N = pq$. Weiter sei e sei Bobs öffentlicher Schlüssel, d sein geheimer Schlüssel (also $\mathrm{ggT}(e, \varphi(N)) = 1$ und $ed \equiv 1 \bmod \varphi(N)$). Alice möchte, dass Bob die Nachricht m signiert, ohne m zu erfahren.

1. Alice wählt zufällig ein $r \in \{2, 3, \ldots, N-1\}$ mit $\mathrm{ggT}(r, N) = 1$.
2. Alice schickt $x = m \cdot r^e \bmod N$ an Bob (mal r^e entspricht dem „m in den Briefumschlag packen").
3. Bob schickt $x^d \bmod N$ an Alice. Alice gibt x^d bekannt.
4. Alice berechnet $x^d \cdot r^{-1} \bmod N$. (Das ist nun m^d, wenn alles korrekt lief. Dies entspricht dem „Dokument aus dem Briefumschlag holen". Beachte: Bob sieht weder m noch m^d).
5. Bei Bedarf kann Alice nun $(x^d \cdot r^{-1})^e \bmod N$ berechnen. Falls da m herauskommt, muss Bob unterschrieben haben, denn nur Bob kann d kennen.

Bemerkung 9.3. Das Verfahren ist effizient, und so korrekt und sicher wie RSA, vgl von zur Gathen. Zur Korrektheit überlegt man sich — neben den üblichen RSA-Überlegungen — noch Folgendes: Es ist $x^d \equiv (m \cdot r^e)^d \equiv m^d r^{ed} \equiv m^d \cdot r \bmod p$, also $x^d r^{-1} = m^d r r^{-1} = m^d$ (Schritt 4 ist also korrekt). Weiter ist

$$(x^d \cdot r^{-1})^e \equiv (m^d r r^{-1})^e \equiv (m^d)^e \equiv m^{de} \equiv m \mod p,$$

also ist Schritt 5 korrekt. Bob kennt nur $m \cdot r^e$, aber nicht r oder r^e, und somit nicht m. Alice kann m geheim halten, bis sie irgendwann (öffentlich) m bekannt gibt, und $(x^d \cdot r^{-1})^e \bmod p$ berechnet. Sie hat eine Zahl, die hoch $e \bmod p$ dieses m liefert. Eine solche Zahl kann — unter den üblichen Annahmen — nur Bob mittels seines geheimen Schlüssels erzeugt haben.

9.5 Elektronische Münzen

5. Juli

Wozu digitale Münzen? Es gibt doch Onlinebanking, Kreditkarten... Aber im Gegenteil zu Letzterem kann man den Geldfluss von Bargeld nicht zurückverfolgen! Das Ziel hier ist also Anonymität. Daher auch Bit-Coin: Digitale Münzen sollen anonym sein. Weiterhin sollen auch andere Eigenschaften von echtem Bargeld gelten. Wir wollen (zunächst mal):

- Zentrale Erzeugung: Nur die Bank kann Münzen herstellen (Wir stellen uns hier der Einfachheit halber nur eine einzelne Bank vor; es geht auch mit mehreren, dann wird es entsprechend komplizierter.)
- Echtheit: Alle Beteiligten sollen die Echtheit einer Münze verifizieren können.
- Eindeutigkeit: Niemand soll dieselbe Münze zweimal ausgeben können.

- Anonymität: Niemand darf erkennen können, wer mit einer Münze mal früher etwas bezahlt hat.

Wir schildern hier zunächst ein einfaches Protokoll. Im nächsten Abschnitt stellen wir Bitcoin vor.

Elektronische Münzen nach Chaum

(1985) Vorab: Es gibt nur eine Bank. Der Geldkreislauf ist immer nur Bank → Kunde → Händler → Bank. Für jeden Wert (z.B. 1, 2 ,5, 10, 20, 50, 100, 200, 500 Euro) hat die Bank einen privaten RSA-Schlüssel d_i (geheim) und einen öffentlichen RSA-Schlüssel e_i (allgemein bekannt, zusammen mit der zugehörigen Zahl $N = pq$). Echte Münzen liefern beim Entschlüsseln immer ein bestimmtes vereinbartes öffentlich bekanntes Muster (z.B. ein Hexadezimal-String, der sich 32-mal wiederholt, wie 4711 4711 $\cdots$ 4711, oder BD09 BD09 $\cdots$ BD09).

Erzeugen: Der Kunde wählt einen String m, der in das Muster passt, und lässt ihn von der Bank blind signieren: $s = f(d_i, m)$. Der Kunde berechnet aus s (z.B. $s \equiv m^{d_i} r \bmod N$) das $c = m^{d_i}$ (also z.B. $sr^{-1} = m^{d_i} r r^{-1} = m^{d_i}$). Die Bank sieht das m und das c nicht, vgl Kap. 9.4.

Bezahlen: Der Kunde gibt c an den Händler. Der Händler prüft, ob $c^{e_i} \bmod N$ das richtige Muster hat. Falls ja, akzeptiert er die Zahlung.

Einlösen: Der Händler gibt c an die Bank. Die prüft auch, ob $c^{e_i} \bmod p$ das richtige Muster hat. Falls ja, schreibt sie ihm den Betrag auf seinem Konto gut.

Die Anonymität ist hier gewahrt, da die Bank weder das m noch das c dem Kunden zuordnen kann: mit den Bezeichnungen aus Kap. 9.4 kennt die Bank beim Erzeugen jeweils nur $s = m^d \cdot r \bmod N$, aber nicht $c = m^d \bmod N$.

Ist das Stringmuster geeignet gewählt, so liefert ein zufällig gewähltes c das richtige Muster nur mit einer extrem winzigen Wahrscheinlichkeit. (Im Bsp oben: mit Wahrscheinlichkeit $\frac{1}{16^{28}} \approx 0,0000000000000000000000000000002$). Also kann nur die Bank zuverlässig echte Münzen erzeugen.

Jeder — insbesondere der Händler — kann die Echtheit prüfen. Das Problem beim Protokoll oben ist die Eindeutigkeit: Der Kunde könnte die Münze fast gleichzeitig bei vielen Händlern einlösen. Der Betrug fällt erst auf, wenn diese Händler alle dasselbe c zur Bank geben.

9.6 Blockchain und Bitcoin

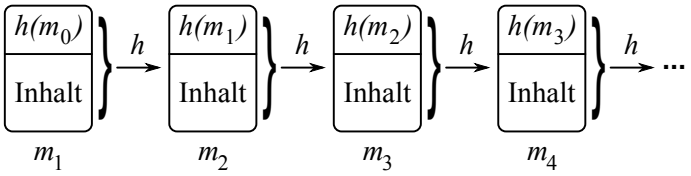
11. Juli

Das Problem der Eindeutigkeit wird offenbar von Bitcoin bzw. Blockchain gelöst. Dabei ist „Bitcoin“ (Zeichen: ₿) sozusagen der Markenname (wie „Tesa“ oder „Thomapyrin“) und „Blockchain“ ist die Bezeichnung des Objekts (wie „transparentes Klebeband“ oder „Paracetamol“). Es gibt andere Kryptowährungen, die die Blockchain-Idee nutzen (oder ganz anders funktionieren, oder einfach nur Betrug sind), und es gibt andere Anwendungen für Blockchain, die keine Kryptowährungen sind (sondern z.B. sowas wie Aktien).

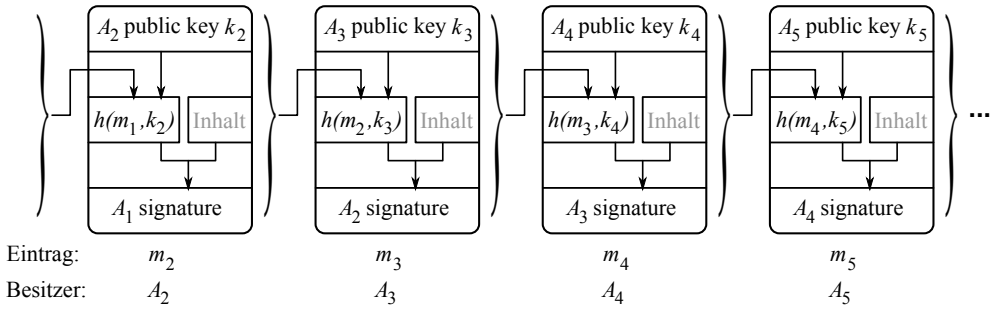
Die Idee wurde 2008 in einem Fachartikel in einem Kryptographieforum und auf der bitcoin-Webseite `https://bitcoin.org/bitcoin.pdf` veröffentlicht: Satoshi Nakamoto, „Bitcoin: a peer-to-peer...“ und 2009 von Satoshi Nakamoto implementiert und als open-source-code öffentlich verfügbar gemacht. Es weiß aber niemand, wer Satoshi Nakamoto ist (außer er/sie selbst?).

Die Ziele bei Bitcoin sind etwas anders als oben. Genauer gesagt will man Echheit, Anonymität und Eindeutigkeit (wie oben, aber Eindeutigkeit wurde nicht wirklich erreicht), aber man will gerade keine zentrale Bank. Vielmehr will man ein dezentrales, verteilt verwaltetes Kassenbuch, das alle jemals in Bitcoin getätigten Transaktionen enthält, in das (im Prinzip) jeder schreiben darf, und in dem kein Eintrag je gelöscht oder verändert werden kann.

Ein Baustein zum Erreichen dieser Ziele ist eine **Hashchain**. Diese dient als das Kassenbuch. Sie funktioniert so:



Dabei ist h eine Hashfunktion; genauer: h ist SHA-256. Der Hash $h(m_i)$ des Eintrags m_i wird Teil von m_{i+1}. Zusätzlich kann jeweils ein beliebiger Inhalt in m_i geschrieben werden (Z.B. A hat von B eine Bitcoin bekommen). Kenne ich den letzten Eintrag der Hashchain, z.B. m_{17}, so kann mir niemand einen falschen Inhalt von z.B. m_5 andrehen: Ändert er m_5, so ändert sich $h(m_5)$, also m_6, also $h(m_6)$ usw. Noch kann aber jeder einen neuen Eintrag m_{18} anlegen (z.B. “A hat 1000 bitcoin bekommen.”). Daher die folgende Verfeinerung:



Der Eintrag m_0 enthält den ersten Besitzer A_0 der Bitcoin (später mehr dazu). Bei einer Zahlung von A_i an A_{i+1} autorisiert A_i den neuen Block m_{i+1} mit seiner Signatur (private key). A_{i+1} schreibt seinen public key in den neuen Block m_{i+1}. (Der Inhalt könnte nun sein “A_{i+1} hat von A_i 2,6 bitcoin bekommen”. Im Originalartikel entspricht der Inhalt “dies ist 1 bitcoin”; wer die von wem bekam, geht aus dem Rest des Eintrags hervor.)

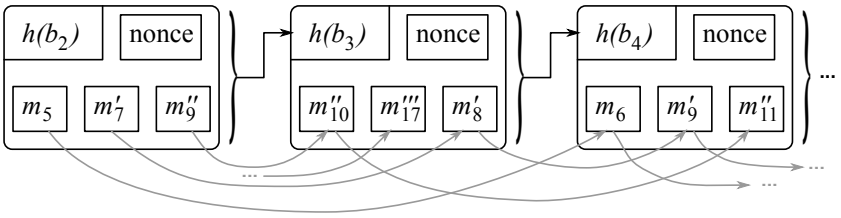
Jeder kann nun die Gültigkeit einer jeden Transaktion $A_i \to A_{i+1}$ prüfen, indem die Korrektheit der Signatur von m_{i+1} mit dem public key aus m_i geprüft wird. Nur der Besitzer von m_i kann m_{i+1} korrekt angelegt haben, denn nur er kann die korrekte Signatur erzeugt haben. Genau wie oben kann mir auch niemand einen falschen Inhalt m_i vorgaukeln, falls ich m_j kenne, für $i < j$. Das Verfahren erfüllt bereits fast alle der Anforderungen:

- Echtheit: zumindest falls m_0 als echt akzeptiert wird, werden nur echte bitcoins weitergegeben. Das kann jeder prüfen. Und nur der korrekte Besitzer kann sie (mittels seines

private keys) weitergeben.

- Anonymität: die public keys dürfen hier natürlich nicht ihren Besitzern zuzuordnen sein; man veröffentlich die nicht. Die private keys sind sowieso geheim.
- Jeder darf (seine) Transaktionen schreiben.
- Kein Eintrag kann gelöscht werden.

Es fehlt noch die Eindeutigkeit. Außerdem ist noch unklar, wie neue bitcoins (also oben das m_0) erzeugt werden. Beides könnte eine zentrale Stelle erledigen, aber das verstieße dann gegen die Forderung nach Anonymität. (Die zentrale Stelle *kann* den public keys die Besitzer zuordnen.) Der Trick ist folgender.



Angenommen, es gibt eine globale Hashchain, die laufend neue Blöcke b_i erzeugt. Verpacke darin unsere kleinen Hashchains, die die Transaktionen m_i enthalten (naiv: für jede bitcoin eine Transaktions-Hashchain). Diese globale Hashchain heißt **Blockchain**. Auch für solche Blockchains wurden verschiedene Lösungen vorgeschlagen. Die im Artikel von Satoshi Nakamoto ist diese:

- Die Blockchain läuft öffentlich und verteilt auf allen Rechnern (jeder, der die Bitcoin-Mining-Software nutzt, hat alle jemals gemachten Transaktionen auf seinem Rechner).
- Jeder kann Transaktionen vorschlagen zur Aufnahme in den neuen Block b_{i+_1}.
- Im Prinzip kann auch jeder neue Blöcke b_{i+1} der Blockchain hinzufügen.

Damit der letzte Punkt nicht zu Chaos und Betrug führt, gelten drei Bedingungen:

1. Einen neuen Block b_{i+1} aus b_i zu berechnen ist aufwendig.
2. Einen neuen Block b_{i+1} aus b_i zu berechnen wird mit Bitcoins bezahlt (**"Mining"**).
3. Bei Verzweigungen der Blockchain ist der längste Pfad der gültige.

Zu 1.: Jeder Block b_j enthält einen Zufallsstring r_j ("nonce"). Der hat keine Zusammenhang mit dem sonstigen Inhalt von b_j, kann also zunächst völlig beliebig sein. Aber: Ein neuer Block b_j ist nur gültig, falls $h(b_j)$ mit einer vorgegebenen Zahl von Nullen startet. (Wieder ist h hier SHA-256.) Man muss also viele r_j in b_j ausprobieren, bis $h(b_j)$ die Bedingung erfüllt. Die benötigte Anzahl der Nullen steigt mir den Jahren: sie dient als Stellschraube, um die Zahl der erzeugten Blöcke auf dem gewünschten Niveau zu halten. (Idealerweise soll alle 10 min ein neuer Block erzeugt werden. Am 1.3.2014 benötigte man dazu im Schnitt $16 \cdot 10^{18}$ Versuche, am 1.3.2015 waren es $200 \cdot 10^{18}$ Versuche.)

Zu 2.: Von 2016 bis 2020 gab es 12,5 ₿ pro erzeugtem Block. Das halbiert sich alle vier Jahre, genauer: alle 210 000 Blöcke. Ab dem 11. Mai 2020 sind es nur noch 6,25 ₿ pro Block. Bis zum Jahr 2140 sollten dann 21 Mio ₿ erzeugt worden sein. Das ist die Obergrenze, danach werden keine neuen bitcoins mehr erzeugt. Aber:

Es gibt auch Transaktionsgebühren: Wer eine Transaktion m_i einreicht, kann eine Gebühr anbieten, damit m_i in einen neuen Block b_j geschrieben wird. 2013 lag das im Schnitt bei 0,17 ₿ pro Block, 2020 wohl bei 0,4 ₿, also bringt das viel weniger ein als Mining.

Zu 3.: Das soll das Problem der Eindeutigkeit lösen: mehrfaches Ausgeben derselben Bitcoin soll unmöglich gemacht werden. Es ist klar, dass mehrfaches Ausgeben derselben bitcoin innerhalb eines Blocks geprüft und aufgedeckt werden kann. Ebenso kann das innerhalb eines Zweigs der Blöcke aufgedeckt werden. Und 3. sagt nun: nur ein Zweig ist gültig.

Im Artikel von Satoshi Nakamoto steht dazu: "Falls die Mehrzahl der Nutzer ehrlich ist..." ("The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.") Der beste Beweis, dass das klappt, ist, dass das seit acht Jahren zu funktionieren scheint.

Literatur

- Johannes Buchmann: Einführung in die Kryptographie. (Deckt alle hier behandelten Themen sehr gut ab.)
- Joachim von zur Gathen: CryptoSchool. (Sehr umfassend, und ein schönes Buch.)
- Bruce Schneier: Angewandte Kryptographie. (Der Klassiker, umfasst Theorie und Anwendung in epischer Breite und Tiefe.)
- Klaus Schmeh: Kryptografie: Verfahren, Protokolle, Infrastrukturen. (Umfassend für Theorie und Praxis, der Versuch, den Klassiker von Schneier zu ersetzen bzw. zu aktualisieren.)
- Douglas R. Stinson: Cryptography - Theory and Practice.
- Burnett, Paine: Kryptographie (schlägt den Bogen zur Anwendung: Implementierung, sichere Schlüssellängen, Rechenzeiten..., ist daher eher komplementär zur Vorlesung.)
- Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra. (Das ist ein Buch zu einem anderen Thema, es enthält reiches Material etwa zu Primzahltests und zu Faktorisierung, aber auch zu FFT, schneller Multiplikation u.a. Ist auch sehr umfassend, und auch ein schönes Buch.)