Kryptographie

Verschlüsselungsverfahren (VV)

Die Kryptografie wurde schon in primitiver Form von den Römern verwendet:

Cäsars Rotier-das-Alphabet Verschlüsselung:

Schlüssel war die Anzahl der Stellen, um die man rotiert hat, dieser wurde abhängig vom Wochentag geändert. An Montagen wurde um 1 rotiert (a -> b, b -> c, …, z -> a), an Dienstagen um 2 usw.

Montag +1: `Reinhard Stix -> Sfjibse Tujy`

Freitag + 5: `Reinhard Stix -> Wjnsmfwi Xync`

Moderne Verfahren:

- sind bekannt und werden nicht geheim gehalten
- die Sicherheit beruht nur auf der Geheimhaltung des Schlüssels (ein Parameter, der den Output des Verfahrens maßgeblich beeinflusst)

Brute-Force-Angriff:

- systematisches Ausprobieren aller möglichen Schlüssel
- da heute sehr schnelle Computersysteme vernetzt sind, kann man sehr viele Schlüssel/s ausprobieren.
- Vor allem moderne Grafikkarten sind um ein Vielfaches schneller beim Schlüssel-Knacken als CPUs, es gibt aber (NSA!) auch spezielle Schaltkreise, die für diesen Zweck entworfen wurden und daher extrem schnell Schlüssel ausprobieren.
- Schutz bietet nur eine große Anzahl von möglichen Schlüsseln => Schlüssellänge in Bits muss ausreichend groß sein. Das Spreadsheet `Dauer einer Brute-Force-Attacke durch Geheimdienste.xlsx` schätzt die Zeit, bis man alle Schlüssel einer vorgegebenen Bitzahl auf einem gigantischen fiktiven Computernetzwerk durchprobieren kann, bis man auf den richtigen stößt. Überprüfen Sie damit, wie sicher ein 128-Bit Schlüssel bei einer Brute-Force-Attacke ist!

1) Symmetrische VV:

verwenden denselben Schlüssel zum Verschlüsseln und Entschlüsseln. Beispiele:

- DES (Digital Encryption Standard): veraltet, 56-Bit Schlüssel
- 3DES (Triple DES): 3 DES-Duchläufe nacheinander mit verschiedenen DES Schlüsseln

- AES (Advanced Encryption Standard): derzeit fast immer verwendet, sehr schnell und sicher, verschiedene Schlüssellängen >= 128 möglich

es gibt unzählige weitere, z.B. Serpent, Blowfish, Twofish, Idea, RC4 ...

AES ist derzeit der beste Kompromiss zwischen Schnelligkeit und Sicherheit. Das Verfahren wurde in einem Wettbewerb der besten Kryptographie-Experten ausgewählt, da das Verfahren den besten Kompromiss zwischen Sicherheit und Schnelligkeit aller getesteten Teilnehmer aufwies und auch verschiedene Schlüssellängen (128, 196, 224, 256 Bits) zulässt.

Vorteile von symmetrischen VV:

- sehr schnell
- sehr sicher
- einfach (kurzer C-Code, AES hat ca. 500 Zeilen)
- kurze Schlüssellängen (ab 128 Bit)
- Hardware-Beschleunigung ist möglich (z.B. Intel CPUs neueren Datums haben solche Funktionen eingebaut: AES-NI), verwenden nur Bitoperationen (Parallelverarbeitung möglich, da kein Überlauf in die nächsten Stellen möglich sind)

Nachteile von symmetrischen VV:

- sicherer Transport des Schlüssels zum Empfänger

Grundsätzlich kann ein symmetrisches Verschlüsselungsverfahren auf mehrere Arten eingesetzt werden und so mehrere Anwendungsvarianten besitzen. AES z.B. chiffriert und dechiffriert immer einen ganzen Block von 16 Bytes an Daten. Man kann eine Datei auf folgende naheliegende Weise verschlüsseln: **ECB (Electronic Code Book)**

Man zerlegt die Datei in Blöcke von 16 Bytes Länge, diese werden einzeln chiffriert/dechiffriert. Den letzten Block (der u.U. kürzer als 16 Bytes sein kann), muss man speziell behandeln (auf 16 Bytes auffüllen = Padding und die Art des Paddings auch spezifizieren)

Eine andere, deutlich sicherere Variante wäre das **Blockchain-Verfahren** (bekannt durch Bitcoins):

Die heutzutage an häufigsten benutzte Anwendungsart ist der Counter-Modus (Zähler-Modus). Dabei
verschlüsselt man nicht die eigentlichen Daten, sondern den Wert eines Zählers, den man nach jedem Gebrauch hochzählt. Man beginnt mit einem öffentlichen Startwert (z.B. Zähler = 1000) und verschlüsselt die Zahlen 1000, 1001, 1002, … mittels AES und dem gewählten Schlüssel. So erhält man eine Folge von unvorhersagbaren (wenn man den Schlüssel nicht kennt) Daten, d.h. einen sogenannten

Stream Cipher (Strom-Verschlüsselung): Hierbei versteht man eine bestimmte Unterart der symmetrischen Verfahren. Dabei handelt es sich um eine Maschine (oder Software), die man mit einem geheimen Schlüssel und mit öffentlichen Parametern initialisiert, und die daraufhin eine Reihe von Zahlen fester Länge ausgibt (den Stream), die man ohne Schlüssel nicht vorhersagen kann. Diese Zahlen betrachtet man je nach Bedarf als Strom von Bytes oder auch Einzelbits.

Beispiel: ChaCha20, AES in diversen Counter-Modi z.B. AES-GCM (Galois Counter Mode)

Man verschlüsselt/entschlüsselt eine Datei, indem man das 1. Byte (oder Bit) XOR-verknüpft mit dem 1. Byte (oder Bit) des Stroms, das 2. Byte (Bit) XOR-verknüpft mit dem 2. Byte (Bit) des Stroms usw. Da 2 XOR-Verknüpfungen mit demselben Strom sich aufheben, entschlüsselt man gleich wie man verschlüsselt.

Vorteile der Stromverschlüsslung:

- Man kann die Zahlen des Stream Cipher vorher berechnen, bevor man sie braucht. Auch kann man diese Berechnungen leicht parallelisieren (hoher Durchsatz möglich)
- Man hat keine feste Blocklänge der Daten, weil man jedes Byte (Bit) einzeln verschlüsseln kann: es ist überhaupt kein Padding nötig
- die eigentliche Verschlüsselung/Entschlüsselung ist nur eine XOR-Operation, also sehr schnell

Nachteile der Stromverschlüsselung:

- Man muss auf jeden Fall sicherstellen, dass man niemals mit denselben Initialisierungswerten beginnt. Ein bekannter Angriff auf die WPA-Verschlüsselung des Wlan funktionierte dadurch, dass ein Angreifer immer wieder einen Reset des Stream Ciphers auslösen konnte, sodass dieser jedes Mal mit denselben Werten startete.

2) Asymmetrische VV: verwenden 2 Schlüssel

ein Schlüssel zum Verschlüsseln (Public Key), den man ausplaudern darf/soll.
ein Schlüssel zum Entschlüsseln (Private Key), den man geheim halten **muss.**

Fast alle asymmetrischen VV verwenden oft folgende Rechenoperation:

$B^E \bmod M$ (B Basis, E Exponent, M Modul)

das ist der Divisionsrest der Potenz B^E bei Division durch M. Hierbei handelt es sich um arithmetische Operationen, die nicht parallel berechnet werden können und daher langsam sind. Natürlich sind alle beteiligten Zahlen B, E, M riesig (hunderte dezimale Stellen). Wie kann man einen solchen gigantischen Wert einigermaßen schnell berechnen?

Zunächst kann man die Potenz B^E nicht wirklich berechnen, weil diese Zahl viel zu groß für unsere Computer wäre, der Speicherplatz würde nicht für alle Ziffern reichen! Aber eigentlich braucht man nur seinen Divisionsrest modulo M, und der ist kleiner als der Modul M.

Man stellt daher die Potenz als eine Folge von Produkten dar und verwendet für jedes Produkt die gültige Mathe-Formel: $a * b \bmod M = (a \bmod M) * (b \bmod M) \bmod M$ (d.h. man darf beliebig oft den Modulo-Operator einsetzen, um die Teilprodukte klein zu machen ($< M$). Aber wie viele Produkte muss man eigentlich berechnen? Wenn man es geschickt macht, erstaunlich wenige. Berechnen wir z.B. $x = 333333333^{2222222222} \bmod 1111111110$, diese Zahlen haben natürlich nicht hunderte von dezimalen Stellen sondern nur 10, aber man erkennt das Prinzip (die komplette Potenz hätte etwa 20 Milliarden Stellen). Alle unsere Konstanten lassen sich als 32-Bit Zahl darstellen (`unsigned` in C++). Um eine solches Modulo-Produkt in C++ zu berechnen, würde ich die folgende C++ schreiben:

```
unsigned mul_and_mod(unsigned a, unsigned b, unsigned M)
{    unsigned long p = static_cast<unsigned long>(a) * b;
     return p % M;
}
```

Wie oft und mit was muss man diese Funktion aufrufen, um x zu berechnen? Zunächst stellt man den Exponenten als Summe von 2er Potenzen dar, das entspricht der Binärdarstellung dieser Zahl:
$1111111110 = (1000010001110100011010111000110)_2$
Jede Binärziffer 1 bedeutet, dass diese 2er-Potenz ein Summand ist. Von hinten nach vorn:
$1111111110 = 2^1 + 2^2 + 2^6 + 2^7 + 2^8 + \ldots + 2^{31}$

Aus der Mathematik kennt man folgende Rechenregel für Potenzen:

$B^{E+F+G+\ldots} = B^E * B^F * B^G * \ldots$

Da wir den Exponenten 1111111110 als Summe von 14 Summanden (2er Potenzen) dargestellt haben, können wir $B^{1111111110}$ als Produkt von 14 Faktoren darstellen, jeder von ihnen ist von der Form B hoch Zweierpotenz, das macht 13 „Multiplikationen“ (+ modulo). Also müssen wir nur noch die 14 Faktoren berechnen, die alle von der Gestalt B hoch 2er-Potenz sind. Nun gilt:

B hoch $2^0 = B^1 = B$	keine Multiplikation nötig
B hoch $2^1 = B^2 = B*B$	eine Multiplikation
B hoch $2^2 = B^4 = B^2*B^2$	eine Multiplikation
B hoch $2^3 = B^8 = B^4*B^4$	eine Multiplikation
B hoch $2^4 = B^{16} = B^8*B^8$	eine Multiplikation
B hoch $2^5 = B^{32} = B^{16}*B^{16}$	eine Multiplikation

…

Wir müssen alle Potenzen bis zur maximalen Hochzahl 2^{31} berechnen, das geht mit 31 Multiplikationen. Insgesamt sind also „nur“ 14 + 31 „Multiplikationen“ nötig. Man kann diesen Algorithmus auch als rekursive Funktion schreiben und erspart sich dadurch alle Schleifen:

```
unsigned power_and_mod(unsigned B, unsigned E, unsigned M)
{
     if (E == 0)
          return 1;
     if (E % 2 != 0)          // Exponent ist ungerade
          return multiply_and_mod(B, power_and_mod(B, E-1, M),
M);
                // B^E = B * B^(E-1)
     // else E ist gerade -> B^E = (B^2)^(E/2)
     auto B_square = mul_and_mod(B, B, M);
     return power_and_mod(B_square, E/2, M);
}
```

Auch bei viel größeren Zahlen, z.B. 2048 Bit, kommt man mit maximal 2047 + 2047 solcher Operationen aus, wobei man natürlich diese größeren Zahlen auch multiplizieren und dividieren können muss, d.h. `mul_and_mod()` muss man für solche großen Zahlen neu schreiben.

Der RSA-Algorithmus:

RSA (ist nach den Erfindern benannt) ist das meistverwendete asymmetrische VV. Der öffentliche Schlüssel (public Key)besteht aus dem öffentlichen Exponenten $ö$ und dem öffentlichen Modul M. Der geheime Schlüssel (secret Key) besteht aus dem privaten Exponenten p:

Verschlüsseln:	$y = x^{ö} \bmod M$	Potenz und Restoperation!
Entschlüsseln:	$x = y^{p} \bmod M$	Potenz und Restoperation!

Natürlich müssen $ö$ und p zueinander passen, damit obige 2 Gleichungen für jedes Datum x gelten. Bei der Generierung des RSA Schlüsselpaares werden u.a. obige Exponenten berechnet.

$$x = y^{p} \bmod M = (x^{ö} \bmod M)^{p} \bmod M = (x^{ö})^{p} \bmod M = x^{ö*p} \bmod M$$

$$x = x^{1} \bmod M = x^{ö*p} \bmod M$$

Damit 2 Potenzen für jede Basis x mod M gleich sind, müssen die Hochzahlen zueinander kongruent mod Phi(M) sein.

Phi ist die Eulersche Phi-Funktion, Phi(M) ist definiert als die Anzahl der zu M teilerfremden Zahlen unter den Zahlen 1, 2, 3, …, M

Also muss gelten:

$$1 = ö*p \bmod \mathrm{Phi}(M)$$

Man muss also ein Lösungspaar ö, p dieser Gleichung finden. Das Problem dabei ist folgendes:
ö und M sind öffentlich, d.h. ein Angreifer kennt 2 der drei involvierten Größen. Trotzdem sollte er die Gleichung nicht in endlicher Zeit lösen können, während der Besitzer des Schlüssels dieselbe Gleichung bei der Schlüsselgenerierung lösen muss und das natürlich in kurzer Zeit möglich sein soll.

Idee: Der Schlüsselbesitzer in Spe wählt 2 sehr große Primzahlen a, b und berechnet damit

$$M = a*b$$

Aus der Zahlentheorie folgt nun:

$$\mathrm{Phi}(M) = \mathrm{Phi}(a*b) = (a-1)*(b-1)$$

Phi(M) lässt sich daher auf diese Weise leicht berechnen. Ein Angreifer kennt lediglich M aus dem öffentlichen Schlüssel und müsste daraus die Faktoren a und b bestimmen (anders ist Phi(M) nicht zu berechnen). Und das kann dauern. Die Sicherheit von RSA liegt also darin, dass man für riesige Zahlen M die Primzahlzerlegung nicht schnell genug durchführen kann. Es ist dadurch nicht jeder Wert von M möglich, sondern nur Produkte von 2 Primzahlen. Deshalb sind solche „asymmetrischen" Schlüssel viel länger als Schlüssel für symmetrische VV.

Derzeit empfiehlt man als Schlüssellänge mindestens 2048 Bits, wodurch eine Brute Force-Attacke vom Zeitaufwand ungefähr einer Brute- Force Attacke auf einen 113-Bit symmetrischen Schlüssel entspricht.

Vorteile der asymmetrischen VV:

- kein Schlüsseltransport nötig, jeder kann für mich verschlüsseln (mit meinem öffentlichen Schlüssel), aber nur ich kann die Nachricht entschlüsseln.

Nachteile der symmetrischen VV:

- viel langsamer als symmetrische VV (um mehr als den Faktor 1000)
- viel längere Schlüssel nötig
- mathematische Verbesserungen könnten die Sicherheit der Verfahren gefährden
- es könnte bessere und schlechtere Schlüssel geben (z.B. wegen der Primzahlzerlegung)

Der **klassische Diffie-Hellman Algorithmus: DHE**

Seine Erfindung wird den Kryptologen Diffie und Hellman zugeschrieben, tatsächlich war das Verfahren schon früher bekannt und im militärischen Einsatz, weshalb dieser Umstand geheim blieb. Dieser Algorithmus ist sehr beliebt, weil man die Schlüsselpaare sehr schnell berechnen kann (viel schneller als bei RSA) und es so möglich ist, für jede Verbindung neue Schlüssel zu generieren. Im Gegensatz zu RSA ist so ein DH-Schlüsselpaar zumeist nur einmal im Einsatz, während RSA Schlüssel jahrelang verwendet werden. Man spricht bei DH daher von flüchtigen Schlüsseln (ephemeral Keys).

Wenn Alice mit Bob über eine ungesicherte Leitung kommunizieren möchte, verwenden beide das DH-Verfahren wie folgt:

1) Alice wählt öffentliche (riesige) Zahlen M (Modul) und G (Generator) sowie eine geheime private Zahl a (privater Schlüssel von Alice)
2) Alice berechnet $G^a \bmod M = A$ (öffentlicher Schlüssel von Alice)
3) Alice sendet an Bob die Werte M, G, A.
4) Bob wählt eine geheime private Zahl b (privater Schlüssel von Bob)
5) Bob berechnet $G^b \bmod M = B$ (öffentlicher Schlüssel von Bob)
6) Bob sendet an Alice den Wert von B.
7) Alice berechnet nun $B^a \bmod M$, Bob berechne t $A^b \bmod M$

Mit ein wenig Mathematik kann man relativ einfach zeigen, dass gilt:

$$B^a \bmod M = A^b \bmod M = G^{a*b} \bmod M$$

Das bedeutet, dass beide dieselbe Zahl berechnen können, indem Sie Ihren eigenen privaten Key und den öffentlichen Schlüssel des Partners verwenden. Ein Lauscher kennt lediglich die beiden öffentlichen Schlüssel und müßte einen von ihnen knacken, z.B. jenen von Alice:

$$A = G^a \bmod M \rightarrow a = \log_M(A) \,/\, \log_M(G)$$

$\log_M$ bezeichnet den diskreten Logarithmus modulo M, und dieser Logarithmus ist für große Exponenten nicht zu berechnen sondern eigentlich nur per Brute-Force zu bestimmen, d.h. man berechnet alle Potenzen von G der Reihe nach, bis sich A ergibt. Die Sicherheit des Verfahrens liegt hier nicht in der Primzahlzerlegung sondern in der Berechnung der diskreten Logarithmen.

Die nötige Schlüssellängen (M, A, B) liegen auch im Bereich 2048 Bits.

Der **Elliptic Curve Diffie-Hellman Algorithmus: ECDHE**

Diese moderne Variante verwendet statt Zahlen und deren Multiplikation (Potenzen) Punkte einer elliptischen Kurve, das ist die Lösungsmenge einer Gleichung der Form

$$y^2 = x^3 + p*x + q$$

Man definiert die Multiplikation von Kurvenpunkten geometrisch und kann so das klassische Verfahren nachbauen (die meisten Autoren sprechen statt einer Multiplikation von

Kurvenpunkten von einer Addition, obwohl es weder das eine noch das andere ist. Dann muss man die Formeln des klassischen DH-Verfahrens umschreiben, d.h. aus Produkt wird Summe, aus Potenz wird Vielfaches.

Die genauere Ausarbeitung des Verfahrens sollen von Teilnehmern ausgearbeitet werden. Der Ablauf eines EC-DH Handshake läuft so ab:

1) Alice wählt aus einer Menge von vordefinierten elliptischen Kurven eine aus, dadurch sind auch M (Modul) und G (der Generator-Kurvenpunkt ist ein fixer Punkt auf dieser elliptischen Kurve) bestimmt sie und wählt eine geheime private Zahl a (privater Schlüssel von Alice)
2) Alice berechnet $G^a \bmod M = A$ (bzw. $a*G \bmod M = A$ bei additiver Schreibweise) (öffentlicher Schlüssel von Alice)
3) Alice sendet an Bob die gewählte elliptische Kurve (d.h. auch M, G) sowie A.
4) Bob wählt eine geheime private Zahl b (privater Schlüssel von Bob)
5) Bob berechnet $G^b \bmod M = B$ (bzw. $b*G \bmod M = B$ bei additiver Schreibweise) (öffentlicher Schlüssel von Bob)
6) Bob sendet an Alice den Wert von B.
7) Alice berechnet nun $B^a \bmod M$, Bob berechne t $A^b \bmod M$ (bzw. $a*B \bmod M = b*A \bmod M$ bei additiver Schreibweise)

Ebenso wie beim klassischen Verfahren sind beide Werte in 7) identisch.

Wegen der deutlich größeren Komplexität reicht hier eine Schlüssellänge von 256 Bits völlig aus. Wegen der kürzeren Schlüssellängen sind die Berechnungen beim Handshake viel kürzer als beim klassischen Verfahren, obwohl die Komplexität größer ist.

Nach dem Ende der DH-Verfahren:

Bei beiden Varianten haben Alice und Bob ein Geheimnis berechnet, das nur Sie kennen. Aus diesem Geheimnis generieren beide nun den Session-Key für die symmetrisch verschlüsselte Kommunikation. In der Praxis sind die DH-Handshakes noch etwas komplexer: beide Partner übermitteln in den Schritten 3 und 6 zusätzlich öffentliche Parameter an den Partner, die zusammen mit dem Geheimnis den Session-Key bestimmen.

3) Hybrides Verfahren

Das Hybridverfahren verwendet Vorteile beider VV. Die Verschlüsselung der Nachricht erfolgt symmetrisch (z.B. mit AES) mit einem Einmal-Schlüssel (= Session Key). Dieser wird für jeden Empfänger mit dessen Public Key asymmetrisch verschlüsselt und mitgeschickt.

Vorteile:
die lange Nachricht wird symmetrisch verschlüsselt
nur der kurze Session Key (128 oder 256 Bit) wird asymmetrisch verschlüsselt

z.B. Pay-TV

> Der Private Key ist in den Smartcards der Anbieter gespeichert (unauslesbar :-(
> die Session-Keys für jeden legalen Benutzer sind im Programmstrom eingebettet,
> allerdings nicht im Klartext sondern mit dem Public Key verschlüsselt
> diese Darstellung ist etwas vereinfacht, in Wirklichkeit läuft es mehrstufig ab.

Früher wurde für den asymmetrischen Teil des Hybridverfahren RSA eingesetzt, d.h. der Session Key wurde von einer Seite erzeugt und mit dem public Key der Gegenseite RSA-verschlüsselt übertragen. Die Gegenseite entschlüsselt den Session Key mit dem eigenen privaten Schlüssel und kann damit die Daten entschlüsseln.

Dieses Verfahren hatte aber einen entscheidenden Nachteil:
Kommt ein Angreifer an den privaten Schlüssel (die NSA kann z.B. von jeder amerikanischen Firma deren private Schlüssel per Gesetz anfordern und das muss sogar geheim bleiben!) und besitzt er Aufzeichnungen über vergangene Sessions, so lassen sich diese Sessions nachträglich entschlüsseln.

Wir verwenden heute ein besseres Verfahren:

Perfect forward Security:

Der Session Key einer Kommunikation wird aus dem DH-Handshake berechnet (klassisch oder EC). Bob übermittelt in Schritt 6 nicht nur seinen public DH-Key an Alice sondern auch eine Signatur davon. Anhand dieser Signatur kann Alice verifizieren, dass Sie tatsächlich mit Bob spricht.

Warum ist das sicherer? Der DH-Handshake wird bei jeder Kommunikation neu initiiert (da flüchtige Keys verwendet werden) und müsste vom Angreifer einzeln gehackt werden. Der public Key von Bob nützt hierbei gar nichts, weil der nur zur Validierung von Bobs Identität aber nicht zum Schlüsseltransport eingesetzt wird.

ABER: Hat ein Hacker den privaten Schlüssel von Bob erlangt, kann er sich Alice gegenüber als Bob ausgeben und damit auch einen zukünftigen DH-Handshake aushebeln. Das geht aber nur in der Zukunft (ab dem Besitz von Bobs Schlüssel), nicht für vergangene aufgezeichnete Kommunikationen.