Cryptography & Coding Theory

Zusammenfassung v1.3

Kälin Thomas, MSE ICT
HS 08/09

1. ALGEBRAIC AND NUMBER THEORETIC BASISCS

1.1. GCD = GGT = Grösster gemeinsamer Teiler

```
Faktorisierung von 18:          2*3*3       = 18
Faktorisierung von 63:          3*3*7       = 63
Grösster gemeinsamer Teiler:    3*3         = 9
```

ACHTUNG: Der GGT(0, x) ist x.

1.2. Lineare Gleichungen

```
ax + by = n                 wenn a, b und n Integer
```

Die obige Gleichung kann genau dann gelöst werden, wenn n ein mehrfaches vom `ggT(a,b)` ist, auch wenn es sich beim ggT um den Wert 1 handeln sollte. Die Werte x und y sind mittels des erweiterten euklidschen Algorithmus bestimmbar (Kapitel 1.3).

```
Beispiel:     5x + 6y = 77              =>    ggT(5,6) = 1

              Es muss eine Lösung existieren, da 77 ein Mehrfaches von 1 ist

              5*x + 6*y = 77
              5*(-77) + 6*(+77) = 77
```

1.3. Euklidscher Algorithmus

Mittels des euklidschen Algorithmus kann der GGT von zwei positiven Ganzzahlen berechnet werden.

```
Algorithmus:                          Beispiel:
while b <> 0                          1) a = 105, b = 25
 r = a mod b                          2) a = 25, b = 5
 a = b                                3) a = 5, b = 0
 b = r                                4) 0 erreicht, GGT = 5!
end while;
```

Der **ERWEITERTE EUKLIDSCHE ALGORITHMUS** berechnet zusätzlich zum ggT auch die Ganzzahlen x und y einer linearen Gleichung `ax + by = ggt(a, b)`. Diese Gleichung ist immer dann lösbar, wenn a und m teilerfremd sind. In diesem Fall gibt es unendlich viele Lösungen.

```
(int, int) erweiterterEuklid(int a, int b) {
 if (b == 0) {
       return (signum(a), 0);
 } else {
       rest = a modulo b;
       (xt, yt) = erweiterterEuklid(b, rest);
       q = (a - rest) / b;
       x = yt;
       y = xt - q * yt;
       return (x, y);
 }
}
```

Beispiel erweiterter euklidscher Algorithmus: a = 100, b = 85

a	b	rest	q	x	y
100	85	15	1	**6**	**-7**
85	15	10	5	-1	6
15	10	5	1	1	-1
10	5	0	2	0	1
5	0				

Resultat: x*a + y*b = ggt(a,b) = **6***100 + **-7***85 = 5

1.4. Eulers Phi-Function

Wenn N eine natürliche Zahl ist, so berechnet $\varphi(n)$ die Anzahl der natürlichen Zahlen, welche kleiner oder gleich n und zugleich teilerfremd („coprime") sind.

Regel 1: **n ist eine Primzahl**
$\varphi(n) = n - 1$

Regel 2: **n ist aus den Primzahlen p*q zusammengesetzt und p != q**
$\varphi(n) = \varphi(p*q) = (p-1)*(q-1)$

Regel 3: **n ist das Quadrat der Primzahl p**
$\varphi(n) = \varphi(p^2) = p^2 - p$

Regel 4: **n ist ein exponentieller Wert der Primzahl p**
$\varphi(n) = \varphi(p^n) = p^n - p^{n-1}$

Regel 5: **n besteht aus mehreren Primzahlen (Faktorzerlegung)**
$\varphi(n) = \varphi(\text{Faktor1}) * \ldots * \varphi(\text{FaktorN})$

Regel 6: **n ist x*y, wobei mindestens ein Faktor keine Primzahl ist**
$\varphi(n) = \varphi(x*y) = n - (n/x) - (n/y) + (n / (x*y))$

Aus obigen Formeln resultiert, dass die Faktorisierung von n genau so kompliziert wie die Berechnung von $\varphi(p*q)$ [Regel 2] ist. Regel 6 ist mit Vorsicht zu geniessen, da diese von mir selber erarbeitet wurde. ☺ Es gibt auch ein allgemeines Verfahren zur Bestimmung der Zahlen:

Allgemeines Vorgehen zur Bestimmung von Eulers Phi-Function:
1) Alle Zahlen, welche kleiner als n sind, aufschreiben
2) Alle Zahlen, welche ein Mehrfaches von x oder y sind, streichen
3) Die übrig gebliebenen Zahlen zählen

Beispiel: **$\varphi(15) = \varphi(3*5)$**
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2 ~~3~~ 4 ~~5~~ ~~6~~ 7 8 ~~9~~ ~~10~~ 11 ~~12~~ 13 14 ~~15~~
1 2 4 7 8 11 13 14 = 8 Zahlen übrig
$\varphi(15) = 8$

Sind sowohl n als auch $\varphi(n)$ bekannt, so können mittels einer quadratischen Gleichung die ursprünglichen Primzahlen p und q berechnet werden.

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

Beispiel: **n = 667, $\varphi(n) = 616$**
$q^2 - (667 - 616 + 1) * q + 667 = 0$
$q = 29 \;||\; 33 \Rightarrow p = 29, q = 23$

1.5. Modulare Kongruenzen

Modulo-Berechnungen sind die Basis der meisten Public Key Verschlüsselungen. **KONGRUENZEN** sind die Basis der Modulo-Arithmetik.

Aussprache: „a und b sind kongruent Modulo m"
$a \equiv b \bmod m \Leftrightarrow a \bmod m = b \bmod m$
$\Leftrightarrow a = b + k*m$
$\Leftrightarrow m$ ist Teiler von $|b-a|$

Beispiel: **a = 15, b = 1, m = 7**
$15 \equiv 1 \bmod 7$ (7 teilt $|1-15| = 14$ ganzzahlig)

1.6. Modulo Inverses (a^{-1})

Eine Ganzzahl a ist dann eine „invertible modulo" zur natürlichen Zahl m, wenn die nachfolgende Kongruenz für eine passende ganze Zahl x erfüllt ist:

$ax \equiv 1 \bmod m$ => x mod m genau dann, wenn ggT(a,m) = 1
=> x mod m genau dann, wenn a und m teilerfremd

Beispiel: **a = 5, m = 12**
1) Bedingung: ggT(a,m) = ggt(5,12) = 1
2) Erweiterter Euklid mit (a,m) bestimmt uns x: a*x + m*y = 1
Der Wert für x lautet 5.
3) Wir können das Überprüfen: a*x ≡ 1 mod m
5*5 ≡ 1 mod 12

Wenn durch eine Primzahl Modulo gerechnet wird, so existiert für jedes Element ein modulares Inverses. Eine weitere Möglichkeit zur Bestimmung bietet Fermat's little Theorem (siehe Kapitel 1.7).

Modulo Inverses mit Fermat's little Theorem: $a^{-1} \equiv a^{p-2} \bmod p$

Beispiel: **a = 3, p = 5**
1) $a^{p-2} \bmod p = 3^{5-2} \bmod 5 = 3^3 \bmod 5 = 2$
2) $a^{-1} * a = 2*3 \equiv 1 \bmod 5$

Resultat: a^{p-2} ist in der tat das modulare Inverse a^{-1} von a.

1.7. Modulo Exponentiale (Fermat's little Theorem)

In der Modulo-Arithmetik mit Modulus m ist der Bereich der möglichen Zahlen auf [0..m-1] beschränkt. Daher resultieren auch die Resultate von Exponentialberechnungen mit Modulo in diesem Bereich.

Fermats kleines Theorem:	$a^p \equiv a \bmod p$		(p = Primzahl)
Falls a und p teilerfremd:	$a^{p-1} \equiv 1 \bmod p$		(Falls: ggT(a,p) = 1)
Verbesserung von Euler:	$a^{\varphi(m)} \equiv 1 \bmod m$		(Falls: ggT(a,m) = 1)

Beispiel: a = 3, p = 7

Da a und p teilerfremd sind, muss **$a^{p-1} \equiv 1 \bmod p$** sein.

$a^6 = 2187$ => 2187 mod 7 = 1

Fermat's kleines Theorem gibt übrigens **NUR EINE OBERE GRENZE FÜR DEN EXPONENT E**, so dass irgendwann gilt: a^e ≡ 1 mod p. Dieser Exponent kann aber auch früher erreicht werden, wie nachfolgendes Beispiel zeigt:

Beispiel: a = 2, p = 7

$a^3 = 8$ => 8 mod 7 = 1

Die Verbesserung von Euler soll auch noch anhand eines Beispiels gezeigt werden:

Beispiel: a = 2, m = 5 (Bedingung: ggt(a=2, m=5) = 1)
$\varphi(5) = 4$

$2^4 \equiv 1 \bmod 5$
$16 \equiv 1 \bmod 5$
$1 = 1$

1.8. Order of Elements

Die kleinst mögliche Zahl e, für die nachfolgende Gleichung gilt, wird „order of g mod m" genannt.

Aussprache: e ist „order of g mod m"
$g^e \equiv 1 \bmod m$ g = Ganzzahl, e und m sind natürliche Zahlen

Sei p eine Primzahl, so ist g ein **GENERATOR** oder **PRIMITIVES ELEMENT** zum Modulus p, wenn die „Order of g mod p" den Wert $p-1$ annimmt. Dazu müssen bei fortlaufender Potenzierung alle Werte im Bereich $\{0 \ldots p-1\}$ erzeugt werden.

Aussprache: g ist „generator mod p" oder g „ist primitives Element mod p"
$g^{p-1} \equiv 1 \bmod p$ g = Ganzzahl, p = Primzahl

Beispiel: m = p = 13, g = 2

$2^1 \bmod 13 = 2$
$2^2 \bmod 13 = 4$
…
$2^{12} \bmod 13 = 1 \quad \Rightarrow 2^{12} \equiv 1 \bmod m$

Erst bei e = 12 wird der Rest 1, somit ist die Zahl 12 die „order of 2 mod 13". Weiter ist g „generator mod 13", da erst bei p-1 = 13-1 = 12 der Rest 1 resultiert.

Ausserdem ist die Bedingung erfüllt, das durch die fort-Laufende Potenzierung alle Werte zwischen {0…12} erzeugt Werden.

Es kann gezeigt werden, dass für jede Primzahl passende Generatoren gefunden werden können. Die **ANZAHL DER GENERATOR** zu einem Modulus p kann aus Eulers Phi-Function bestimmt werden.

Anzahl Generatoren = $\varphi(p-1)$

Beispiel: Anzahl Generatoren zum Modulos p = 13
Anzahl Generatoren = $\varphi(p-1) = \varphi(13-1) = \varphi(12) = 4$
Tatsächlich lassen sich die 4 Generatoren {2, 6, 7, 11} finden.

1.9. Faster Exponentiation: Square and Multiply

Mittels „Square and Multiply" sollen aufwendige Exponentialberechnungen einfacher und effizienter gemacht werden.

Vorgehen: Square and Multiply
1) Binäre Darstellung des Exponents e
2) Iterativ die Quadrate der Basis $g^{2\hat{}i} \bmod m$ berechnen
3) Produkt der vorkommenden Quadrate bilden und mod m berechnen

Beispiel: $6^{73} \bmod 100 = ?$
1) $e = 73 = 64 + 8 + 1 = 2^{\underline{6}} + 2^{\underline{3}} + 2^0$
2) $g^{2\hat{}0} \bmod 100 \equiv 6^1 \bmod 100 = \underline{6}$
   $g^{2\hat{}1} \bmod 100 \equiv 6^2 \bmod 100 = \overline{36}$
   $g^{2\hat{}2} \bmod 100 \equiv 36^2 \bmod 100 = 96$ (Oder: -4)
   $g^{2\hat{}\underline{3}} \bmod 100 \equiv 96^2 \bmod 100 = \underline{16}$
   $g^{2\hat{}4} \bmod 100 \equiv 16^2 \bmod 100 = \overline{56}$
   $g^{2\hat{}5} \bmod 100 \equiv 56^2 \bmod 100 = 36$
   $g^{2\hat{}\underline{6}} \bmod 100 \equiv 36^2 \bmod 100 = \underline{96}$ (Oder: -4)
3) (96 * 16 * 6) mod 100 = 16

1.10. Faster Exponentiation: Modular Reduction

Eine zweite Möglichkeit, um Potenzen schneller zu berechnen, ist die so genannte „Modular Reduction". Dieses Verfahren kann bedeutend einfacher programmiert werden, als die Square and Multiply-Methode. Sie eignet sich deshalb prima für den Taschenrechner. ☺

Beispiel: $2^5 \bmod 5 = 2$

1) $2*2 \bmod 5 = 4$
2) $4*2 \bmod 5 = 3$
3) $3*2 \bmod 5 = 1$
4) $1*2 \bmod 5 = 2$

1.11. Chinese Remainder Theorem

Mit dem Chinese Remainder Theorem können Kongruenz-Systeme der Form $x \equiv a \bmod p$ gelöst werden.

Chinese Remainder Theorem: **$m_1 \ldots m_n$ = Paarweise teilerfremd**
$a_1 \ldots a_n$ = Ganzzahlen

$$x = \left(\sum_{i=1}^{n} a_i * y_i * M_i \right) \bmod m$$

$m = m_1 * m_2 * \ldots * m_n$ $\quad$ $M_i = m / m_i$ $\quad$ $y_i = M_i^{-1} \bmod m_i$

Beispiel: $x \equiv 5 \bmod 7$ $\quad$ $x \equiv 3 \bmod 11$ $\quad$ $x \equiv 10 \bmod 13$

1) $n = 3$, $a_1 = 5$, $a_2 = 3$, $a_3 = 10$, $m_1 = 7$, $m_2 = 11$, $m_3 = 13$
2) $m = m_1 * m_2 * m_3 = 7 * 11 * 13 = 1001$
3) $M_1 = m / m_1 = 1001 / 7 = 143$
   $M_2 = m / m_2 = 1001 / 11 = 91$
   $M_3 = m / m_3 = 1001 / 13 = 77$
4) Bestimmen der Modular-Inverses mittels erweiterten Euklidschen Algorit.:
   $y_1 = M_1^{-1} \bmod m_1 = 143^{-1} \bmod 7 = -2 \bmod 7 = 5$
   $y_2 = M_2^{-1} \bmod m_2 = 91^{-1} \bmod 11 = 4 \bmod 11 = 4$
   $y_3 = M_3^{-1} \bmod m_3 = 77^{-1} \bmod 13 = -1 \bmod 13 = 12$
5) $x = (a_1*y_1*M_1 + a_2*y_2*M_2 + a_3*y_3*M_3) \bmod m$
   $x = (5*5*143 + 3*4*91 + 10*12*77) \bmod 1001$
   $x = 894 \bmod 1001$

1.12. PI-Function

Die π-Funktion ist so definiert, dass sie für einen Input die Anzahl der Primzahlen ausgibt, welche nicht grösser als der Input sind.

Beispiel: n = 13

1) Primzahlen <= 13 sind: {2, 3, 5, 7, 11, 13}
2) $\pi(n) = \pi(13) = 6$

Ein bekanntes Theorem sagt: $x > 17 \Rightarrow \pi(x) > \frac{x}{\ln(x)}$.

Beispiel: Probedivision für RSA mit 512 Bit-Schlüssel

Die Anzahl der zu testenden Primzahlen bei der Probedivision (siehe Kapitel 3.1) beträgt für einen 512-Bit-Schlüssel somit im Minimum:

$$\frac{\sqrt{n}}{\ln(\sqrt{n})} = \frac{\sqrt{2^{512}}}{\ln(\sqrt{2^{512}})} = \frac{2^{256}}{\ln(2^{256})} > 2^{248}$$

2. POLYNOMIALS UND FINITE FIELDS

2.1. Finite Fields (Galois Field)

Die bekanntesten unendlichen Körper (=Field) sind die rationalen und realen Zahlen. Für unsere Fälle konzentrieren wir uns auf endliche Körper, auch Galois-Fields (GF) genannt. Wird ein Feld GF(p) mit einer Primzahl gebildet, so entsprechen die Berechnungen der Modulo-Arithmetik.

Beispiel: GF(5)

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	**1**	2	3	4
2	0	2	4	**1**	3
3	0	3	**1**	4	2
4	0	4	3	2	**1**

Wenn x*y mod p = 1, dann sind x und y inverse Elemente zueinander.

2.2. Polynomials

Polynomials werden zur Konstruktion von grösseren Feldern aus einem Ausgangsfeld benötigt. Der **GRAD** eines Polynoms wird durch seine höchste Potenz n bestimmt und mit `deg[p(x)]` bezeichnet. Die Koeffizienten (an … a0) können Werte aus dem zugehörigen GF annehmen.

Form eines Polynoms: $p(x) = a_n*x^n + a_{n-1}*x^{n-1} + \ldots + a_1*x + a_0$

Auf Polynome können normale Rechenoperationen angewandet werden, wie folgendes Beispiel zeigt:

Beispiel: **GF(2), p(x) = x+1, q(x) = x^3+x+1**

$p(x)+q(x) = x^3$ *//1+1 mod 2 = 0*

$p(x)*q(x) = x^4 + x^3 + x^2 + 1$ *//1+1 mod 2 = 0*

Beispiel: **GF(3), p(x) = $2x^2$ + x + 1, q(x) = 2x + 1**

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

*	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

$p(x)*q(x) = (2x^2 + x + 1) * (2x + 1)$

$= x^3 + 2x^2 + 2x^2 + x + 2x + 1$

$= x^3 + x^2 + 1$

2.2.1. Irreducible Polynoms

Was die Primzahlen für die normalen Zahlen bedeutet sind die **IRREDUCIBLE POLYNOMS** (nicht-reduzierbare Polynome) für endliche Felder. Diese lassen sich nicht weiter teilen.

Beispiel: **n = Grad = 2, F = GF(2)**

Das einzige irreduzible Polynom vom Grad 2 über GF(2) ist: **$m(x)=x^2+x+1$**

Die Überprüfung, ob ein Polynom über ein Feld wirklich irreduzibel ist kann auf zwei Methoden erfolgen:

- Probedivision mit allen irreduziblen Polynomen bis zum halben Grad des Polynoms, Rest $\neq 0$
- (!) Funktioniert nicht immer: Überprüfung der Eigenschaft $\forall a \in Z_n : f(a) \bmod n \neq 0$

Beispiel: **Ist $p(x) = x^2+1$ irreduzibel über F = GF(3)?**

$p(0) = (0^2+1) \bmod 3 = 1 \bmod 3 = 1$

$p(1) = (1^2+1) \bmod 3 = 2 \bmod 3 = 2$

$p(2) = (2^2+1) \bmod 3 = 5 \bmod 3 = 2$

p(x) ist irreduzibel über GF(3), da der Rest nirgends 0 ergibt.

2.2.2. Liste von irreduziblen Polynomen über GF(2)

Grad 1:	x	**Grad 4:**	$x^4 + x + 1$
	$x + 1$	**Grad 5:**	$x^5 + x^2 + 1$
Grad 2:	$x^2 + x + 1$	**Grad 8:**	$x^8 + x^4 + x^3 + x + 1$
Grad 3:	$x^3 + x + 1$		

2.2.3. Grössere Felder erzeugen

Zur Konstruktion von grösseren Feldern aus bestehenden Feldern werden nicht-reduzierbare Polynome benötigt.

Beispiel: n = 2, F = GF(2), E = GF(2^2), m(x) = x^2+x+1,

1) Die Elemente des erweiterten Feldes bestehen aus allen Kombinationen bis zum um eins reduzierten Grad des irreduziblen Polynoms. Da der Grad des Polynoms 2 ist, müssen alle Polynom-Kombinationen bis zum Grad 1 erzeugt werden. Die Koeffizienten a des Polynoms können wegen GF(2) nur die Werte 0 und 1 annehmen.

   => {0, 1, x, x+1}

2) Anschliessend werden die Additions- und Multiplikationstabellen des neuen Feldes berechnet. Enstehen dabei Resultate, welche einen höheren Grad als das irreduzible Polynom aufweisen, so müssen diese durch Polynomdivision reduziert werden.

+	0	1	x	1+x
0	0	1	x	1+x
1	1	0	1+x	x
x	x	1+x	0	1
1+x	1+x	x	1	0

·	0	1	x	1+x
0	0	0	0	0
1	0	1	x	1+x
x	0	x	1+x	1
1+x	0	1+x	1	x

Beispiele Additionstabelle:

(1) + (1+x) = 2+x = x (1+x) + (1+x) = 2+2x = 0

Beispiele Multiplikationstabelle:

Da x*x = x^2 den Grad 2 aufweist, muss eine Polynomdivision durchgeführt werden:

x^2 / m(x)
x^2 / x^2 + x + 1 = 1
x^2 − x − 1

− x − 1 äquivalent zu: **x + 1**

2.2.4. Generatoren

Wie bei den normalen Ganzzahlen gibt es auch in endlichen Feldern so genannte Generatoren (primitive Elemente). Diese erzeugen bei fortlaufender Potenzierung jegliche Elemente.

Beispiel: E = GF(2^3), F = GF(2), m(x) = x^3+x+1, a = Generator = x

$a^1 = x$			*//Zeilenweise x multiplizieren*
$a^2 = x^2$			$= x^2$
$a^3 = x^3$	$= x^3$	mod m(x)	$= x + 1$
$a^4 = x^2 + x$			$= x^2 + x$
$a^5 = x^3 + x^2$	$= x^3 + x^2$	mod m(x)	$= x^2 + x + 1$
$a^6 = x^3 + x^2 + x$	$= x^3 + x^2 + x$	mod m(x)	$= x^2 + 1$
$a^7 = x^3 + x^2$	$= x^3 + x^2$	mod m(x)	$= 1$

3. TESTEN VON PRIMZAHLEN

3.1. Probedivision

Das einfachste Verfahren zur Überprüfung, ob eine Zahl prim ist, ist die Probedivision. Dabei wird die Zahl durch alle Primzahlen beginnend mit der 2 dividiert bis sich eine Primzahl als deren Teiler erweist oder bis der Probedivisor grösser als $\sqrt{n}$ ist (falls `n=a*b`, so muss entweder a oder b kleiner $\sqrt{n}$ sein). Das Verfahren ist zwar sehr aufwändig, eignet sich allerdings sehr gut zur Bestimmung kleiner Primfaktoren.

```
Beispiel:    n = 191
1) Sqrt(n) = Sqrt(191) = 13.82       => alle Primzahlen <= 13 testen
2) Primzahlen <= 13 sind 2, 3, 5, 7, 11 und 13.
3) 191 ist durch keine der Zahlen teilbar, somit muss n eine Primzahl sein.
```

Für eine Abschätzung der benötigten Divisionen sei hier auf das Kapitel 1.12 hingewiesen.

3.2. Miller-Rabin Test

Beim Miller-Rabin Primzahlentest handelt es sich um einen effizienten „Monte Carlo"-Algorithmus. Zum Verständnis sind zuerst einige Definitionen nötig:

- **DECISION-PROBLEM:** Ein Problem, dessen Antwort „ja" oder „nein" ist.
- **PROBALISTIC ALGORITHMUS:** ein Algorithmus, der zur Lösung der Aufgabe Zufallszahlen verwendet.
- **YES-BASED:** ein probalistischer Algorithmus für ein Entscheidungs-Problem. Die „ja"-Antwort ist dabei immer korrekt, die „nein"-Antwort kann mit einer bestimmten Wahrscheinlichkeit falsch sein.

Beim MR-Test handelt es sich um ein „yes-based" Monte Carlo Verfahren zur Überprüfung grosser Zahlen auf ihre Prim-Eigenschaft. Beantwortet der Algorithmus die Frage „Ist n Composite?" mit ja, so ist diese Aussage garantiert korrekt. Die Antwort nein („n ist prim"), kann jedoch falsch sein.

```
Algorithmus: Miller-Rabin Test
1) Eingabe n als n-1 = 2^k*m schreiben. Der Wert m muss ungerade sein.
   Einfache Methode: solange durch 2 dividieren, bis Rest ungerade ist.
2) Zufällige Zahl a wählen, wobei: 1 <= a <= n-1
3) Berechnen von b = a^m mod n
4) Wenn b ≡ 1 mod n gilt, dann stoppen wir. N ist vermutlich eine Primzahl.
5) Für alle i in {0 … k-1}:
 5a) Gilt b ≡ -1 mod n, dann stoppen. N ist vermutlich eine Primzahl.
 5b) Andernfalls berechnen von b ≡ b^2 mod n und fortfahren bei 5.
6) Erzeugte keine Iteration einen Abbruch, so ist n keine Primzahl!
```

```
Beispiel: Miller-Rabin Test für n = 561
1) n-1 = (2^k)*m     =>  560 = 2*280 = 2^2*140 = 2^3*70 = 2^4*35
                     =>  k = 4, m = 35
2) Zufällige Zahl: a = 2
3) b = a^m mod n = 2^35 mod 561 = 263
4) 263 ≡ 1 mod 561 gilt nicht, somit wird nicht gestoppt.
5) i = 0:    b = b^2 mod n = 263^2 mod 561 = 166
   i = 1:    b = b^2 mod n = 166^2 mod 561 =  67
   i = 2:    b = b^2 mod n =  67^2 mod 561 =   1
   i = 3:    b = b^2 mod n =   1^2 mod 561 =   1

   b wird nie (-1 mod n), somit wird nicht gestoppt. Weiter bei 6.
6) Wenn wir hier sind, ist n sicher keine Primzahl. Ende des Algorithmus.
   Anmerkung: n = 561 = 3*11*17
```

Es kann gezeigt werden, dass der MR-Test den Output „n ist Composite" nicht produzieren kann, wenn n eigentlich eine Primzahl ist. Hingegen existiert eine Restwahrscheinlichkeit von maximal 25%, dass der Test eine Zahl fälschlicherweise als Primzahl definiert, obwohl diese eigentlich ein Composite wäre. Aus diesem Grund wird der Test für unterschiedliche Zufallszahlen `a` mehrmals wiederholt.

3.3. Faktorisierungsmethode von Fermat

Dieser Algorithmus funktioniert nur für ungerade Zahlen. Das gibt auch Sinn, wer möchte schon eine gerade Zahl auf die Primzahlen-Eigenschaft testen? ☺ Am besten wird das Verfahren an Hand eines Beispiels demonstriert:

Beispiel: n = 1729

1) x = ceil(Wurzel(1729)) = ceil(41.6) = 42
2) $r = x^2 - n = 42^2 - 1729 = 35$ (Keine Quadratzahl, x++)
   $r = x^2 - n = 43^2 - 1729 = 120$ (Keine Quadratzahl, x++)
   …
   $r = x^2 - n = 55^2 - 1729 = 1296$ (Quadratzahl, Abbruch)
3) f1 = x + Wurzel(r) = 55 + 36 = 91
   f2 = x - Wurzel(r) = 55 - 36 = 19

4) Da f1=19 und f2=91 Primzahlen sind, wird hier abgebrochen. Falls dies nicht der Fall sein sollte, so würde der Algorithmus mit der Ausgangszahl von vorne beginnen.

4. PUBLIC KEY KRYPTOGRAPHIE

In der klassischen, symmetrischen Kryptographie müssen n*(n-1)/2 Schlüssel auf einem sicheren Kanal ausgetauscht werden. PK-System versuchen das durch die Verwendung von so genannten Public- und Private-Keys zu vermeiden, wobei der Public-Key selbstverständlich keinen Rückschluss auf den Private-Key erlauben darf.

4.1. Diffie Hellmann

DH erlaubt die Erzeugung eines symmetrischen Schlüssels über einen unsicheren Kanal, wird aber selber nicht zur Verschlüsselung eingesetzt. Es basiert auf Einweg-Funktionen, heisst also, dass die Berechnung in eine Richtung sehr einfach, in die andere sehr schwierig ist

Vorgehen beim Schlüsselaustausch
0) Wählen einer grossen Primzahl p und einem Generator α mod p.
1) A wählt eine grosse Zahl x und berechnet **$A = \alpha^x \bmod p$**
2) B wählt eine grosse Zahl y und berechnet **$B = \alpha^y \bmod p$**
3) A berechnet Schlüssel mit **$k = B^x \bmod p$**
   B berechnet Schlüssel mit **$k = A^y \bmod p$**

Beispiel: $\alpha = 3$, $p = 17$
1) A wählt $x = 7$ und berechnet $A = \alpha^x \bmod p = 3^7 \bmod 17 = 11$
2) B wählt $y = 4$ und berechnet $B = \alpha^y \bmod p = 3^4 \bmod 17 = 13$
3) A berechnet Schlüssel mit $k = B^x \bmod p = 13^7 \bmod 17 = \underline{4}$
   B berechnet Schlüssel mit $k = A^y \bmod p = 11^4 \bmod 17 = \underline{4}$

Obwohl sowohl α, p, A und B offen übertragen werden kann ein Angreifer, wegen des diskreten Logarithmus-Problem, keinen Rückschluss auf den geheimen Schlüssel machen. Denkbar sind allerdings **MAN-IN-THE-MIDDLE** Attacken: in diesen handelt ein Angreifer sowohl mit A als auch B einen eigenen Schlüssel aus. Dies kann nur durch eine vorgängige Authentifizierung der Benutzer vermieden werden.

Diskretes Logarithmus-Problem: Umkehrung von $A = \alpha^x \bmod p \Rightarrow$ **$x = \log_\alpha(A)$**

4.2. RSA

Mit RSA können Nachrichten ohne vorherigen Tausch von symmetrischen Schlüsseln verschlüsselt übertragen werden. Das System basiert auf der Annahme, dass es schwierig ist, das Produkt von zwei Primzahlen zu faktorisieren. Es ist nicht geeignet zur Verschlüsselung langer Texte.

Interessant ist, dass die **KOMPLEXITÄT** mit steigender Bitzahl immer weniger stark zunimmt. Ist beim Schritt von 512 auf 768 bit noch eine Steigerung um den Faktor 10'000 möglich, so ist die Zunahme beim Wechsel von 768 auf 1024 bit nur noch 1'000.

Public Elemente: Exponent e, Modulus n
Private Elemente: Primzahl p, Primzahl q, Exponent d
Beziehungen: $n = p*q$
$e*d = 1 \bmod \phi(n)$ *//d = ModInverse von e!*
$0 <= m < n$

Vorgänge beim Verschlüsseln:
1) A verschlüsselt Nachricht m mit: **$c = m^e \bmod n$**
2) B entschlüsselt Cypher c mit: **$m = c^d \bmod n$**

Vorgänge beim Signieren:
1) A signiert Nachricht m mit: **$s = m^d \bmod n$**
2) B überprüft Signatur s mit: **$m = s^e \bmod n$**

Üblicherweise ist die Nachricht m länger als der Modulus n. In diesem Fall wird die Signatur über einen Fingerprint (Hash-Funktion) anstatt das Dokument selber gebildet:

1) A signiert Nachricht m mit: **$s = h(m)^d \bmod n$**
2) B überprüft Signatur mit: **$h(m) = s^e \bmod n$**

Beispiel: p = 11, q = 23, m = 165
1) Öffentliche Modulus n = p*q = 11 * 23 = 253
2) $\varphi(n) = \varphi(253) = (p-1)*(q-1) = 10 * 22 = 220$
3) Kleinste Primzahl teilerfremd zu $\varphi(n)$ ist e = 3.
4) Da $e*d \equiv 1 \bmod \varphi(n)$ folgt d = 147. Das kann mit dem erweiterten Euklidschen Algorithmus für EEA(e, $\varphi(n)$) berechnet werden, da d ein „Modulo Inverse" zu e ist.
5) A verschlüsselt Nachricht mit $c = m^e \bmod n = 165^3 \bmod 253 = 110$
6) B entschlüsselt Nachricht mit $m = c^d \bmod n = 110^{147} \bmod 253 = \underline{165}$

4.2.1. Anzahl öffentliche Exponenten e bestimmen

Bestimmung aller möglichen Exponenten e zum Modulus n:
1) phi_N = $\varphi(n)$
2) Es muss gelten: ggT(e, phi_N) = 1
3) Daraus folgt, dass die Anzahl der möglichen Exponententen e gerade durch die eulersche Funktion auf phi_N gegeben ist.
4) Anzahl Exponenten e = **$\varphi(\varphi(n))$**

Beispiel: q = 3, p = 5
1) n = q*p = 3*5 = 15
2) phi_N = $\varphi(n) = (p-1)*(q-1) = 2*4 = 8$
3) Anzahl Exponenten e = $\varphi(\text{phi\_N}) = \varphi(8) = 4$
   Es handelt sich um die Exponenten e = {1, 3, 5, 7}.

4.2.2. Common Modulus Attack

Wird dieselbe Nachricht mit zwei Schlüsseln verschlüsselt, welche denselben Modulus n aufweisen, so ist es einem Angreifer möglich, die ursprüngliche Textnachricht zu berechnen. Es sollte aus diesem Grund auf keinen Fall immer derselbe Modulus n verwendet werden.

Common Modulus Attack: pk1 = (n,e), pk2 = (n,f)
1) Es gilt ggT(e,f) = 1, woraus über den erweiterten euklidschen Algorithmus zwei Zahlen x und y berechnet werden können, so dass gilt:

   $x*e + y*f = 1$

2) Mittels der Zahlen x und y kann nun die ursprüngliche Nachricht berechnet werden:

   $m = c1^x * c2^y \bmod n = m^{(x*e+y*f)}$

3) Da eine Zahl (x oder y) den Wert −1 ergeben wird, so muss für diese Zahl das Modulo Inverse mittels erweitertem euklidschen Algorithmus berechnet werden.

   Modulo Inverse von y: ggT(y, m) = 1, somit: r*y + s*m = 1

Beispiel: pk1 = (n=493,e=3), pk2 = (n=493,f=5), ce=293, cf=421
1) Euklid: $x*3 + y*5 = 1 \Rightarrow x = 2, y = -1$
2) Klartext: $m = 293^2 * 421^{-1} \bmod 493 = 67 * 89 \bmod 493 = 47$
3) Modulo Inverse: 421^{-1} mittels $r*421 + s*493 = 1$ bestimmen

5. SYMMETRISCHE KRYPTOGRAPHIE

In der symmetrischen Kryptographie wird für die Verschlüsselung und Entschlüsselung derselbe Schlüssel verwendet. Dabei sollte die Sicherheit einer Verschlüsselung grundsätzlich nur vom Schlüssel, nicht aber vom Algorithmus abhängen. Symmetrische Verschlüsselungen sind immer injektiv, also eindeutig. Das macht auch Sinn, da andernfalls aus einem Cipher kein eindeutiger Text ermittelt werden könnte.

- **CIPHERTEXT-ONLY:** Ein Angreifer kennt nur verschlüsselte Texte
- **KNOWN PLAINTEXT:** Ein Angreifer besitzt einige (Klartext,Cipher)-Paare
- **CHOSEN PLAINTEXT:** Ein Angreifer kann beliebige (Klartext,Cipher)-Paare produzieren

5.1. Block Ciphers

Diese Verschlüsselungen arbeiten mit festen Blockgrössen. Der Ursprungstext wird also in Blöcke aufgeteilt und jeder einzeln verarbeitet. Die Verschlüsselung kann als Permutation des Klartextes verstanden werden.

5.2. Historisch: Vigenere, Hill und Permutation Cipher

Diese Verschlüsselungen sind nicht länger in Verwendung. Sie zeigen jedoch, dass Linearitäten in sicheren Verschlüsselungssystemen vermieden werden sollten.

$f : Z_m^n \to Z_m^l$

m = Anzahl der Zeichen im Alphabet (A..Z = 26)
n = Länge der Zeichenkette unverschlüsselt
l = Länge der Zeichenkette verschlüsselt

Eine Funktion f ist dann linear, wenn für alle Vektoren $v, w \in Z_m^n$ gilt: $f(\alpha v + \beta w) = \alpha f(v) + \beta f(w)$.

Weiter ist diese **AFFINE LINEAR**, wenn gilt: $f(v) - f(0)$.

5.2.1. Vigenere Cipher

Die Vigenère-Verschlüsselung galt lange als sicherer Chiffrieralgorithmus. Ein Schlüsselwort bestimmt, wie viele Alphabete genutzt werden. Die Alphabete leiten sich aus der Caesar-Substitution ab. Es wird mit affine linearen Maps gearbeitet.

$$E_{\mathbf{z}} : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \ \mathbf{v} \mapsto \mathbf{v} + \mathbf{z} \mod m.$$

$$D_{\mathbf{z}} : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \ \mathbf{v} \mapsto \mathbf{v} - \mathbf{z} \mod m.$$

Unvollständiges Beispiel für Vigenere

	Text																									
	A	B	C	D	**E**	F	**G**	**H**	**I**	J	K	L	**M**	**N**	O	P	Q	R	**S**	T	U	V	W	X	Y	Z
1	A	B	C	D	E	F	**G**	H	**I**	J	K	L	M	N	O	P	Q	R	**S**	T	U	V	W	X	Y	Z
2	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
3	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
4	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
5	E	F	G	H	I	J	K	**L**	M	N	O	P	Q	**R**	S	T	U	V	W	X	Y	Z	A	B	C	D
6	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
7	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
8	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G

```
Text:       geheimnis
Schlüssel:  AKEYAKEYA
Chiffrat:   GOLCIWRGS
```

5.2.2. Hill Cipher

Der Hill-Cipher wurde 1929 eingeführt. Er verwendet für jede Zeichen von A – Z einen Zahlenwert, üblicherweise beginnend bei A = 0. Anschliessend werden aus dem Input Vektoren der Länge n gebildet, welche mit einer n*n-Schlüsselmatrix multipliziert und anschliessend Modulo 26 gerechnet wird.

$$E_M : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \ \mathbf{v} \mapsto M\mathbf{v} \mod m.$$

Beispiel: Verschlüsselung von w = ACT mit M = GYBNQKURP

$$C = (M * w) \bmod 26$$

$$C = \begin{pmatrix} G & Y & B \\ N & Q & K \\ U & R & P \end{pmatrix} * \begin{pmatrix} A \\ C \\ T \end{pmatrix} \equiv \begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} * \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} \equiv \begin{pmatrix} 67 \\ 222 \\ 319 \end{pmatrix} \bmod 26 = \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} = \begin{pmatrix} P \\ O \\ H \end{pmatrix}$$

Zur Entschlüsselung wird die Schlüssel-Matrix invertiert und diese mit dem Cipher multipliziert. Zur Bestimmung der inversen Matrix stehen zwei Möglichkeiten zur Verfügung: der Gauss-Jordan-Algorithmus sowie die Adjunkte. Letztere Methode gibt fixe Muster für 2x2 und 3x3-Matrizzen vor. Dabei ist zu erwähnen, dass nicht eine beliebige Matrix verwendet werden darf. Vielmehr darf die Diskriminante der Matrix nicht 0 und auch kein gemeinsamer Teiler der Zahl 26 = 2*13 sein.

Beispiel: Entschlüsselung von C = POH mit M^{-1} = IFKVIVVMI

$$w = (M^{-1} * C) \bmod 26$$

$$w = \begin{pmatrix} I & F & K \\ V & I & V \\ V & M & I \end{pmatrix} * \begin{pmatrix} P \\ O \\ H \end{pmatrix} \equiv \begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} * \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} \equiv \begin{pmatrix} 260 \\ 574 \\ 539 \end{pmatrix} \bmod 26 = \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} = \begin{pmatrix} A \\ C \\ T \end{pmatrix}$$

5.2.3. Permutation Cipher

Diese Verschlüsselung ist ein Spezialfall des Hill-Ciphers. Er verwendet im Gegensatz zu den anderen Algorithmen Permutationen auf Zeichen- anstatt Stringebene.

$$Bit - Permutationen = n! \qquad String - Permutationen = 2^n!$$

5.3. Kryptoanalyse der historischen Cipher

Es kann gezeigt werden, dass jegliche historischen Verschlüsselungen wegen ihrer Linearität mittels einer Known-Plaintext Attacke geknackt werden können. Bei diesen Attacken wird darauf gezielt, die verwendete Schlüsselmatrix M aus `n+1` (Plaintext, Cipher)-Paaren zu bestimmen.

Beispiel: Hill Cipher, n = 2, w = FRIDAY, c = PQCFKU

$$w1 = \begin{pmatrix} f \\ r \end{pmatrix} = \begin{pmatrix} 5 \\ 17 \end{pmatrix} \qquad w2 = \begin{pmatrix} i \\ d \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \end{pmatrix} \qquad c1 = \begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} 15 \\ 16 \end{pmatrix} \qquad c2 = \begin{pmatrix} c \\ f \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

$$W = \begin{pmatrix} f & i \\ r & d \end{pmatrix} = \begin{pmatrix} 5 & 8 \\ 17 & 3 \end{pmatrix} \qquad C = \begin{pmatrix} p & c \\ q & f \end{pmatrix} = \begin{pmatrix} 15 & 2 \\ 16 & 5 \end{pmatrix}$$

$$W^{-1} = \frac{1}{\det(W)} * \begin{pmatrix} w22 & -w12 \\ -w21 & w11 \end{pmatrix} = \frac{1}{(5*3-8*17)} * \begin{pmatrix} 3 & -8 \\ -17 & 5 \end{pmatrix} = \frac{1}{9} * \begin{pmatrix} 3 & -8 \\ -17 & 5 \end{pmatrix} = \begin{pmatrix} 9 & 2 \\ 1 & 15 \end{pmatrix}$$

$$M = C * W^{-1} \equiv \begin{pmatrix} 15 & 2 \\ 16 & 5 \end{pmatrix} * \begin{pmatrix} 9 & 2 \\ 1 & 15 \end{pmatrix} \equiv \begin{pmatrix} 137 & 60 \\ 149 & 107 \end{pmatrix} \bmod 26 = \underline{\underline{\begin{pmatrix} 7 & 8 \\ 19 & 3 \end{pmatrix}}}$$

5.4. Sichere Block-Cipher

Shannon hat zwei Prinzipien eingeführt, welche für sichere Blockcipher erfüllt sein müssen:

- **CONFUSION:** Verschlüsselungen sollen komplex und die resultierenden Gleichungen nicht-linear sein.
- **DIFFUSION:** Bei der Änderung eines einzelnen Bits sollten durchschnittlich 50% der verschlüsselten Bits ändern. Es sollte keine Aussage darüber möglich sein, welche Bits das wechseln.

Üblicherweise werden diese beiden Attribute durch „Product Cipher" erreicht. Diese verwenden mehrere Runden, welche als einzelnes unsicher sind und erst durch mehrmaliges Anwenden eine gute Sicherheit erreichen. In jeder Runde wird aus dem privaten Schlüssel ein „Rundenschlüssel" abgeleitet.

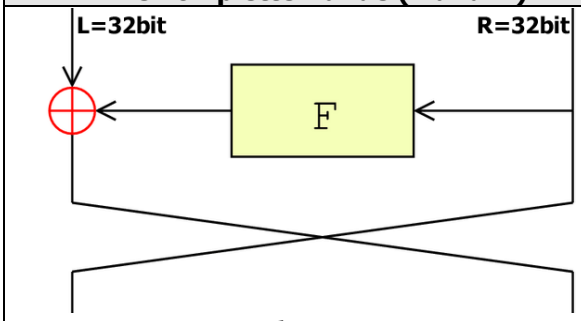
5.5. Modes von Block-Ciphern

- **ELECTRONIC CODE BOOK MODE (ECB):** der Klartext wird auf verschieden Blöcke fixer Länge aufgeteilt. Anschliessend wird jeder dieser Blöcke separat verschlüsselt. Problematisch: ein Klartext resultiert immer in demselben Ciphertext und die Blockgrenzen sind für einen Angreifer klar erkenn- und somit auch manipulierbar.
- **CIPHER BLOCK CHANING MODE (CBC):** Auch hier wird der Klartext auf Blöcke fixer Länge aufgeteilt, allerdings basiert jeder Block auf den Resultaten des vorherigen Blocks (XOR von Input `n` mit Output `n-1`). Das bedingt, dass ein Startblock („Initialisierungsvektor") benötigt wird. Vorteil: Änderungen an einem Block wirken auf den nachfolgenden Block.
- **OUTPUT FEEDBACK MODE (OFB):** Verwandelt einen Block-Cipher in einen Stream-Cipher. Wie bei CBC hat der berechnet Keystream Einfluss auf die nachfolgenden XOR-Berechnungen. Der verwendete Schlüssel muss periodisch gewechselt werden, um gegen Attacken sicher zu sein.

5.6. DES

DES arbeitet mit 64bit-Blöcken, verwendet einen geheimen Schlüssel mit 56bit Länge und ist sehr performant in Hardware. Ingesamt werden 16 Runden mit jeweils 2 Operationen (T = Transformation und M = Mixing) durchgeführt. Der Input wird dabei auf zwei Hälften (L und R) mit je 32bit aufgeteilt.

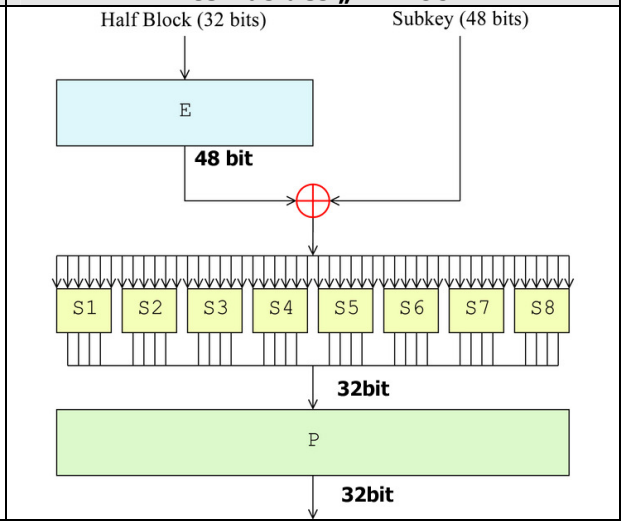
Eine komplette Runde (T und M)



$$Runde = M \circ T$$
$$Runde^{-1} = (M \circ T)^{-1} = T \circ M$$

$$T(L,R) = (L \oplus F_K(R), R)$$
$$M(L,R) = (R,L)$$

Internas des „F"-Block



Sowohl die T als auch die M-Operation sind „involutions", das heisst, dass eine zweimalige Anwendung der Funktion wiederum zum Ausgangstext führt. Dies führt dazu, dass die Entschlüsselung gerade die Umgekehrte Reihenfolge der Verschlüsselung ist.

$$Encryption = IP^{-1} \circ T_{16} \circ M \circ T_{15} \circ \ldots \circ M \circ T_1 \circ IP(X)$$
$$Decryption = IP^{-1} \circ T_1 \circ M \circ \ldots T_{15} \circ M \circ T_{16} \circ IP(Y)$$

Die **F-FUNKTION** garantiert die Nicht-linearität. Die 32bit des Inputs werden dabei mit der Expansion-Funktion E zuerst auf 48bit erhöht und mit dem Rundenschlüssel XORed. Anschliessend wird über 8 nicht-lineare **S-BOXEN** die Datenmenge wieder auf 32bit reduziert und ausgegeben.

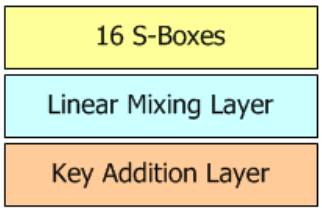
5.7. Triple DES, IDEA

Grosses Problem von DES ist die Tatsache, dass die Schlüsselgrösse von 56bit für den sicheren Einsatz nicht mehr genügend ist. Als Folge davon entwickelte sich 3DES: die Verkettung von 3 normalen DES-Operationen. Daraus erhöht sich die Schlüssellänge auf 112 oder 168bit. Allerdings wird für diese dreifache Berechnung sehr viel Zeit benötigt (auch in HW).

Ein anderer Ansatz verwendet IDEA: hier ist der Schlüssel 128bit gross, die Blockgrösse ebenfalls 64bit. Es werden 8 Berechnungsrunden verwendet. IDEA verwendet Multiplikationen, da diese in Software schneller sind als auf HW. Dementsprechend findet IDEA auch hauptsächlich in programmierte Form Anwendung.

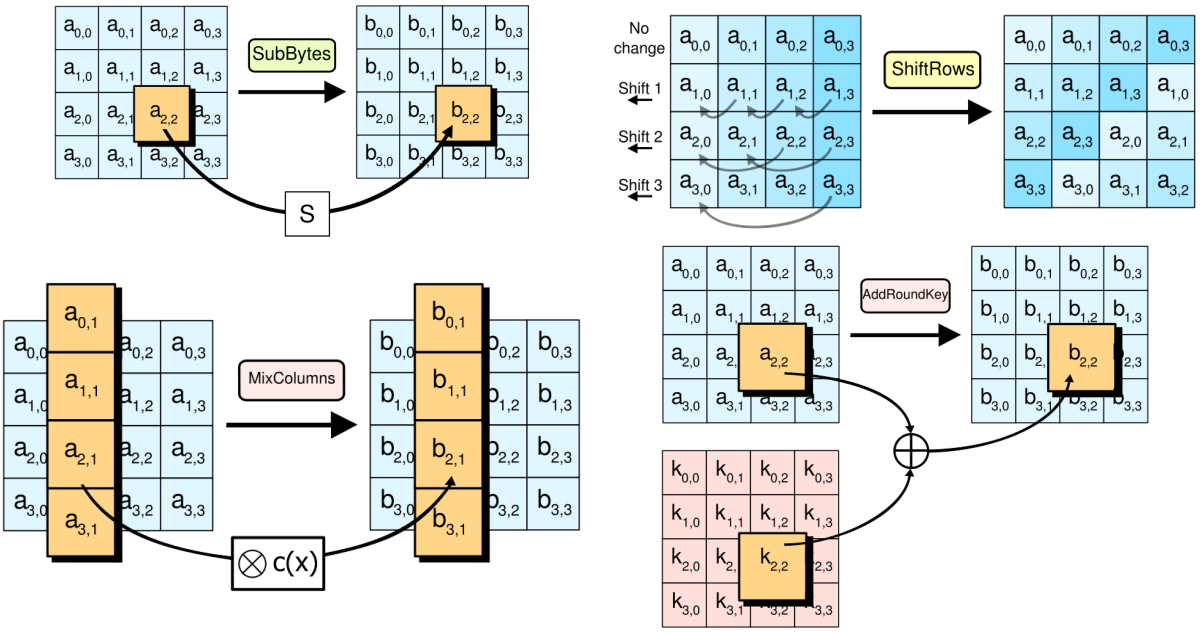
5.8. AES

AES wurde 1998 als Nachfolger von DES über eine öffentliche Ausschreibung bestimmt. Abhängig von der Schlüssellänge (128, 192 oder 256bit) ändert sich auch die verwendete Rundenzahl (10, 12 oder 14 Runden). AES verwendet als Grundlage Substitution-Permutation Netzwerke, welche den Nachteil besitzen, dass sich die Entschlüsselung von der Verschlüsselung unterscheidet. Dafür erreichen sie eine schnellere Diffusion.



Jede AES-Runde besteht aus 3 Schichten. Durch die S-Boxen wird die Nicht-Linearität (ByteSub) erreicht, durch den Linear Mixing Layer eine gute Diffussion (ShiftRows, MixColumns). Im Key Addition Layer wird schliesslich der Rundenschlüssel einbezogen (AddRoundKey). Eine Ausnahme stellt die letzte Runde dar: in dieser wird MixColumn weggelassen.

Die 4 Transformationen arbeiten auf Byte-Ebene. Der aktuelle Zustand kann immer mittels 4x4-Matrizen dargestellt werden.



5.9. PGP – Pretty Good Privacy

PGP ist eine Anwendung, welche in den frühen 90er Jahren entwickelt wurde. PGP verwendet sowohl Block-Cipher (Datenverschlüsselung, z.B. IDEA) als auch Public-Key-Systeme (Schlüsselaustausch, digitale Signaturen, z.B. RSA). Es gibt keine PKI-Infrastruktur, sondern nur ein „Web of Trust".

5.9.1. Session Keys

Einen Session-Schlüssel generieren, Nachricht mit diesem symmetrisch verschlüsseln, Public-Key von RSA zur Verschlüsselung des Session-Schlüssels verwenden, beides an Empfänger senden.

5.10. Kryptografische Hash-Funktionen

Hash-Funktionen bilden einen String einer beliebigen Länge auf eine vordefinierte Länge ab. Hieraus können **KOLLISIONEN** (selber Hash aus zwei unterschiedlichen Inputs) resultieren. Eine Hash-Funktion gilt als „kollisionsresistent", wenn es systematisch unmöglich ist, eine Kollision zu erzeugen.

Die schnellstmögliche Attacke auf Hash-Funktionen ist die Vorberechnung und Tabellierung von $2^{n/2}$ Hashwerten. Dieser Wert resultiert aus dem **GEBURTSTAGSPARADOXON**. Dieses Paradoxon sagt, dass für eine 50%ige Kollisions-Wahrscheinlichkeit etwa $k \approx \sqrt{n}$ Werte genügen. Um solchen Attacken vorzubeugen sollte die Schlüssellänge mindestens 160 bit betragen. Damit müssten etwa 2^{80} Werte tabelliert werden (was heute noch zu lange dauert).

Bekannte Hash-Funktionen sind MD5 (128-bit) und SHA-1 (160-bit), welche beide geknackt wurden.

6. ELLIPTISCHE KURVEN

Die bekanntesten Algorithmen in Public-Key Verschlüsselung (DH, RSA) basieren auf Problemen (Diskreter Logarithmus, Faktorisierung), welche mit ähnlichen Verfahren gelöst werden können, sollte irgendwann jemand wirklich eine Lösung für das Problem finden. Aus diesem Grund bemüht man sich um andere Ansätze. Der bisher erfolgsversprechendste ist die Verwendung von elliptischen Kurven.

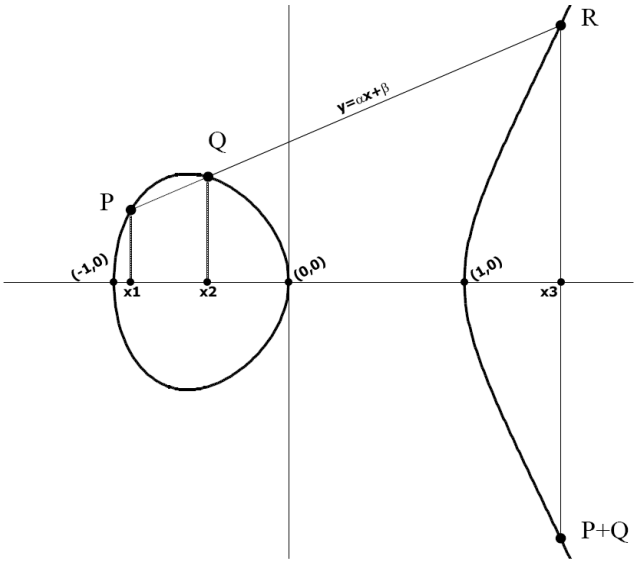
6.1. Grundlagen

6.1.1. Elliptische Kurven über die realen Zahlen

In der Kryptographie werden üblicherweise elliptische Kurven über endliche Körper betrachtet. Die grundlegende Gleichung von elliptischen Kurven lautet wie folgt (a, b ε GF):

$$y^2 = x^3 + ax + b$$

Das Polynom $x^3 + ax + b$ sollte keine mehrfachen Nullstellen im endlichen Körper F besitzen. Das nachfolgendes Beispiel zeigt die elliptische Kurve $y^2 = x^3 - x$:



Die Steigung α der Gerade kann aus den Punkten P(x1,y1) und Q(x2,y2) berechnet werden. Mit Hilfe dieser Steigung kann anschliessend die X-Komponente von R (x3) und somit sofort auch den zugehörigen Y-Wert (y3) berechnet werden.

$$\alpha = \frac{y_2 - y_1}{x_2 - x_1} \qquad x_3 = \alpha^2 - x_1 - x_2 \qquad y_3 = -y_1 + \alpha(x_1 - x_3)$$

Beispiel: $y^2 = x^3 - 36x$

1) Wir wählen zwei zufällige Punkte x1 = -3 und x2 = -2
2) Die zugehörigen y-Werte sind y1 = 9 und y2 = 8
3) Die Steigung beträgt α = -1
4) Jetzt kann der Punkt x3 = 6 berechnet werden
5) Der zum Punkt x3 passende Wert beträgt y3 = 0

6.1.2. Elliptische Kurven über GF(2^n)

Über diesem endlichen Körper wird zwischen zwei „Normalformen" unterschieden:

$$y^2 + b_1 y = x^3 + a_1 x + a_0 \qquad y^2 + xy = x^3 + a_2 x^2 + a_0$$

Im ersten Fall ist die Addition der Punkte einfacher zu realisieren, allerdings ist dieser anfälliger für Attacken. Aus diesem Grund wird in der Praxis üblicherweise die zweite Variante eingesetzt.

6.1.3. Anzahl Punkte einer Kurve (Hasse's Theorem)

In der Kryptologie werden Kurven mit einer grossen Zahl an Punkten bevorzugt. Um die **ANZAHL MÖGLICHER PUNKTE** einer Kurve zu bestimmen müssen alle (x/y)-Kombinationen über einem Feld ausprobiert und die resultierende Gleichung auf Gültigkeit überprüft werden.

Beispiel: Anzahl Punkte der Kurve E = ($y^2 + y = x^3$) über F = GF(2)

(0/0): $0^2 + 0 = 0$ => $\underline{0 \equiv 0}$ (1/0): $0^2 + 0 = 1$ => $0 \equiv 1$

(0/1): $1^2 + 1 = 0$ => $\underline{0 \equiv 0}$ (1/1): $1^2 + 1 = 1$ => $0 \equiv 1$

Die Kurve besitzt somit 3 Punkte: (0/0), (0/1) und den „Point of Infinity".

Bei **GRÖSSEREN FELDERN** kann es ratsam sein, zuerst alle Werte der einen Gleichungshälfte zu berechnen. Anschliessend kann man die andere Hälfte berechnen und identischen Resultate miteinander kombinieren.

Beispiel: Anzahl Punkte der Kurve E = ($y^2 = x^3 - 1$) über F = GF(5)

1) Mögliche Werte für y:

$0^2 \% 5 = 0$
$1^2 \% 5 = 1$
$2^2 \% 5 = 4$
$3^2 \% 5 = 4$
$4^2 \% 5 = 1$

2) Mögliche Werte für x:

$0^3 - 1 \% 5 = \underline{4}$
$1^3 - 1 \% 5 = \underline{0}$
$2^3 - 1 \% 5 = \underline{2}$
$3^3 - 1 \% 5 = \underline{1}$
$4^3 - 1 \% 5 = 3$

3) Für die X-Werte 0, 1 und 3 haben wir Resultate bekommen, welche auch auf der y-Seite anzutreffen waren. Wir können diese also miteinander kombinieren. Punkte der Kurve sind: (0/2), (0,3), (1/0), (3/1), (3/4), PoI

Eine weitere Variante zur Bestimmung der Anzahl Punkte N einer Kurve E über dem Körper GF(q) ist die Verwendung von Hasse's Theorem:

$N = q + 1 - a$	Hasse' Theorem (mit $\lvert a \rvert \leq 2\sqrt{q}$)
$N_r = -\alpha^r - \beta^r + q^r + 1$	α und β sind die Nullstellen des charakteristischen Polynoms
$T^2 - aT + q$	Charakteristisches Polynom

Beispiel: E = ($y^2 + y = x^3$) F = GF(2)

1) Wir wissen aus dem vorhergehenden Beispiel, dass die Kurve 3 Punkte hat.
2) Aus Formel 1 geht hervor, dass: a = q + 1 – N = 2 + 1 – 3 = 0
3) Das charakteristische Polynom lautet somit: T^2–aT+q = T^2–0*T+2 = T^2+2
4) Nullstellen bestimmen: T^2+2 = 0, ergibt α = –j*Sqrt(2), β = j*Sqrt(2)
5) Für ungerade r gilt: $N_r = 2^r + 1$
   Für gerade r gilt: $N_r = 2^r + 1 - 2(-2)^{r/2}$

6.2. Vorteile und Nachteile

- (+) Es gibt viele Kurven und somit viele verschiedene Gruppen (endliche Körper). Beim klassischen Diffie-Hellmann darf nur über die Primzahlen gerechnet werden.
- (+) Es gibt noch keine effiziente Methode gegen elliptische Kurven. Folglich können kleinere Gruppen („Schlüssel") verwendet werden (200 bit in EC ~ 1024 bit in klassischen Kryptoverfahren)
- (+) Kurven über GF(2^n) können effizient berechnet werden
- (–) Im Allgemeinen ist die Arithmetik aufwändiger als bei der Modulo-Rechnung.

6.3. Anwendungen

6.3.1. Diffie-Hellmann

Vorgehen beim Schlüsselaustausch mittels elliptischen Kurven

0) Wählen eines endlichen Körpers F=GF(q), einer elliptischen Kurve E über F und einem Punkt P auf der Kurve E.
1) A wählt eine zufällige grosse Zahl a und berechnet **A = aP**
2) B wählt eine zufällige grosse Zahl b und berechnet **B = bP**
3) A berechnet Schlüssel mit **k = aB = a(bP)**
   B berechnet Schlüssel mit **k = bA = b(aP)**

6.3.2. ElGamal

Bei ElGamal handelt es sich um ein Public-Key System, welches sowohl über Primzahlen, als auch mittels elliptischen Kurven verwendet werden kann.

Vorgehen beim Versenden einer verschlüsselten Nachricht M von A nach B

0) Wählen eines endlichen Körpers F=GF(q), einer elliptischen Kurve E über F und einem Punkt P auf der Kurve E.
1) B wählt einen geheimen Schlüssel b und berechnet **x = bP**
2) A erhält von B den öffentlichen Teil x.
3) A wählt ein geheime grosse Zahl k und berechnet das Paar **(kP, M+kx)**
4) B errechnet M mittels **M = (M+kx)−b(kP) = (M+kbP)−bkP**

7. DIGITALE SIGNATUREN

Digitale Signaturen erfüllen, mit Ausnahme der ersten, alle der nachfolgenden Bedingungen:

- **AUTHENZITÄT:** Der Besitzer des Dokuments hat dieses Dokument freiwillig unterzeichnet.
- **UNFÄLSCHBARKEIT:** Niemand anderes kann die Signatur unter das Dokument gesetzt haben.
- **WIEDERVERWENDBARKEIT:** Signaturen können nicht verschoben oder kopiert werden.
- **ANPASSUNGEN:** signierte Dokumente können nicht verändert werden.
- **UNLEUGBARKEIT:** der Signierende kann die Signatur nicht abstreiten.

7.1. RSA Signaturen

Um die oft durchgeführte Verifizierung zu beschleunigen ist ein möglichst kleiner öffentlicher Exponent (=e) gewünscht. Eine gebräuchliche Wahl ist $e = 2^{16} + 1$. Dies bedingt, dass der private Exponent (=d) sehr gross und die Erstellung der Signatur somit sehr rechenintensiv wird. Um diesen Vorgang zu Beschleunigen wird das **CHINESE REMAINDER THEOREM** eingesetzt.

Beispiel: p=11, q=23, n=p*q=253, d=147, e=3, x=Hash=119

1) $d_p = d \% (p-1) = 147 \% (11-1) = 7$
   $d_q = d \% (q-1) = 147 \% (23-1) = 15$
2) $s_p = x^{d_p} \% p = 119^7 \% 11 = 4$
   $s_q = x^{d_q} \% q = 119^{15} \% 23 = 3$
3) Euklidscher Algorithmus auf: $y_p*p + y_q*q = 1 \Rightarrow y_p = -2, y_q=1$
4) $s = (s_p*y_q*q + s_q*y_p*p) \% n = (4*1*23 - 3*2*11) \% 253 = 26$

Es kann gezeigt werden, dass diese Variante etwa 4x schneller ist als die direkte Exponentialrechnung mit dem Exponent d. Ausserdem hängen die Werte `yq*q` und `yp*p` im Schritt 4 nicht von der zu signierenden Nachricht ab. Diese können somit einmalig vorberechnet werden.

7.2. ElGamal Signaturen

Wie die ElGamal-Verschlüsselung basiert die Signierung auf dem diskreten Logarithmusproblem über den Primzahlen oder der Gruppe einer elliptischen Kurve. Es ist wichtig für die Sicherheit der Signatur, dass die zufällige Zahl k für jede Signatur neu gewählt wird. Andernfalls könnte aus zwei Texten mit den dazugehörigen Signaturen der private Schlüssel `a` bestimmt werden (k identisch = r identisch!).

Public Elemente: Primzahl p, Generator g in GF(p), $A = g^a \bmod p$
Private Elemente: Exponent a <= p-2

1) A erzeugt einen Hash h der Meldung mit, mit h(m) aus {1,2,…,p-2}
2) A wählt eine Zahl k aus {1,2,…,p-2}, die teilerfremd zu p-1 ist.
3) A berechnet das modulare Inverse k^{-1} zu k mod p-1.
4) A berechnet Signatur: **$r = g^k \bmod p$**
   $s = k^{-1}*(h-a*r) \bmod (p-1)$
6) B verifiziert Signatur (r,s): **$A^r*r^s \equiv g^h \bmod p$** für **$1 <= r <= p-1$**

Beispiel: p = 23, g = 7, a = 6
0) $A = g^a \bmod p = 7^6 \bmod 23 = 4$
1) h = h(m) = 7
2) A wählt k = 5, welches teilerfremd zu p-1 ist.
3) A berechnet das modulare Inverse k^{-1} zu k mod p-1, $k^{-1} = 9$
4) A berechnet Signatur: $r = g^k \bmod p = 7^5 \bmod 23 = 17$
   $s = k^{-1}*(h-a*r) \bmod (p-1) = 9*(7-6*17) \bmod 22 = 3$
6) B verifiziert Signatur (r=17, s=3): $A^r*r^s \equiv g^h \bmod p$
   $4^{17}*17^3 \equiv 7^7 \bmod 23$
   $5 \equiv 5 \bmod 23$

7.3. DSA – Digital Signatur Algorithm

DSA ist ein Standard des NIST (National Institute of Standards). Es ist eine effizientere Version der ElGamal-Signierung. Die Länge des Exponenten beträgt nur gerade 160 bit, die Parameterwahl ist vorgeschrieben. Es gibt ausserdem eine Variante, welche auf elliptischen Kurven basiert (ECDSA).

8. ONE-TIME-PAD

8.1. Allgemeines

Beim One-Time-PAD (=OTP) handelt es sich um eine 1917 von Vernam erfundene symmetrische Verschlüsselung. Verschlüsselung wird durch ein bitweises XOR des Klartexts mit einem Stream von zufälligen Bits erreicht. Da XOR selbstinvertierend ist, handelt es sich bei der Entschlüsselung um dieselbe Operation.

$$c = p \oplus k \qquad\qquad p = c \oplus k$$

Dies bedingt, dass der Schlüsselstream vorgängig auf einem sicheren Kanal übertragen wurde. Da der zufällige Schlüsselstream dieselbe Länge wie der zu verschlüsselnde Klartext besitzt entstehen unter Umständen riesige Schlüssel. Ausserdem muss die Bitfolge wirklich zufällig sein, das heisst:

- 0er und 1er treten mit derselben Wahrscheinlichkeit (=1/2) auf.
- Es kann anhand aufgetretener Bitfolgen keine Voraussage für die nachfolgenden Bits gemacht werden.

8.2. Perfect Security

Der OTP besitzt die „perfect Security" Eigenschaft. Diese Eigenschaft sagt aus, dass bei einem gegebenen Ciphertext die Wahrscheinlichkeit für jeden Ausgangstext gleichgross ist. Oder umgekehrt: ausgehend von einem Klartext kann jeder der möglichen Ciphertexte mit derselben Wahrscheinlichkeit generiert werden. Durch bedingte Wahrscheinlichkeiten ausgedrückt sieht dies wie folgt aus:

$$p(p \mid c) = p(p) \qquad \Rightarrow \qquad p(p \mid c) = \frac{p(p,c)}{p(c)} = \frac{p(p) * p(c)}{p(c)} = p(p)$$

Wir sehen auch, dass die Wahrscheinlichkeiten unabhängig sein müssen. Im konkreten Fall beträgt die Wahrscheinlichkeit $p(c) = p(p) = 1/2^n$, wobei n die Länge des zu verschlüsselnden Textes ist.

Die vollständige mathematische Herleitung lautet wie folgt: jeglicher Ciphertext wird mit derselben Wahrscheinlichkeit $p(c) = 1/2^n$ produziert. Jeder Klartext kann mit derselben Wahrscheinlichkeit auf einen der möglichen Ciphertexte abgebildet werden: $p(c \mid p) = 1/2^n$. Die Definition der bedingten Wahrscheinlichkeit liefert:

$$p(c \mid p) * p(p) = p(p \mid c) * p(c)$$

Durch Einsetzen der vorgängig bestimmten Terme in obige Gleichung folgt direkt die Definition der „Perfect Security"-Eigenschaft:

$$(1/2^n) * p(p) = p(p \mid c) * (1/2^n)$$
$$p(p) = p(p \mid c)$$

8.3. Probleme beim OTP

Der OTP bleibt sicher, solange ein Schlüssel wirklich nur einmal verwendet wird. Wird ein Schlüssel mehrmalig verwendet, so wird ein Rückschluss auf die Inhalte der Nachricht möglich.

Mit der aktuellen Technik wäre es zwar möglich eine grosse Menge an zufälligen Zahlen dem Kommunikationspartner zu übermitteln (beispielsweise mittels CD-Rom), allerdings muss dieser dann immer noch wissen, dass man mit ihm kommunizieren möchte.

Auch die Generierung von realen Zufallszahlen ist nicht ganz trivial. Als Alternative hat sich die Verwendung eines **PSEUDO RANDOM GENERATORS** etabliert. Dieser wird mit einer kurzen Zufallsfolge initialisiert (welche natürlich sicher übertragen werden muss) und produziert anschliessend zufällige Zeichenfolgen.

9. AUTHENTICATION

Es wird zwischen Authentication von Benutzern (RSA, DH) und Authentication von Daten unterschieden. Besonders bei PKI-System müssen wir sicher sein, dass der erhaltene Public-Schlüssel vertrauenswürdig ist, da sonst „Man in the Middle"-Attacken möglich wären. Authentifizieren sich beide Seiten gegenseitig, so spricht man von **MUTUAL** Auth, andernfalls von **ONE-SIDED** Auth.

9.1. Message Authentication Codes (MAC)

Normale Hash-Funktionen (siehe Kapitel 5.10) erlauben die Überprüfung der Integrität einer Nachricht, allerdings kann die Herkunft nicht verifiziert werden. Für diesen Einsatzzweck existieren die so genannten parametrisierten Hash-Funktionen (=MAC). Neben dem zu „hashenden" Text verlangen diese einen symmetrischen Schlüssel als Input. Natürlich sollte der symmetrische Schlüssel nicht aus dem produzierten Hash ableitbar sein, damit die Funktion als sicher betrachtet werden kann.

Beispiel: Hash-Funktion g, MAC h_k, k = Schlüssel

$$g : \sum^* \to \sum^4$$

$$h_k : \sum^* \to \sum^4, x \to g(x) \oplus k$$

Ein Anwendungsfall ist das Versenden von Daten, welche zwar nicht geheim, dennoch nicht verändert werden dürfen (Beispiel: „Wahlliste").

- Die beiden Kommunikationspartner A und B tauschen zuerst einen geheimen Schlüssel aus.
- A sendet die Liste zusammen mit dem MAC unter Verwendung des geheimen Schlüssels.
- B berechnet ebenfalls den MAC unter Verwendung des geheimen Schlüssels und vergleicht.

Obiges Verfahren erinnert Stark an RSA-Signaturen. Es gibt aber einen entscheidenden Unterschied: bei RSA kann die Signatur von jedem validiert werden (Schlüssel öffentlich), in obigem Fall nur von denjenigen Personen, die den privaten Schlüssel k kennen. Ausserdem kann beim MAC-Verfahren jeder Anwender, der in Besitz des Schlüssel k ist, nach Änderung am Dokument einen neuen Hash erzeugen. Bei RSA ist dies nur dem ursprünglichen Erzeuger der Signatur mittels Private-Key möglich.

9.2. Identification and Entity Authentication

Oft müssen sich die Anwender eines Systems gegenüber dem System ausweisen. Der einfachste Fall ist ein **PASSWORT-SYSTEM**, in welchem der Anwender das Passwort eingibt und dieses mit einem verschlüsselt abgelegten Wert verglichen wird. Wir sprechen auch von „Weak-Authentication". Eine mögliche Erweiterung ist die Verwendung von biometrischen Werten (Fingerabdrücke).

Fortgeschrittener ist das **CHALLENGE-RESPONSE-VERFAHREN** („Strong-Authentication). Die Idee dahinter ist, dass der Anwender dem System glaubhaft machen kann im Besitz eines Schlüssels zu sein, der nur der Anwender (und das System) kennen.

- B sendet einen zufälligen String RAND zu A („Challenge")
- A berechnet RES = f(k, RAND) und sendet dieses zurück an B („Response")
- B berechnet ebenfalls f(k, RAND) und vergleicht seinen Wert mit dem empfangenen Wert

Abhängig vom System werden diese Schritte mehrmals ausgeführt.

10. ERROR-CORRECTING CODES

Codierung von Daten hat hauptsächlich drei Anwendungsgebiete: Kompression von Daten, Fehlerkorrektur und –erkennung sowie Geheimhaltung von Daten. Das folgende Kapitel beschäftigt sich mit dem zweiten Anwendungsgebiet.

10.1. Binäre Blockcodes

Bei diesen Blockcodes werden zu den Codewörtern der Länge `k` zusätzliche Bits `e` zur Fehlererkennung angehängt.

$$n = r + e$$

Der einfachste binäre Blockcode ist der **PARITÄTSCHECK**. An das Codewort wird ein zusätzliches Bit angehängt, sodass die Summe aller Bits Modulo 2 den Wert 0 ergibt. Durch diese Massnahme kann ein einzelner Fehler detektiert, nicht aber korrigiert werden. Es kann aber durchaus möglich sein, dass wir mehr als einen, aber eine ungerade Anzahl, an Fehlern erkennen.

$$\left(\sum_k c_k\right) + c_e = 0 \mod 2$$

Ein anderer einfacher Code ist der **REPETITION CODE**. Hier wird die Nachricht in Blöcke der Länge k=1 aufgeteilt. Das Zeichen in jedem Block wird n-1 Mal wiederholt. Für das Beispiel n=5 resultiert:

$$0 \to 00000 \qquad \text{und} \qquad 1 \to 11111$$

Das ist nicht wirklich effizient, allerdings erlaubt uns obige Massnahme bereits die Erkennung und Korrektur von bis zu 2 falschen Bitstellen.

10.2. Fehlererkennung und –korrektur

Blockcode	Ein Blockcode ist eine Liste von `m` unterschiedlichen Codewörtern der Länge `n`.
Rate des Codes	$R = \frac{k}{n}$ (`k` = Länge der „Nutzdaten", `n` = Länge des CW's)
Hamming-Distanz	$d(x, y)$ Anzahl der unterschiedlichen Stellen zweier Codewörter.
Minimale Distanz	$d_{\min}$ Minimale Hamming-Distanz zwischen 2 Codewörtern eines Codes.
Hamming-Gewicht	$w(x)$ Anzahl der Stellen von x, welche nicht den Wert 0 aufweisen.
Fehlermuster	$e = r - c$ Wobei `c` = Codewort und `r` = empfangenes Codewort.

Für obige Definitionen gelten die folgenden Zusammenhänge:

$$d(x, y) = w(x - y) \qquad\qquad w(e) = d(r, c)$$

Die minimale Hammingdistanz $d_{\min}$ eines Codes ist wichtig im Hinblick auf die Fähigkeit eines Codes, Fehler zu erkennen, bzw. sogar korrigieren zu können. Es gelten die folgenden Regeln:

Wenn $d_{\min} > t$ kann ein Code maximal `t` Fehler in einem Codewort erkennen.
Wenn $d_{\min} > 2t$ kann ein Code maximal `t` Fehler in einem Codewort korrigieren.

Beispiel: c1=(0011), c2=(1100)
Der Coderaum besteht in diesem Beispiel aus nur gerade zwei Wörtern. Die minimale Distanz entspricht somit der Distanz d(c1, c2) = 4. Gemäss obigen Regeln können also t = 3 Fehler erkannt und t = 1 Fehler korrigiert werden.

Aufbauend auf diesen Definitionen kann der **MINIMUM HAMMING-DISTANCE DECODER** implementiert werden. Dieser weist ein erhaltenes Codewort `r` demjenigen Codewort `c` zu, welches die geringste Hammingdistanz besitzt. Grösstes Problem dieses Decoders: bei einem Code mit vielen Codewörtern müssen nach jedem empfangenen Wort eine Vielzahl an Hammingdistanzen bestimmt werden, um die „perfekte" Dekodierung zu ermöglichen. Dieser Vorgang ist ziemlich rechenintensiv.

10.3. Lineare Codes

Die Codewörter eines linearen Codes C spannen einen Vektorraum über einem endlichen Körper auf. In jedem linearen Code muss der 0-Vektor immer enthalten sein. Ausserdem ist die minimale Distanz identisch mit dem minimalen Gewicht aller Codewörter. Ferner ergibt die Addition von zwei gültigen Codewörtern und die Multiplikation eines Codeworts mit einer Konstante erneut ein gültiges Codewort.

$$d_{\min} = w_{\min} \qquad c, c' \in C \Rightarrow (c + c') \in C \qquad c \in C, a \in F \Rightarrow (ac) \in C$$

Die **GENERATORMATRIX** eines Codes wird durch zeilenweises Aufschreiben von Basisvektoren aus dem Vektorraum erzeugt. Sie wird oft auch **ENCODING MATRIX** genannt und besteht aus `(k x n)` Komponenten, wobei `k` die Zahl der Informationsstellen und `n` die Länge des gesamten Codeworts repräsentiert. Ein Vektor `c` gehört genau dann zum Code `C`, wenn das Matrixprodukt `c = a*G` gilt.

Beispiel: Codewörter einer Generatormatrix bestimmen (F^3, F = GF(2))

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Mögliche Inputs sind: (0,0) - (0,1) - (1,0) - (1,1)

Variante 1: Ausmultiplizieren aller möglichen Inputs:

$$(0,0)*G = (0*1+0*0 \quad 0*0+0*1 \quad 0*1+0*1) = \underline{(0 \quad 0 \quad 0)}$$
$$(0,1)*G = (0*1+1*0 \quad 0*0+1*1 \quad 0*1+1*1) = \underline{(0 \quad 1 \quad 1)}$$
$$(1,0)*G = (1*1+0*0 \quad 1*0+0*1 \quad 1*1+0*1) = \underline{(1 \quad 0 \quad 1)}$$
$$(1,1)*G = (1*1+1*0 \quad 1*0+1*1 \quad 1*1+1*1) = \underline{(1 \quad 1 \quad 0)}$$

Variante 2: Alle Möglichen Zeilenkombinationen der Generatormatrix

```
=>     0 Vektor,   1. Zeile,   2. Zeile,   1. + 2. Zeile
=>     (0,0,0),    (1,0,1),    (0,1,1),    (1,1,0)
```

Wir sprechen von einer **SYSTEMATISCHEN MATRIX**, wenn die Generatormatrix aus zwei Teilen besteht. Der erste Teil ist die Identitätsmatrix I_k = `(k x k)`, der zweite Teil ist `P = (k x (n-k))`.

Beispiel: Finden einer systematischen Generatormatrix über GF(2).

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \Rightarrow G = \begin{pmatrix} Z_1 + Z_3 \\ Z_1 + Z_2 \\ Z_1 + Z_2 + Z_3 \end{pmatrix} = \begin{pmatrix} (1,1,1,0) + (0,1,1,1) \\ (1,1,1,0) + (1,0,1,1) \\ (1,1,1,0) + (1,0,1,1) + (0,1,1,1) \end{pmatrix} = \underline{\underline{\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}}}$$

Vorgehen: wir können einzelne Zeilen so kombinieren, dass die entstehenden Vektoren die Bedingung der systematischen Matrix erfüllen. Dabei können wir beliebig Zeilen addieren als auch subtrahieren - Hauptsache wir finden Vektoren, welche die Bedingung erfüllen. ☺

Eine zweite Variante geht systematisch vor: durch Berechnung aller möglichen Codewörter können die Vektoren für die systematische Matrix direkt ausgelesen werden:

(0,0,0)*G = (0,0,0,0)	(1,0,0)*G = (1,1,1,0)
(0,0,1)*G = (0,1,1,1)	(1,0,1)*G = (1,0,0,1)
(0,1,0)*G = (1,0,1,1)	(1,1,0)*G = (0,1,0,1)
(0,1,1)*G = (1,1,0,0)	(1,1,1)*G = (0,0,1,0)

Eine dritte, besonders systematische Variante zeigte Dozent Meier in der Übungsbesprechung. In dieser werden auf Basis der unsystematischen Matrix verschiedene Gleichungssysteme aufgestellt.

$$\begin{pmatrix}1\\0\\0\\*\end{pmatrix} = x*\begin{pmatrix}1\\1\\1\\0\end{pmatrix} + y*\begin{pmatrix}1\\0\\1\\1\end{pmatrix} + z*\begin{pmatrix}0\\1\\1\\1\end{pmatrix} = \left|\begin{array}{l}1 = x+y\\0 = x+z\\0 = x+y+z\end{array}\right| \Rightarrow \begin{array}{l}x=1\\y=0\\z=1\end{array}$$

=> Z1 + Z3 = (1,0,0,1)

$$\begin{pmatrix}0\\1\\0\\*\end{pmatrix} = x*\begin{pmatrix}1\\1\\1\\0\end{pmatrix} + y*\begin{pmatrix}1\\0\\1\\1\end{pmatrix} + z*\begin{pmatrix}0\\1\\1\\1\end{pmatrix} = \left|\begin{array}{l}0 = x+y\\1 = x+z\\0 = x+y+z\end{array}\right| \Rightarrow \begin{array}{l}x=1\\y=1\\z=0\end{array}$$

=> Z1 + Z2 = (0,1,0,1)

$$\begin{pmatrix}0\\0\\1\\*\end{pmatrix} = x*\begin{pmatrix}1\\1\\1\\0\end{pmatrix} + y*\begin{pmatrix}1\\0\\1\\1\end{pmatrix} + z*\begin{pmatrix}0\\1\\1\\1\end{pmatrix} = \left|\begin{array}{l}0 = x+y\\0 = x+z\\1 = x+y+z\end{array}\right| \Rightarrow \begin{array}{l}x=1\\y=1\\z=1\end{array}$$

=> Z1 + Z2 + Z3 = (0,0,1,0)

Aus einer systematischen Generatormatrix kann die so genannte **PARITY-CHECK MATRIX H** abgeleitet werden. Für die Matrix H und die Codewörter c gilt die Beziehung $c \bullet H^T = 0$. Die Matrix kann für jede Generatormatrix bestimmt werden, für eine systematische Matrix ist das Vorgehen jedoch besonders einfach. Ausgehend von $G = (I_k : P)$ berechnet sich die Paritätsmatrix H mittels:

$$H = \left(-P^T : I_{n-k}\right)$$

P^T bezeichnet die **TRANSPOSITION** der Matrix P (Spiegelung der Matrix an der Diagonale). Falls die Matrix nicht symmetrisch ist, so kann sie zur leichteren manuellen Verarbeitung mit Dummydaten (z.B. dem Platzhalter x) aufgefüllt werden.

Beispiel: Bestimmen der Paritätsmatrix aus einer system. Generatormatrix

$$G = \begin{pmatrix}1&0&0&1\\0&1&0&1\\0&0&1&0\end{pmatrix} \Rightarrow P = \begin{pmatrix}1\\1\\0\end{pmatrix} \Rightarrow P^T = \begin{pmatrix}1&x&x\\1&x&x\\0&x&x\end{pmatrix} \Rightarrow \begin{pmatrix}1&1&0\\x&x&x\\x&x&x\end{pmatrix} = \underline{(1\ \ 1\ \ 0)}$$

$$H = \left(-P^T : I_{n-k}\right) = \left(-P^T : I_{4-3}\right) = \underline{\underline{(1\ \ 1\ \ 0\ \ 1)}}$$

Hinweis: Über GF(2) ist $-P^T$ dasselbe wie P^T. Bei anderen Feldern ist dies zu berücksichtigen!

Wenn wir nun überprüfen wollen, ob ein Code zu einem Codewort gehört, so müssen wir die Beziehung $c \bullet H^T = 0$ überprüfen.

Beispiel: Überprüfen eines gültigen Codeworts mit Hilfe der Paritätsmatrix

$$c = (1 \quad 0 \quad 0 \quad 1) \qquad H = (1 \quad 1 \quad 0 \quad 1)$$

$$c \bullet H^T = (1 \quad 0 \quad 0 \quad 1) * (1 \quad 1 \quad 0 \quad 1)^T = (1 \quad 0 \quad 0 \quad 1) * \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \underline{\underline{0}}$$

Ausgehend von einer Paritätsmatrix H kann die **MINIMALE HAMMING-DISTANZ** $d_{\min}$ des Codes C bestimmt werden. Die Distanz $d_{\min}$ entspricht der kleinsten Spaltenzahl von H, welche linear abhängig sind. Oder anders gesagt: die minimale Distanz entspricht der Anzahl Spalten von H, welche zusammen kombiniert den 0-Vektor ergeben.

Beispiel: Bestimmen der minimalen Distanz aus einer Paritätsmatrix

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \text{ oder } \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \text{ oder } \ldots$$

Fazit: wir brauchen 3 Spalten für eine lineare Abhängigkeit. $d_{Min} = 3$.

Gehen wir zurück zum Kanal: wenn wir ein Codewort `c` übertragen, so erhalten wir am anderen Ende des Kanals das Codewort `r = c + e`. Für dieses erhaltene Codewort `r` kann nun mit Hilfe der (transponierten) Paritätsmatrix das so genannte **FEHLERSYNDROM** berechnet werden. Dieser Vektor markiert uns bei genau einem Fehler die Fehlerstelle innerhalb der Paritätsmatrix (gilt auch für den Hamming Code im nachfolgenden Kapitel, siehe Beispiel unten).

$$s = \underline{r * H^T} = c * H^T + e * H^T = 0 + e * H^T = \underline{e * H^T}$$

Obige Gleichung zeigt, dass das Fehlersyndrom nur vom Fehlermuster `e` abhängt. Da dieses allerdings nicht bekannt ist (wäre zwar schön!), muss das Fehlersyndrom allerdings über das empfangene Codewort `r` bestimmt werden.

Beispiel: Fehlersyndrom eines falschen Hamming-Codeworts bestimmen

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad r = (0 \quad \underline{1} \quad 1 \quad 1 \quad 1 \quad 0 \quad 0)$$

$$s = r * H^T = (0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0) * \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}^T = \underline{\underline{\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}}}$$

Bei genau einem Fehler ist also das 2. Bit fehlerbehaftet.

11. HAMMING AND BCH CODES

Beide in diesem Kapitel vorgestellten linearen Blockcodes basieren auf endlichen Körpern. Ein bekanntes Anwendungsbeispiel des BCH-Codes sind CD-Player.

11.1. Hamming Codes

Binäre Hamming Codes haben eine Codewortlänge von $n = 2^m-1$. Die Anzahl der Informationsbits k beträgt k = n – m, somit haben wir m Paritybits (was gleichbedeutend mit m Paritätsgleichungen ist). Hammingcodes können einen Fehler korrigieren, dazu ist die Coderate sehr hoch.

Zur Konstruktion einer Paritätsmatrix benötigen wir ein irreduzibles Polynom und ein primitives Element, auch Generator genannt (siehe Kapitel 2.2.4). Im ersten Schritt berechnen wir iterativ alle Potenzen des Generators, wobei wir zu grosse Potenzen mittels dem irreduziblen Polynom reduzieren müssen. Wir verdeutlichen dies mittels des Beispiels:

$$p(x) = \alpha^4 + \alpha + 1 \qquad E = GF(2^4)$$

$$\begin{aligned}
\alpha^0 &: \quad 1 && (1000)\\
\alpha^1 &: \quad \alpha && (0100)\\
\alpha^2 &: \quad \alpha^2 && (0010)\\
\alpha^3 &: \quad \alpha^3 && (0001)\\
\alpha^4 &: \quad \alpha^4 /(\alpha^4 + \alpha + 1) = \alpha + 1 && (1100)\\
\alpha^5 &: \quad \alpha^2 + \alpha && (0110)\\
\alpha^6 &: \quad \alpha^3 + \alpha^2 && (0011)\\
\alpha^7 &: \quad (\alpha^4 + \alpha^3)/(\alpha^4 + \alpha + 1) = \alpha^3 + \alpha + 1 && (1101)\\
\alpha^8 &: \quad \alpha^2 + 1 && (1010)\\
\alpha^9 &: \quad \alpha^3 + \alpha && (0101)\\
\alpha^{10} &: \quad \alpha^2 + \alpha + 1 && (1110)\\
\alpha^{11} &: \quad \alpha^3 + \alpha^2 + \alpha && (0111)\\
\alpha^{12} &: \quad \alpha^3 + \alpha^2 + \alpha + 1 && (1111)\\
\alpha^{13} &: \quad \alpha^3 + \alpha^2 + 1 && (1011)\\
\alpha^{14} &: \quad \alpha^3 + 1 && (1001)\\
\alpha^{15} &: \quad 1
\end{aligned}$$

Für α^4 und α^7 habe ich den Reduzierungsvorgang bei zu grossen Potenzen explizit aufgeschrieben. In den restlichen Fällen habe ich ihn jeweils weggelassen. Aus der dritten Spalte dieser Tabelle kann nun zeilenweise die Paritätsmatrix herausgelesen werden:

$$H = \begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1\\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1
\end{pmatrix}$$

Obige Paritätsmatrix zeigt deutlich, dass die ersten 4 Bits die Paritätsbits (m = 4) darstellen. Für jedes Paritätsbit existiert eine Paritätsgleichung. Für die erste Zeile wäre dies zum Beispiel:

$$c_0 = c_4 + c_7 + c_8 + c_{10} + c_{12} + c_{13} + c_{14}$$

Zur **ÜBERPRÜFUNG EINES CODEWORTS**, bzw. **ERKENNUNG EINES FEHLERS** werden diejenigen Bits der Paritätsmatrix miteinander XORed, welche im empfangenen CW ein 1-Bit aufweisen. Dieser Vorgang ist identisch dem am Ende des Kapitel 10.3 vorgestellten Verfahren.

Beispiel: r = (110 110 101 110 101)

Wir müssen also die Spalten 0, 1, 3, 4, …, 13 und 15 miteinander XOR rechnen, da diese Bitpositionen im empfangenen Codewort eine 1 besitzen.

$$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}\oplus\begin{pmatrix}0\\1\\0\\0\end{pmatrix}\oplus\begin{pmatrix}0\\0\\0\\1\end{pmatrix}\oplus\begin{pmatrix}1\\1\\0\\0\end{pmatrix}\oplus\begin{pmatrix}0\\0\\1\\1\end{pmatrix}\oplus\begin{pmatrix}1\\0\\1\\0\end{pmatrix}\oplus\begin{pmatrix}0\\1\\0\\1\end{pmatrix}\oplus\begin{pmatrix}1\\1\\1\\0\end{pmatrix}\oplus\begin{pmatrix}1\\1\\1\\1\end{pmatrix}\oplus\begin{pmatrix}1\\0\\0\\1\end{pmatrix}=\underline{\underline{\begin{pmatrix}0\\1\\0\\1\end{pmatrix}}}$$

Wir erhalten einen Vektor (Fehlersyndrom), welcher uns die Fehlerstelle im Codewort markiert. In der Paritätsmatrix finden wir den Vektor an der Stelle 9 (beginnend bei 0), somit ist das 9. Bit fehlerhaft.

r-Korrigiert = (110 1101 101 010 101).

Eine Anmerkung: ergibt das Fehlermuster den 0-Vektor, so bedeutet dass entweder, dass kein Fehler aufgetreten ist, oder aber das mehr als ein Fehler im CW enthalten ist.

11.2. BCH Codes

BCH Codes enthalten mehr Paritätsbits, aus diesem Grund können sie auch mehr Fehler als der Hamming-Code korrigieren (mindestens 2 Fehler). Um mehr Paritätsgleichungen zu erhalten, wird die bestehende Bedingung des Hamming-Codes um zwei weitere Bedingungen ergänzt. Nachfolgendes Beispiel gilt für einen Code mit Codewörtern der Länge n = 15 und k = 7 Informations-Bit. Wir haben somit m = 8 Paritätsbits.

$$\sum_{i=0}^{14} c_i * \alpha^i = 0 \qquad \sum_{i=0}^{14} c_i * \alpha^{2i} = 0 \qquad \sum_{i=0}^{14} c_i * \alpha^{3i} = 0$$

Die erste Bedingung galt bereits beim Hammingcode. Die zweite fügt keine neuen Paritätsgleichungen hinzu, da in GF(2) die Quadrierung dasselbe ergibt wie die erste Bedingung. Die dritte Bedingung hingegen ergänzt die 4 bestehenden Paritätsgleichungen des Hammingcodes um 4 zusätzliche Paritätsgleichungen – insgesamt haben wir also 8 Paritätsgleichungen. Das deckt sich mit den oben erwähnten m = 8 Paritätsbits.

Zur **FEHLERKORREKTUR** müssen mit Hilfe der Bitstellen des empfangenen Codeworts die folgenden beiden Syndrome berechnet werden. Diese leiten sich aus den obigen Bedingungen ab:

$$s_1 = \sum_{i=0}^{14} r_i * \alpha^i \qquad s_3 = \sum_{i=0}^{14} r_i * \alpha^{3i}$$

Nach einem mathematischen Zauber (welcher hier nicht erwähnt werden soll oder kann) ergibt sich eine **QUADRATISCHE GLEICHUNG** mit den beiden Syndromen:

$$x^2 + s_1 * x + \frac{({s_1}^3 + s_3)}{s_1} = 0$$

Leider gilt die normale Regel zum Lösen von quadratischen Gleichungen nicht für GF(2). Die für uns einfachste machbare Methode ist das Einsetzen aller möglichen Polynome der Tabelle in x. Bereits in unserem Beispiel sind dies 16 zu überprüfende Fälle.

Mit Hilfe der Syndrome und der folgenden Regeln kann die **ANZAHL DER FEHLER** bestimmt werden:

- **KEINE FEHLER:** $s_1 = s_3 = 0$
- **1 FEHLER:** $s_1^3 + s_3 = 0$ && $s_1 != 0$
- **2 FEHLER:** $s_1 != 0$ && $s_3 != 0$
- **MEHR ALS 2 FEHLER:** $s_1 = 0$ && $s_3 != 0$

Beispiel: Überprüfen des Codeworts r = (100 010 111 000 000)

Zur Überprüfung des Codeworts können wir die in Kapitel 11.1 erarbeitete Tabelle, bzw. die dort abgebildete Paritätsmatrix verwenden. Zu „grosse Potenzen" im Syndrom s_3 werden einfach mit der grössten möglichen Potenz reduziert (in unserem Fall mit 15).

$$s_1 = \alpha^0 + \alpha^4 + \alpha^6 + \alpha^7 + \alpha^8 = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \oplus \begin{pmatrix}1\\1\\0\\0\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\1\end{pmatrix} \oplus \begin{pmatrix}1\\1\\0\\1\end{pmatrix} \oplus \begin{pmatrix}1\\0\\1\\0\end{pmatrix} = \underline{\begin{pmatrix}0\\0\\0\\0\end{pmatrix}}$$

$$s_3 = \alpha^0 + \alpha^{12} + \alpha^3 + \alpha^6 + \alpha^9 = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \oplus \begin{pmatrix}1\\1\\1\\1\end{pmatrix} \oplus \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\1\end{pmatrix} \oplus \begin{pmatrix}0\\1\\0\\1\end{pmatrix} = \underline{\begin{pmatrix}0\\0\\0\\0\end{pmatrix}}$$

Da beide Syndrome 0 ergeben enthält das Codewort **keine Fehler**.

Beispiel: Überprüfen des Codeworts r = (111 010 111 000 000)

$$s_1 = \alpha^0 + \alpha^1 + \alpha^2 + \alpha^4 + \alpha^6 + \alpha^7 + \alpha^8 = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \oplus \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \oplus \begin{pmatrix}1\\1\\0\\0\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\1\end{pmatrix} \oplus \begin{pmatrix}1\\1\\0\\1\end{pmatrix} \oplus \begin{pmatrix}1\\0\\1\\0\end{pmatrix} = \underline{\begin{pmatrix}0\\1\\1\\0\end{pmatrix}} = \alpha^5$$

$$s_3 = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^{12} + \alpha^3 + \alpha^6 + \alpha^9 = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \oplus \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\1\end{pmatrix} \oplus \begin{pmatrix}1\\1\\1\\1\end{pmatrix} \oplus \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\1\end{pmatrix} \oplus \begin{pmatrix}0\\1\\0\\1\end{pmatrix} = \underline{\begin{pmatrix}0\\0\\1\\0\end{pmatrix}} = \alpha^2$$

Weder s_1 noch s_3 sind 0. Gemäss obiger Liste besitzt das Codewort 2 Fehler. Die Positionen der Fehler werden nun mit der quadr. Gleichung bestimmt:

$$x^2 + s_1 * x + (s_1^3 + s_3)/s_1 = 0$$
$$x^2 + \alpha^5 * x + (\alpha^{15} + \alpha^2)/\alpha^5 = 0$$
$$x^2 + \alpha^5 * x + (1 + \alpha^2)/\alpha^5 = 0$$
$$x^2 + \alpha^5 * x + \alpha^8/\alpha^5 = 0$$
$$x^2 + \alpha^5 * x + \alpha^3 = 0$$

Im 4 Schritt wurde gemäss Tabelle $1+\alpha^2$ durch α^8 ersetzt. Nun müssen wir noch alle möglichen Fälle $\alpha^0 \ldots \alpha^{14}$ für x einsetzen und die Bedingung prüfen. Wir finden $x = \alpha^1$ und $x = \alpha^2$, also Fehler in Position 1 und 2.

12. ZYKLISCHE CODES

12.1. Grundlagen

Zyklische Codes sind Block Codes, welche die Eigenschaft aufweisen, dass das zyklische Verschieben eines Codeworts immer ein anderes Codewort erzeugt.

$$1011 \rightarrow 0111 \rightarrow 1110 \rightarrow 1101 \rightarrow 1011$$

Die Variable `r` beschreibt den Grad des Generatorpolynoms `g(x)`, `n` ist die gesamte Länge des Codeworts und `k = n - r` ist die Anzahl der Stellen, welche für die Codierung von Informationen verwendet werden können. Folglich bestimmt der Grad des Generatorpolynoms (= `r`) auch die Anzahl der Paritybits.

Ein Polynom wird **MONIC** genannt, wenn die Koeffizienten des Codeworts, bzw. des Codeworts für jegliche Indexe grösser als der Grad des Generatorpolynoms, den Wert 0 aufweisen. Hat das Generatorpolynom beispielsweise den Grad `r = 4` und das gesamte Codewort die Länge `n = 6`, so wäre folgendes Codewort ein monic Codewort:

$$001011$$

Ausgehend vom Generatorpolynom kann durch bitweises Verschieben eine zugehörige Generatormatrix bestimmt werden:

$$G = \begin{pmatrix} 1 & g_{n-k-1} & \dots & g_1 & g_0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & g_2 & g_1 & g_0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & \dots & .. & \dots & g_1 & g_0 \end{pmatrix}$$

Das Generatorpolynom `g(x)` hat eine wichtige Eigenschaft: es teilt das Polynom $x^n - 1$, wobei `n` die Länge der CW's ist. Diese Eigenschaft machen wir uns zur Bestimmung des **PARITY-CHECK POLYNOMS** `h(x)` zu Nutze:

$$h(x) = \frac{x^n - 1}{g(x)}$$

Das so ermittelte Polynom wird zur Erstellung der $(n-k) \times n$ Paritymatrix `H` verwendet. Durch bitweises Verschieben des Parity-Check Polynoms wird diese Matrix mit Werten gefüllt:

$$H = \begin{pmatrix} h_0 & h_1 & \dots & h_{k-1} & 1 & 0 & \dots & 0 & 0 \\ 0 & h_0 & \dots & h_{k-2} & h_{k-1} & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & h_{k-1} \end{pmatrix}$$

Vielmehr kann ich zu der Theorie anhand der Vorlesung leider nicht sagen – diese kam für meinen Geschmack etwas zu kurz. Gemäss Dozent Meier sollten die Beispiele im nächsten Kapitel allerdings für dieses Thema völlig genügen (auch wenn ich nicht ganz dieser Meinung bin ☺).

12.2. Beispiele

Faktorisierung des Polynoms: $1+x^5$
Für die Faktorisierung müssen wir die geometrische Summe kennen. Diese lautet wie folgt:

$$\frac{x^n - 1}{x - 1} = x^{n+1} + ... + x^2 + x^1 + 1$$

Dadurch ergibt sich sofort:

$$x^5 - 1 = (x - 1) * (x^4 + x^3 + x^2 + x^1 + 1)$$

Einen zyklischen Code der Länge n = 3 mit zugehörigen CW's auflisten:
1) Zuerst brauchen wir ein Generatorpolynom g(x). Beispiel: g(x) = x+1
2) Nun müssen wir alle möglichen Terme mit diesem Generatorpolynom multiplizieren. Erhalten wir einen Grad der höher als n ist, so können wir mit dem Vorgang aufhören.

0*(x+1)	= 0	= 000
1*(x+1)	= x+1	= 110
x*(x+1)	= $x+x^2$	= 011
(1+x)*(x+1)	= $1+x^2$	= 101

3) Die Binärdarstellung erhalten wir ganz einfach dadurch, indem wir die Koeffizienten der einzelnen Potenzen als „Binärstelle" verwenden. In der obigen Auflistung entspricht dies der dritten Spalte.

Einen zyklischen Code der Länge n = 3 mit zugehörigen CW's auflisten:
1) Zuerst brauchen wir ein Generatorpolynom g(x). Beispiel: g(x) = x+1
2) Nun erstellen wir die zugehörige Generatormatrix. Dazu wird das Generatorpolynom g(x) in seine Binärdarstellung umgewandelt (1,1) und gemäss Theorie bitweise verschoben.

$$G = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

3) Nun multiplizieren wir alle möglichen Inputs mit dieser Generatormatrix:

(0/0) = (0*1 + 0*0) (0*1 + 0*1) (0*0 + 0*1) = (0 0 0) = 0
(0/1) = (0*1 + 1*0) (0*1 + 1*1) (0*0 + 1*1) = (0 1 1) = $x+x^2$
(1/0) = (1*1 + 0*0) (1*1 + 0*1) (1*0 + 0*1) = (1 1 0) = $1+x$
(1/1) = (1*1 + 1*0) (1*1 + 1*1) (1*0 + 1*1) = (1 0 1) = $1+x^2$

Wie wir sehen, erhalten wir dieselben Lösungen wie in der vorherigen Aufgabe, allerdings mit einer anderen Herleitung.

Bestimmen eines Generatorpolynoms für den Code: {0, $1+x^2$, $x+x^3$, $1+x+x^2+x^3$}
Ein Generatorpolynom zeichnet sich dadurch aus, dass es in allen CW's enthalten ist (siehe auch vorherige Aufgabe). Daraus folgt, dass ein mögliches Generatorpolynom $1+x^2$ ist.

$1+x^2$	=	$1+x^2$ * 1
$x+x^3$	=	$1+x^2$ * x
$1+x+x^2+x^3$	=	$1+x^2$ * (1+x)

Bestimmen der Paritycheck Matrix für n = 4 und g(x)=x^2+1 über GF(3):

$$h(x) = \frac{x^4 - 1}{g(x)} = \frac{x^4 - 1}{x^2 + 1} = \underline{x^2 - 1} = \underline{\begin{pmatrix} 1 & 0 & -1 \end{pmatrix}}$$

Mit Hilfe des Paritycheck-Polynoms kann durch bitweises Verschieben eine Paritycheck Matrix H der Dimension $(n-k)\times n = (4-2)\times 4 = 2\times 4$ ermittelt werden. Achtung: das Polynom muss dabei „umgedreht" werden, also der tiefste Koeffizient folgt an erster Stelle.

$$\underline{\underline{H = \begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}}}$$