cv act *library*™

V1R4

Handbuch

cv cryptovision gmbh
http://www.cryptovision.com

Einführung

Dieses Handbuch gliedert sich in zwei Teile:

1. Kryptographiehandbuch
2. Referenzhandbuch
    a) Algorithmenkatalog
    b) Schnittstellenbeschreibung,

wobei sich das Referenzhandbuch in die angegebenen Teile untergliedert.

Die **cv act *library*** richtet sich an Software-Entwickler. Deshalb sind Kenntnisse in der Programmiersprache C++ vorausgesetzt, um mit der kryptographischen Bibliothek effektiv arbeiten zu können. Außerdem sollte jeder Benutzer, der die **cv act *library*** anwenden möchte, über ein Grundwissen in der Kryptographie und deren Verfahren verfügen. Deshalb besteht diese Dokumentation aus einem Kryptographiehandbuch und einem Referenzhandbuch, so dass im ersten Teil eine Gesamtdarstellung aller verwendeten kryptographischen Verfahren zu finden ist. Im Referenzhandbuch im zweiten Teil werden alle, in der **cv act *library*** enthaltenen Objekte in alphabetischer Reihenfoge nochmal erklärt.

Zur leichteren Navigation durch das Handbuch findet man die Symbole Glossar, Lupe und Literatur.


Alle im Text definierten Begriffe sind im Glossar alphabetisch angeordnet und in einer anderen Form erklärt, als es im Text der Fall ist.


weist darauf hin, dass der zugehörige Text weiterführende, vertiefende Informationen enthält.


ist ein Verweis auf Bücher, Standards oder Web-Pages. Falls an der entsprechenden Stelle nur ein Verweis auftaucht, kann man das Buch anklicken und findet so direkt zum gesuchten Hinweis. Falls mehrere Verweise aufgeführt sind, kann man diese jeweils separat anklicken. Das Buch führt in diesem Fall als Hyperlink zum Anfang des Literaturverzeichnisses.

Die Begriffe, die neu eingeführt werden, sind in der Definition unterstrichen. Englische Begriffe, die als allgemein bekannt gelten, sind *kursiv* dargestellt. Die Inhalte orientieren sich am Stand der wissenschaftlichen Forschung, sind aber an manchen Stellen, z.B. bei mathemamtischen Definitionen mehr populär, also allgemeinverständlich verfasst.

Zur leichteren Navigation durch die Online-Hilfe kann man Hyperlinks benutzen. Diese sind grün markiert und beim Mouse-Over verwandelt sich die Maus in eine Hand. Durch Anklicken landet man an der entsprechenden Stelle. Falls man wieder an die vorherige Stelle zurück möchte, kann man den Button zurück in der Symbolleiste benutzen. In dieser befindet sich noch der cryptovision-Button, durch den man sofort auf die cryptovision-Webpage gelangen kann.

Produktstruktur

Die **cv act *library*** ermöglicht einem Programmierer die Benutzung der wichtigsten Verfahren zur symmetrischen und asymmetrischen Verschlüsselung sowie zur Erstellung digitaler Signaturen, inklusive der benötigten Hashfunktionen und Zufallsgeneratoren. Die Verwendung der Bibliothek ist abgestimmt auf den Einsatz von entsprechenden Smartcards. Desweiteren orientiert sich die **cv act *library*** an den derzeitigen Standards und Richtlinien, wie z.B. ANSI, ISO/IEC oder SigG/SigV.

Die verwendeten Verfahren werden im **Kryptographiehandbuch** beschrieben.

Durch eine modulare Softwarearchitektur werden die Algorithmen über eine einheitliche Schnittstelle verwendet. Damit ist der Code nicht nur unabhängig vom jeweils speziell eingesetzten Algorithmus, sondern auch vom grundsätzlich angewandten Verfahren. Außerdem ist es möglich, eigene kryptographische Algorithmen einzubinden; aufbauende übergeordnete Schemata sind direkt in der Lage, diese zusätzlichen Algorithmen zu verwenden. In der vorliegenden Schnittstelle wird spezifiziert, welche Dienstleistungen die einzelnen Module exportieren, d.h. dem Benutzer zur Verfügung gestellt werden. Im **Referenzhandbuch** wird im Anschluss an den alphabetisch geordneten Algorithmenkatalog diese Schnittstelle beschrieben.

Technischer Support

Für Fragen zur Installation und Funktionalität der **cv act *library*** wenden Sie sich bitte an unseren telefonischen Support **0209/167-2499** oder schicken Sie eine E-Mail an support@cryptovision.com.

Bitte halten Sie Ihre Lizenznummer bereit, wenn Sie den Support in Anspruch nehmen!

Für weitere Informationen zur **cv act** Produktfamilie, für allgemeine Auskünfte über die cv cryptovision gmbh sowie für die neusten Software Downloads besuchen Sie uns im Internet unter www.cryptovision.com.

Teil 1 Kryptographiehandbuch

Kapitel 1 Grundlagen der Kryptographie

Information ist der Rohstoff des 21. Jahrhunderts. Der Austausch und die Beschaffung von Informationen sind zu einem zentralen Bestandteil privater und wirtschaftlicher Aktivitäten geworden. Informationen werden in Dokumenten abgelegt und durch Kommunikation übertragen. Geraten diese Informationen in die falschen Hände, kann daraus ein großer Schaden erwachsen.

Der Schutz vor Missbrauch, Manipulation und Abhören gilt schon heute als die grundlegende Herausforderung dieses Zeitalters. Neue kryptographische Verfahren, die flexibel einsetzbar sind, effizient arbeiten und hohe Sicherheit bieten, sind der Schlüssel dazu.

Kryptographisches Modell

Die Kryptographie ist die Lehre von der Verschlüsselung des geschriebenen Wortes. Ausgangspunkt für die Betrachtung kryptographischer Verfahren ist das Problem, dass zwei Kommunikationspartner Informationen über einen möglicherweise unsicheren Kanal austauschen wollen. Solch ein Kanal kann eine Telefonleitung, die Verbindung zweier Computer oder aber einfach auch das Papier sein, auf dem die Information notiert wird. Unsicherheit bedeutet, dass Dritte diese Nachricht entweder abhören oder verändern könnten. Um dies zu verhindern, soll die Nachricht entsprechend verschlüsselt werden.

Die klassischen Akteure solcher Modelle sind in erster Linie

Alice und

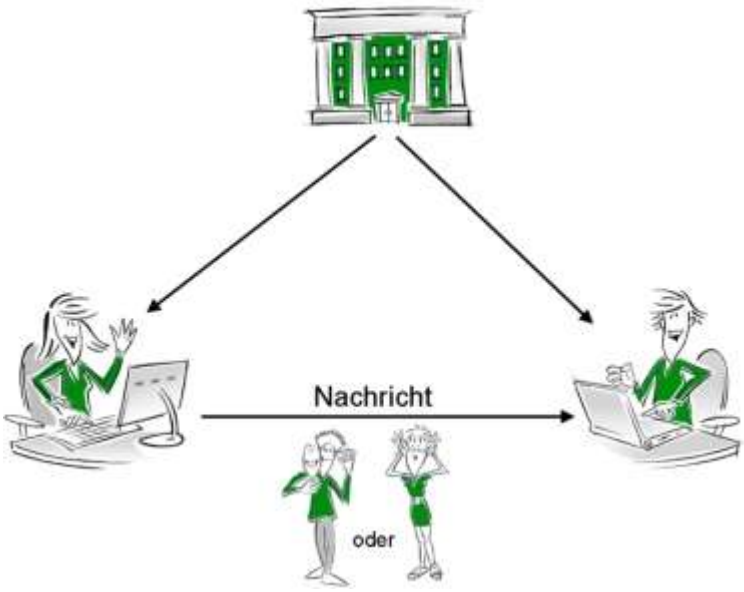
Bob

In der Regel wird Alice eine Nachricht an Bob schicken und Bob wird ihr antworten. Weitere Akteure sind

Eve als Lauscherin

Mallory als ein bösartiger Angreifer

und Trent als Treuhänder und vertrauenswürdiger Vermittler



Die benötigten Begriffe sind:

- Eine Nachricht besteht aus Klartext.
- Das Verfahren, eine Nachricht unverständlich zu machen, heißt Verschlüsselung.
- Eine verschlüsselte Nachricht besteht aus dem Chiffretext.
- Die Umkehrung zur Verschlüsselung ist die Entschlüsselung oder Dechiffrierung.

Ein Kryptosystem ist damit der Versuch, einen Klartext mit Hilfe einer, von einem Schlüssel abhängigen Weise so auf einen Chiffretext abzubilden, dass die Sicherheit dieser Vorgehensweise einzig und allein von der Wahl des Schlüssel abhängt.

Bei den hier beschriebenen Verfahren wird davon ausgegangen, dass dem Angreifer sämtliche Informationen über das verwendete System (bis auf den Schlüssel) zur Verfügung steht. Früher eingesetzte Verfahren zur Verschlüsselung von Informationen verletzten diesen Anspruch und vertrauten zum Teil lediglich auf die Geheimhaltung der verwendeten Methode.

> „Wenn ich von Ihnen verlange, einen Brief zu lesen, den ich in einen Safe gelegt und dann irgendwo in New York versteckt habe, dann hat das nichts mit Sicherheit zu tun, sondern ist ein einfaches Verstecken. Wenn ich diesen Brief jedoch in den Safe lege, Ihnen dann die Entwicklungspläne des Safes und noch hundert Safes gleicher Bauart mitsamt Ihrer Kombinationen gebe, so dass Sie mit Hilfe der weltbesten Safeknacker den Sperrmechanismus ausgiebig studieren können, aber immer noch nicht in der Lage sind, den Safe zu öffnen und den Brief zu lesen-dann ist das Sicherheit. “
>
> -- Bruce Schneier -- „Angewandte Kryptographie“, Vorwort

Neben der Geheimhaltung soll die Kryptographie noch andere Ansprüche erfüllen:

Authentifizierung: dem Empfänger sollte es möglich sein, die Herkunft einer Nachricht zu ermitteln

Integrität: der Empfänger sollte überprüfen können, ob eine Nachricht bei der Übermittlung verändert wurde

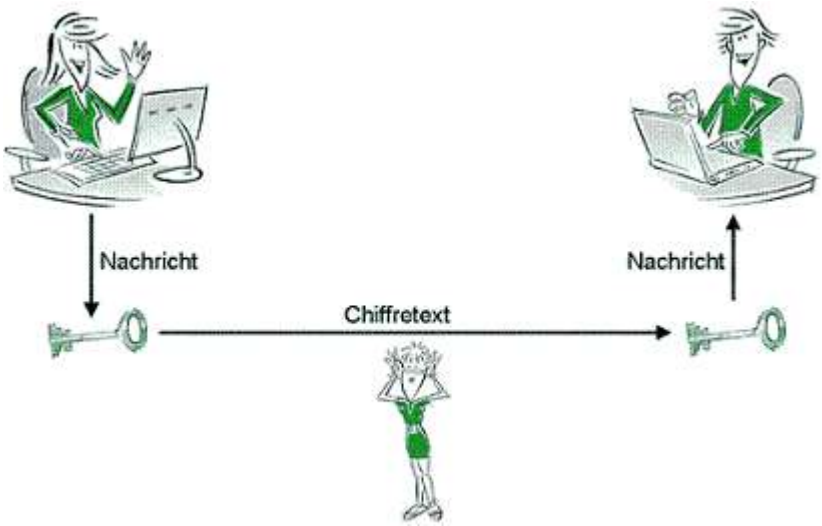
Verbindlichkeit: ein Sender sollte später nicht leugnen können, dass er eine Nachricht gesendet hat.

Verfahren

Es gibt verschiedene Bereiche der Kryptographie, wie die symmetrischen und asymmetrischen Verfahren, Hashfunktionen und unterstützende Verfahren:

Symmetrische Kryptographie:

Zum Ver- und Entschlüsseln von Nachrichten oder Dokumenten wird der gleiche Schlüssel benutzt.



Problem:

- Wie kommen Alice und Bob an den Schlüssel?

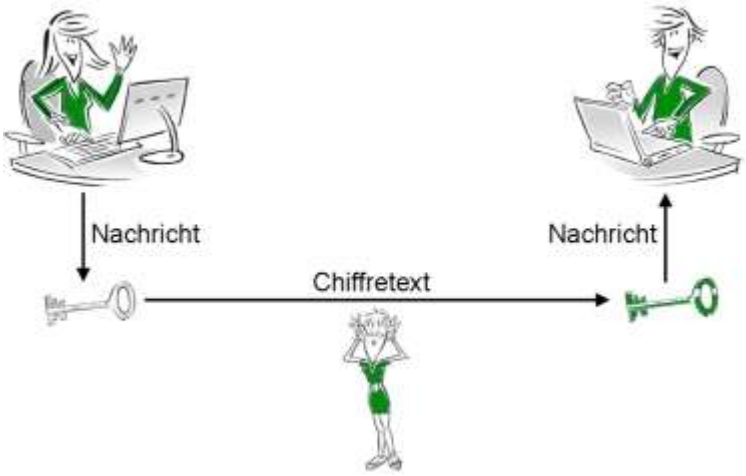
Darauf wird in dem Kapitel 1. 3. 2 über Schlüsselaustausch eingegangen.

- Wie verwaltet man diese Anzahl von Schlüsseln?

  Die Anzahl der zu verwaltenden Schlüssel beträgt bei n Teilnehmern n(n-1)/2, was bei großem n problematisch ist. Darauf wird im Kapitel 2. 3 über Public Key-Infrastrukturen eingegangen.

Asymmetrische Kryptographie: Public Key-Verfahren

Jeder Teilnehmer hat einen öffentlichen Schlüssel, der dem Kommunikationspartner bekannt sein muss (*public key*), und einen privaten, geheimzuhaltenden Schlüssel. Mit asymmetrischer Kryptographie kann man verschlüsseln und digitale Signaturen erstellen.



Probleme:

- Asymmetrische Verfahren sind um Größenordnungen langsamer (in der Praxis mehr als Faktor 1000) als symmetrische Verfahren.

- Die öffentlichen Schlüssel müssen in öffentlichen Datenbanken verwaltet werden und Zertifikate müssen vergeben und verteilt werden.

  Darauf wird in Kapitel 2. 3 über Public Key-Infrastrukturen eingegangen

Hashfunktionen:

Diese Funktionen werden dazu benutzt, das elektronische Gegenstück eines Fingerabdruckes zu erzeugen, welches ein Dokument eindeutig identifiziert. Dazu wird kein Schlüssel verwendet.

> Darauf wird in Kapitel 1. 2. 1 eingegangen.

Unterstützende Verfahren:

Um Sicherheit gewährleisten zu können, bedient man sich zusätzlicher Verfahren wie Zufallszahlengeneratoren oder Techniken wie die Speicherung von Schlüsseln auf Smartcards.

> Darauf wird in den Kapiteln 1. 4 und 2. 2 eingegangen.

Diese kryptographischen Verfahren kann man der Dokumentensicherheit und der Kommunikationssicherheit zuordnen:

Dokumentensicherheit	Kommunikationssicherheit	Verfahren
Verschlüsselung	Verschlüsselung	Symmetrische und Asymmetrische Verfahren
Digitale Signatur	Authentifizierung	Asymmetrische Verfahren
Urheberschutz	Verdeckte Information	Steganographie

Tabelle 1: Zuordnung der Verfahren

Die **cv act *library*** legt die Schwerpunkte auf Verschlüsselung, Digitale Signaturen und Authentifizierung:

Verschlüsselung	Digitale Signaturen	Authentifizierung
- mit symmetrischer Kryptographie - mit asymmetrischer Kryptographie - mit hybriden Verfahren	- mit symmetrischer Kryptographie - mit asymmetrischer Kryptographie - mit asymmetrischer Kryptographie + Hashfunktionen	- mit asymmetrischer Kryptographie - mit symmetrischen oder schlüssellosen Verfahren (MACs)

Tabelle 2: Schwerpunkte der cv act *library*

1. 1 Symmetrische Verschlüsselung

Bei symmetrischer Verschlüsselung wird eine Nachricht mit einem geheimen Schlüssel verschlüsselt und vom Empfänger wieder entschlüsselt. Der Chiffrierschlüssel und der Dechiffrierschlüssel sind also identisch oder stehen in einem einfachen Zusammenhang.

Die konkrete Verschlüsselung verläuft wie folgt:

- Alice und Bob einigen sich über einen als sicher angesehenen Kanal auf einen gemeinsamen Schlüssel
- Alice chiffriert ihre Klartext-Nachricht anhand des Algorithmus und Schlüssels und sendet die Chiffretext-Nachricht an Bob
- Bob dechiffriert die Chiffretext-Nachricht anhand desselben Algorithmus und Schlüssels und erhält die Klartext-Nachricht

Der wesentliche Punkt bei diesen Verfahren ist, dass einem potentiellen Angreifer die Umkehrung der Verschlüsselung ohne Kenntnis des geheimen Schlüssels nicht möglich sein darf. Die Sicherheit dieser Algorithmen liegt also in der Geheimhaltung des Schlüssels (*secret key*). Daraus ergibt sich die Problematik, wie Alice und Bob sicher in Besitz des Schlüssels kommen können.

So ist der Transport des Schlüssels über digitale Netze ohne zusätzliche Sicherheitsvorkehrungen nicht möglich, da er in unbefugte Hände gelangen kann und somit die gesamte verschlüsselte Kommunikation lesbar wird. Der Schlüssel kann über den herkömmlichen postalischen Weg - z. B. per Abholung eines versiegelten Kuverts transportiert werden (typisch bei Online-Banking Verfahren).

Es gibt jedoch auch kryptographische Verfahren zum Schlüsselaustausch, wo als bekanntestes das Protokoll von Diffie-Hellman zu nennen ist, welches in Kapitel 1. 3. 2. 1 beschrieben wird. Oder der symmetrische Schlüssel wird über ein zusätzliches Verfahren, ein sogenanntes

hybrides Verfahren gesichert, das den sicheren Transport über digitale Netze ermöglicht. Dieses wird in Kapitel 1. 3. 3 über asymmetrische Verschlüsselung näher ausgeführt.

Auf jeden Fall wird in der Praxis bei symmetrischer Verschlüsselung ein Schlüssel, der sogenannten Sitzungsschlüssel, für nur eine oder wenige Kommunikationssitzungen verwendet.

1. 1. 1 Strom- und Blockchiffren

Längere Nachrichten müssen in der Regel vor der Chiffrierung aufgespalten werden, da ein Chiffreverfahren normalerweise einen Input fester Länge benötigt. In der Literatur werden symmetrische Algorithmen in zwei Kategorien unterteilt:

Die einen bearbeiten den Klartext zeichenweise und heißen Stromalgorithmen oder Stromchiffren. Dabei kann ein Zeichen z.B. ein Bit oder ein Byte sein. Die anderen bearbeiten den Klartext in Bitgruppen, die Blöcke genannt werden. Sie heißen Blockalgorithmen oder Blockchiffren. Stromchiffren sind in der Regel bei Hardware-Implementationen schneller als die blockorientierten Verfahren, da die bit- bzw. byteweise Verarbeitung einer Hardware-Lösung besser angepasst sind. Bei Software-basierten Systemen finden aber normalerweise Blockchiffren Anwendung. Es ist jedoch auch für Blockchiffren sinnvoll, die Chiffrierung von der Bearbeitung vorhergehender Blöcke abhängig zu machen. Damit werden auch identische Klartextblöcke auf unterschiedliche Chiffretexte abgebildet und damit einem potentiellem Angreifer die Arbeit erschwert.

Diese Verkettung aufeinanderfolgender Blöcke wird als kryptographischer Modus bezeichnet. Die verschiedenen Varianten dabei sind der *electronic codebook* (ECB), *cipher block chaining* (CBC), *cipher feedback* (CFB) oder *output feedback* (OFB). Beim ECB-Modus wird ein Codebuch mit Klartexten und zugehörigen Chiffretexten anlegt. Er bezeichnet den Modus ohne Verknüpfung. Beim CBC-Modus wird andererseits ein Klartextblock vor der Verschlüsselung mit dem vorherigen Block XOR -verknüpft.

1. 1. 2 Schlüsselerzeugung

Bei Strom- und Blockchiffren kann man den geheimen Schlüssel erzeugen, indem man (am besten mit einem Zufallszahlengenerator) eine Bitfolge zufällig auswählt, die dann als Schlüssel fungiert.

1. 1. 3 Die in der cv act *library* enthaltenen symmetrischen Algorithmen

DES

Triple-DES

AES(sowie die Kandidaten Mars, Serpent, Twofish)

CAST

Blowfish

• DES (Data Encryption Standard)

DES ist der Klassiker der kryptographischen Verfahren und stellt die Geburtsstunde der modernen Kryptographie dar. Er wurde aufgrund einer Ausschreibung vom *National Bureau of Standards* (NBS, heute NIST) für einen einheitlichen Verschlüsselungsstandard bei IBM entwickelt und 1976 vorgestellt. Diese Vorstellung bedeutete damals eine kleine Sensation, da DES das erste, standardisierte und vor allen Dingen auch publik gemachte Verschlüsselungsverfahren für wirklich hohe Sicherheitsanforderungen war.

Alle 5 Jahre zertifiziert NIST den DES-Algorithmus. Die letzte Zertifizierung fand 1999 statt, jedoch unter der Auflage, dass die DES-Variante Triple-DES verwendet wird, da DES nicht mehr den heutigen Sicherheitsanforderungen genügt.

Beschreibung von DES

DES ist eine symmetrische Blockchiffre, die Daten in Blöcken von 64 Bit verschlüsselt. Der Algorithmus erhält als Eingabe einen Block von 64-Bit-Klartext und liefert als Ausgabe einen 64-Bit-Chiffretext. Die Schlüssellänge beträgt 56 Bit, wobei der Schlüssel zwar als gewöhnliche 64-Bit-Zahl ausgedrückt wird, doch dient jedes achte Bit einer Paritätsprüfung und wird deshalb ignoriert.

Bei DES gelten einige Schlüssel als schwach, die allerdings vermieden werden können.

Ein Grundbaustein von DES ist, dass die gleichen, schlüsselabhängigen Chiffrierfunktionen mehrfach hintereinander auf einen Block angewendet werden. Solche Anwendung dieser Funktion nennt man Runde.

Des weiteren ist DES ein sogenanntes Feistel-Netzwerk. Die Grundidee dabei ist, dass man den Klartextblock in zwei gleich große Hälften aufteilt. Die linke Hälfte wird in der i-ten Runde mit L_i und die rechte mit R_i bezeichnet. Der Schritt zur nächsten Runde i+1 besteht daraus, dass man den rechten Block der jetzigen Runde zum linken Block der nächsten Rund macht:

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \text{ XOR } f_{S,i}(R_i)$$

Auf die linke Hälfte der i-ten Runde wendet man durch eine XOR-Verknüpfung eine Funktion $f_{S,i}$ an, die vom geheimen Schlüssel S abhängt und erhält die rechte Hälfte der nächsten Runde.

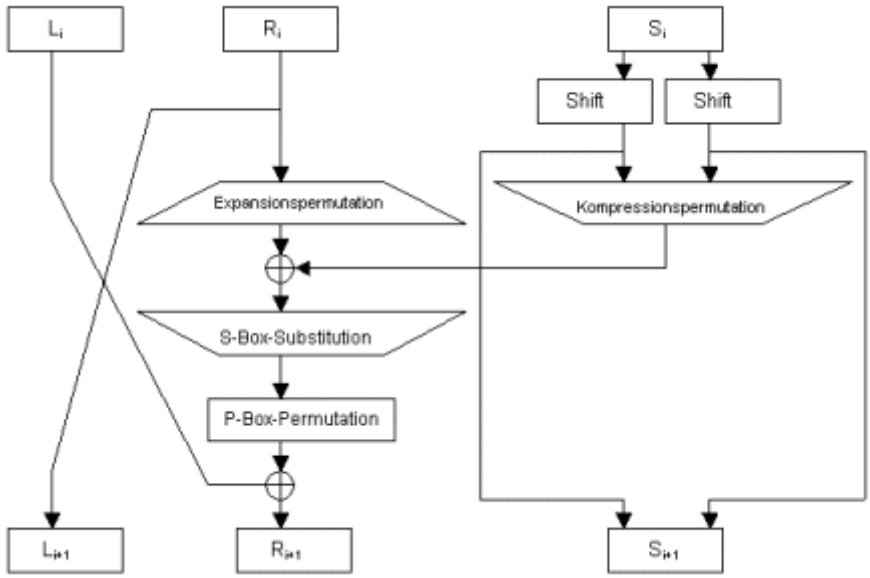


Abbildung: Eine Runde von DES

Ein wesentlicher Punkt beim Design von Chiffrieralgorithmen ist, dass diese eine Bijektion darstellen müssen, um von Chiffretext wieder auf den Klartext schließen zu können und dass sie kryptographisch sicher sind. Der Vorteil von Feistel-Netzwerken liegt darin, dass man sich nur noch um die kryptographische Sicherheit bemühen muss, während die 1-1-Beziehung zwischen Ver- und Entschlüsselung automatisch durch diese Methode gegeben ist.

Auf der untersten Stufe besteht der Algorithmus aus einer vom Schlüssel abhängige Kombination der zwei Grundtechniken der Verschlüsselung: nämlich Konfusion und

Diffusion. Dabei bedeutet Konfusion die Verschleierung des Zusammenhangs zwischen Klartext- und Chiffretextzeichen und die Diffusion verteilt die im Klartext enthaltenen Informationen auf den Geheimtext.

Ein stärkeres Kriterium als die Diffusion ist der Lawineneffekt (*avalanche-effect)*. Hierbei soll jedes Bit des Chiffretextblocks von jedem Bit des Klartextblocks und jedem Bit des Schlüssels abhängen. DES besitzt durch die Expansionspermutation und die Kompressionspermutation die erwünschte Eigenschaft des Lawineneffekts. Bei der ersten werden die 32 Bit des Blocks R_i auf einen 48-Bit-Block erweitert, während durch die Kompressionspermutation die 56 Bit des Schlüssels auf 48 Bit reduziert werden. Jedes geänderte Schlüssel- oder Klartextbit soll nach möglichst wenigen Runden alle Chiffretextbits beeinflusst haben.

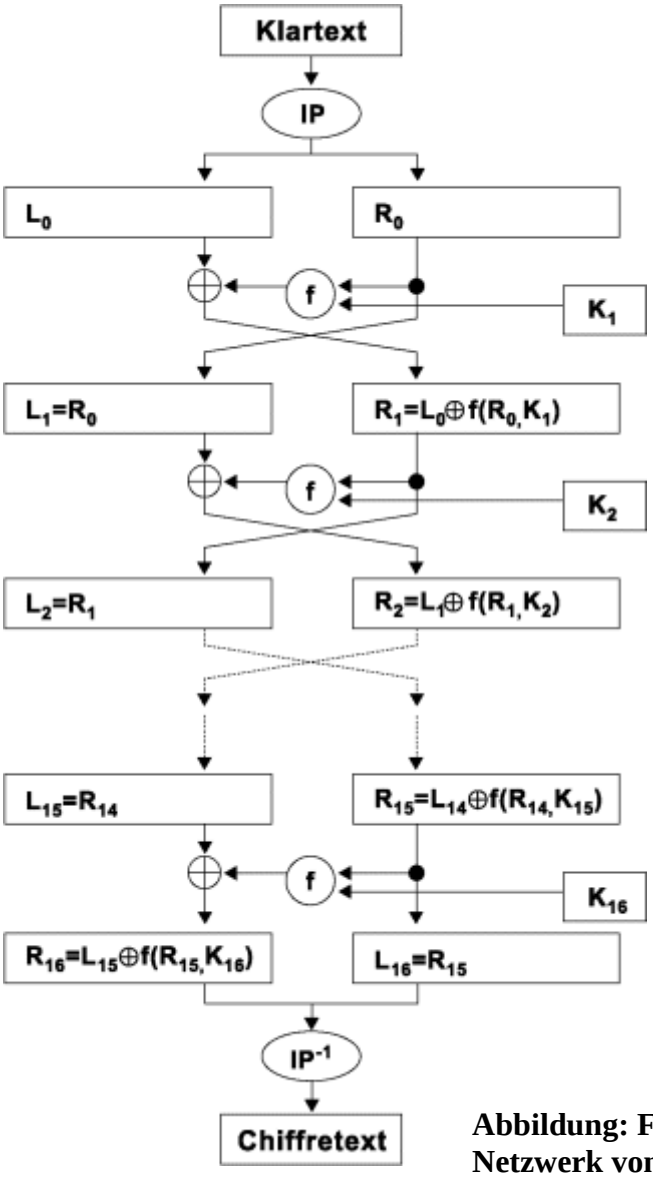


Abbildung: Feistel-Netzwerk von DES

Das Feistel-Netzwerk bei DES besteht aus 16 Runden. Nach einer Eingangspermutation (IP) wird der 64 Bit lange Block des Klartextes in eine jeweils 32 Bit lange rechte Hälfte R_1 und linke Hälfte L_1 zerlegt. Jetzt folgen 16 Runden identischer Operationen, in denen die Daten mit dem Schlüssel kombiniert werden.

Der Schlüssel wird zunächst in zwei 28 - Bit - Hälften geteilt (wobei die übrigen 8 Bit als Paritätsbits verwendet werden). Die DES – Software rotiert den Schlüssel in Abhängigkeit von der Rundennummer um ein oder zwei Bit zyklisch. Nach einer festen Kompressionsfunktion werden anschließend 48 Bit für den Rundenschlüssel S_i ausgewählt.

Auf R_i wird eine Rundenfunktion f mit dem Rundenschlüssel S_i angewendet. Das Ergebnis hiervon wird mit L_i durch die XOR – Funktion verknüpft. In der nächsten Runde werden dann die Ausgaben L_{i+1} und R_{i+1} vertauscht und obiges wiederholt. Nach den 16 Runden wird das Ergebnis einer Schlußpermutation, die zur Eingangspermutation invers ist, unterworfen.

Die Rundenfunktion f besteht aus einer Expansionsfunktion, wobei durch Verdoppelung einiger Bits die 32 Eingabe - Bits auf 48 Ausgabe – Bits erweitert werden. Anschließend werden diese 48 Bit mit den 48 Bit des Rundenschlüssels XOR – verknüpft. Bis zu dieser Stelle ist der Prozeß kryptographisch nicht besonders sicher. Die dann folgenden S-boxen bilden das Herz des Algorithmus. Durch 8 parallele S-Boxen werden 48 Bit auf 32 Bit reduziert, welche dann mit dem unverändert gebliebenen Block L_i XOR-verknüpft werden.

Die Entschlüsselung eines Blocks geschieht mit dem gleichen Algorithmus, d.h. man kann die gleichen Funktionen benutzen mit dem Unterschied, dass man die Schlüssel in umgekehrter Reihenfolge anwenden muss.

Sicherheit

Für den DES-Algorithmus ist die Existenz schwacher Schlüssel bekannt.

Diese theoretische Schwäche ist allerdings für praktische Angriffe kaum geeignet und DES hat sich trotz intensiver Analysen als sehr robust erwiesen.

Aus heutiger Sicht ergeben sich dennoch kryptographische Mängel: der Schlüsselraum von DES umfasst nur 2^{56} Schlüssel. Diese bisher einzig bekannte echte Schwäche ist allerdings verheerend: die zu geringe Schlüssellänge und die ständig schnellere Hardware ermöglichen es, den gesamten Schlüsselraum systematisch abzusuchen.


[FIPS 46]

- **Triple-DES**

Eine Variante von DES ist der sogenannte Triple-DES. Um ein vollständiges Absuchen des Schlüsselraums zu verhindern, verwenden manche DES-Implementationen Triple-DES, d.h. der DES-Algorithmus wird dreimal mit verschiedenen Schlüsseln angewendet. Dabei nutzt man aus, dass DES keine mathematische Gruppe ist, d.h. nach Anwenden der drei Schlüssel landet man in einem anderen Schlüsselraum.

Die Dreifachverschlüsselung mit zwei Schlüsseln funktioniert folgendermaßen: ein Block wird erst mit dem ersten Schlüssel chiffriert, dann mit dem zweiten dechiffriert und dann wieder mit dem ersten chiffriert. Dieses Verfahren wird mit als *encrypt – decrypt – encrypt* (EDE)-Modus bezeichnet, welcher zur Verbesserung von DES für die Standards X9.17 und ISO 8732 modifiziert wurde [ANSI X9.17],[ISO 8732].

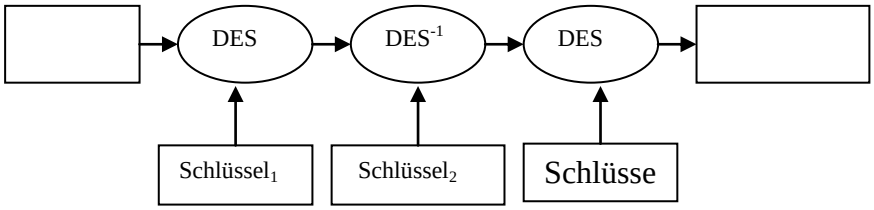


Abbildung: EDE-Modus von Triple-DES

Triple-DES findet unter anderem bei der Berechnung der neuen EC-PIN und beim Homebanking-Standard (HBCI) Anwendung. Für Multimedia-Applikationen ist er allerdings zu langsam.

[FIPS 46]

- **AES**

Das *National Institut of Standards and Technology* (NIST) hat einen DES-Nachfolger als neuen Standard für symmetrische Verschlüsselungsverfahren festlegt: AES, A*dvanced Encryption Standard.* Das Hauptziel ist die Festlegung eines *Federal Information Processing Standard* (FIPS). Am 12. September 1997 wurde ein formaler Aufruf für die Algorithmen veröffentlicht. Bestimmte Auflagen gaben für den Algorithmus unter anderem vor:

- ein symmetrisches Kryptographieverfahren
- eine Blockchiffre
- eine Blockgröße von 128 Bit
- eine Schlüsselgröße von 128, 192 und 256 Bit

Die AES-Finalisten der dritten und letzten Runde sind

- MARS
- RC6
- Rijndael
- Serpent
- Twofish

 [AES99]

- **CAST-128**

CAST ist nach seinen Entwicklern Carlisle Adams und Stafford Tavares benannt und wurde am 23. April 1996 zum Patent angemeldet. Die symmetrische Blockchiffre CAST ist wie der DES-Algorithmus ein Feistel-Netzwerk. Als ein Vorteil gegenüber DES gilt das Fehlen von schwachen Schlüsseln.

Anwendung findet CAST in *PGPhone* und in einem an der Universität Mannheim entwickelten sicheren Videokonferenzsystem. Auch Northern Telecom, IBM, Tandem und Microsoft verwenden CAST in ihren Produkten. Außerdem ist CAST die Standardchiffre in *PGP*.

 [RFC 2144]

- **Blowfish**

Blowfish wurde von dem bekannten Kryptographen Bruce Schneier entwickelt und im Dezember 1993 vorgestellt. Blowfish ist wie DES ein symmetrisches Verschlüsselungsverfahren mit einem Feistel-Netzwerk. Allgemein als kryptographisch stark empfundene Design-Kriterien untermauern das allgemeine Vertrauen in Schneiers Algorithmus und bis heute ist kein für die Praxis relevanter Angriff veröffentlicht.

Wegen bestimmten Modifikationen und der freien Verfügbarkeit ohne Lizenzgebühren ist Blowfish weit verbreitet. Die bekanntesten Produkte sind wohl die beiden Programme für abhörsicheres Telefonieren *PGPhone* und Nautilus.

 [Sch94]

1. 2 Hashfunktionen & MACs

Im allgemeinen kann man nicht davon ausgehen, dass die Verschlüsselung einer Nachricht automatisch deren Authentizität oder Integrität garantiert. Deshalb beschäftigt sich dieser Abschnitt mit Funktionen, die für solche Zwecke einzusetzen sind. Dabei handelt es sich zum Teil um schlüssellose Funktionen, d.h. für die Bearbeitung sind keine geheimen Parameter notwendig.

Hashfunktionen werden häufig für die Überprüfung der Integrität von Daten und MACs für die Sicherstellung der Authentizität eingesetzt.

1. 2. 1 Hashfunktionen

Eine Hashfunktion komprimiert Daten in einer bestimmten, praktisch schwer umkehrbaren Art. Für die exakte Definition braucht man den Begriff der Einwegfunktion: so wird eine Funktion f bezeichnet, die schnell berechenbar ist, für die es aber praktisch nicht möglich ist, zu gegebenem Funktionswert y ein Argument x mit der Eigenschaft f(x)=y zu bestimmen. Also sind dies Funktionen, die sich leicht berechnen lassen, ihre Umkehrung aber erheblich schwieriger ist.

Eine (kryptographische) Hashfunktion ist eine Einwegfunktion, die für eine Eingabe beliebiger Länge eine - in der Regel kürzere Ausgabe mit fest vorgegebener Länge (den Hashwert) liefert. Diese Kompressionseigenschaft von Hashfunktionen ist in vielen Anwendungen nützlich. Als Beispiel sei die Berechnung elektronischer Unterschriften genannt, bei der anstelle der gesamten Daten nur deren Hashwert dem Signaturprozess unterworfen wird.

Zusammenfassend ist eine kryptographische Hashfunktion also ein Algorithmus, der zu einer Nachricht einen nicht manipulierbaren Prüfwert („Fingerabdruck“) fester Länge erzeugt. Der einzige Parameter einer Hashfunktion ist die Nachricht selbst, d.h. in die Berechnung des Hashwerts geht kein Schlüssel ein.

Beim heutigen Stand der Technik sollte eine Hashfunktion mit einem Hashwert von mindestens 160 Bit Länge verwendet werden.

Das Hauptproblem beim Entwurf einer Hashfunktion besteht darin, die Anforderungen an die Sicherheit des Algorithmus mit dem Ziel einer effizienten Berechenbarkeit in Einklang zu bringen. Man fordert deshalb von einer Hashfunktion h folgende Eigenschaften:

a) Der Hashwert h(M) ist bei gegebenem Input M „leicht“ zu berechnen.

b) Es ist praktisch nicht möglich, bei gegebenem Hashwert X eine Nachricht M zu bestimmen, die diesen Hashwert liefert, d.h. mit h(M) = X.

Man kann die Forderung in b) verschärfen:

b´) Es ist praktisch nicht möglich, zwei beliebige Nachrichten M und M´ zu finden, deren Hashwerte h(M) und h(M´) übereinstimmen.

Eine Hashfunktion mit den Eigenschaften a) und b´) wird häufig als kollisionsfrei oder kollisionsresistent bezeichnet.

Generell sind für Hashfunktionen die folgenden Angriffsmöglichkeiten zu betrachten:

- Zu gegebenem Hashwert sucht man eine Nachricht mit identischem Hashwert.
- Man sucht zwei Nachrichten mit identischem Hashwert.

Der zweite Angriff ist also eine Attacke auf die Kollisionsresistenz und das er einfacher ist beruht auf dem sogenannten Geburtstagsparadoxon dem Standardproblem in der Statistik, zwei Leute zu finden, die am gleichen Tag (egal welchen) Geburtstag haben. Jemanden zu finden, der am gleichen Tag (also einem bestimmten) wie Sie Geburtstag hat, entspricht dem ersten Angriff und ist schwieriger zu lösen. Diese grundsätzliche Problematik ist zu beachten, wenn man die Sicherheit von Systemen beurteilen will, in denen Hashfunktionen eingesetzt werden: für einen Angriff auf Hashfunktionen mit 160 Bit Hashwertlänge sind um die 2^{80} Operationen notwendig, um Kollisionen zu erzeugen.

1. 2. 1. 1 Allgemeine Konstruktionsprinzipien

Es gibt verschiedene Vorgehensweisen, eine Hashfunktion zu konstruieren. Bei der verbreitesten arbeitet der Algorithmus folgendermaßen:

- Die Nachricht wird in eine Sequenz von gleich großen Blöcken passender Länge aufgeteilt. Bei Bedarf wird der letzte Block nach einem vorgegebenem Muster aufgefüllt (das sogenannte *Padding*).

- Der zugrundeliegende Algorithmus wird mit einem bestimmten, festen Initialisierungswert IV (*Initial Value*) gestartet.

- Die Nachrichtenblöcke werden dann nacheinander verarbeitet, indem sie als Input für eine Kompressionsfunktion benutzt werden. Der jeweilige Output dieser Funktion ist dann der Initialisierungswert für deren nächste Anwendung.

- Der Hashwert ist der letzte Output der Kompressionsfunktion.

Diese Vorgehensweise basiert auf Runden, d.h. die Kompressionsfunktion wird mehrfach angewendet.

Daraus ergibt sich typischerweise die erste Klasse von Hashfunktionen, die für 32-Bit Architekturen entwickelt wurden, also für den Einsatz auf PCs. Der Vorreiter war MD4, dem MD5, RIPEMD-128, RIPEMD-160, SHA-0, SHA-1, SHA-256, SHA-384 und SHA-512 folgten.

Das ursprüngliche Design-Ziel der Entwickler war es, dass man als Angriff einen Geburtstagsangriff durchführen muss: wegen der 128-Bit-Ausgabe sind ca 2^{64} Operationen erforderlich, um zwei verschiedene Nachrichten mit identischem Hashwert zu konstruieren, bzw. 2^{128}-1 Operationen, um zu einem gegebenen Hashwert eine passende Nachricht - d.h. eine mit dem gleichen Hashwert zu bestimmen.

1. 2. 1. 2 Die in der cv act *library* enthaltenen Hashalgorithmen

MD2

MD4

MD5

RIPEMD-128

RIPEMD-160

SHA-0

SHA-I

SHA-224, SHA-256, SHA-384 und SHA-512

Zusammenfassung

- **MD2**

MD2 ist eine 1989 von Ron Rivest entwickelte Hashfunktion. Das Kürzel MD steht für *Message Digest* und wurde für eine 8-Bit Architektur optimiert, weshalb MD2 langsamer als die meisten anderen Algorithmen ist. MD2 ist in RFC 1319 standardisiert und wird in den PEM1-Protokollen benutzt.

 [RFC 1319]

- **MD4**

MD4 wurde 1990 von R.L. Rivest entwickelt. Obwohl relativ schnell Schwächen im Design sichtbar wurden, war die Idee wegweisend: MD4 ist der Vorläufer fast aller Hashfunktionen und basiert auf dem oben beschriebenen Konstruktionsprinzip. Bei den Entwicklungen der nachfolgenden Verfahren wurde jeweils auf die Erkenntnisse zurückgegriffen, die sich beim Angriff auf die aktuellen Algorithmen ergaben.

Von der Verwendung von MD4 muss abgeraten werden, da potentielle Angriffsmöglichkeiten bekannt sind und schon zum Erfolg geführt haben. 1995/96 wurden neue Methoden zur Analyse von Hashfunktionen der MD4-Familie entwickelt. Die Ergebnisse findet man in [Dob96a], [Dob96b] und [Dob97].

 [RFC1320]

---

1 Privacy Enhanced Mail

- **MD5**

Diese Weiterentwicklung von MD4 wurde von R.L. Rivest 1991 vorgestellt. Dieses war wohl die am weitesten verbreitete Funktion dieser Art und wurde u.a. in *PGP* verwendet.

Auch bei MD5 sind aufgrund der großen Fortschritte bei der Kryptoanalyse einige Schwachstellen gefunden worden. Es kann für Einsatzzwecke, wo keine Kollisionsresistenz gefordert wird, noch eingesetzt werden.

 [RFC 1321]

- **RIPEMD-128**

Eine weitere Hashfunktion mit einem ähnlichen Design wie dem von MD4 ist das im Rahmen des 1992 von der EG geförderten Forschungsprojekts RIPE (*RACE2 Integrity Primitives Evaluation*) entwickelte Hashverfahren RIPEMD.

Für RIPEMD-128 kann man Kollisionen für die Kompressionsfunktion finden und der Geburtstagsangriff ist realisierbar, wenn auch mit hohen Kosten. Deshalb ist von der Anwendung abzuraten.

 [DBP96]

- **RIPEMD-160**

Dieses ist die Weiterentwicklung von RIPEMD-128 mit einem Hashwert von 160 Bit und wurde aufgrund des Sicherheitsaspekts entwickelt, dass eine Hashfunktion mit 128 Bit nicht mehr als sicher gelten kann. Auf Grund der aktuellen Veröffentlichungen zu Angriffsmöglichkeiten auf SHA-1 wird vom Einsatz einer Hashfunktion mit 160 Bit abgeraten.

---

2 *RACE* ist wiederum die Abkürzung von *Research and Development in Advanced Communication Technologies*

 [DBP96]

- **SHA-0**

Der *Secure Hash Algorithm* (SHA-0) wurde von der *National Security Agency* (NSA) entworfen. Er wurde 1992 vom NIST veröffentlicht und ist in den USA seit Mai 1993 ein *Federal Information Processing Standard* (FIPS). Entwickelt wurde SHA-0 als Hash-Algorithmus für den Signaturalgorithmus DSA.Früher hieß dieses Verfahren nur SHA, was öfters zu Missverständnissen geführt hatte, weshalb SHA immer häufiger SHA-0 genannt wird. SHA-0 wird nur noch aus Gründen der Rückwärtskompatibilität in kryptographischen Bibliotheken aufgenommen.

- **SHA-1**

SHA-0 wurde ohne Angabe von Gründen von den Entwicklern modifiziert und SHA-1 genannt. Auf Grund der aktuellen Veröffentlichungen zu Angriffsmöglichkeiten auf SHA-1 wird vom Einsatz einer Hashfunktion mit 160 Bit abgeraten.

- **SHA-224, SHA-256, SHA-384 und SHA-512**

Diese drei (SHA-256, SHA-384, SHA-512) Hashalgorithmen wurden im Novemver 2000 vom NIST vorgeschlagen und die entsprechenden Hashwerte der Länge 256, 384 bzw. 512 Bit gelten zur Zeit als sicher. Sie sind seit 2001 Teil des Federal Information Processing Standard (FIPS).SHA-224 wurde 2002 in den FIPS Standard aufgenommen und ist bis auf andere Initialwerte und die Verkürzung des Hashwertes auf 224 Bit mit SHA-256 identisch.

- **Zusammenfassung**

Name	Bitlänge	Runden x Schritte pro Runde

MD4	128	3 x 16
MD5	128	4 x 16
RIPEMD-128	128	4 x 16 zweimal (parallel)
RIPEMD-160	160	5 x 16 zweimal (parallel)
SHA-0	160	4 x 20
SHA-1	160	4 x 20
SHA-2242244 x16		
SHA-256	256	4 x16
SHA-384	384	5 x 16
SHA-512	512	5 x 16

Tabelle: Vergleich der Hashfunktionen, die auf dem Design von MD4 basieren

Nach dem heutigen Stand der Analyse kann man von den Hashfunktionen nur noch

- SHA-256, SHA-384 und SHA-512

ohne Vorbehalte und Einschränkungen für den Einsatz bei digitalen Signaturen empfehlen, also bei Verfahren, die eine kollisionsresistente Hashfunktion voraussetzen.

1. 2. 2 Authentifizierungs-Codes (MAC)

Für eine elektronische Authentifizierung von Daten wird eine Nachricht M um spezielle redundante Informationen erweitert, die mittels eines kryptographischen Verfahrens aus M berechnet und zusammen mit der Nachricht gespeichert bzw. übertragen werden. Damit ein Angreifer die angefügte Redundanz nicht gezielt modifizieren kann, muss diese in geeigneter Weise geschützt sein.

Erfolgt die Ermittlung der redundanten Informationen unter Verwendung eines geheimen Schlüssels, so spricht man von einem *Message Authentication Code* (MAC). Deshalb wird ein MAC auch als Hashfunktion mit einem zusätzlichen geheimen Schlüssel bezeichnet. . Die Theorie der MACs unterscheidet sich nicht von der Theorie der Hashfunktionen, außer dass der Hashwert nur mit dem entsprechenden Schlüssel verifiziert werden kann.

Für einen MAC werden folgende Eigenschaften gefordert:

a) MAC(K,M) ist bei gegebener Nachricht M und Schlüssel K „leicht“ zu berechnen.

b) Es ist praktisch nicht möglich, aus der Nachricht M und dem zugehörigen Authentifizierungs-Code MAC(K,M) den Schlüssel K zu bestimmen.

c) Es ist praktisch nicht möglich, bei gegebener Nachricht M und gegebenem MAC(K,M) eine von M verschiedene Nachricht M´ zu finden, die denselben Authentifizierung-Code wie M liefert, d.h. mit MAC(K,M) = MAC(K,M´).

In jedem Fall basiert der Nachweis der Integrität einer Nachricht auf der Geheimhaltung bzw. der Integrität des kryptographischen Schlüssels. Der einfachste Fall besteht darin, den Hashwert mit einem symmetrischen Algorithmus zu verschlüsseln. Ein potentielles Sicherheitsproblem besteht darin, dass der Empfänger den Schlüssel kennen muss. Mit diesem Schlüssel kann er aber andere Nachrichten mit demselben Hashwert erzeugen.

Ein MAC lässt sich aus einer Hashfunktion oder einer symmetrischen Blockchiffre (CBC-MAC) erzeugen.

[ISO 9797]

1. 2. 2. 1 Die in der cv act *library* enthaltenen MACs

Bei beiden, im folgenden vorgestellten Verfahren müssen die Kommunikationspartner über einen gemeinsamen Schlüssel verfügen.

Dieser muss vorab mit einem geeigneten Schlüsselaustauschverfahren vereinbart werden.

- **MAC basierend auf Hashfunktionen**

Man erzeugt die Ausgabe des MACs (im folgenden *Tag* genannt), indem man Schlüssel, Nachricht und wiederum den Schlüssel zusammenfasst und dies einer Hashfunktion als Eingabe übergibt. Diese Zusammenfassung (Konkatenierung) wird mit dem Operator II bezeichnet.

Der *Tag* ergibt sich durch die Berechnung von H (K II M II K).

Die Sicherheit dieses MAC-Verfahrens hängt von der Geheimhaltung des Schlüssels und vom verwendeten Hashverfahren ab.

 [Tsu92]

- **CBC-MAC**

Eine andere und sehr einfache Möglichkeit zur Konstruktion eines MAC ist die Verschlüsselung einer Nachricht mit einem Blockalgorithmus im CBC-Modus. Der letzte Chiffreblock wird als *Tag* verwendet.

Die Sicherheit des CBC-MAC hängt von der Geheimhaltung des Schlüssels und vom verwendeten symmetrischen Verfahren ab.

 [ANSI X9.9], [ANSI X9.19] [ISO 9797] und [ISO 8731]

1. 3 Asymmetrische Kryptographie

1. 3. 1 Mathematische Grundlagen

Im folgenden werden erst die mathematischen Grundlagen, die für das weitere Verständnis notwendig sind und dann einige mathematischen Probleme, welche in der asymmetrischen Kryptographie Anwendung finden, kurz vorgestellt.

1. 3. 1. 1 Modulares Rechnen und endliche Körper

Modulares Rechnen ist aus der Schule bekannt. „Klein Hänschen soll um 12.00 Uhr zu Hause sein und verspätet sich um 13 Stunden - um welche Zeit ist er nach Hause gekommen?“ Bei dieser „Uhrenarithmetik“ gibt es keine 25 Uhr, sondern nur 1 Uhr. Man rechnet 25 modulo 24 und erhält 1. Anders ausgedrückt ergibt sich

$$25 \equiv 1 \bmod 24.$$

Es gilt $a \equiv b \bmod n$, falls es eine ganze Zahl k gibt mit $a = b + k\,n$ und man sagt, dass a kongruent zu b modulo n ist; n bezeichnet man als Modulus. Wenn a nicht negativ ist und b zwischen 0 und n liegt, kann man sich b als Rest vorstellen, der bei der Division von a durch n übrigbleibt.

Für einen Modulus n bezeichnet man die Menge der ganzen Zahlen { 0,1,2, ...,n-1 } mit $\mathbf{Z}_n$. Zusammen mit der Addition modulo n formt diese Menge eine mathematische Struktur, die formal Gruppe genannt wird. Eine mathematisch exakte Definition ist folgende: Eine Gruppe G ist ein mathematisch abstraktes Objekt, das sich aus einer Menge G und einer Operation ° zusammensetzt, die auf einem Paar von Elementen aus der Menge definiert ist.

Diese Operation muss folgende Eigenschaften haben:

1. $a \circ b \in G$ für alle $a, b \in G$ (Abgeschlossenheit)
2. $a \circ (b \circ c) = (a \circ b) \circ c$ für alle $a, b, c \in G$ (Assoziativität)
3. Es gibt ein $e \in G$ mit $e \circ a = a \circ e = a$ für alle $a \in G$.
   Dieses Element wird neutrales Element genannt.
4. Zu jedem $a \in G$ gibt es ein $b \in G$ mit $a \circ b = b \circ a = e$.
   Dieses Element b heißt inverses Element zu a und wird mit mit a^{-1} bezeichnet

.

Die Ordnung einer Gruppe G ist die Anzahl der Elemente der Gruppe G. Die Ordnung eines Elements einer Gruppe $g \in G$ ist die kleinste, positive, ganze Zahl n mit $g^n = e$, wobei e das neutrale Element der Gruppe ist.

Beispiel

Um dies zu verdeutlichen, schauen wir uns die Gruppe

$$\mathbf{Z}_p := \{ 0, 1, 2, ..., p-1 \}$$

der ganzen Zahlen modulo p an, wobei die Operation ° die Addition modulo p sei.
Diese Gruppe hat die Ordnung p.

Das neutrale Element bezüglich dieser Addition ist die 0.
Die multiplikativ geschriebene Gruppe

$$\mathbf{Z}_p^* := \{ 1, 2, ..., p-1 \},$$

mit der Multiplikation modulo p hat die Ordnung p-1
und das neutrale Element bezüglich der Multiplikation ist 1.
Speziell für p=7 ergibt sich das Beispiel

$$\mathbf{Z}_7^* := \{ 1, 2, 3, 4, 5, 6 \},$$

d.h. die Ordnung der Gruppe $\mathbf{Z}_7^*$ ist 6 und das Element
g=4 hat die Ordnung 3, denn

$$4^1 \equiv 4 \mod 7,$$
$$4^2 \equiv 16 \mod 7 \equiv 2 \mod 7$$
$$4^3 \equiv 64 \mod 7 \equiv 1 \mod 7.$$

Für eine Gruppe ist immer eine Operation definiert, aus der man aber andere Operationen herleiten kann. Hat man z.B. eine additive Gruppe, kann man auch immer subtrahieren, nämlich über die Addition mit dem Inversen und die Multiplikation ist die wiederholte Anwendung der Addition: 4*g = g+g+g+g.

Wenn man in einer Gruppe Inverse bezüglich der Multiplikation und der Addition für jedes von 0 verschiedene Element finden kann, spricht man von einem Körper. Also kann man in dieser Struktur sowohl addieren (und subtrahieren), als auch multiplizieren (und dividieren).

Die Gruppe $\mathbf{Z}_p$ mit einer Primzahl p als Modulus formt zusammen mit der Addition und der Multiplikation modulo p einen endlichen Körper, in dem die üblicherweise bekannten Rechenregeln für Addition, Subtraktion, Multiplikation und Division gelten. Das Wort „endlich“ resultiert daraus, dass er eine endliche Anzahl von Elementen besitzt. Eine andere Bezeichnung für $\mathbf{Z}_p$ ist auch GF(p) von *Galois Field.*

Die Vorteile der modularen Rechnung sind u.a., dass die Länge der zu betrachtenden Zahlen durch die Länge des Modulus begrenzt wird. Besonders die in der Kryptographie weit verbreitete Operation der Exponentation oder Potenzierung von Elementen profitiert von diesem Umstand. Trotzdem ist auch bei dieser der Art der Arithmetik für die Verwendung innerhalb der Kryptographie der Einsatz einer sogenannten Langzahlarithmetik notwendig, d.h. einer Arithmetik, die in der Lage ist, mit Zahlen zu rechnen, die den üblichen Darstellungsbereich von Zahlen auf Computern übersteigen.

1. 3. 1. 2 Das Knapsack-Problem

Dies war wohl einer der ersten Versuche, ein asymmetrisches Kryptosystem zu konstruieren. Man hat einen Rucksack (*knapsack*), welcher exakt eine vorgegebene Masse fassen kann, sowie eine unbestimmte, große Anzahl von Objekten mit verschiedenen Massen. Die Frage ist, welche Objekte man auswählen muss, um den Rucksack optimal zu füllen.

Die Lösung dieses Problems ist für eine große Anzahl an zur Auswahl stehenden Objekten kaum zu berechnen. Das erste veröffentlichte Konzept [MH78] eines Public Key-Kryptosystems mit diesem Problem als Grundlage wurde dennoch nach relativ kurzer Zeit geknackt. Allerdings war damit das Interesse an asymmetrischen Verfahren geweckt und es dauerte nicht lange, bis ein Algorithmus entwickelt wurde, der auch heute noch den Standard innerhalb der Public Key-Kryptographie darstellt. Grundlage dafür ist das folgende Problem:

1. 3. 1. 3 Das Problem der Faktorisierung

Beim Problem des Faktorisierens (IF- von *Integer Factorization*) geht es um die Tatsache, dass sich zwei große Primzahlen (in der Größenordnung von über 100 Dezimalstellen) relativ problemlos multiplizieren lassen, die Umkehrung jedoch, also die Bestimmung der Primfaktoren bei einem vorgegebenen Produkt, ist ungleich schwieriger.

Trotz gewisser Angriffsmöglichkeiten basiert auf diesem Problem das wohl bekannteste Public Key-Verfahren, der 1978 veröffentlichte RSA-Algorithmus. Dieser beruht auf der (allgemein akzeptierten) Vermutung, dass für großes zusammengesetztes n die Potenzierung einer Zahl modulo n eine Falltürfunktion (*trapdoor*-Funktion) darstellt. Die für die Invertierung notwendige geheime Information besteht dabei gerade aus der Kenntnis der Faktorisierung von n.

Ein Angriff auf dieses Verfahren der Verschlüsselung über die Faktorisierung des Modulus bei den heute verwendeten Schlüssellängen von 1024 Bit ist wohl für die nächste Zeit unrealistisch: Der momentan gültige Rekord für die längste Zahl, welche mit einem allgemein einsetzbaren Algorithmus faktorisiert wurde, liegt bei 155 Dezimalstellen, also bei 512 Bit.

1. 3. 1. 4 Das Problem des Diskreten Logarithmus

Bei dem Problem des Diskreten Logarithmus (DL) versucht man die Umkehrung der Exponentation zu berechnen, d.h. zu einem gegebenem $y = g^x$, wobei g allgemein bekannt und x „groß“ ist, den Wert $x = \log_a y$ zu berechnen. Wesentlich dabei ist, dieses Problem in einer geeigneten, diskreten Struktur zu betrachten (daher der Begriff). In Strukturen wie den reellen Zahlen stellt die entsprechende Berechnung keine größeren Schwierigkeiten dar.

Zusammenfassend hat das DL-Problem in GF(p) also die Form:

> Bei gegebenem a, b und p bestimme x mit $a^x \equiv b \bmod p$.

Die Zahl x heißt diskreter Logarithmus von b. Für eine mathematisch exakte Definition und weiterführende Erklärungen siehe die Bücher von Koblitz [Kob94] oder Forster [For96].

Ein Verfahren, das auf dem DL-Problem basiert, ist der Unterschriftenalgorithmus von ElGamal, der eine Variation des ursprünglichen Schlüsselaustauschs nach Diffie und Hellman ist, der nun beschrieben wird.

1. 3. 2 Schlüsselaustausch

Eines der Hauptprobleme bei der symmetrischen Verschlüsselung besteht darin, dass Alice und Bob beide über denselben geheimen Schlüssel verfügen müssen. Sie müssen einen geheimen Schlüssel austauschen, bevor sie verschlüsselte Nachrichten verschicken können. So ist der direkte Transport des Schlüssels über digitale Netze problemlos nicht möglich, da er in unbefugte Hände gelangen kann; verschlüsselt übertragen kann man den Schlüssel natürlich auch nicht, da es noch keinen gemeinsamen Schlüssel für die Dechiffrierung gibt.

Es gibt Methoden, wie Alice und Bob sich dennoch über einen geheimen Schlüssel verständigen können, ohne dass ein Dritter Kenntnis von dem Schlüssel erhält.

1. 3. 2. 1 Die in der cv act *library* enthaltenen Schlüsselaustauschprotokolle

- **Schlüsselaustausch nach Diffie-Hellman**

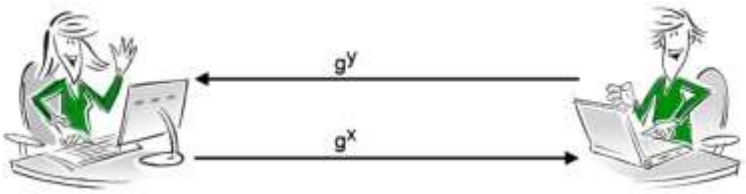
Das Schlüsselaustausch-Protokoll von Diffie-Hellman ist das bekannteste Protokoll und seine Vorstellung im Jahre 1976 kann als die Geburtsstunde der Public Key-Kryptographie bezeichnet werden [DH76].

Es eignet sich für die Erzeugung und Verteilung von Schlüsseln, jedoch nicht zur Ver- oder Entschlüsselung. Es beruht auf dem DL-Problem und die Schlüsselvereinbarung hat folgende Form:

Alice und Bob einigen sich auf eine geeignete Gruppe, in der das DL-Problem rechentechnisch nicht beherrschbar ist. Außerdem wird ein festes Gruppenelement g gewählt, das wie die Gruppe selber öffentlich bekannt sein darf:

- Alice wählt eine zufällige Zahl x, berechnet g^x und sendet dies an Bob
- Bob wählt eine zufällige Zahl y, berechnet g^y und sendet dies an Alice
- Alice bildet $(g^y)^x = g^{yx}$
- Bob bildet $(g^x)^y = g^{yx}$

Dann ist g^{yx} ihr gemeinsamer, geheimer Schlüssel für ein symmetrisches Verfahren.



Der beste bekannte Ansatzpunkt für einen allgemeinen Angriff auf dieses Schema ist, das Problem des diskreten Logarithmus in der zugrundeliegenden Gruppe zu lösen, d.h. für ein gegebenes f und g den passenden Exponenten e in $f=g^e$ zu finden.

Für eine konkrete Implementierung in $\mathbf{Z}_p$ berechnet man g^x mod p, bzw. g^y mod p mit x und y kleiner als p. Die Sicherheit dieses Verfahrens beruht auf der Wahl von p und g. Die Primzahl p sollte entsprechend groß sein und (p-1)/2 sollte auch eine Primzahl sein, die im folgenden q genannt wird. Für g wählt man dann ein erzeugendes Element der Untergruppe von Ordnung q in $\mathbf{Z}_p$*.

Dieses Protokoll zählt zu den asymmetrischen Verfahren, da man die Werte g^x bzw. g^y als die öffentlichen sowie x und y als die zugehörigen privaten Schlüssel von Alice und Bob ansehen kann.

Bemerkung: Diffie-Hellman war bis zum 29. April 1997 in den Vereinigten Staaten patentiert.

- **EC-DH**

EC-DH ist die Entsprechung des Protokolls für den Schlüsselaustausch von Diffie-Hellman auf elliptischen Kurven. Anstatt in einer Untergruppe von $\mathbf{Z}_p^*$ zu arbeiten, ist die zugrundeliegende mathematische Struktur eine elliptische Kurve.

 [ElG84], [ElG85]

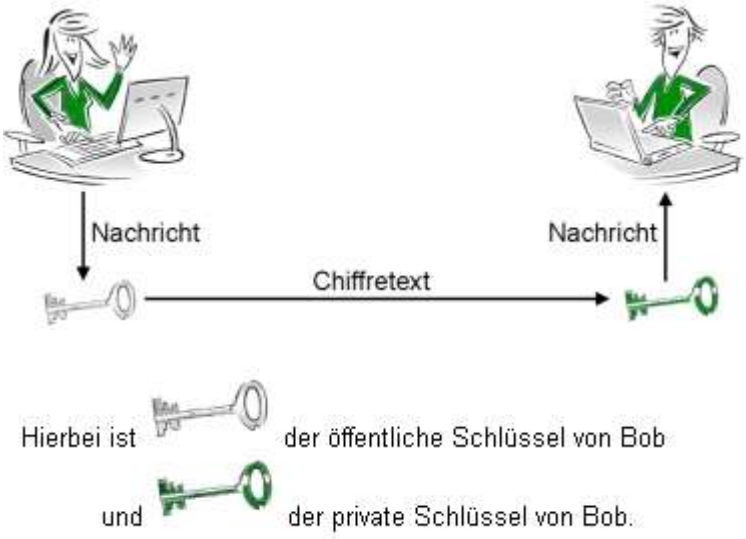
1. 3. 3 Asymmetrische Verschlüsselung

Bei den symmetrischen Verfahren zur Verschlüsselung gab es vor allem zwei Dinge zu beachten:

- Zum Chiffrieren und zum Dechiffrieren einer Nachricht werden derselbe Schlüssel verwendet. Dies bedeutet, dass jeder, der in der Lage ist, eine Nachricht zu verschlüsseln, diese Nachrichten auch entschlüsseln kann.

- Um eine sichere Verbindung zu gewährleisten, müssen sich zwei Kommunikationspartner vorab auf einen gemeinsamen Schlüssel einigen. Der Austausch dieses gemeinsamen Schlüssels könnte abgehört werden.

Diese beiden Umstände und vor allem die Gefahren, welche aus dem zweiten Punkt entstehen, sind deutliche Einschränkungen für die praktische Verwendbarkeit. Bei den nun behandelten Verfahren sieht die Situation dagegen anders aus: Hierbei handelt es sich um ein kryptographisches Verschlüsselungssystem, bei dem die Kommunikationspartner jeweils über ein Schlüsselpaar verfügen. Der eine Schlüssel heißt öffentlicher Schlüssel (*public key)* und wird zum Verschlüsseln benutzt und der andere ist der geheime Schlüssel *(private key)*, mit dem der Empfänger die Nachricht entschlüsselt.



Das Senden einer verschlüsselten Nachricht sieht folgendermaßen aus:

- Alice besorgt sich auf vertrauenswürdige Weise Bobs öffentlichen Schlüssel

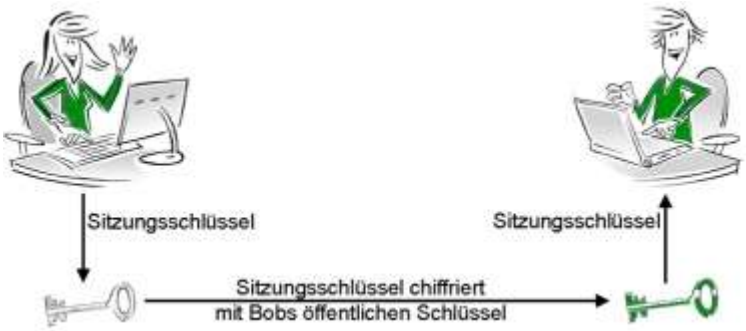
- Alice chiffriert ihre Nachricht mit Bobs öffentlichen Schlüssel und sendet sie an Bob

- Bob dechiffriert die Nachricht mit seinem privaten Schlüssel

Im Gegensatz zu symmetrischen Verfahren entfällt eine der größten Schwachstellen für eine gesicherte Kommunikation, nämlich der Schlüsselaustausch über einen unsicheren Kanal. Der potentielle Empfänger, in diesem Fall Bob, veröffentlicht einfach den für die Verschlüsselung notwendigen Teilschlüssel und behält den anderen, privaten Schlüssel unter Verschluss. Dann kann jeder Daten verschlüsseln, aber nur der rechtmäßige Empfänger kann sie entschlüsseln.

Außerdem verringert sich bei asymmetrischen Verfahren die Anzahl der benötigten Schlüssel: bei n Kommunikationspartnern reicht die Verwaltung einer für jedermann lesbaren Datenbank mit n Schlüsseln (im Gegensatz zu n(n-1)/2 Schlüsseln bei symmetrischen Verfahren). Dabei wird allerdings eine grundsätzliche Schwachstelle dieser Public Key-Verfahren sichtbar: Alice muss in den Besitz von Bobs öffentlichen Schlüssel kommen. Für einen Angreifer ist es ein lohnendes Ziel, die bei der Abfrage dieser Datenbank übermittelten Informationen abzufangen und zu verfälschen. Auf diese Problematik und deren Lösungen wird im Kapitel 2.3 über Public Key-Infrastrukturen eingegangen.

In der Praxis aber sind Public Key-Algorithmen kein Ersatz für symmetrische Algorithmen, da sie um Größenordnungen langsamer sind. Deshalb werden sie beispielsweise zur Chiffrierung von Schlüsseln für symmetrische Verfahren verwendet. Bei diesen sogenannten **hybriden Verfahren** handelt es sich um Kryptographie mit einem öffentlichen Schlüssel zum Schützen und Verteilen von Sitzungsschlüsseln (= Schlüssel nur für eine oder wenige Kommunikationssitzungen). Der Sitzungsschlüssel ist ein geheimer Schlüssel, d.h. man verwendet zur Verschlüsselung ein symmetrisches Verfahren und zum Schlüsselaustausch oder zur Schlüsselübermittlung wird ein asymmetrisches Verfahren verwendet.



Das Protokoll hat dann folgende Form:

- Alice besorgt sich auf vertrauenswürdige Weise Bobs öffentlichen Schlüssel
- Alice generiert einen zufälligen Sitzungsschlüssel, chiffriert ihn mit Bobs öffentlichen Schlüssel und sendet ihn an Bob

- Bob dechiffriert Alices Daten mit seinem privaten Schlüssel, um den Sitzungsschlüssel zu erhalten

- Beide chiffrieren ihre Kommunikation nun mit demselben Sitzungsschlüssel.

 [DH76]

1. 3. 3. 1 Die in der cv act *library* enthaltenen asymmetrischen Verschlüsselungsalgorithmen

RSA

DLIES

EC-IES

- **RSA**

Unter all den asymmetrischen Algorithmen, die im Laufe der Jahre veröffentlicht wurden, ist dieser bei weitem am einfachsten zu verstehen und zu implementieren. Das Verfahren wurde 1976 von den Forschern Ron Rivest, Adi Shamir und Leonard Adleman vorgestellt und nach ihnen benannt.

Beschreibung von RSA

RSA ist ein asymmetrischer Algorithmus, der auf dem IF-Problem beruht.

Zur Erzeugung der beiden Schlüssel wählt man zufällig zwei große Primzahlen p und q, die in etwa gleich lang sein sollten, um maximale Sicherheit zu gewähren und berechnet

$$n = p\,q.$$

Dann wählt man zufällig einen Chiffrierschlüssel e mit e relativ prim zu (p-1)(q-1). Dann berechnet man den Dechiffrierschlüssel d durch

$$e\,d \equiv 1 \bmod ((p-1)(q-1)) \;\Leftrightarrow\; d \equiv e^{-1} \bmod ((p-1)(q-1)) .$$

Der öffentliche Schlüssel besteht dann aus e und n und der private Schlüssel aus d.

Die Verschlüsselung hat dann die Form

$$c \equiv m^e \bmod n$$

und die Entschlüsselung hat die Form

$$m \equiv c^d \bmod n.$$

Dazu wird die Nachricht m in Blöcke m_i zerlegt, die kleiner als n sind. Dann besteht die verschlüsselte Nachricht c aus Nachrichtenblöcken c_i gleicher Größe.

Es sollte beachtet werden, dass die Parameter p, q und e so „zufällig wie möglich“ gewählt und p und q geheim gehalten werden sollten, damit das Verfahren sicher ist.

Beispiel:

An einem kleinen Beispiel wird das Verfahren klarer. Der Buchstabenfolge „cv“ entspricht nach der ASCII-Tabelle das Wort 6376H in hexadezimaler Schreibweise, also 25462D in Dezimaldarstellung. Zur Verschlüsselung dieser Zahl als Block wird ein n gebraucht:

Mit p = 223 und q = 281 ergibt sich

n = pq = 62663.

Der Chiffrierschlüssel e darf keine gemeinsamen Faktoren mit

(p-1)(q-1) = 222 * 280 = 62160

haben. Mit der gewählten Zufallszahl e = 14321 ist dies erfüllt. Damit ist das Inverse von 14321 mod 62160 zu berechnen und mit Hilfe des euclidischen Algorithmus ergibt sich d = 32801. Dies ist der geheim zu haltende, private Schlüssel, während der öffentliche Schlüssel, bestehend aus n = 62663 und e = 14321 veröffentlicht wird und p und q vergessen werden können.
Zur Verschlüsselung von „cv“ =2 5462 braucht dieser Block nicht weiter zerlegt werden, da er kleiner als n ist. Es wird wie folgt verschlüsselt:

$c = 25462^{14321} \bmod 62663 = 15571$

Zur Entschlüsselung der Nachricht muss man die Potenzierung des Chiffretexts mit dem geheimen Dechiffrierschlüssel, also 32801 durchführen:

$m = 15571^{32801} \bmod 62663 = 25462.$

Sicherheit von RSA

Die Kryptoanalyse konnte die Sicherheit von RSA weder beweisen noch widerlegen, aber sie stärkt das Vertrauen in den Algorithmus.

Die Sicherheit beruht auf der Schwierigkeit, große Zahlen zu faktorisieren. Öffentlicher und privater Schlüssel hängen von einem Paar großer Primzahlen ab (mehr als 150 Stellen). Man vermutet, dass die Wiederherstellung des Klartextes äquivalent ist zur Faktorisierung des Produkts der Primzahlen. Es wurde jedoch nie mathematisch bewiesen, dass man n faktorisieren muss, um m aus c und e zu berechnen. Es ist also denkbar, dass eine völlig andere Art, RSA zu kryptanalysieren, entdeckt wird.

Es zeigt sich aber immer deutlicher, dass die derzeit oftmals eingesetzte 512 Bit RSA-Verschlüsselung nicht mehr sicher ist. Am 22. August 1999 hat eine Wissenschaftlergruppe bekanntgegeben, einen Schlüssel dieser Länge knacken zu können.

Das Problem ist, dass sich stärkere RSA-Verschlüsselungen von 2048 Bit oder größer in Smartcards, wie sie oftmals zur Autorisierung und Identifizierung verwendet werden, nur schwer oder gar nicht einsetzen lassen.

Standards und Patente für RSA

RSA ist in weiten Teilen der Welt ein De-Facto-Standard und in einem informativen Anhang zu ISO 9796 enthalten. In den Vereinigten Staaten ist der RSA-Algorithmus patentiert, für das Public Key Partners (PKP) die Lizenzen vergibt. Das US-Patent läuft am 20. September 2000 aus.

Außerdem ist RSA Bestandteil von PKCS, das von RSA Security, Inc. entwickelt wurde. Der Entwurf für einen ANSI-Standard im Bankwesen spezifiziert RSA.

 [RFC 2313], auch [ISO 9796]

- **DLIES**

DLIES ist ein asymmetrisches Chiffrierverfahren, welches aus einem Schlüsselaustauschverfahren und einem MAC besteht und auf dem Diffie-Hellmann Problem basiert. Die frühere Bezeichnung war DHAES, die dann aufgrund der Kollision der Namen mit dem DES-Nachfolger AES geändert wurde.

Die prinzipielle Funktionsweise dieses Schemas kann man der folgenden Figur entnehmen:

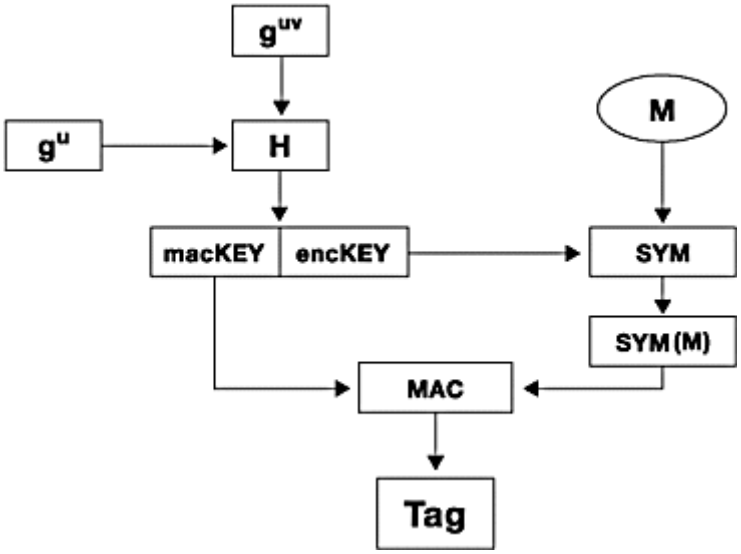


Abbildung: DLIES

Dabei ist die Notation folgende:

- g^u sei ein (eventuell temporärer) öffentlicher Schlüssel von Alice
- g^v sei der öffentliche Schlüssel von Bob
- g^{uv} ist dann das gemeinsame Geheimnis von Alice und Bob
- H sei eine Hashfunktion
- die Ausgabe der Hashfunktion H wird aufgeteilt in macKEY und encKEY
- MAC sei ein Authentifizierungscode
- SYM(M) sei ein symmetrischer Verschlüsselungalgorithmus

Alice möchte Bob eine Nachricht M schicken. Dazu wird ein gemeinsames Geheimnis g^{uv} vereinbart, welches zusammen mit Alice öffentlichem Schlüssel g^u durch eine Hashfunktion H komprimiert wird. Dieser Hashwert wird aufgeteilt in einen Anteil, der in den MAC geht. Der andere Anteil wird zusammen mit der Nachricht M durch einen symmetrischen

Algorithmus zum Chiffretext SYM(M) verschlüsselt, der der zweite Input für den MAC ist. Nach Anwenden von MAC bekommt man als Output das *Tag*.

Die Daten, die Alice an Bob schickt, bestehen dann aus dem öffentlichen Schlüssel g^u, dem Chiffretext SYM(M) und dem *Tag*.

Die Sicherheit von DLIES beruht auf der Schwierigkeit des Diffie-Hellman-Problems und in der Annahme, dass der zugrundeliegende symmetrische Algorithmus, die Hashfunktion und der MAC sicher sind.

 [ANSI X9.63]

- **EC-IES**

EC-IES ist eine Variante des DLIES-Schematas, bei der man über elliptischen Kurven arbeitet. Es wird im IEEE-Standard beschrieben.

 [ANSI X9.63]

1. 3. 4 Digitale Signaturen

Die Signierung eines in digitaler Form vorliegenden Dokuments ist das Äquivalent zu einer eigenhändigen Unterschrift eines auf Papier vorliegenden Dokuments. Bei der zunehmenden Bedeutung der elektronischen Medien ist es notwendig, solche Entsprechung zu der traditionellen Unterschrift zur Verfügung zu haben. Die Probleme dabei bestehen aus folgenden Fragestellungen:

1. Ist das Dokument nach Fertigstellung noch verändert worden?

2. Stammt das Dokument vom vorgegebenen Absender?

Außerdem ist die rechtliche Seite dabei zu beachten. In Deutschland wurde 1997 das Signaturgesetz und eine ergänzende Signaturverordnung (SigG/SigV) verabschiedet, dass unter anderem die rechtlichen Rahmenbedingungen für fälschungssichere digitale Signaturen und für die Wahrung der Rechte der Teilnehmer am elektronischen Rechtsverkehr schafft.

Auch auf EU-Ebene gibt es seit geraumer Zeit diverse Ansätze für Regelungen zu elektronischen Signaturen. Am 19.1.2000 wurde die Richtlinie 1999/93/EG veröffentlicht und trat somit in Kraft. Die EU-Mitgliedsstaaten haben nun bis zum 19. Juli 2001 Zeit, die Richtlinie in nationales Recht umzusetzen.

Es gibt Algorithmen zum Unterzeichnen von Dokumenten mit Public Key-Kryptographie. Bei einem asymmetrischen Verschlüsselungsverfahren wird normalerweise mit dem öffentlichen Schlüssel chiffriert und mit dem privaten dechiffriert. Beim digitalen Signieren wird der Gebrauch der Schlüssel umgedreht:

- Alice chiffriert die Daten mit Hilfe ihres privaten Schlüssels.

- Alice sendet das unterzeichnete Dokument an Bob.

- Bob dechiffriert das Dokument mit Alices öffentlichem Schlüssel, wodurch er die Echtheit der Unterschrift überprüft.

Jedoch gibt es bei großen Dokumenten das Problem, dass das Verschlüsseln mit einem asymmetrischem Verfahren einige Zeit in Anspruch nimmt. Deshalb wird in der Praxis nicht das ganze Dokument verschlüsselt, sondern nur der Hashwert. Dieses stellt eine ähnliche Situation dar wie bei hybriden Verfahren, da es sich dabei auch um eine Kombination von verschiedenen Verfahren handelt.

Die digitale Signatur hat nun folgendes Protokoll:

- Alice chiffriert den Hashwert ihres Dokuments mit ihrem privaten Schlüssel, womit sie das Dokument unterzeichnet

- Alice sendet das Dokument und den signierten Hashwert an Bob

- Bob berechnet den Hashwert des von Alice gesendeten Dokuments mit derselben Hashfunktion. Mit Alice öffentlichem Schlüssel und dem Algorithmus für elektronische Unterschriften dechiffriert er den signierten Hashwert. Stimmt dieser mit dem von ihm generierten Hashwert überein, kann er Vertrauen in die Echtheit des Dokuments gewinnen.

Blickt man nun auf die oben angeführten Fragestellungen, erkennt man, dass das Problem der Verfälschung schon durch Anwenden von Hashfunktionen gelöst ist, da diese Änderungen erkennbar machen.

Das andere Problem ist durch die Anwendung von asymmetrischen Kryptosystemen gelöst, da die Identität einer Person durch seinen öffentlichen Schlüssel gewährleistet ist, falls dieser durch eine Zertifizierungsinstanz bestätigt sind (wie man in Kapitel 2. 3 erfahren kann).

1. 3. 4. 1 Die in der cv act *library* enthaltenen digitalen Signaturverfahren

DSA

RSA

EC-DSA

- **DSA**

Im August 1991 schlug das *National Institute of Standards und Technologie* (NIST) den *Digital Signature Algorithm* (DSA) für den Einsatz im *Digital Signature Standard* (DSS) vor.

Diese Ankündigung provozierte eine Flut von kritischen Bemerkungen, die zum einen politischer Natur waren, denn die RSA Security Inc. verdient viel Geld mit den Lizenzen für RSA und machte Front gegen den Algorithmus. Zum anderen sah die ursprüngliche Fassung

eine Modullänge mit maximal 512 Bit vor, was auf Druck von unterschiedlichen Seiten auf 512-1024 Bit gesteigert wurde.

Am 19. Mai 1994 wurde der Standard schließlich festgelegt.

Beschreibung von DSA

DSA ist eine Variante der Unterschriftenalgorithmen von Schnorr und ElGamal. Es handelt sich um Beispiele des allgemeinen Verfahrens für digitale Signaturen, das auf dem Problem des Diskreten Logarithmus (DLP) in endlichen Körpern basiert.

DSA benutzt folgende Parameter:

p	Primzahl mit 512 bis 1024 Bit Länge (in 64-Bit-Schritten)
q	Primzahl, 160 Bit langer Faktor von p-1
g	erzeugendes Element mit Ordnung q
x	privater Schlüssel, beliebige Zufallszahl kleiner q
$y = g^x \mod p$	öffentlicher Schlüssel

Die ersten drei Parameter p, q, g und natürlich der öffentliche Schlüssel y sind öffentlich bekannt . Der private Schlüssel x muss geheim gehalten werden.

Weiter sei:

M	Nachricht
H	Hashfunktion (SHA-1)

Zum Erstellen und Verifizieren von Signaturen beim DSA sind die wesentlichen Gleichungen:

(1) $r = (g^k \bmod p) \bmod q$ k zufällig gewählt
(2) $sk = H(m) + xr \pmod q$

Die Idee ist, dass der Inhaber r und s bestimmen kann. Der Empfänger verifiziert die Signatur, indem er prüft, ob (2) gilt. Die Zufallszahl k ist als temporärer, geheim zu haltender Schlüssel anzusehen und nur mit Kenntnis dieses Geheimnisses ist (2) nach s auflösbar, was für die Erstellung einer korrekten Signatur notwendig ist.

Es sollte beachtet werden, dass der private Schlüssel x und der Parameter k so „zufällig wie möglich“ gewählt und geheim gehalten werden sollten, damit das Verfahren sicher ist. Dazu sollten Zufallszahlengeneratoren benutzt werden und eine erneute Verwendung von k ist auszuschliessen.

Sicherheit von DSA

Mit 512 Bit ist DSA nicht stark genug, um Sicherheit zu bieten. Mit 1024 Bit ist dies jedoch der Fall. Die Sicherheit entspricht der von RSA mit vergleichbaren Parametern.

Auf jeden Fall beruht die Sicherheit des DSA auf zwei verschiedenen, aber verwandten Problemen:

Einerseits beruht sie auf dem allgemeinen DL-Problem in $\mathbf{Z}_p^*$ (für welches ähnlich wie beim Faktorisierungsproblem subexponentielle Angriffsmethoden existieren).

Andererseits beruht die Sicherheit von DSA auf dem DL-Problem in der von g erzeugten Untergruppe mit Ordnung q. Für das letztere benötigen die besten bekannten Angriffe in der Größenordnung von $\sqrt{(q)}$ Operationen; die Parameter der aktuellen Implementierung sind so gewählt, daß beide Angriffsmethoden ungefähr gleichen Aufwand erfordern.

 [FIPS 186]

- **RSA**

Man kann RSA nicht nur als Verschlüsselungsverfahren einsetzen, sondern auch zum digitalen Signieren. Der Algorithmus hat eine analoge Form wie in Kapitel 1.3.3 über asymmetrische Verschlüsselung beschrieben. Es sind jedoch die Rollen von e und d vertauscht: Mit dem eigenen privaten Schlüssel wird signiert und der Empfänger überprüft die Unterschrift mit dem zugehörigen, öffentlichen Schlüssel.

Wie allgemein bei der Erstellung digitaler Signaturen mit Hilfe asymmetrischer Verfahren wird in der Praxis nicht die gesamte Nachricht unterschrieben, sondern nur deren Hashwert. Dabei muss beachtet werden, dass nach Anwenden einer Hashfunktion der zu signierende Wert eine geeignete Länge haben muss. Dazu verwendet man sogenannte Padding-Verfahren wie z.B. PKCS 1 [PKCS #1] oder gemäß ISO 9796 zum Auffüllen des Hashwerts auf die geforderte Länge.

 [RSA78], [RSA79]

- **EC-DSA**

EC-DSA ist die Entsprechung von DSA auf elliptischen Kurven. Anstatt in einer Untergruppe der Ordnung q von $\mathbf{Z}_p^*$ zu arbeiten, ist die zugrundeliegende mathematische Struktur eine elliptische Kurve.

Die Notationen von EC-DSA im Vergleich zu DSA sind folgende:

DSA	EC-DSA
q	n
g	P
x	d
$y=g^x$	Q=d P

mit folgender Erklärung:

n	Primzahl von mindestens 160 Bit Länge
P	Punkt auf der elliptischen Kurve der Ordnung n
D	privater Schlüssel, beliebige Zufallszahl kleiner als n
Q=d P	öffentlicher Schlüssel

Die zum Erstellen und Verifizieren von Signaturen beim EC-DSA wesentlichen Gleichungen sind dann:

(1)	kP = (x1,y1) und r = x1 mod n	k zufällig gewählt
(2)	sk = (H(m) + d r) mod n	

Zum Signieren wählt Alice ein zufälliges k, berechnet r und s mit Hilfe von (1) und (2) und sendet (m, r, s) an Bob. Zum Verifizieren der Signatur (m, r, s) hat das Protokoll folgende Form:

- Bob holt sich eine authentifizierte Kopie (E, P, n, Q) von Alice Schlüssel und überprüft, ob r und s ganze Zahlen aus dem Intervall [1,n-1] sind
- Bob verifiziert (m, r, s), indem er (2) prüft.

Die Sicherheit dieser kryptographischen Systeme basiert auf der Schwierigkeit des DL-Problems in Gruppen von Punkten auf einer elliptischen Kurve (EC-DLP). Dazu siehe die Ausführung in Kapitel 3 über ECC.

[ANSI X9.62], [ISO 15946]

1. 4 Sonstige Funktionen

1. 4. 1 Zufallszahlen

In vielen kryptographischen Mechanismen werden Zufallszahlen benötigt. So müssen z.B. bei der Schlüsselerzeugung bestimmte Parameter oder Bitfolgen „so zufällig wie möglich“ gewählt werden. Von dieser Zufälligkeit kann die Sicherheit des gesamten Verfahrens abhängen.

Echte Quellen von Zufällen bietet die Natur. Von ihr kann man atmosphärisches Rauschen oder die Zeiten für einen tropfenden Wasserhahn übernehmen. Oder man schließt einen Geigerzähler an den Computer an. Bei Online-Banking wird oft die Mausbewegung des Benutzers als Zufallssequenz genommen. Solche physikalischen Ereignisse sind in einer Software nicht leicht zu realisieren und das Problem ist, dass es keine echten Zufallszahlengeneratoren gibt. Es kann bestenfalls ein Generator für sogenannte Pseudozufallszahlen realisiert werden. Dieser Pseudozufallszahlengenerator (*Pseudo Random Number Generator* PRNG) sollte für kryptographische Zwecke folgende Eigenschaften besitzen:

1. Er scheint zufällige Ausgaben zu erzeugen, d.h. dass er sämtliche bekannten statistischen Tests besteht

2. Die Pseudozufallsfolge ist nicht voraussagbar. Es muss unmöglich zu berechnen sein, welches Zufallsbit als nächstes kommt, selbst wenn der Algorithmus oder die Hardware, die die Folge erzeugt, sowie alle vorankommenden Bits bekannt sind.

3. Der Generator ist nicht zuverlässig reproduzierbar. Wenn man ihn zweimal mit exakt derselben Eingabe laufen läßt, erhält man zwei Folgen, die keinerlei Ähnlichkeit haben

Da die in der Praxis verwendeten Verfahren deterministisch sind, muss der letzte Punkt durch eine geeignete Initialisierung des Algorithmus durch einen, bei jedem Ablauf unterschiedlichen, nicht vorhersehbaren Wert (den sogenannten *seed*) erfolgen. Da die Sicherheit der Vorgehensweise damit direkt von diesem *seed* abhängt, wird dieser Wert normalerweise durch Auslesen verschiedenster, möglichst nicht vorhersehbarer Daten erfolgen.

1. 4. 1. 1 Die in der cv act *library* enthaltenen Pseudozufallszahlengeneratoren

Blum, Blum und Shub-Generator

Linearer Kongruenzgenerator

FIPS 186 Generator

- **Blum, Blum und Shub-Generator**

Dieser Generator ist nach seinen Erfindern benannt und wird im weiteren mit BBS abgekürzt. Seine Theorie basiert auf den quadratischen Resten modulo p.

Für den Startwert wählt man zwei große Primzahlen p und q, die kongruent 3 modulo 4 sind.

Das Produkt n=pq nennt man Blumsche Zahl und man wählt eine andere, zufällige Zahl x, (den *seed*), die zu n prim ist und erhält x_0 durch

$$x_0 \equiv x^2 \bmod n.$$

Die restlichen Elemente x_i der Pseudozufallsbitfolge werden sukzessiv berechnet durch

$$x_i \equiv x_{i-1}^2 \bmod n.$$

Als Zufallsbit verwendet man das niederwertigste Bit dieses Folgegliedes. Die Sicherheit dieses Verfahrens basiert auf der schwierigen Faktorisierung von n.

Das BBS besitzt die starke Eigenschaft, dass man zu einer erzeugten Folge weder das vorherige, noch das nachfolgende Bit dieser Folge vorhersagen kann.

Dafür ist der Algorithmus langsam jedoch mit der Möglichkeit der Beschleunigung z. B. darin, dass man pro Iteration mehr als ein Bit entnimmt.

 [BBS86]

- **Lineare Kongruenzgeneratoren**

Bei Linearen Kongruenzgeneratoren nimmt man einen beliebigen Startwert X_0. Die restlichen Folgeglieder der Pseudozufallszahlenfolge haben die Form

$$X_i = (a\ X_{i-1} + b) \bmod m,$$

wobei die Variablen a, b und m konstant sind. Wie man diese geschickt wählt, kann man in [Knu81] nachlesen. Die Periode dieses Generators ist höchstens m, d.h. in diesem Fall gilt $X_{m+i} = X_i$. Sind a,b und m gut gewählt, ist die Periode gleich m, denn je länger desto besser.

Man muss jedoch beachten, dass dieses Verfahren für den Einsatz in kryptographischen Routinen ungeeignet ist und wird vielmehr wegen der schnellen und reproduzierbaren Bereitstellung von Daten zu Testzwecken eingesetzt.

 [Knu81]

- FIPS 186 Generator

Dieser Generator ist in seiner SHA-1 und DES basierten Variante implementiert. Ausgehend von einem 160 Bit Anfangsstatus X_i wird mit Hilfe der Hashfunktion SHA-1 oder der Blockchiffre DES eine 160 Bit Zufallszahl $G(X_i)$ generiert. Der aktualisierte Anfangsstatus für die Erzeugung der nächste Zufallszahl wird wie folgt bestimmt:

$$X_{i+1} = G(X_i) + X_i + 1 \bmod 2^{160}$$

wobei G wiederum die verwendete SHA-1 oder DES basierten Einwegsfunktion ist.

 [FIPS186]

1. 4. 2 Padding

Will man eine Nachricht mit einer symmetrischen Blockchiffre verschlüsseln, ergibt sich das Problem, dass die Länge des letzten Blocks kleiner sein kann als die Blocklänge des Algorithmus. Also muss der letzte Block mit dem sogenannten Padding aufgefüllt werden.

Eine einfache Methode ist das Auffüllen des letzten Blocks mit der Zeichenkette d,c,...,c, wobei c und d verschiedene Zeichen darstellen. Durch d wird der Anfang des Paddingbereichs gekennzeichnet. Eine weitere Methode ist das Anhängen einer Zeichenkette c,...,c,m, wobei m die Anzahl der angefügten Zeichen oder die ursprüngliche Länge der Nachricht ausdrückt. Außerdem gibt es noch die Kombination aus beiden, die durch die Zeichenkette d,c...c,m ausgedrückt wird.

Unelegant werden diese Verfahren, falls der Klartext den letzten Block gerade füllt. In diesem Fall muss ein zusätzlicher Block generiert werden. Außerdem ist der Chiffretext länger als der Klartext, was problematisch werden kann, falls z.B. ein Teilstück einer Datei verschlüsselt werden soll. Ebenso kann das Problem des Aufüllens bei Hashfunktionen und digitalen Signaturen auftreten.

Kapitel 2 Reale Anwendungen

Bei jedem Kryptosystem versucht ein potentieller Angreifer alle Schwachpunkte so gut wie möglich auszunutzen. Die verschiedenen Möglichkeiten und Verfahren dazu werden im ersten Unterabschnitt erklärt.

Allerdings gibt es auch Schwachpunkte in der Umgebung, in der Kryptosysteme eingesetzt werden: Bei Computern kann man mangels Schutzmechanismen Informationen manipulieren. Deshalb sollten die eigentlichen Berechnungen auf besonders geschützter Hardware durchgeführt werden. Als Lösung werden in Abschnitt 2.2 Smartcards vorgestellt, auf denen man seinen privaten Schlüssel sicher speichern kann.

Eine andere Schwachstelle ist, dass man durch digitale Signaturen zwar gewährleisten kann, dass man immer mit demselben Kommunikationspartner kommuniziert, dies verschafft aber keine Gewähr über die tatsächliche Identität. Dieses wird dadurch gelöst, dass man sich bei einem Trustcenter (Kapitel 2. 3 über PKIs) persönlich anmelden muss. In diesem Abschnitt wird außerdem eine Lösung für das Problem der Schlüsselverwaltung bei symmetrischen Verfahren aufgeführt.

Im letzten Unterabschnitt werden einige Beispiele gegeben, wie man in der alltäglichen Praxis Verschlüsselung oder die Mechanismen der digitalen Signatur nutzen kann, um seine Daten, seine Kommunikation oder anderes zu schützen.

2. 1 Mögliche Attacken

Bei den bisher betrachteten Verfahren haben zwei Kommunikationspartner Daten über einen unsicheren Kanal ausgetauscht. Für die Sicherheitsbetrachtungen wird davon ausgegangen, dass einem potentiellen Angreifer alle Informationen über das verwandte System (bis auf den eigentlichen Schlüssel) zur Verfügung stehen, so dass er völligen Zugriff auf die Kommunikation hat.

Die Kryptoanalyse ist die Lehre, ohne Kenntnis des Schlüssels an die geheimen Daten zu gelangen.

Für die Kryptoanalyse eines Verfahrens kann man nun verschiedene Unterscheidungen hinsichtlich der Rolle des Angreifers und der ihm vorliegenden Informationen treffen.

Bei den Angreifern unterscheidet man:


Den passiven Angreifer (Eve): dieser hört den Übertragungskanal ab mit dem Ziel, Nachrichten zu entschlüsseln.


Den aktiven Angreifer (Mallory): dieser ist in der Lage, Übertragungen auch zu manipulieren.

Für die dem Angreifer vorliegenden Informationen unterscheidet man:

- Nur Chiffretext (*known ciphertext*): dies ist die schwächste aller Voraussetzungen. Hierbei wird davon ausgegangen, dass der Angreifer eine bestimmte Anzahl von Chiffretexten ohne den zugehörigen Klartext besitzt.

- Bekannter Klartext (*known plaintext*): der Angreifer besitzt eine gewisse Anzahl von Chiffretexten inklusive der zugehörigen Klartexte.

- Frei wählbarer Klartext (*chosen plaintext*): bei dieser ist der Angreifer in der Lage, zu frei gewählten Nachrichten die Chiffrierung zu erhalten. Diese (auf den ersten Blick

unrealistisch erscheinende) Möglichkeit des Angriffs ist jedoch die Regel bei Public Key-Verschlüsselung.

Grundsätzlich sollte man für die kritische Analyse eines gegebenen Kryptosystems immer von der (aus Sicht des Angreifers) günstigste Situation ausgehen.

Einem Angreifer stehen nun verschiedene Methoden zur Verfügung:

2. 1. 1 Generische Attacken

Bei Verfahren, die von einem geheimen Schlüssel abhängen, versucht man, diesen zu bestimmen. Unzureichend für heutige Verhältnisse sind natürlich Systeme, bei denen der zur Verfügung stehende Schlüsselraum nicht groß genug ist. Solche können nämlich schon dem einfachsten aller Angriffe, dem systematischen Ausprobieren aller Möglichkeiten, nicht widerstehen. Bei diesem sogenannten *brute force*-Angriff werden nacheinander alle Schlüssel zur Verschlüsselung eingesetzt und man ist erfolgreich, falls der erhaltene Klartext einen Sinn ergibt. Dazu muss man allerdings sinnvollen von sinnlosen Klartext unterscheiden können.

Um solche Attacken zu verhindern, sollte die Anzahl der Operationen zur Berechnung des Schlüssels mindestens in der Größenordnung von 10^{25} liegen[3]. Dies ist natürlich nur eine Richtlinie, die gewährleisten soll, dass diese so entstandenen Verfahren auch bei wachsender Rechenleistung in einigen Jahren noch als sicher gelten.

Der Erfolg eines solchen Angriffs hängt von der Länge des Schlüssels und von der Rechenleistung der eingesetzten Computer ab. So hat man 1995 geschätzt, dass ein *brute force*-Angriff mit einem Kostenaufwand von 100 000 Dollar auf einen 56-Bit-Schlüssel (DES) 35 Stunden, auf einen 64-Bit-Schlüssel ca. 1 Jahr und auf einen 80-Bit- Schlüssel 70 000 Jahre dauert. Die Anzahl der dabei jeweils veranschlagten Operationen entspricht dabei der Anzahl der Elemente im Schlüsselraum, also 2^{56}, 2^{64} bzw. 2^{80} Operationen.

Bei Hashfunktionen, die schlüsselunabhängig sind, versucht man mit einer Geburtstagsattacke das Verfahren zu knacken. Dabei handelt es sich um einen Angriff auf die Kollisionsresistenz: Man versucht, zwei beliebige Nachrichten M und M´ zu finden, deren Hashwerte h(M) und h(M´) übereinstimmen. Um bei einer Attacke 2^{80} Operationen durchführen zu müssen, was der Sicherheit eines 80-Bit-Schlüssel bei symmetrischen Verfahren entspricht, sollte eine Hashfunktion eine Ausgabe von 160 Bit Länge erzeugen.

Bei Public Key-Verfahren ist nicht so ganz klar, wie ein Angriff aussehen kann. Da dabei versucht wird, die zugrundeliegenden Probleme aus der Mathematik zu knacken, spricht man auch von „harten“ Problemen. Nach dem heutigen Stand der Erkenntnisse sollten z.B. beim RSA-Algorithmus eine Schlüssellänge von 1024 Bit und beim DSA von 1024 bzw. 160 Bit für die verschiedenen Parameter verwendet werden.

---

3 Um sich die ungeheure Größe dieser Zahl vorstellen zu können, sollte man sich überlegen, dass unser Universum 10^{10} Jahre alt ist.

2. 1. 2 Angriffe auf die Mechanismen

Bei symmetrischen Verschlüsselungsverfahren hat man noch die Möglichkeit, einen Algorithmus anzugreifen, indem man sich die Chiffre selber anguckt. Die Verfahren, die am gefährlichsten sind, sind folgende:

- die differentielle Kryptoanalyse: hierbei betrachtet man zwei Chiffretexte, deren zugehörige Klartexte gewisse Differenzen aufweisen. Man versucht also, durch Verändern des Klartextes Informationen über den Zusammenhang zwischen Chiffretext und Klartext zu erlangen.

- die lineare Kryptoanalyse: hierbei handelt es sich im Gegensatz zu der differentiellen Kryptoanalyse um eine Attacke, die nur bekannten Klartext benötigt. Man versucht, einfache („lineare“) Abhängigkeiten zwischen den Bits des Klartextes und des Chiffretextes zu entdecken und auszunutzen, um Informationen über den Schlüssel zu erhalten.

Beim Design moderner Algorithmen wird darauf geachtet, sich gegen diese beiden Arten von Angriffen zu schützen.

2. 2 Smartcards

Man sollte es potentiellen Angreifern nicht zu leicht machen und seinen Schlüssel für ein symmetrisches Verfahren oder seinen privaten Schlüssel sichern. Die sicherste Lösung dazu ist die Speicherung auf einer Smartcard.

2. 2. 1 Was ist eine Smartcard?

Eine Smartcard ist im wesentlichem ein eigenständiger Computer auf einem Siliziumchip. Dieser Chip kann eigenständig rechnen und besitzt ein eigenes Betriebssystem.

Nur Computer können mit kryptographischen Schlüsseln etwas anfangen, jedoch sind diese zu leicht zu manipulieren. Man muss den Computer nur aufschrauben und kann mitprotokollieren, was gerechnet wird.

Den Chip auf einer Smartcard kann man nicht ohne großen Aufwand manipulieren und es besteht eine hohe Wahrscheinlichkeit, ihn dabei zu zerstören. Die Nachteile sind, dass ein Chip auf einer Smartcard wenig Speicherplatz zur Verfügung hat und die Rechenleistung relativ gering ist.

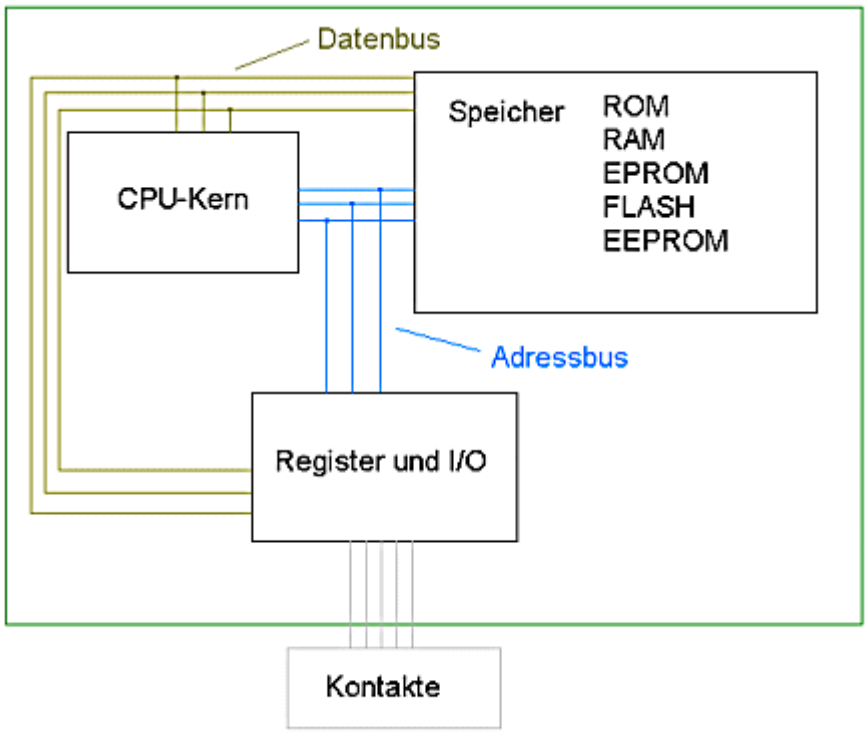


Abbildung: Das Innenleben einer Smartcard

2. 2. 2 Warum sind Smartcards für uns interessant?

Der private Schlüssel wird auf einer Smartcard nicht nur gespeichert, sondern kann sogar auf der Karte generiert werden. Man kann sich also seinen privaten Schlüssel von der Karte erzeugen lassen.

Dieser Vorteil wird klarer, wenn man sich die Alternative anschaut: Dabei wird der Schlüssel extern generiert und erst danach auf der Karte gespeichert. Das bietet natürlich mehr Angriffsmöglichkeiten, da es einen Zeitpunkt gibt, zu dem der private Schlüssel außerhalb der Karte bekannt ist.

Hinzu kommt, dass selbst nach der Erstellung des privaten Schlüssels dieser im Normalfall auf der Karte verbleibt: Anstatt dass der Schlüssel für die kryptographischen Operationen (wie z.B. das Signieren eines Dokuments) zur Benutzung in den Computer übertragen wird, werden die zu bearbeitenden Daten der Smartcard übermittelt, die das Resultat zurückgibt.

Wegen der begrenzten Speicherkapazität auf der Karte ist eine kleine Schlüssellänge von Vorteil, da es sich bei dem zu behebenden Problem um die Speicherung eines privaten Schlüssels handelt. Also kann man die Eigenschaft der Verschlüsselung mit elliptischen Kurven ausnutzen, mit deutlich geringeren Schlüssel- und Parameterlängen auszukommen, ohne Abstriche hinsichtlich der Sicherheit in Kauf nehmen zu müssen.

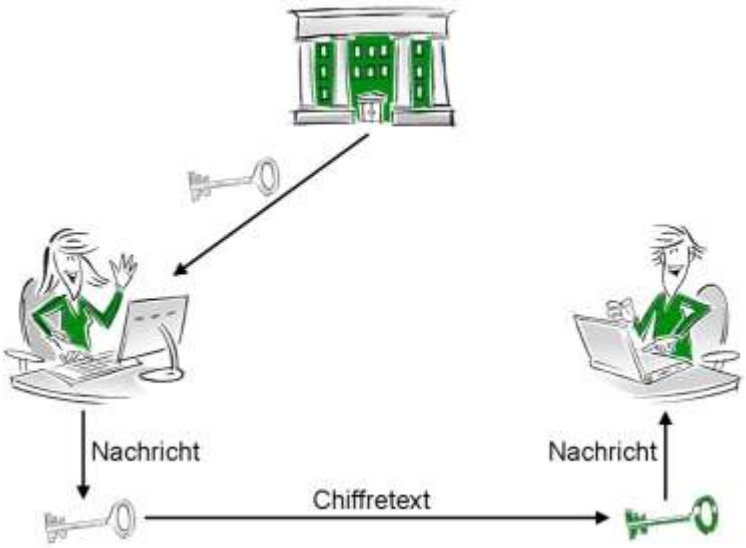
2. 3 Public Key-Infrastruktur

Nachdem man im vorherigen Unterabschnitt eine Lösung für die Verwaltung von geheimen oder privaten Schlüssel kennengelernt hat, geht es in diesem Abschnitt um die Problematik des Managements von öffentlichen Schlüsseln. Falls diese öffentlichen Schlüssel in einer Datenbank verwaltet und jedem zugänglich gemacht werden, spricht man von einer Public Key-Infrastruktur (PKI).

2. 3. 1 Grundbegriffe

Das Protokoll einer Kommunikation mit Public Key-Kryptographie ist folgende:

- Alice bezieht Bobs öffentlichen Schlüssel aus der Datenbank
- Alice chiffriert ihre Nachricht mit Bobs öffentlichen Schlüssel und sendet sie an Bob
- Bob dechiffriert Alices Nachricht mit seinem privaten Schlüssel



Das Problem ist, dass die öffentlichen Schlüssel in sogenannten PKIs verwaltet und von dort abgeholt werden müssen.

2. 3. 2 Definitionen

Eine Public Key-Infrastruktur (PKI) ist eine Kombination aus Hardware- und Software-Komponenten, Policen und verschiedenen Prozeduren. Sie basiert auf Objekten, welche Zertifikate genannt werden und als digitale Ausweise operieren, indem sie einen Benutzer mit seinem öffentlichen Schlüssel verbinden.

Abbildung.: Zertifikat

Eine PKI besteht aus:

- einer Sicherheitspolicy: definiert den Sicherheitsgrad, die Prozesse und Verwendungen der Kryptographie. Sie beinhaltet die Angaben, wie Schlüssel und wertvolle Informationen gehandhabt werden.

- einer Zertifizierungsstelle (*Certification Authority* - CA): diese Instanz ist die Vertrauensbasis der PKI, da sie die Zertifikate erstellt.

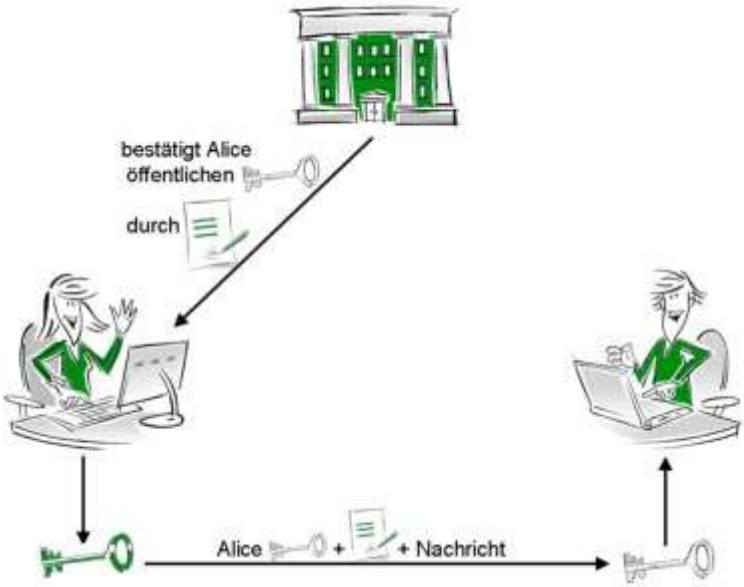
- einer Registrierungsstelle (*Registration Authority* - RA): ist die Schnittstelle zwischen Benutzer und der CA. Sie erfasst und authentifiziert die Identität der Benutzer und reicht die Anfrage nach einem Zertifikat an die CA weiter.

- einem Verteilungssystem für die Zertifikate: die Verteilung von Zertifikaten hängt von der PKI-Umgebung ab. Beispiele sind Verteilungen durch den Benutzer selber oder durch einen Verzeichnis-Server.

- PKI-Applikationen: Anwendungen sind z.B. E-Mails, Kommunikation zwischen Web-Server und Browser oder der Einsatz von Kreditkarten im Internet.

Die Einheit aus CA und RA wird auch Trustcenter genannt und wird für die Authentifizierung von Personen und ihren Nachrichten verwendet. Im grundlegenden Modell steht der Treuhänder „Trent“ für das Trustcenter.

Ein Trustcenter kann zum Signieren und Verschlüsseln in Anspruch genommen werden. Signieren kann z.B. folgende Form haben:



Elektronische Dokumente enthalten oft Zeitstempel, wenn der Zeitpunkt der Bearbeitung einer Datei von Bedeutung ist: an die Nachricht werden Datum und Zeit der Unterschrift angehängt und gemeinsam mit der übrigen Nachricht unterschrieben.

Beispiel:

> Ein Vertrag kann ab dem Zeitpunkt des Unterzeichnens gültig sein. Dieses kann man in elektronischen Dokumenten durch einen Zeitstempel gewährleisten.

2. 3. 3 Zertifikate

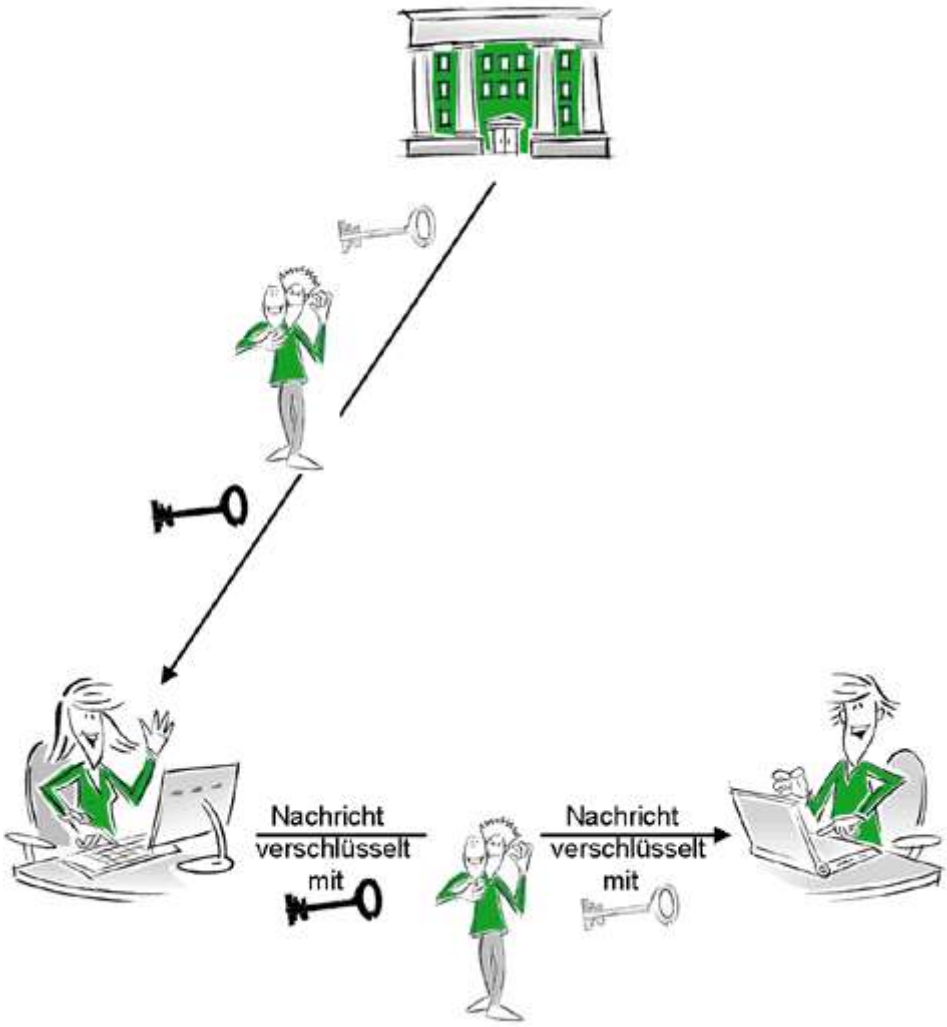
Um übertragene Schlüssel vor unbefugter Veränderung zu schützen, werden öffentliche Schlüssel von vertrauenswürdigen Personen, meist der CA einer PKI, zertifiziert, d.h. mit einem oben schon genannten Zertifikat versehen. Man kann Zertifikate mit einem amtlichen Ausweis vergleichen. Ein Beispiel ist der Reisepass. Mit dem Reisepass ist das Recht verbunden, in bestimmte Länder einreisen zu dürfen. Das Dokument enthält den Namen des Inhabers, ein Foto des Inhabers, die Unterschrift des Inhabers, einen Gültigkeitsvermerk der ausstellenden Behörde und viele weitere Daten. So kann jeder erkennen, wem dieser Ausweis gehört, und wem die damit verbundenen Rechte zustehen. Wichtig dabei ist, dass diese Rechte unverwechselbar mit einer bestimmten Person verbunden sind. Außerdem ist es relativ schwierig, einen Ausweis zu fälschen. Daher wird ein gültiger Ausweis in unserem Rechtssystem als ausreichend zur Identifikation von Personen angesehen.

Das Problem der Schlüsselverteilung bei asymmetrischen Kryptosystemen wird oft nicht erkannt, da man öffentliche Schlüssel über einen unsicheren Kanal austauschen kann. Dies ist jedoch nur richtig, wenn man den passiven Angriff „Abhören“ als die einzige Gefährdung der Kommunikationssicherheit betrachtet.

Es muss jedoch mit aktiven Angriffen gerechnet werden:

- Mallory fängt z.B. Bobs öffentlichen Schlüssel K_B ab und ersetzt ihn durch einen eigenen Schlüssel K_{B*}
- Es befindet sich in der Datenbank der ausgetauschte Schlüssel K_{B*}, den Alice sich dort abholt
- Sie verschlüsselt damit ihre Nachricht und schickt sie an Bob
- Mallory fängt diese Nachricht ab, entschlüsselt sie mit K_{B*}. Er kann sie also lesen und verändern
- Dann verschlüsselt Mallory diese Nachricht mit Bobs richtigen öffentlichen Schlüssel K_B und sendet sie an Bob
- Bob entschlüsselt die Nachricht mit seinem privaten Schlüssel.

Hierbei haben weder Alice noch Bob den Angriff bemerkt.



Bei Ausweisen gibt es das Problem, dass sie fälschungssicher sein sollten, was durch Spezialpapier, Spezialdruck, Wasserzeichen und anderes erreicht wird. Außerdem müssen sie vor Betrug geschützt werden, wenn z.B. ein Reisepass gestohlen wird und der Dieb versucht, damit das Land zu verlassen. Dieses wird dadurch erreicht, dass ein Ausweis eine begrenzte Gültigkeitsdauer hat und an den Grenzen die Einträge mit einer Datenbank verglichen werden.

Dieses Szenario kann man auch auf Zertifikate anwenden:

- Fälschungssicherheit wird bei Zertifikaten durch die Unterschrift der autorisierten Stelle erreicht.

- Missbrauch: Auch bei Zertifikaten wird eine Gültigkeitsspanne angesetzt. Diese kann sehr kurz sein, da die Ausstellung neuer Zertifikate relativ unproblematisch ist (*key pair update*).

Die Datenbank mit den Einträgen von gesperrten Zertifikaten heißt *Certificate Revocation List* (CRL). Diese Datenbank muss zentral verwaltet werden und kann im Laufe der Zeit wegen der ansteigenden Anzahl von Zertifikaten sehr umfangreich werden.

2. 3. 4 Aufbau eines Zertifikates nach X.509 v3

Ein Standardisierungsvorschlag für digitale Zertifikate ist das vom CCITT und der ISO vereinbarte *Directory Authentication Framework* [CCITT509], das auch X.509-Protokoll genannt wird. Diese Empfehlung beschreibt den Aufbau von Zertifikaten. Zur Zeit ist X.509 in der Version 3 vorhanden und wurde von ANSI und ISO übernommen. Es ist im Moment der am meisten verbreitete Standard.

Ein Zertifikat muss mindestens folgende Daten enthalten:

- Inhaber
- öffentlicher Schlüssel des Inhabers
- Signatur der CA

Ansonsten sind noch zusätzliche Informationen notwendig. Das Format eines X.509-Zertifikats ist folgendes:

Version
Seriennummer
Algorithmus
Name des Ausstellers
Geltungsdauer
Name des Benutzers
Öffentlicher Schlüssel des Benutzers
Unique Identifier des Ausstellers$^{+\oplus}$
Unique Identifier des Benutzers$+\oplus$
Erweiterungen+
Signatur

Abbildung: Aufbau eines X.509-Zertifikats

$\oplus$ Diese Felder wurden vorgesehen, damit die Namen hinterher wiederverwendet werden können

\+ Diese Felder wurden mit der Version 3 eingeführt

In dem Feld Erweiterungen sind als wichtigste folgende mögliche Einträge zu nennen:

- *Certificate Policies:*

  Beinhaltet die Konditionen, unter denen die CA arbeitet (z.B. Sicherheitsvorkehrungen bei der Zertifizierung …). Dies soll den Applikationen ermöglichen, anhand einer Vergleichsliste zu entscheiden, ob ein Zertifikat den geforderten Sicherheitsanforderungen entspricht oder zurückgewiesen werden soll.

- CRL *Distribution Points:*

  Bezeichnet den Punkt (IP-Adresse), wo die Informationen über widerrufene Zertifikate (die CRL) gefunden werden können, d.h. die zuständige Zertifizierungsstelle.

2. 3. 5 Widerrufene Zertifikate (*Certificate Revocation*)

Missbrauchte Zertifikate müssen erkannt und für alle erkennbar widerrufen werden. Die Erkennung liegt außerhalb des Bereichs von X.509. Hier wird nur der Widerrufungsmechanismus von missbrauchten Zertifikaten beschrieben.

Die missbrauchten Zertifikate selbst werden in speziellen Listen (CRLs) abgelegt, bis die Gültigkeitsdauer des Zertifikats abgelaufen ist. Der Aufbau der *Certificate Revocation List* Version 2 ist in folgender Abbildung dargestellt.

Version
Algorithmus
Name des Ausstellers
Ausgabezeitpunkt dieser CRL
Ausgabezeitpunkt einer neuen CRL
Seriennummern der widerrufenen Zertifikate
Erweiterungen
Signatur

Abbildung: Aufbau einer CRL

Problem der CRLs:

1. Wenn ein Zertifikat widerrufen werden soll, kann das erst bei der Ausgabe der nächsten CRL bekannt gemacht werden. Die Zeit, bis eine neue CRL erstellt wird, kann einige Stunden bis zu einigen Wochen betragen.

2. Außerdem hängt die Größe der CRL von der Anzahl der Benutzer einer Zertifizierungsstelle und der Gültigkeitsdauer der ausgestellten Zertifikate ab. Je mehr Benutzer eine CA hat und je länger die Gültigkeitsdauer der Zertifikate ist, desto mehr Einträge sind in einer CRL.

2. 4 Beispielhafte Anwendungen

Die **cv act *library*** umfasst alle Algorithmen und Funktionen, die für die Erstellung sicherheitsrelevanter Anwendungen notwendig sind. Es gibt zahlreiche Einsatzmöglichkeiten, wie man Geheimhaltung oder Authentifizierung durch Verschlüsselung oder digitale Signaturen zur Sicherung von vertraulichen Daten oder Kommunikation einsetzen kann.

Das anschaulichste Beispiel ist die Entwicklung Ihres eigenen E-Mail-Programms, das Ihre Nachrichten verschlüsseln oder digital signieren kann. Nachdem Sie sich für ein Verfahren, z.B. CAST zur Verschlüsselung oder EC-DSA für digitale Signaturen entschieden haben, benutzen Sie dazu die in der **cv act *library*** implementierten Objekte.

Jedoch findet dieses Beispiel in der Realität nicht häufig Anwendung, da bereits viele E-Mail-Programme vorhanden sind, die man einfacher um die gewünschten Funktionalitäten erweitern kann oder schon erweitert wurden, wie z.B. das **cv act *s/mail*™** entsprechend als Erweiterung vom Microsoft Outlook, Novell Groupwise oder Lotus Notes.

Im folgenden werden weitere Beispiele beschrieben, um Sie auf zum Teil einfache Maßnahmen zur Erhöhung der IT-Sicherheit hinzuweisen.

2. 4. 1 Safe

Bei diesem Objekt handelt es sich um einen Ordner, der wie zum Beispiel der Papierkorb auf dem Desktop oder in dem Verzeichnis liegt, in dem Sie primär arbeiten. Man will Daten löschen, so dass kein anderer auf sie zugreifen kann, andererseits aber, wie beim Papierkorb, sie doch wiederherstellen. In den Safe kann man Daten ablegen, die dann automatisch verschlüsselt werden und auf die man nur mittels einer PIN (*personal identification number*) oder eines Schlüssels, der am besten auf einer Smartcard gespeichert ist, zugreifen kann. Damit ist die Geheimhaltung der Daten gewährleistet und die Möglichkeit ihrer Manipulation verhindert. Dabei ist zu beachten, dass beim Ziehen der Daten in den Safe kein Schlüssel oder PIN notwendig ist, da man ein Public Key-Verfahren verwenden kann.

Dies ist eine sinnvolle Anwendung für Personen, die z.B. ihren Arbeitsplatz häufiger für andere zur Verfügung stellen (gewollt oder ungewollt), ohne sich vorher auszuloggen. Oder es dient beispielsweise Außendienstmitarbeitern, die auf einem Laptop oder einem anderen Small Device arbeiten, bei denen die Wahrscheinlichkeit hoch ist, das diese abhanden kommen oder gestohlen werden. Natürlich sind bei dieser Methode nur die Daten verschlüsselt, die in den Safe gezogen wurden.

2. 4. 2 Zugangskontrollen

Hierbei handelt es sich um eine Art der Authentifizierung, die in vielen Bereichen Anwendung findet. Es kann sich um Zugangskontrollen handeln, bei denen der Zutritt in bestimmte Räume nur für ausgesuchte Personen - oder auch Tieren - möglich sein soll. Dieses kann durch Smartcards realisiert werden, bei denen das Sicherheitsniveau skalierbar ist. Es gibt auch die Möglichkeit der kontaktlosen oder der biometrischen Authentifizierung, bei denen beim jetzigen Stand der Technik jedoch das Sicherheitslevel nach den Vorgaben des deutschen Signaturgesetzes noch nicht erfüllt ist, was bei solchen Kontrollen in der Regel auch nicht sinnvoll ist. Desweiteren sind Zugangskontrollen möglich, die den Zugriff auf Softwareprodukte oder die Art der zur Verfügung gestellten Funktionalitäten regeln.

Beispiel:

> Ein Kunde bekommt von Ihnen eine Software-CD oder den Zugang zu einer Datenbank, auf der sich alle möglichen Funktionalitäten befinden. Er soll jedoch nur die Funktionen nutzen können, für die er auch bezahlt hat. Also muss eine Person mit seinen Berechtigungen verbunden werden, die dem System bei jedem Zugang bekannt sind. Diese Art der Authentifizierung kann man durch den Einsatz digitaler Signaturen erreichen. Diese Art der Berechtigung kann jederzeit aktualisiert werden.

2. 4. 3 E-Commerce

Es wird immer beliebter, über das Internet Geschäftsprozesse abzuwickeln, wie z.B. Waren einzukaufen. Jedoch ist es den meisten Menschen zu unsicher, die Nummer ihrer Kreditkarten über diesen unsicheren Kanal preiszugeben. Durch die Presse sind hinreichend Fälle bekannt, bei denen diese Nummer abgefangen und missbraucht wurde. Für dieses Problem gibt es jedoch die einfache Lösung, sich über eine Smartcard zu identifizieren und mit einer digitalen Signatur den Kaufvertrag digital zu unterschreiben oder seine Bankverbindung verschlüsselt zu übermitteln, solange man auf gesicherte Art an den öffentlichen Schlüssel des Empfängers gelangen kann.

Auch kann man beim Online-Banking die Eingabe der nur einmal zu verwendeten PINs, den sogenannten TANs (*transaction numbers*) vermeiden, indem man solche Verfahren einführt.

2. 4. 4 Mobile Commerce

Mobile Commerce ermöglicht es beispielsweise, mit seinem Handy zu bezahlen oder andere Arten von Geschäftsprozessen, wie z.B. Verträge abzuschließen. Die vorhandene SIM-Karte, welche eine bestimmte Form der Chipkarte darstellt, wird um die Funktionalität der digitalen Unterschrift erweitert. Ein zusätzlicher Sicherheitsmechanismus bei jeder Signatur, wie die Eingabe einer PIN, verhindert den Missbrauch der zusätzlichen Funktionalität beim Verlust des Handys.

In der Praxis stellt sich das wie folgt dar: Man möchte in einem Restaurant mit seinem Handy bezahlen. Dazu gibt man seine Telefonnummer an. Daraufhin wird das Handy angerufen und der zu zahlende Betrag mit dem Namen und der Adresse des Restaurants erscheint auf dem Display. Durch Eingabe seiner PIN bestätigt man diese und bezahlt.

Der Vorteil dieser Methode ist, dass man nicht mehr zwei Objekte, nämlich Handy und Kredit- oder EC-Karte bei sich haben muss, sondern nur noch das Handy. Der Trend geht auch weiter dahin, dass man mit einem Multifunktionsgerät viele Aufgaben erledigen kann. In Verbindung mit einem Palm-Top kann man dann ganze Verträge komfortabel schreiben und auf diese Weise digital verschicken und unterschreiben.

Kapitel 3 Elliptic Curve Cryptography (ECC)

Der Einsatz von elliptischen Kurven in der Public Key-Kryptographie wurde 1985 unabhängig voneinander von N. Koblitz und V. Miller angeregt. Das Interesse an elliptischen Kurven für kryptographische Zwecke, d.h. an ECC - *Elliptic Curve Cryptography* hat seitdem stark zugenommen.

Da ECC-Implementationen eine höhere Effizienz besitzen und auch langfristig als sicher gelten, hat auch deren Anwendung zunehmend an Bedeutung gewonnen. Viele infrastrukturelle Probleme, wie z. B. die Schlüsselgenerierung, sind wesentlich eleganter und schneller gelöst, als dies im derzeit eingesetzten RSA-Verfahren der Fall ist. Außerdem ist im Falle der Verwendung von ECC das Problem der wachsenden Schlüssellänge bei steigenden Sicherheitsanforderungen deutlich besser zu handhaben: wo im Falle von RSA oder DSA über eine Verdoppelung der Schlüssellänge nachgedacht werden muss, reichen im Fall von ECC wenige Bit mehr aus, um die Sicherheit des Systems deutlich zu erhöhen.

Der Vorteil beim Einsatz in Kryptosystemen mit elliptischen Kurven liegt in der schnellen Verschlüsselung und der größeren Flexibilität. Ohne Abstriche hinsichtlich der Sicherheit in Kauf zu nehmen, kommt man mit deutlich geringeren Parameterlängen aus. Dies wirkt sich besonders beim Einsatz in Situationen aus, wo Speicher- oder Rechenkapazität knapp sind, wie z.B. bei Smartcards und anderen Small Devices.

3. 1 Mathematischer Hintergrund

Eine elliptische Kurve über einem endlichen Körper **K** ist die Menge der Punkte (x,y) welche z.B. eine Gleichung der Form

$$y^2 = x^3 + A\,x^2 + B$$

erfüllen. Hat der endliche Körper die Charakteristik 2, so lautet die entsprechende Gleichung

$$y^2 + xy = x^3 + A\,x^2 + B$$

In beiden Fällen gilt: x, y, A, B $\in$ **K**. Zusammen mit einem speziellen Punkt *0*, der „Punkt an Unendlich“ genannt wird, bildet eine elliptische Kurve eine Gruppe. Man bezeichnet sie mit E(**K**) oder kurz mit E, falls der zugrundeliegende Körper bekannt ist.

Die zugrundeliegende Operation dieser Gruppe ist die Addition und das Additionsgesetz für elliptische Kurven kann man graphisch folgendermaßen definieren:

Seien $P = (x_1,y_1)$ und $Q = (x_2,y_2)$ zwei verschiedene Punkte auf der elliptischen Kurve E, dann ist die Summe P + Q graphisch definiert durch:

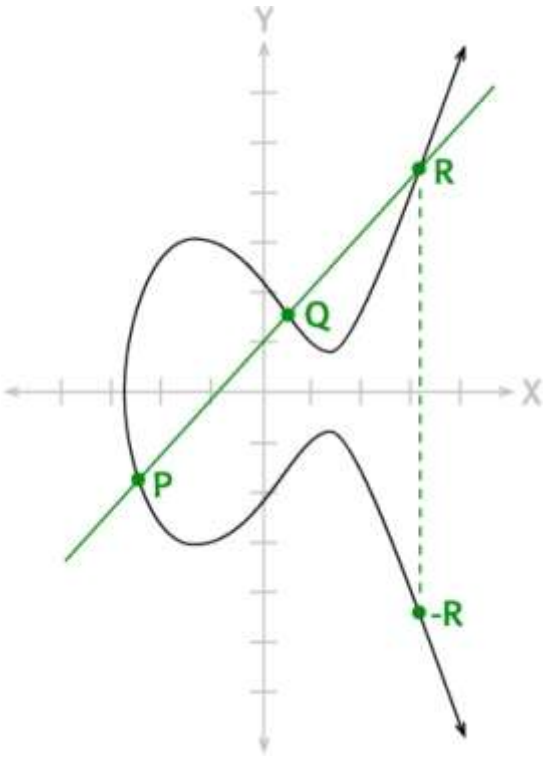


Abbildung: Additionsgesetz für elliptische Kurven

d.h. man legt eine Gerade durch die Punkte P und Q, die die Kurve in genau einem weiteren Punkt R schneidet. Die Summe P + Q ist definiert durch -R d.h. durch die Spiegelung des Punktes R an der x-Achse.

Es gibt drei Spezialfälle:

Falls P = *0* der Punkt an Unendlich ist, dann sei –P wieder *0* und P + Q = Q.

Falls P = -Q, dann bekommt man P + Q = *0.*

Falls P = Q, dann nimmt man als Gerade die Tangente an den Punkt P und verfährt wie oben. Außerdem ist 2*P = P + P, so dass auch k*P für eine ganze Zahl k definiert ist.

Bemerkung: Die Punktegruppe einer elliptischen Kurven ist eine additive Gruppe, in der sich aber alle Protokolle für die Schreibweise einer multiplikativen Gruppe übertragen lassen: was beim originalen DL-Problem die Bestimmung von x in g^x bedeutet, ist im Falle der Verwendung der elliptischen Kurven die Berechnung von k in k*P.

3. 2 Sicherheit

Die Sicherheit dieser kryptographischen Systeme basiert auf der Schwierigkeit des DL-Problems in Gruppen von Punkten auf einer elliptischen Kurve (EC-DLP):

Gegeben seien eine elliptische Kurve E über $\mathbf{Z}_p$,

ein Punkt $P \in E(\mathbf{Z}_p)$ der Ordnung n

und ein Punkt $Q \in E(\mathbf{Z}_p)$

Gesucht ist ein $k \in \mathbf{Z}$ mit $0 \leq k \leq n-1$, so dass $Q = k*P$ (unter der Voraussetzung dass man weiß , dass eine solche Darstellung von Q existiert).

Die Forschung im Bereich der elliptischen Kurven wird seit ca. 150 Jahren betrieben. Seit ungefähr 1985 ist bekannt, dass das EC-DLP für die Kryptographie genutzt werden kann und kein führender Mathematiker hat bis heute eine bemerkenswerte Schwachstelle entdeckt.

Es muss jedoch beachtet werden, dass es Klassen von ungeeigneten Kurven gibt, und zwar sind sogenannte supersinguläre Kurven, so wie auch sogenannte anomale Kurven nicht geeignet. Die Attacken auf die letzteren wurden unabhängig voneinander von Semaev [Sem], Smart [Sma97], Satoh und Araki [SA97], und für den allgemeinen Fall von Rück [Rüc97] gefunden.

Miller [Mil86], Frey und Rück [FR94], Adleman, DeMarrais und Huang [ADH94], Boneh und Lipton [BL92], Stein [Ste96], Balasubramanian und Koblitz [BK], Zuccherato [Zuc96], Flassenberg und Paulus [FP97] oder Voloch [Vol97].

Ansonsten bietet ein System, das auf elliptischen Kurven mit einem Modulus von 160 Bit basiert, die gleiche kryptographische Sicherheit wie ein RSA- oder DSA-System mit 1024-Bit-Modulus. Ein 256 Bit ECC-Schlüssel ist mit einer 3072 Bit RSA-Verschlüsselung vergleichbar und eine ECC-Schlüssellänge von 512 Bit bietet dieselbe Sicherheit wie ein utopischer 15000 Bit RSA-Schlüssel.

[NR96], [ISO 14888], [ISO 9796]

3. 3 Konstruktion geeigneter Kurven

Für die Feststellung, ob eine gegebene elliptische Kurve für den Einsatz innerhalb der Kryptographie geeignet ist, gibt es gewisse Kriterien. Das wichtigste dabei ist die Anzahl der Punkte auf dieser Kurve. Diese muss hinreichend groß sein (in der Regel mehr als 2^{160}, was eine Zahl mit fast 50 Dezimalstellen ist) und sollte eine Primzahl sein oder einen entsprechend großen Primteiler enthalten.

Daneben existieren weitere Eigenschaften, welche gewährleisten, dass das aufbauende Kryptosystem die gewünschte Sicherheit bietet. Diese sind in Standards (wie z.B. ANSI X9.62 oder IEEE P1363) festgeschrieben.

Die heutzutage bevorzugte Methode zur Konstruktion besteht in der Berechnung sogenannter Zufallskurven: dabei werden in einem ersten Schritt die Parameter zufällig gewählt. Der zweite Schritt besteht in der Berechnung der Punkteanzahl dieser konkreten Kurve. Damit hat man dann alle Informationen zur Verfügung, um die Eignung dieser Kurve beurteilen zu können. Diese beiden Schritte sind dann einfach so lange zu wiederholen, bis eine geeignete Kurve gefunden wurde.

Andere Verfahren (neben der Wahl von Zufallskurven) zur Konstruktion von geeigneten Kurven sind die Methode der Komplexen Multiplikation (CM) sowie die Wahl von sogenannten Koblitz-Kurven. Bei letzteren berechnet man die Anzahl der Punkte mittels des sogenannte Weil-Theorems.

Ein Nachteil der CM-Methode besteht darin, dass man nicht in der Lage ist, den zugrunde liegenden Körper, über dem die Kurve definiert wird, zu fixieren. Dies kann ein Problem darstellen, falls man eine besonderes effiziente Arithmetik in diesem Grundkörper verwenden will. Der Grund hierfür ist, dass in dem (zweigeteilten) Algorithmus im ersten Schritt eine Primzahl gesucht wird, so dass sichergestellt ist, dass eine geeignete Kurve über $\mathbf{Z}_p$ existiert. Erst im zweiten Schritt wird dann die konkrete Gleichung der Kurve konstruiert.

Das zweite (wesentlichere) Problem mit diesem Konstruktionsverfahren ist, dass für die so konstruierten Kurven eventuell Angriffsmöglichkeiten bestehen könnten, wie man im Falle anderer Kurven gesehen hat. Deshalb werden bei den in die Berechnung eingehenden Parametern Größenordnungen verwendet, die den ursprünglichen Vorteil (schnellere Konstruktion gegenüber Zufallskurven) zunichte machen können.

Der Nachteil der speziellen Struktur von Kurven trifft im noch größeren Maße auf die Konstruktionsmethoden mit Koblitz-Kurven zu: Dieses sind Kurven, die über einem relativ kleinen Körper definiert werden, deren Punktegruppe aber erst über einem hinreichend großen Erweiterungskörper betrachtet wird.

Für die Erstellung allgemeiner Anwendungen ist dagegen der Gebrauch standardisierter Parametersätze zu empfehlen: NIST (*National Institute of Standards),* ANSI und andere Standardisierungsgremien haben sich für die verschiedenen Sicherheitsstufen auf

Parametersätze geeinigt. Diese sind ebenfalls in der **cv act *library*** enthalten und können sofort verwendet werden.

Teil 2 Referenzhandbuch

Kapitel 4 Algorithmenkatalog

In diesem Kapitel werden alle Algorithmen alphabetisch aufgelistet und alle für die Implementierung erforderlichen, kryptographischen Verfahren, die in der **cv act *library*** enthalten sind, anhand ihrer wichtigsten Eigenschaften spezifiziert.

AES

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 128-Bit Blockchiffre

Schlüssellänge: 128, 192 oder 256 Bit

Name `AES`

Beschreibung Beim AES handelt es sich um eine iterierte Blockchiffre. Jede Runde besteht aus drei unabhängigen, invertierbaren Schichten. Die erste Schicht ist eine lineare Vertauschung, die zweite Schicht besteht aus der parallelen Anwendung von S-Boxen, die nichtlineare Eigenschaften haben. Die dritte Schicht besteht aus einer XOR-Verknüpfung des Rundenschlüssels mit dem Ergebnis der zweiten Schicht.

Vor der ersten Runde wird der Eingabeblock mit dem Schlüssel verknüpft.

Konstruktion AES ist kein Feistel-Netzwerk. Statt dessen besteht jede Runde aus drei unabhängigen, invertierbaren Transformationen, die jedes Eingabebit in gleicher Weise umwandeln. Die Wahl der einzelnen Transformationen ist so gewählt, dass sie Schutz gegen lineare und differentielle Kryptoanalysen bieten.

Komplexität In jeder Runde werden eine byteweise Substitution, ein byteweiser zyklischer Shift, eine modulare Multiplikation mit einem Polynom und eine byteweise XOR-Verknüpfung durchgeführt.

Sicherheit Es sind bisher keine Angriffe auf AES bekannt.

An dieser Stelle ist allerdings darauf hinzuweisen, dass der Algorithmus noch zu neu ist, um eine abschließende Beurteilung abgeben zu können.

Andererseits erfreut sich der Algorithmus als US-Regierungsstandard besonderer Aufmerksamkeit. Die Tatsache, dass er sich im AES-Wettbewerb durchgesetzt hat, deutet auf einen kryptographisch starken Entwurf hin.

Restriktionen keine

Bemerkungen AES wurde von Joan Daemen und Vincent Rijmen aus Belgien entwickelt.

Blowfish

Verfahren	symmetrisches Chiffrierverfahren
Parameter	Blocklänge/-art: 64-Bit Blockchiffre Schlüssellänge: 8-448 Bit
Name	BlowFish
Beschreibung	Blowfish besteht aus zwei Teilen, nämlich Schlüsselexpansion und Datenverschlüsselung. Zur Datenverschlüsselung wird eine einfache Funktion verwendet, die sechzehnmal durchlaufen wird. Jede Runde ist aufgeteilt in eine schlüsselabhängige Permutation sowie eine schlüssel- und datenabhängige Substitution.

Konstruktion Blowfish ist ein Feistel-Netzwerk mit variabler Rundenzahl.

Komplexität Der Algorithmus verwendet folgende Operationen auf 32 Bit Wörtern: Addition, XOR, Tabellenindizierung.Während jeder Verschlüsselung wird eine große Anzahl von Teilschlüsseln benötigt, die im voraus zu berechnen sind. An dieser Stelle ist ein Prozessor mit großem

Datencache von Vorteil.

Sicherheit Bisher ist keine erfolgreiche Kryptanalyse bekannt, zur sicheren Verwendung sollte der Algorithmus mit voller Rundenzahl implementiert werden. Die Existenz schwacher Schlüssel ist bekannt. Die Frage, ob es sich bei einem gegebenen Schlüssel um einen schwachen Schlüssel handelt, läßt sich allerdings erst nach der Schlüsselexpansion entscheiden. Unklar ist, ob sich diese Tatsache zur Kryptanalyse ausnutzen läßt.

Restriktionen keine

Bemerkungen Blowfish wurde von B. Schneier entwickelt, der frei verfügbare Sourcecode in C findet sich in [Sch97].Der Algorithmus ist wegen des hohen Speicherplatzbedarfs ungeeignet für Chipkarten. Ebenso ist Blowfish nicht geeignet für Anwendungen, bei denen der Schlüssel häufig gewechselt werden muss.

CAST-128

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 64-Bit Blockchiffre

Schlüssellänge: 40-128 Bit

Name `Cast128`

Beschreibung CAST 128 besteht aus einem Feistel-Netzwerk mit bis zu 16 Runden. Zur Verschlüsselung wird zunächst der Klartextblock in zwei Hälften zerlegt. In jeder Runde wird die rechte Hälfte mit Hilfe einer der drei Funktionen f, g und h mit bestimmten Bestandteilen des Schlüssels kombiniert und mit der linken Hälfte XOR-verknüpft, was die neue rechte Hälfte ergibt. Die ursprüngliche rechte Hälfte (vor der jeweiligen Runde) wird zur neuen linken Hälfte. Nach der sechzehnten Runde werden die beiden Hälften zusammengefügt und ergeben zusammen den Chiffretext.Welche der drei Funktionen f, g oder h in der jeweiligen Runde verwendet wird, hängt von der Runde ab.

Konstruktion Der Algorithmus wurde zur Verwendung mit unterschiedlichen Schlüsselängen entwickelt. Erlaubt sind 40 bis 128 Bit in 8 Bit Schritten. Ist die Schlüssellänge kleiner als 128 Bit, werden von rechts Nullen eingefügt.

Die Zahl der Runden hängt von der Länge des Schlüssels ab

(zwischen 12 und 16).

Komplexität Der Algorithmus verwendet folgende Operationen auf 32 Bit Wörtern:

Links-shift, Addition und Subtraktion modulo 2^{32}.

Für die im Rahmen der Funktionen f, g und h verwendeten acht S-Boxen werden acht KByte Speicherplatz benötigt. Falls die Generierung der Rundenschlüssel im voraus durchgeführt wird, werden während der Verschlüsselung nur vier KByte benötigt.

Sicherheit CAST-128 erfüllt das starke Avalanche-Kriterium.

Es sind keine schwachen Schlüssel bekannt.

Restriktionen CAST-128 ist in RFC 2144 beschrieben.
Der Algorithmus ist patentiert, eine abgabenfreie Lizenz ist erhältlich.

Bemerkungen In der Regel werden Schlüssellängen von 40, 64, 80, 128 Bit verwendet.

CBC-MAC

Verfahren MAC basierend auf symmetrischen Verfahren

Parameter Hashwert: Die Länge des erzeugten Tags ist abhängig vom verwendeten symmetrischen Chiffrierverfahren

Name CBCMAC

Beschreibung Eine einfache Möglichkeit zur Konstruktion eines MAC ist die Verschlüsselung einer Nachricht mit einem Blockalgorithmus im CBC-Modus. Der letzte Block findet als Tag Verwendung.

Konstruktion Ein CBC-MAC ist als schlüsselabhängige Einweg-Hashfunktion zu sehen. Neben den üblichen Eigenschaften einer Hasfunktion wird zur Verifizierung der Schlüssel benötigt.

Komplexität Der Aufwand für die Berechnung des Tags hängt vom gewählten Hashalgorithmus ab.

Sicherheit Die Sicherheit des CBC-MAC hängt vom verwendeten Verschlüsselungsverfahren ab.

Ein potentielles Sicherheitsproblem besteht darin, dass der Empfänger den Schlüssel kennen muss. Mit diesem Schlüssel kann er aber andere Nachrichten mit demselben Tag erzeugen.

Restriktionen Die patentrechtliche Frage hängt von der gewählten Blockchiffre ab.

Bemerkungen MAC Algorithmen werden z. B. beim Homebanking zur Authentifizierung eingesetzt.

DES

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 64-Bit Blockchiffre

Schlüssellänge: 64 Bit, davon 8 Paritätsbits

Name DES

Beschreibung DES arbeitet auf einem 64 Bit Block des Klartextes. Nach einer Eingangspermutation wird dieser Block in eine jeweils 32 Bit lange rechte und linke Hälfte zerlegt. Jetzt folgen 16 Runden identischer Operationen, in denen die Daten mit dem Schlüssel kombiniert werden. Nach der sechzehnten Runde werden rechte und linke Hälfte zusammengefügt. Eine Schlußpermutation, die zur Eingangspermutation invers ist, schließt den Algorithmus ab.

Konstruktion Feistel Netzwerk mit Permutationen, Expansionen und Substitutionen.

Komplexität Für jeden Ausgabeblock des Algorithmus ist ein 64-Bit String zu übertragen.

Sicherheit Der Schlüsselraum des DES umfaßt nur 2^{56} verschiedene Schlüssel, davon sind einige als schwache Schlüssel bekannt.

Der DES ist invariant unter binärer Komplementierung. Dadurch reduziert sich der Aufwand für die Suche nach einem unbekannten Schlüssel um den Faktor 2, wenn ein Kryptanalytiker über zwei Klartext/Chiffretext Paare zu binär komplementären Klartexten verfügt.

Mit der Methode der differentiellen Kryptanalyse wird ein Komplexitätsgrad von 2^{47} erreicht. Da die Mächtigkeit des Schlüsselraumes des DES 2^{56} beträgt, wird durch diese Methode die Sicherheit des DES

verringert.

Mit der linearen Kryptanalyse ist es möglich, eine *known plaintext*-Attacke mit 2^{47} bekannten Klartexten zu unternehmen.

DES ist vollständig, d.h. jedes Bit der Ausgabe hängt von jedem Eingabebit ab. Der DES erfüllt das strikte AvalancheKriterium in hohem Maße.

DES ist mit seinen 56 Bit effektiver Schlüssellänge inzwischen als nicht mehr sicher einzustufen; gleichwohl ist er natürlich in der Bibliothek enthalten, um Kompatibilität mit bestehenden Anwendungen zu gewährleisten.

Restriktionen IBM besitzt das Patent zu DES, dieses ist jedoch bereits abgelaufen.

Bemerkung DES stellt seit über zwanzig Jahren einen internationalen Standard dar, der jahrelange Kryptanalyse erstaunlich gut überstanden hat.

Diffie-Hellman (DH)

Verfahren Verfahren zum Schlüsselaustausch

Parameter Man arbeitet mit folgenden Parametern:

- Blocklänge: abhängig von den gewählten *domain parametern*
- Schlüssellänge: abhängig von den gewählten *domain parametern*

Name DH

Beschreibung Erstes und bekanntestes Schlüsselaustauschverfahren; mit der Vorstellung dieses Protokolls begann die Entwicklung der Public Key-Kryptographie. Zwei Kommunikationspartner einigen sich auf eine Primzahl p, so dass p-1 einen großen Primteiler besitzt, und eine Zahl g.

> Jeder Kommunikationspartner wählt eine Zufallszahl (x bzw. y) und berechnet g^x mod p bzw. g^y mod p. Diese Werte tauschen sie aus und berechnen den Wert g^{xy} mod p, der dann als gemeinsamer Schlüssel fungiert.

Konstruktion Der Algorithmus beruht auf dem Problem der Berechnung

diskreter Logarithmen (DL) in endlichen Körpern.

Anstelle von GF(p) kann auch eine beliebige andere Gruppe verwendet werden.

Komplexität Jeder Kommunikationspartner muss eine Zufallszahl erzeugen und zwei (modulare) Exponentiationen durchführen.

Sicherheit Die Sicherheit von Diffie-Hellman hängt von der Schwierigkeit des DL-Problems in der gewählten Gruppe ab. Entwicklungen auf diesem Gebiet müssen deshalb berücksichtigt werden. Bei Implementierung in $\mathbf{Z}_p$ sollte p groß genug gewählt werden (mindestens 1024 Bit).

Ein Angriff auf das Verfahren resultiert daraus, daß die Authentizität der Teilnehmer durch das Protokoll nicht gewährleistet ist. Es existieren Abwandlungen bzw. Ergänzungen des Protokolls, die diesen Angriff verhindern. Das Verfahren benötigt keine vertrauenswürdige dritte Instanz. Damit ist nur den beiden Kommunikationspartnern der ausgehandelte geheime Schlüssel bekannt.

Restriktionen Diffie-Hellman war in den USA und Kanada patentiert, das US-Patent ist am 29.04.1997 abgelaufen.

Bemerkung Der Diffie-Hellman-Algorithmus war der erste Public-Key-Algorithmus, der patentiert wurde. Er kann nicht zur Verschlüsselung von Nachrichten verwendet werden, sondern nur zum sicheren Austausch von Schlüsseln.

Diffie-Hellman kann auch in anderen Strukturen verwendet werden (siehe EC-DH). Die Berechnungen sind eventuell deutlich schneller.

DLIES

Verfahren asymmetrisches Chiffrierverfahren, welches aus einem Schlüsselaustauschverfahren und einem MAC besteht.

Parameter abhängig von den gewählten Algorithmen

Name IES mit DH als Austauschverfahren

Beschreibung Nach der Auswahl der zugehörigen Parameter wird ein Schlüsselpaar erzeugt. Aus dem eigenen geheimen Schlüssel und dem öffentlichen Schlüssel des Kommunikationspartners wird ein gemeinsamer Schlüssel erzeugt. Die Nachricht wird mit diesem

Schlüssel verschlüsselt und mit einem MAC versehen, um die Authentizität zu gewährleisten.

Konstruktion DLIES ist ein Verschlüsselungsschema, das einen Rahmen für eine asymmetrische Verschlüsselung vorgibt, der einige Wahlmöglichkeiten läßt. Die Bestandteile sind vom Benutzer auszuwählen.

Komplexität Der Rechenaufwand hängt von den gewählten Algorithmen ab.

Sicherheit Die Sicherheit beruht auf dem Problem des Diskreten Logarithmus auf der gewählten Gruppe, sowie auf der Sicherheit der anderen Verfahren.

Restriktionen keine

Bemerkungen Das Verfahren ist in IEEE 1363 standardisiert.

Domain Parameter

Die auf dem Diskreten Logarithmus basierenden Verfahren brauchen zur Laufzeit Informationen über die Gruppe, in welcher zu rechnen ist. Diese Parameter, welche den eigentlichen Algorithmus festlegen, nennt man *domain parameter;* sie werden in jedem DL-Schema gebraucht und sind implizite Komponenten von jedem DL-Schlüssel.

Diese Parameter müssen nicht geheim gehalten werden, so sie vom Sicherheitsstandpunkt aus gesehen unkritisch sind. Allerdings ist zu beachten, dass diese Daten authentisch sein müssen, um eine korrekte Kommunikation zu gewährleisten. Prinzipiell könnte dabei jeder Kommunikationspartner seine eigenen *domain parameter* verwenden. In der Praxis werden diese aber für einen größeren Benutzerkreis identisch sein, so dass die Notwendigkeit für einen Austausch dieser Informationen entfällt.

Die *domain parameter* unterscheiden sich im Fall der beiden zugrundeliegenden Probleme: Der eine Fall, wie z.B. im DSA verwandt, ist das DL-Problem in einem endlichen Körper und der andere Fall ist das entsprechende Problem in der Punktegruppe elliptischer Kurven. Wir bezeichnen die entsprechenden Datensätze im folgenden mit DL-*domain parameter* und EC-DL-*domain parameter.*

Die DL-*domain parameter* sind:

- eine Primzahl p
- ein Element g des Körpers mit einer Ordnung r.
- evtl. der *Cofactor* h (gegeben durch die Beziehung n = h*r, wobei n die Ordnung der gesamten Gruppe ist, also n=p-1)

Die EC-DL- *domain parameter* sind analog dazu:

Falls der zugrundeliegende Körper GF(p) (p prim) ist:

- eine Primzahl p, die festlegt, über welchem Körper die Kurve betrachtet wird.
- die Koeffizienten A und B der elliptischen Kurve E: $y^2 = x^3 + A\,x + B$.

Falls der zugrundeliegende Körper GF(2^m) ist:

- eine Zahl m, die festlegt, über welchem Körper die Kurve betrachtet wird.
- die Koeffizienten A und B der elliptischen Kurve E: $y^2 + xy = x^3 + A\,x^2 + B$.

Zusätzlich in beiden Fällen:

- eine positive, ganze Zahl r, die die Anzahl n der insgesamt auf der elliptischen Kurve vorhandenen Punkte teilt.

- ein Punkt P auf der elliptischen Kurve mit der Ordnung r, d.h. ein erzeugendes Element der Untergruppe von Ordnung r; dieser wird auch Basispunkt genannt.

- evtl. der *Cofactor* h (gegeben durch die Beziehung n = h*r).

Die Erzeugung dieser *domain parameter* ist in den beiden geschilderten Fällen mit deutlich unterschiedlichem Rechenaufwand verbunden. Da sie aber in der Regel Bestandteil der Infrastruktur sind, bedeutet dies kein Problem. Im Falle elliptischer Kurven (was der rechentechnisch komplexere Fall ist) kann auch auf Datensätze verschiedener Institutionen (z.B. ANSI, NIST) zurückgegriffen werden; die entsprechenden Daten sind in der **cv act *library*** enthalten.

DSA

Verfahren digitaler Signaturalgorithmus

Parameter Man arbeitet mit folgenden Parametern:

- n_k= 512-1024 Bit (in 64 Bit-Schritten)
- p, q und g haben eine jeweilige Länge von n_k, 160 und n_k Bit
- der öffentlicher Schlüssel y hat die Länge n_k Bit
- der geheime Schlüssel x hat die Länge 160 Bit
- Länge der Signatur (r, s) : (160,160) Bit

Name DSA

Beschreibung Mittels modularer Arithmetik wird unter Verwendung des öffentlichen Schlüssels, einer zufällig gewählten Zahl k und der Hashfunktion SHA-1 eine Signatur erzeugt.

Konstruktion DSA beruht auf dem Problem des Diskreten Logarithmus in GF(p).

Komplexität Im wesentlichen wird eine modulare Exponentation zur Erzeugung der Signatur und zwei zur Verifizierung benötigt.

Sicherheit Der private Schlüssel lässt sich berechnen, falls k bekannt ist oder ein k für zwei Signaturen verwendet wird. Deswegen muss für jede Signatur ein neues k mit einem Zufallszahlengenerator ermittelt werden.

Die Sicherheit des DSA beruht auf zwei verschiedenen, aber

verwandten Problemen:

Einerseits beruht sie auf dem allgemeinen DL-Problem in $\mathbf{Z}_p^*$ (für welches ähnlich wie beim Faktorisierungsproblem subexponentielle Angriffsmethoden existieren).

Andererseits beruht die Sicherheit von DSA auf dem DL-Problem in der von g erzeugten Untergruppe mit Ordnung q. Für das letztere benötigen die besten bekannten Angriffe in der Größenordnung von $\sqrt{(q)}$ Operationen; die Parameter der aktuellen Implementierung sind so gewählt, daß beide Angriffsmethoden ungefähr gleichen Aufwand erfordern. DSA mit 512 Bit Schlüssel ist nicht mehr als sicher einzustufen. Um bestmögliche Sicherheit zu erlangen, ist die Verwendung eines 1024 Bit Schlüssels empfehlenswert.

Restriktionen Der Algorithmus ist Teil des vom NIST herausgegebenen DSS Standards.
D. Kravitz ist Inhaber des Patents für DSA.

Bemerkungen Aufgrund der erzielten Ergebnisse bei Faktorisierungs-problemen ist inzwischen klar, daß DSA mit 512 Bit nicht mehr als sicher einzustufen ist (Angriffe hätten eine vergleichbare Komplexität wie bei den erfolgreichen Attacken auf 512 Bit RSA-Schlüssel). Schlüssellängen um 768 Bit sind ebenfalls nur für eine Übergangszeit verwendbar; es erscheint auf jeden Fall sinnvoll, diesen Algorithmus nur mit der maximal möglichen Schlüssellänge von 1024 Bit zu verwenden.

EC-DH

Verfahren Verfahren zum Schlüsselaustausch

Parameter Man arbeitet mit folgenden Parametern:

- Blocklänge: abhängig von den domain parametern.
- Schlüssellänge: abhängig von den domain parametern.
- Länge von x und y: n Bit (empfohlen > 160 Bit)

Name ECDH

Beschreibung Zwei Kommunikationspartner einigen sich auf eine elliptische Kurve und einen Punkt W auf der Kurve. Jeder Kommunikationspartner wählt eine Zufallszahl (x bzw. y) und berechnet x*W bzw. y*W. Diese Werte tauschen sie aus und berechnen x*y*W. Sollte das Ergebnis einer der drei Berechnungen der Punkt Unendlich sein, muss das Verfahren mit anderen x und y wiederholt werden.

Konstruktion Ein Verfahren, dessen Sicherheit auf dem diskreten Logarithmusproblem (DLP) in endlichen Körpern beruht, kann auf elliptische Kurven übertragen werden, so daß die Sicherheit auf dem

diskreten Logarithmusproblem über elliptischen Kurven beruht.

Komplexität Jeder Kommunikationspartner muss eine Zufallszahl erzeugen und zwei Multiplikationen auf elliptischen Kurven durchführen.

Sicherheit Die Wahl der elliptischen Kurve muss geeignet erfolgen.

Es sollten keine supersinguläre oder anomalen Kurven verwendet werden, da für diese Kurven ein Reduktionsalgorithmus auf das Problem des Diskreten Logarithmus über endlichen Körpern gefunden wurde.

Die Sicherheit hängt entscheidend von der Länge in Bit von x und y ab. Zur Zeit gelten Bitlängen ab 160 als sicher.

Restriktionen keine

EC-DSA

Verfahren digitaler Signaturalgorithmus

Parameter Man arbeitet mit folgenden Parametern:

- n_k= 160 Bit (mindestens)
- P Punkt auf der elliptischen Kurve der Ordnung n_k
- d privater Schlüssel, beliebige Zufallszahl kleiner als n_k
- Q = d*P öffentlicher Schlüssel

Name ECDSA

Beschreibung DSA beruht auf dem Diskreten Logarithmusproblem in endlichen Körpern. Diese Verfahren wurde für EC-DSA auf die Verwendung elliptische Kurven übertragen.

Mittels Arithmetik auf einer elliptischen Kurve wird unter Verwendung des öffentlichen Schlüssels, einer zufällig gewählten Zahl k und der Hashfunktion SHA-1 eine Signatur erzeugt.

Konstruktion EC-DSA beruht auf dem Diskreten Logarithmusproblem auf elliptischen Kurven über endlichen Körpern.

Komplexität Der Rechenaufwand besteht im Wesentlichen aus einer Gruppenoperation bei der Erstellung einer digitalen Signatur und zwei Operationen bei der Verifizierung.

Sicherheit Der private Schlüssel lässt sich berechnen, falls k bekannt ist

oder ein k für zwei Signaturen verwendet wird. Deswegen muss für jede Signatur ein neues k mit einem Zufallszahlengenerator ermittelt werden. Bei der Verwendung des gleichen q von vielen Benutzern bestehen eventuell Angriffsmöglichkeiten.

Restriktionen keine

Bemerkung Die Wahl der elliptischen Kurve muss geeignet erfolgen.

EC-IES

Verfahren	asymmetrisches Chiffrierverfahren, welches aus einem Schlüsselaustauschverfahren und einem MAC besteht.
Parameter	abhängig von den gewählten Algorithmen
Name	IES mit ECDH als Austauschverfahren
Beschreibung	Nach der Auswahl einer elliptischen Kurve und der zugehörigen Parameter wird ein Schlüsselpaar erzeugt. Aus dem eigenen geheimen Schlüssel und dem öffentlichen Schlüssel des Kommunikationspartners wird ein gemeinsamer Schlüssel erzeugt. Die Nachricht wird mit diesem Schlüssel verschlüsselt und mit einem MAC versehen.
Konstruktion	EC-IES ist ein Verschlüsselungsschema, das einen Rahmen für eine asymmetrische Verschlüsselung vorgibt, der einige Wahlmöglichkeiten lässt.
Komplexität	Der Rechenaufwand hängt von den gewählten Algorithmen ab.
Sicherheit	Die Sicherheit beruht auf dem Problem des Diskreten Logarithmus in der gewählten Gruppe, sowie auf der Sicherheit der anderen Verfahren.
Restriktionen	keine
Bemerkungen	Das Verfahren ist in IEEE 1363 standardisiert.

EC-MQV

Verfahren Verfahren zum Schlüsselaustausch

Parameter Man arbeitet mit folgenden Parametern:

- Blocklänge: abhängig von den domain *parametern*
- Schlüssellänge: abhängig von Kurve
- Länge von x und y: n Bit (empfohlen > 160 Bit)

Name Nicht im Lieferumfang der library enthalten.

Beschreibung Zwei Kommunikationspartner einigen sich auf eine elliptische Kurve. Jeder der beiden Kommunikationspartner errechnet den gemeinsamen geheimen Schlüssel aus seinen zwei geheimen Schlüsseln und den beiden öffentlichen Schlüsseln des anderen Kommunikationspartners. Dabei wird mit Hilfe von Arithmetik auf elliptischen Kurven, insbesondere der Multiplikation, der gemeinsame Sitzungsschlüssel errechnet.

Konstruktion Der Algorithmus beruht auf dem Problem der Berechnung diskreter Logarithmen (EC-DL) auf elliptischen Kurven über endlichen Körpern.

Komplexität Von jedem Kommunikationspartner wird im wesentlichen eine Multiplikation auf elliptischen Kurven durchgeführt.

Sicherheit Das Verfahren benötigt keine vertrauenswürdige dritte Instanz. Damit ist nur den beiden Kommunikationspartnern der ausgehandelte geheime Schlüssel bekannt.

Eine der Stärken des Verfahrens besteht in der Authentifikation der beiden Teilnehmer.

Restriktionen keine

Bemerkungen Dieses Protokoll für den Schlüsselaustausch wurde 1995 von A. Menezes, M. Qu und S. Vanstone entwickelt. Es basiert im Wesentlichen (wie fast alle Schlüsselaustauschprotokolle) auf dem DH-Protokoll.

EC-MQV ist in IEEE 1363 standardisiert.

EC-NR

Verfahren digitaler Signaturalgorithmus

Parameter Man arbeitet mit folgenden Parametern:

- n_k= 160 Bit (mindestens)
- P Punkt auf der elliptischen Kurve der Ordnung n_k

- d privater Schlüssel, beliebige Zufallszahl kleiner als n_k
- Q = d*P öffentlicher Schlüssel

Name Nicht im Lieferumfang der library enthalten.

Beschreibung Signaturalgorithmus vom ElGamal-Typ mit der Möglichkeit des *message recovery* (in Übereinstimmung mit den existierenden Standards wird in der vorliegenden Implementierung auf diese Möglichkeit verzichtet).

Nach Wahl der zugrundeliegenden Kurve wird ein temporäres Schlüsselpaar (u,V) bestimmt, dabei ist V ein Punkt auf der elliptischen Kurve. Aus der x-Koordinate dieses Punktes, dem privaten Schlüssel und dem Hashwert der Nachricht wird die Signatur berechnet.

Konstruktion EC-NR beruht auf dem Diskreten Logarithmusproblem auf elliptischen Kurven über endlichen Körpern.

Komplexität Zur Berechnung der Signatur ist im wesentlichen eine Multiplikation auf der elliptischen Kurve erforderlich.

Sicherheit Entscheidend für die Sicherheit ist, dass für jede Signatur ein neues, temporäres Schlüsselpaar gewählt wird. Ansonsten basiert die Sicherheit auf dem DL-Problem in der betrachteten Gruppe.

Restriktionen Wie bei allen ElGamal-Signaturschemen besteht die Möglichkeit des Konfliktes mit dem Schnorr-Patent.

Bemerkungen Die Wahl der elliptischen Kurve muss geeignet erfolgen.

Der Algorithmus ist von ISO standardisiert.

Hash-MAC

Verfahren MAC basierend auf Hashfunktionen

Parameter Der Hashwert ist abhängig vom verwendeten Hashverfahren

Name `HashMAC`

Beschreibung Bei diesem Verfahren müssen beide Kommunikationspartner über einen gemeinsamen Schlüssel K verfügen. Dann berechnet man von der Konkatenierung den Hashwert: H (K || M || K), der dann den Tag darstellt, wobei M die Nachricht ist.

Konstruktion Ein Hash-MAC ist als eine schlüsselabhängige Einweg-Hashfunktion zu sehen. Neben den üblichen

Eigenschaften einer Hashfunktion wird zur Verifizierung eines Hashwertes der Schlüssel eines symmetrischen Verschlüsselungsverfahrens benötigt.

Komplexität Der Aufwand für die Berechnung des Tags hängt vom gewählten Hashalgorithmus ab.

Sicherheit Die Sicherheit dieses MAC-Verfahrens hängt von der Geheimhaltung des Schlüssels und vom verwendeten Hashverfahren ab.

Restriktionen Die patentrechtliche Frage hängt von dem gewählten Hashverfahren ab.

Bemerkungen MAC-Algorithmen werden z. B. im Homebanking zur Authentifizierung eingesetzt.

IDEA

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 64-Bit Blockchiffre

Schlüssellänge: 128 Bit

Name Nicht im Lieferumfang der library enthalten.

Beschreibung Aus dem Schlüssel werden 52 Teilschlüssel à 16 Bit erzeugt. Die Ver- und Entschlüsselung der 64-Bit Blöcke erfolgt durch die Aufteilung in je vier 16-Bit-Blöcke, die dann mit den Teilschlüsseln durch die Verwendung dreier untereinander inkompatibler Gruppenoperationen auf 16-Bit verbunden werden: bitweises XOR, Addition modulo 16 und Multiplikation modulo $2^{16}+1$.

Konstruktion IDEA verwendet eine Mischung von Operationen unterschiedlicher algebraischer Gruppen (s. o.). Die Implementation in Hardware ist besonders einfach. Es handelt sich um eine Feistel-Netzwerk.

Komplexität Rechenaufwand beim Ver- bzw. Entschlüsseln:

48 XOR-Verknüpfungen (16 Bit), 34 Additionen modulo 2^{16}, 34 Multiplikationen modulo $2^{16}+1$.

Sicherheit Die einzige bisher bekannte Angriffsmöglichkeit stellen sogenannte schwache Schlüssel dar, deren Verwendung sich durch Unterschieben ausgewählter Klartexte beweisen lässt.

Restriktionen Der Patentinhaber ist Ascom Tech AG.

Bemerkungen Der Algorithmus ist in einer Zusammenarbeit der ETH Zürich und des Patentinhabers entstanden.

Der Bekanntheitsgrad wurde durch die Implementierung in der Software PGP deutlich gefördert.

MARS

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 128-Bit Blockchiffre

Schlüssellänge: 128, 192 oder 256 Bit

Name Mars

Beschreibung Nach einer Schlüsselexpansion teilt MARS jeden 128-Bit Eingabeblock in vier 32-Bit Teilblöcke auf.

Der eigentliche, 16 Runden lange Verschlüsselungsalgorithmus verwendet modulare Arithmetik, um Daten und Schlüssel miteinander zu verknüpfen. Er wird von einer Eingangs- und einer Ausgangspermutation eingeschlossen. MARS war ein AES-Kandidat.

Konstruktion Der Algorithmus arbeitet auf 32-Bit Wörtern mit folgenden Operationen:

XOR, Addition, Subtraktion, Multiplikation (alle modulo 2^{32}) und Permutation.

MARS ist ein Feistel-Netzwerk und benutzt eine einzelne

S-Box. Diese S-Box soll lineare und differentielle Kryptanalyse verhindern und den Lawineneffekt verstärken.

Komplexität Es werden drei Permutationen, 58 XOR-Verknüpfungen, 161 Shift-Operationen, 58 Additionen, 15 Multiplikationen und 28 Subtraktion (mod 2^{32}) durchgeführt.

Sicherheit Es sind bisher keine Angriffe auf MARS bekannt.

An dieser Stelle ist allerdings darauf hinzuweisen, dass der Algorithmus noch zu neu ist, um eine abschließende Beurteilung abgeben zu können.

Andererseits erfreut sich der Algorithmus als AES-Kandidat besonderer Aufmerksamkeit. Die Tatsache, dass er sich für die letzte AES-Runde qualifiziert hat, deutet auf einen kryptographisch starken Entwurf hin.

Restriktionen IBM besitzt ein Patent, vergibt aber nach eigenen Angaben kostenlose Lizenzen.

Bemerkungen Der Algorithmus wurde in IBM-Forschungszentren entwickelt.

MD2

Verfahren Hashfunktion

Parameter Hashwert: 128 Bit

Name MD2

Beschreibung MD2 wurde für eine 8-Bit Architektur entwickelt.

Die Nachricht M wird (nach Auffüllen, falls notwendig) in Blöcke der Länge 16 Byte unterteilt. MD2 wird mit dem ersten Block und einer XOR-Verknüpfung aus erstem und zweitem Block gestartet. Die weiteren Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung.

Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion MD2 ist definiert als iterative Anwendung einer Kompressionsfunktion.

Komplexität Es werden folgende Operationen verwendet:

Byteweise XOR-Verknüpfung und Addition modulo 256.Der Algorithmus ist langsamer als die meisten anderen Hashfunktionen.

Sicherheit Eine Hashfunktion mit Hashwerten von 128 Bit Länge ist nach dem heutigen Stand der Technik nicht mehr als sicher einzuschätzen.

Restriktionen keine

Bemerkung MD2 ist aus Kompatibilitätsgründen enthalten. Der Algorithmus wurde von R. Rivest bei RSA entwickelt, ist in RFC 1319 standardisiert und wird in den PEM-Protokollen benutzt.

MD4

Verfahren Hashfunktion

Parameter Blocklänge: 512 Bit

Hashwert: 128 Bit

Name MD4

Beschreibung Eine speziell für den Einsatz auf 32-Bit Maschinen konzipierte Hashfunktion mit 3 Runden. Nach der Aufteilung des Eingabetextes in Blöcke fester Länge (512 Bit) wird in drei Runden unter Verwendung von Verkettungsvariablen und nichtlinearen Funktionen als Ausgabe ein Hashwert der Länge 128 Bit erzeugt.

Konstruktion MD4 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden folgende Operationen auf 32 Bit Wörtern verwendet: Addition modulo 2^{32}, bitweise definierte Boolesche Operationen und zyklische Shifts. Diese Operationen können auf den heute üblichen 32-Bit Prozessoren sehr schnell ausgeführt werden. Der Algorithmus ist für *Little Endian*-Architekturen entwickelt worden.

Sicherheit Eine Hashfunktion mit Hashwerten von 128 Bit Länge ist nach dem heutigen Stand der Technik nicht mehr als sicher einzuschätzen.

Restriktionen keine

Bemerkung Dieser Algorithmus ist seit einigen Jahren geknackt. MD4 wurde in die Bibliothek lediglich aufgenommen, um gegebenenfalls vorhandene Daten bearbeiten zu können. Der Algorithmus ist in RFC 1320 definiert; das Dokument enthält eine Beispielimplementierung in C.

MD5

Verfahren Hashfunktion

Parameter Blocklänge: 512 Bit

Hashwert: 128 Bit

Name MD5

Beschreibung Eine Hashfunktion der MD4-Familie, jedoch mit vergrößerter

Rundenanzahl. Nach der Aufteilung des Eingabetextes in Blöcke fester Länge (512 Bit) wird in vier Runden unter Verwendung von Verkettungsvariablen und nichtlinearen Funktionen als Ausgabe ein Hashwert der Länge 128 Bit erzeugt.

Konstruktion MD5 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden folgende Operationen auf 32 Bit Wörtern verwendet: Addition modulo 2^{32}, bitweise definierte Boolesche Operationen und zyklische Shifts. Diese Operationen können auf den heute üblichen 32-Bit Prozessoren sehr schnell ausgeführt werden. Der Algorithmus ist für *Little Endian*-Architekturen entwickelt worden.

Sicherheit Ende 2008 wurde ein Kollisionsangriff auf MD5 erfolgreich durchgeführt, wenn auch mit vergleichsweise hohem Aufwand. Aus diesem Grund ist von einer Verwendung des Verfahrens soweit möglich abzusehen. Aufgrund seiner weiten Verbreitung wurde das Verfahren trotzdem in die Bibliothek aufgenommen.

Restriktionen keine

Bemerkung In RFC 1321 findet sich eine Referenzimplementierung in C. Der Algorithmus findet sich in diversen Dienstprogrammen, wie z. B. PGP.

MQV

Verfahren Verfahren zum Schlüsselaustausch

Parameter Man arbeitet mit folgenden Parametern:

- Blocklänge: abhängig von den gewählten *domain parametern*
- Schlüssellänge: abhängig von den gewählten *domain parametern*

Name Nicht im Lieferumfang der library enthalten.

Beschreibung Erweiterung des DH-Protokolls mit dem Ziel der impliziten Authentifizierung der erzeugten Sitzungsschlüssel.

Konstruktion Der Algorithmus beruht auf dem Problem der Berechnung diskreter Logarithmen (DL) in endlichen Körpern.

Komplexität Von jedem Kommunikationspartner werden im wesentlichen

zwei Exponentiationen durchgeführt.

Sicherheit Das Verfahren benötigt keine vertrauenswürdige dritte Instanz. Damit ist nur den beiden Kommunikationspartnern der ausgehandelte geheime Schlüssel bekannt.

Eine der Stärken des Verfahrens besteht in der Authentifikation der beiden Teilnehmer.

Restriktionen keine

Bemerkungen Dieses Protokoll für den Schlüsselaustausch wurde 1995 von A. Menezes, M. Qu und S. Vanstone entwickelt. Es basiert im Wesentlichen (wie fast alle Schlüsselaustauschprotokolle) auf dem DH-Protokoll.

MQV ist in IEEE 1363 standardisiert.

NR

Verfahren digitaler Signaturalgorithmus

Parameter Schlüssellänge: abhängig von den verwendeten *domain parametern*

Name Nicht im Lieferumfang der library enthalten.

Beschreibung Signaturalgorithmus vom ElGamal-Typ mit der Möglichkeit des *message recovery* (in Übereinstimmung mit den existierenden Standards wird in der vorliegenden Implementierung auf diese Möglichkeit verzichtet).

Konstruktion NR beruht auf dem Diskreten Logarithmusproblem.

Komplexität Zur Berechnung der Signatur ist im wesentlichen eine modulare Exponentiation erforderlich.

Sicherheit Entscheidend für die Sicherheit ist, dass für jede Signatur ein neues, temporäres Schlüsselpaar gewählt wird. Ansonsten basiert die Sicherheit auf dem DL-Problem in der betrachteten Gruppe.

Restriktionen Wie bei allen ElGamal-Signaturschemen besteht die Möglichkeit des Konfliktes mit dem Schnorr-Patent.

Bemerkungen Der Algorithmus ist von ISO standardisiert.

RC6™

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 128-Bit Blockchiffre

Schlüssellänge: 128, 192 oder 256 Bit

Name Nicht im Lieferumfang der library enthalten.

Beschreibung Nach einer Schlüsselexpansion teilt RC6 jeden 128-Bit Eingabeblock in vier 32-Bit Teilblöcke auf.

Der eigentliche, 20 Runden lange Verschlüsselungsalgorithmus verwendet unter anderem modulare Arithmetik, um Daten und Schlüssel miteinander zu verknüpfen. Er wird von einer Eingangs- und einer Ausgangspermutation eingeschlossen.

War ein AES-Kandidat.

Konstruktion Der Algorithmus arbeitet auf 32-Bit Wörtern mit folgenden Operationen: XOR, Addition, Subtraktion, Multiplikation (alle modulo 2^{32}) und Permutation.

Komplexität Es werden 84 Addition, 40 Multiplikation (beide modulo Blockgröße), 40 XOR-Verknüpfungen, 80 shift-Operationen und eine Permutation der Teilblöcke durchgeführt.

Sicherheit Es sind bisher keine Angriffe auf RC6 bekannt.

An dieser Stelle ist allerdings darauf hinzuweisen, dass der Algorithmus noch zu neu ist, um eine abschließende Beurteilung abgeben zu können.

Andererseits erfreut sich der Algorithmus als AES-Kandidat besonderer Aufmerksamkeit. Die Tatsache, dass er sich für die letzte AES-Runde qualifiziert hat, deutet auf einen kryptographisch starken Entwurf hin.

Restriktionen RSA-Labs besitzt ein Patent, vergibt aber nach eigenen Angaben kostenlose Lizenzen. Der Name RC6 ist geschützt.

Bemerkungen Der Algorithmus wurde von RSA-Labs entwickelt.

Rijndael

siehe AES

RIPEMD-128

Verfahren Hashfunktion

Parameter Blocklänge: 512 Bit

Hashwert: 128 Bit

Name `Ripemd128`

Beschreibung Ripemd-128 ist eine Hashfunktion der MD4-Familie. Die verwendete Kompressionsfunktion besteht aus insgesamt acht Runden.

Die zu hashende Nachricht wird in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion RIPEMD-128 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden folgende Operationen auf 32 Bit Wörtern verwendet:

Addition modulo 2^{32}, bitweise definierte Boolesche Operationen und zyklische Shifts. Diese Operationen können auf den heute üblichen 32-Bit Prozessoren sehr schnell ausgeführt werden.

Sicherheit Es gelten die Bemerkungen wie zu MD5; eine Hashwertlänge von 128 Bit ist für heutige Verhältnisse eindeutig zu kurz. Zu RIPEMD-128 sind bisher weder Kollisionen der Kompressionsfunktion noch Kollisionen der Hashfunktion selber bekannt.

Restriktionen keine

Bemerkung Wie bei MD5 gilt auch hier, daß der Einsatz in bestimmten Anwendungen, in denen keine starke Hashfunktion benötigt wird, denkbar ist. RIPEMD-128 ist von ISO/IEC standardisiert worden.

Der Algorithmus kann genutzt werden, falls 128 Bit Hashwerte benötigt werden und diese Bitlänge für die Sicherheit als ausreichend angesehen wird.

RIPEMD-160

Verfahren Hashfunktion

Parameter Blocklänge: 512 Bit

Hashwert: 160 Bit

Name `Ripemd160`

Beschreibung RIPEMD-160 ist eine Hashfunktion der MD4-Familie. Die Kompressionsfunktion besteht aus insgesamt zehn Runden.

Die zu hashende Nachricht wird in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion RIPEMD-160 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden folgende Operationen auf 32 Bit Wörtern verwendet:

Addition modulo 2^{32}, bitweise definierte Boolesche Operationen und zyklische Shifts. Diese Operationen können auf den heute üblichen 32-Bit Prozessoren sehr schnell ausgeführt werden.

Sicherheit Auch wenn bislang noch keine erfolgreichen Kollisisonsangriffe auf diesen Algorithmus bekannt sind schränkt die deutsche Signaturverordnung die Benutzung dieses Algorithmus für die Erzeugung von Zertifikaten und Nachrichtensignaturen auf bis Ende 2010 ein. Der Grund ist die nach heutigem Stand eher geringe Länge des Hashwerts sowie die ‚Nähe' zu Algorithmen, zu denen bereits Kollisionsangriffe existieren.

Restriktionen keine

Bemerkungen Der Algorithmus wurde von Dobbertin, Bosselaers und Preneel entwickelt.

RSA

Verfahren asymmetrisches Chiffrierverfahren

Parameter Schlüssellänge bis zu 4096 Bit

Name RSA

Beschreibung Zur Erzeugung eines Schlüsselpaares (öffentlicher und privater Schlüssel) wird das Produkt zweier großer Primzahlen p und q gebildet. Dann wählt man zufällig den Chiffrierschlüssel, der zu (p-1)(q-1) relativ prim sein muss. Der Dechiffrierschlüssel ist das Inverse dazu bezüglich des betrachteten Modulus. Bei der Verschlüsselung wird dann die Nachricht mit dem öffentlichem Schlüssel potenziert.

Konstruktion Der Algorithmus beruht auf dem Faktorisierungsproblem.

Komplexität Es werden eine Langzahl-Multiplikation, eine modulare Invertierung und eine modulare Exponentiation durchgeführt. Der Aufwand für die Erzeugung der Primzahlen hängt vom gewählten Verfahren ab.

Sicherheit Die Sicherheit von RSA hängt von der Sicherheit des IF-Problems. Entwicklungen auf diesem Gebiet müssen deshalb berücksichtigt werden. Die Schlüssellänge sollte mindestens 1024 Bit betragen.

Restriktionen RSA ist in einem informativen Anhang zu ISO 9796 enthalten. (Vorstufe zur Standardisierung).

Der Algorithmus ist nur in den USA patentiert, das Patent läuft am 20.09.00 aus.

Bemerkungen Der Algorithmus wurde nach seinen Erfindern Rivest, Shamir und Adleman benannt. Durch Vertauschen der Rollen von öffentlichem und privaten Schlüssel erhält man ein Verfahren zur digitalen Signatur.

Serpent

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 128-Bit Blockchiffre

Schlüssellänge: 128, 192 oder 256 Bit

Name Serpent

Beschreibung Der Algorithmus besteht aus einer Eingangspermutation, 32 Runden und einer Ausgangspermutation, die zur Eingangspermutation invers ist. Jede der 32 Runden besteht aus einer Schlüsselpermutation, S-Box Substitutionen und einer linearen Transformation. In der letzten Runde wird die lineare Transformation durch eine weitere Schlüsselpermutation ersetzt.

War ein AES-Kandidat

Konstruktion Serpent ist in Anlehnung an DES entwickelt worden.

Komplexität Es werden 161 XOR-Verknüpfungen, 16 Matrix-multiplikationen, 96 bitweise Rotationen, 32 Shift-Operationen durchgeführt.

Sicherheit Es sind bisher keine Angriffe auf Serpent bekannt.

An dieser Stelle ist allerdings darauf hinzuweisen, dass der Algorithmus noch zu neu ist, um eine abschließende Beurteilung abgeben zu können.

Andererseits erfreut sich der Algorithmus als AES-Kandidat besonderer Aufmerksamkeit. Die Tatsache, dass er sich für die letzte AES-Runde qualifiziert hat, deutet auf einen kryptographisch starken Entwurf hin.

Restriktionen keine

Bemerkungen Der Algorithmus wurde von Ross Anderson, Eli Biham und Lars Knudsen entwickelt.

SHA-0

Verfahren Message Digest/ Hashalgorithmus

Parameter Blocklänge: 512 Bit

Hashwert: 160 Bit

Name SHA0

Beschreibung SHA-0 ist eine Hashfunktion der MD4-Familie. Die Kompressionsfunktion besteht aus vier Runden mit je 20 Schritten.

Die zu hashende Nachricht wird in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion SHA-0 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden folgende Operationen auf 32 Bit Wörtern verwendet:

Addition modulo 2^{32}, bitweise definierte Boolesche Operationen und zyklische Shifts. Diese Operationen können auf den heute üblichen 32-Bit Prozessoren sehr schnell ausgeführt werden.

Sicherheit SHA-0 wurde 1994 ohne nähere Angabe von Gründen modifiziert.

Restriktionen keine

Bemerkung SHA-0 ist nur aus Kompatibilitätsgründen enthalten.

SHA-0 wurde von der NSA entwickelt, vom NIST 1992 veröffentlicht und ist in den USA seit `93 standardisiert (NIST). Der Algorithmus wurde 1994 ohne nähere Angabe von Gründen modifiziert, die so entstandene Hashfunktion heißt SHA-1.

SHA-1

Verfahren Message Digest/ Hashalgorithmus

Parameter Blocklänge: 512 Bit

Hashwert: 160 Bit

Name SHA1

Beschreibung SHA-1 ist eine Hashfunktion der MD4-Familie. Die Kompressionsfunktion besteht aus vier Runden mit je 20 Schritten.

Die zu hashende Nachricht wird in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

SHA-1 ist Teil des DSS.

Konstruktion SHA-1 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden folgende Operationen auf 32 Bit Wörtern verwendet:

Addition modulo 2^{32}, bitweise definierte Boolesche Operationen und zyklische Shifts. Diese Operationen können auf den heute üblichen 32-Bit Prozessoren sehr schnell ausgeführt werden.

Sicherheit Laut der deutschen Signaturverordnung (speziell Anlage 1 Abschnitt 1 Nr. 2 SigV) darf SHA-1 nur noch bis Ende 2009 (in bestimmten Fällen bis 2010) zur Erzeugung von Zertifikatssignaturen verwendet werden.
Grund sind hier bereits bekannte Kollisionsangriffe auf diesen Algorithmus bei größeren Datenmengen, weswegen SHA-1 auch nicht mehr zur Nachrichtensignatur verwendet werden darf.

Restriktionen keine

Bemerkungen SHA-1 wurde von der NSA entwickelt, vom NIST 1992 veröffentlicht und ist in den USA seit 1993 standardisiert.

SHA-256

Verfahren Hashfunktion

Parameter Blocklänge: 512 Bit

Hashwert: 256 Bit

Name SHA256

Beschreibung SHA-256 ist eine Hashfunktion der MD4-Familie.

Die zu hashende Nachricht wird in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion SHA-256 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Es werden Operationen auf 32-Bit-Wörtern verwendet. Diese Operationen können auf den heute üblichen 32-Bit-Prozessoren sehr schnell ausgeführt werden.

Sicherheit Erfolgreiche Angriffe auf SHA-256 sind derzeit nicht bekannt.

Restriktionen keine

Bemerkungen SHA-256 wurde im Oktober 2000 vom NIST veröffentlicht und soll im Jahr 2001 in den USA standardisiert werden.

SHA-384

Verfahren Hashfunktion

Parameter Blocklänge: 1024 Bit

Hashwert: 384 Bit

Name SHA384

Beschreibung SHA-384 ist eine Hashfunktion der MD4-Familie. Der Algorithmus basiert auf SHA-512, benutzt aber andere Initialisierungswerte und beschneidet den Output auf 384 Bit.

Die zu hashende Nachricht wird also in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion SHA-384 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Siehe SHA-512.

Sicherheit Erfolgreiche Angriffe auf SHA-384 sind derzeit nicht bekannt.

Restriktionen keine

Bemerkungen SHA-384 wurde im Oktober 2000 vom NIST veröffentlicht und soll im Jahr 2001 in den USA standardisiert werden.

SHA-512

Verfahren Hashfunktion

Parameter Blocklänge: 1024 Bit

Hashwert: 512 Bit

Name SHA512

Beschreibung SHA-512 ist eine Hashfunktion der MD4-Familie.

Die zu hashende Nachricht wird in gleich große Blöcke fester Länge unterteilt. Diese Blöcke werden nacheinander als Input für die Kompressionsfunktion benutzt. Die jeweilige Ausgabe davon ist dann der Initialisierungswert für die nächste Anwendung. Der Hashwert ist die letzte Ausgabe der Kompressionsfunktion.

Konstruktion SHA-512 ist definiert als iterative Anwendung einer Kompressionsfunktion, deren zugrundeliegender Algorithmus mit einem bestimmten festen Initialisierungswert gestartet wird.

Komplexität Im Gegensatz zu den anderen Mitgliedern der MD4-Familie arbeitet SHA-512 und damit auch SHA-384 mit 64-Bit-Wörtern. Dies kann gegebenenfalls die Performance auf 32-Bit-Prozessoren beeinflussen.

Sicherheit Erfolgreiche Angriffe auf SHA-512 sind derzeit nicht bekannt.

Restriktionen keine

Bemerkungen SHA-512 wurde im Oktober 2000 vom NIST veröffentlicht und soll im Jahr 2001 in den USA standardisiert werden.

Triple-DES

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge: 64-Bit Blockchiffre

Schlüssellänge: 112 Bit, d.h. zwei Schlüssel K und L je 64 Bit

Name TripleDES

Beschreibung Als Triple-DES bezeichnet man die Klasse von Chiffrierverfahren durch dreimalige Anwendung des DES-Verfahrens. Der Name findet für verschiedene Vorgehensweisen Verwendung: beim bekanntesten Vertreter werden zwei Schlüssel verwendet (insgesamt 112 Bit). Ein Nachrichtenblock wird zuerst mit dem ersten Schlüssel chiffriert, danach mit dem zweiten Schlüssel dechiffriert und abschließend noch einmal mit dem ersten Schlüssel chiffriert. Eine andere Variante setzt dazu drei verschiedene Schlüssel ein.

Konstruktion Hintereinanderschaltung von DES-Schlüsseln

Komplexität Für jeden Ausgabeblock des Algorithmus ist ein 64-Bit String zu übertragen.

Sicherheit Durch den erweiterten Schlüsselraum sind Angriffe gegen Triple-DES im Gegensatz zu Angriffen gegen DES deutlich erschwert.

Restriktionen IBM besitzt das Patent zu DES, eine kostenlose Lizenz ist Bestandteil des DSS Standards. Dieses Patent betrifft auch Triple-DES.

Bemerkungen Außer Triple-DES gibt es noch einige andere Weiterentwicklungen von DES mit dem Ziel, die Sicherheit zu verbessern.

Twofish

Verfahren symmetrisches Chiffrierverfahren

Parameter Blocklänge/-art: 128-Bit Blockchiffre

Schlüssellänge: 128, 192 oder 256 Bit

Name TwoFish

Beschreibung Der Twofish -Verschlüsselungsalgorithmus ist ein Feistel-Netzwerk mit 16 Runden. Eine bijektive Funktion F, die aus 4 schlüsselabhängigen 8x8 S-Boxen, einer festen 4x4 separablen Matrix, einer Pseudo-Hadamard Transformation, bitweisen Permutationen und einer Schlüsseltransformation besteht, bildet den Verschlüsselungsalgorithmus.

War ein AES-Kandidat.

Konstruktion Feistel-Netzwerk

Komplexität Es werden 40 XOR-Verknüpfungen, 48 Shift-Operationen, 64 Additionen, 128 Matrix-/Vektormultiplikationen und 16 Permutationen durchgeführt.

Sicherheit Es sind bisher keine Angriffe auf Twofish bekannt.

An dieser Stelle ist allerdings darauf hinzuweisen, dass der Algorithmus noch zu neu ist, um eine abschließende Beurteilung abgeben zu können.

Andererseits erfreut sich der Algorithmus als AES-Kandidat besonderer Aufmerksamkeit. Die Tatsache, dass er sich für die letzte AES-Runde qualifiziert hat, deutet auf einen kryptographisch starken Entwurf hin.

Restriktionen keine

Bemerkungen Der Algorithmus wurde von B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall und N. Ferguson entwickelt.

Kapitel 5 So benutzen Sie die cv act *library*

Die **cv act *library*** ist eine C++ Bibliothek und umfasst alle Objekte, die für die Programmierung kryptographischer Anwendungen notwendig sind. Sie stellt eine einheitliche Schnittstelle zur Verfügung, deren Quellcode unabhängig vom eingesetzten Algorithmus und vom angewandten Verfahren ist. Zusätzlich liefert die **cv act *library*** eine Unterstützung für die Verwendung von eigenen kryptographischen Primitiva und ist abgestimmt auf den Einsatz entsprechender Smartcards.

Eine technische Eigenschaft der **cv act *library*** ist, dass es sich um eine statisch zu bindende Bibliothek handelt, d.h. dass Sie im Lieferumfang die Headerdateien und die `.lib`-Dateien, aber keine DLLs finden. Deshalb lassen sich in Anwendungen, die mit dieser Bibliothek erstellt wurden, die kryptographischen Funktionalitäten nicht austauschen, indem man die DLL ersetzt. Ein offensichtliches Angriffsszenario, das auf dem Ersetzen von DLLs beruht, ist das Verwenden von Wrapper-DLLs, die beispielsweise geheime Schlüssel abfangen.

Die Bedienung der **cv act *library*** kann man in folgende Punkte unterteilen:

- Für die Programmierung mit der Library auf höherem Level können Sie Handle-Klassen verwenden. Dieses minimiert die ansonsten notwendige Benutzung von Zeigern und dynamischem Speicher auf Applikationsebene.

- Der Zugriff auf die Funktionalitäten der Library erfolgt über Interface-Klassen, die das abstrakte Verwenden der kryptographischen Verfahren gestatten.
- Sie können die Library anpassen oder erweitern, indem Sie eigene Klassen erstellen, die von solchen Interfaces abgeleitet werden. Diese Klassen können dann von den, in der Library enthaltenen Verfahren benutzt werden.

Im Abschnitt 5. 1 werden einige Hinweise zur Installation angegeben. Die Handle-Klassen werden dann im Unterkapitel 5. 2 an Beispielen erklärt. Desweiteren wird die Fehlerbehandlung im Abschnitt 5. 3 beschrieben und in Unterkapitel 5. 4 finden Sie alle in der Library enthaltenen Objekte und deren Parameterlisten. Die Erläuterungen der Interface-Klassen und wie man die **cv act *library*** erweitert oder anpasst, finden Sie in Kapitel 6.

5. 1 Installation

Zur Installation legen Sie die CD **cv act *library*** in das CD-ROM-Laufwerk ein und starten Sie setup.exe im Hauptverzeichnis der CD-Rom. Die weitere Vorgehensweise entnehmen Sie den Anweisungen auf dem Bildschirm. Lesen Sie auch die Datei `readme.txt` auf der CD, um weitere Informationen zur Installation und Compiler-spezifische Benutzungshinweise zu erhalten.

5. 2 Programmierung mit Hilfe der Handle-Klassen

In diesem Abschnitt wird anhand von Beispielen ein Überblick gegeben, wie Sie die **cv act *library*** für die Programmierung von Anwendungen benutzen können. Die lauffähigen Beispiele können Sie der CD entnehmen.

Dieses Kapitel ist so aufgebaut, dass die Erklärungen zu dem ersten Verfahren, der symmetrischen Verschlüsselung, am ausführlichsten sind und die dort erklärten Begriffe in den nachfolgenden Beispielen nicht mehr wiederholt werden.

Vorbemerkung

Alle Klassen und Funktionen, die in der **cv act *library*** enthalten sind, befinden sich im Namespace `act`. Bevor Sie Objekte erzeugen können, muss die Funktion `act::Init()` aufgerufen werden. Dadurch werden die Registries, Zufallszahlengeneratoren und andere Konfigurationen initialisiert.

Sie sollten beachten, dass Algorithmen- oder Schlüsselobjekte nur solange existieren sollten, wie sie benötigt werden, damit geheime Informationen nicht unnötig lange unverschlüsselt in Ihrem Speicher vorhanden sind. Sie sollten auch vermeiden, Schlüsselobjekte wiederzuverwenden, indem Sie beispielsweise nach dem Nichtmehrbenötigen mit der Methode `Generate()` einen neuen Schlüssel erzeugen.

5. 2. 1 Symmetrische Verschlüsselung

Bei der symmetrischer Verschlüsselung werden Daten mit einem geheimen Schlüssel ver- und entschlüsselt. Im folgenden Beispiel werden Daten aus einem Stream ausgelesen, mit dem DES-Algorithmus verschlüsselt und in einen anderen Stream ausgegeben (Streams sind Bestandteil der *Standard Template Library (STL)* des C++ Standards). Die Entschlüsselung und die Benutzung anderer symmetrischer Kryptoverfahren verläuft analog.

```
void encrypt_from_stream_to_stream(std::istream &in, std::ostream &out,
const act::Blob &passphrase)
{
        act::Key key("BlockCipher");           // Schluessel für
                                        //Blockchiffre anlegen
        key.SetParam(act::CIPHER,"DES");       // Algorithmus auswaehlen
        key.SetParam(act::BCMODE,"CBC");       // Modus auswaehlen

        key.Derive(passphrase);                // Schluessel aus Text
ableiten

        act::Algorithm Encrypt(key,act::ENCRYPT);// Verschluesselungs-
                                               //algorithmus anlegen

        act::Blob PlainData (BUFFER_SIZE);     // Blob fuer Eingabe anlegen
        act::Blob CipherData;                  // Blob fuer Ausgabe anlegen

        std::streamsize count;

        //es werden blockweise Daten aus dem Stream in eingelesen,
        //verschluesselt und in den Stream out ausgegeben
        do {

        in.read(reinterpret_cast<char*>(&PlainData[0]),BUFFER_SIZE);//Einle
sen
                count=in.gcount();

                if( count<BUFFER_SIZE ) { //Sonderbehandlung des letzten
                                // Blocks, der nicht mehr volle Laenge hat

                        PlainData.resize(count);
                }
                Encrypt << PlainData;  die unverschluesselten Daten
                                // werden an den Algorithmus uebergeben
                Encrypt >> CipherData;         die verschluesselten Daten
                                //werden an den Blob uebergeben
                out.write(reinterpret_cast<char*>(&CipherData[0]),
CipherData.size());
                // Ausgabe
                                //vom Blob in den Stream

   } while(count == BUFFER_SIZE);

        Encrypt << act::final; // es kommen keine weiteren Daten, die
                                //verschluesselt werden sollen
        Encrypt >> CipherData;         // restliche Daten abholen
        out.write(reinterpret_cast<char*>(&CipherData[0]),
CipherData.size());//wie oben
 }
```

Die Funktion `encrypt_from_stream_to_stream` bekommt als Parameter Referenzen auf den Eingabestream, den Ausgabestream und einen Blob, der die Passphrase enthält, übergeben. In der **cv act *library*** stellt der `Blob` den universellen Datentyp dar, um beliebige Daten zu repräsentieren. Er entspricht dem `std::vector<unsigned char>` aus der C++ Standardbibliothek.

Zunächst wird ein Schlüsselobjekt angelegt. In dieser Library werden Schlüssel durch die Klasse `Key` repräsentiert. Dabei wird im Konstruktor der Typ, in diesem Falle "`BlockCipher`", übergeben. Dann werden die benötigten Parameter für ein Primitiva und einen Modus durch die Methode `SetParam` gesetzt.

Die in der **cv act *library*** enthaltenen symmetrischen Algorithmen sind

`BlowFish, CAST128, DES, Mars, AES (Rijndael),`

`Serpent, TripleDES` und `TwoFish`

und die enthaltenen Betriebsmodi sind `ECB, CBC` und `CFB`. Aufgrund des IDEA-Patents wurde auf dessen Auslieferung verzichtet. Falls Sie diesen Algorithmus verwenden möchten, können Sie die Library um diesen Algorithmus erweitern, wie Sie in Kapitel 6 nachlesen können.

Der Schlüssel wird mit der Methode `Derive` erzeugt. Dabei ist zu bemerken, dass zu derselben Passphrase immer derselbe Schlüssel erzeugt wird. Danach wird ein Algorithmenobjekt angelegt. Algorithmen werden durch die Klasse `Algorithm` repräsentiert. Dem Konstruktor wird das zu verwendende Schlüsselobjekt übergeben und durch `ENCRYPT` mitgeteilt, dass verschlüsselt werden soll. Mit dem << - Operator werden Daten-Blobs an den Algorithmus zur Verschlüsselung übergeben und mit dem >> - Operator die verschlüsselten Daten an einen Blob übergeben. Mit dem Modifizier `final` wird dem Algorithmus mitgeteilt, dass keine weiteren Eingabedaten mehr folgen.

5. 2. 2 Hashfunktionen

Eine Hashfunktion bekommt einen Eingabestring beliebiger Länge und wandelt diesen in einen - in der Regel kürzeren Ausgabestring fester Länge um.

Im folgenden Beispiel werden Daten aus einem Stream ausgelesen, gehasht und in einen anderen Stream ausgegeben.

```
act::Blob hash_stream (std::istream &in)
{
        act::Algorithm hash("SHA1");          //Hashfunktion auswählen
        act::Blob InData(BUFFER_SIZE); //Blob fuer Eingabe anlegen
        std::streamsize count;

        //es werden blockweise Daten aus dem Stream in eingelesen,
         //gehasht und in den Stream out ausgegeben
        do
        {
                in.read(reinterpret_cast<char*>(&InData[0]),BUFFER_SIZE);    //Einlesen
                count=in.gcount();

                if(count<BUFFER_SIZE) { //Sonderbehandlung des letzten
                                //Blocks, der nicht mehr volle Laenge hat
                        InData.resize(count);
                }

                hash << InData;        // die Daten werden an den
                                    // Algorithmus uebergeben
    } while(count == BUFFER_SIZE);

        hash << act::final;           //es kommen keine weiteren Daten, die
                                           //gehasht werden sollen
        act::Blob OutData;            //Blob fuer Ausgabe anlegen
        hash >> OutData;                      //berechneten Hashwert
auslesen
        return OutData;
}
```

Die Funktion `hash_stream` bekommt als Parameter Referenzen auf einen Eingabestream. Es wird ein Hashalgorithmus ausgewählt und die in der **cv act *library*** enthaltenen sind

`MD2, MD4, MD5, RIPEMD128, RIPEMD160, SHA0, SHA1, SHA224, SHA256, SHA384, SHA512.`

Mit dem << - Operator werden Daten-Blobs an den Algorithmus zum Hashen übergeben und mit dem >> - Operator die gehashten Daten an einen Blob übergeben. Mit dem Modifier `final` wird dem Algorithmus mitgeteilt, dass keine weiteren Eingabedaten mehr folgen. Zurückgegeben wird der Hashwert in einem Blob.

5. 2. 3 MACs

Ein MAC besteht aus einer Hashfunktion oder einer Blockchiffre zusammen mit einem Schlüssel. Im folgenden Beispiel eines HashMACs werden Daten aus einem Stream ausgelesen, durch den MAC bearbeitet und in einen anderen Stream ausgegeben.

```
act::Blob generate_MAC_from_stream (std::istream &in, const act::Blob
&passphrase)
{
        act::Key key("HashMAC");          //Schluessel für HashMAC anlegen
        key.SetParam(act::HASH,"SHA1");  //Hashalgorithmus auswaehlen
        key.Derive(passphrase);               //Schluessel aus Text
ableiten
        act::Algorithm mac(key,act::MAC);     //MAC-Algorithmus anlegen
        act::Blob InData(BUFFER_SIZE);        //Blob fuer Eingabe anlegen
        std::streamsize count;
        //es werden blockweise Daten aus dem Stream in eingelesen,
        //durch den MAC bearbeitet und in den Stream out ausgegeben
        do      {
                in.read(reinterpret_cast<char*>(&InData[0]),BUFFER_SIZE);
//Einlesen
                count=in.gcount();
                if(count<BUFFER_SIZE) {  //Sonderbehandlung des letzten
                                //Blocks, der nicht mehr volle Laenge hat
                        InData.resize(count);
                }
                mac << InData;

        } while(count == BUFFER_SIZE);

        mac << act::final;            //es kommen keine weiteren Daten, die
                                      //verschluesselt werden sollen
        act::Blob OutData;
        mac >> OutData;               //bearbeitete Daten auslesen
        return OutData;
}
```

An die Funktion `generate_MAC_from_stream` werden als Parameter Referenzen auf den Eingabe-, den Ausgabestream und einen Blob, der die Passphrase enthält, übergeben. Zunächst wird ein Schlüsselobjekt für einen HashMAC oder für einen CBCMAC angelegt. Im Falle des HashMACs wird mit der Methode `SetParam` ein Hashalgorithmus gesetzt und im Falle des CBCMAC eine Blockchiffre.

Der Schlüssel wird wieder mit der Methode `Derive` erzeugt und danach ein MAC-Algorithmus angelegt. Mit dem `<<` - Operator werden Daten-Blobs an den Algorithmus übergeben und mit dem `>>` - Operator die verarbeiteten Daten an einen Blob übergeben. Mit dem Modifier `final` wird dem Algorithmus mitgeteilt, dass keine weiteren Eingabedaten mehr folgen. Zurückgegeben wird ein MAC in einem Blob.

5. 2. 4 Verfahren zum Schlüsselaustausch

Beim Schlüsselaustausch zwischen zwei Kommunikationsteilnehmern wird ein gemeinsamer Sitzungsschlüssel erzeugt. Dabei muss jeder einzelne Partner einen privaten und einen öffentlichen Schlüssel erzeugen. Danach kann ein gemeinsames Geheimnis generiert werden. Es handelt sich also um zwei Funktionen, eine für die Erzeugung des eigenen Schlüssels und eine für die Erzeugung des Sitzungsschlüssels.

```
void generate_ECDH_key_pair (act::Blob &MyPrivateKeyBlob, act::Blob
&MyPublicKeyBlob)
{
        act::Key key("ECDH");          //Schluesselobjekt fuer EC-DH anlegen
        int m = 163;                   //Parameter für flexible Kurve setzen
        int k3 = 8;
        int k2 = 2;
        int k1 = 1;
        act::Blob param = act::EncodePentanomialCurveParam(
                m, k3, k2, k1,

        act::EncodeNumber("0x72546B5435234A422E0789675F432C89435DE5242"),

        act::EncodeNumber("0x0C9517D06D5240D3CFF38C74B20B6CD4D6F9DD4D9"),

        act::EncodeNumber("0x7AF69989546103D79329FCC3D74880F33BBE803CB"),

        act::EncodeNumber("0x1EC23211B5966ADEA1D3F87F7EA5848AEF0B7CA9F"),

        act::EncodeNumber("0x400000000000000000001E60FC8821CC74DAEAFC1"),
        act::EncodeNumber("0x2")
                );

        key.SetParam(act::CURVEPARAM,param);   //Eine flexible Kurve setzen
        key.Generate();                          //Schluesselpaar wird
erzeugt
        key.Export(MyPrivateKeyBlob,act::PRIVATE);    //Schluessel werden
        key.Export(MyPublicKeyBlob,act::PUBLIC);   // in Blobs exportiert
}

void generate_secret (const act::Blob &MyPrivateKeyBlob,const act::Blob
&OtherPublicKeyBlob, act::Blob &SecretBlob)
{
        act::Key key(MyPrivateKeyBlob);       //Schluesselobjekt wird an-
      //gelegt, indem der Schluessel aus einem Blob importiert wird
        key.Derive(OtherPublicKeyBlob);       //Sitzungsschluessel wird
                                                    //erzeugt und
        key.Export(SecretBlob,act::SECRET);  //in einen Blob exportiert

}
```

Die Funktion `generate_ECDH_key_pair` erhält als Parameter Referenzen auf einen Blob, der den eigenen, privaten Schlüssel und auf einen Blob, der den eigenen, öffentlichen Schlüssel repräsentiert. Zunächst wird ein Schlüsselobjekt angelegt, durch das man das Verfahren zum Schlüsselaustausch festlegt und die in der **cv act *library*** enthaltenen Verfahren sind

`DH` und `ECDH`.

Danach wird in diesem Beispiel eine nicht in der **cv act *library*** enthaltene elliptische Kurve angelegt und ein Schlüsselpaar wird mit der Methode `Generate` erzeugt. Der eigene, private Schlüssel und der eigene, öffentliche Schlüssel werden dann jeweils in einen Blob importiert. Die zweite Funktion `generate_secret` erhält als Parameter Referenzen auf drei verschiedene Blobs: auf einen, der den eigenen, privaten Schlüssel repräsentiert, auf einen zweiten Blob, der den anderen, öffentlichen Schlüssel und auf einen dritten Blob, der das gemeinsame Geheimnis repräsentiert. Mit diesem gemeinsamen Geheimnis lassen sich beispielsweise symmetrische Sitzungsschlüssel ableiten.

Zunächst wird ein Schlüsselobjekt angelegt, indem man den privaten Schlüssel aus einem Blob importiert. Danach wird ein Sitzungsschlüssel mit der Methode `Derive` erzeugt und dann mit der Methode `Export` in einen Blob exportiert.

Die beiden Schlüsselpaare in einem Schlüsselaustauschverfahren benötigen die gleichen *domain parameter*. In diesen Beispielen (mit Ausnahme des letzten) ist die elliptische Kurve defaultmäßig `SECGp160r1`. In dem letzten Beispiel wird die Verwendung von elliptischen Kurven, die nicht in der **cv act *library*** enthalten sind, demonstriert.

Bei dem Verfahren DH müssen die *domain parameter* explizit in Übereinstimmung gebracht werden. Dazu kann man nach dem Erzeugen des ersten Schlüsselpaares mit `Export(Blob, DOMAINPARAMS)` die *domain parameter* in einen Blob exportieren und vor dem Erzeugen des zweiten Schlüsselpaares importieren, um die Übereinstimmung der *domain parameter* zu gewährleisten.

5. 2. 5 Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung wird eine Nachricht mit dem öffentlichen Schlüssel des Kommunikationspartners verschlüsselt, der die chiffrierte Nachricht mit seinem privaten Schlüssel wieder entschlüsselt. Im Gegensatz zur symmetrischen Verschlüsselung ist bei asymmetrischen Verschlüsselungsverfahren die maximal zu verschlüsselnde Datenmenge beschränkt: bei RSA auf 1024 Bit, beim RSAES-Padding auf Modulusmenge-80 Bit und bei IES auf 160 Bit.

In diesem Beispiel werden in einer ersten Funktion beide Schlüssel erzeugt und exportiert, dann in der nächsten Funktion mit einem öffentlichem Schlüssel in einen Datenblob verschlüsselt und in der dritten Funktion wird der Blob entschlüsselt.

```
void generate_key_pair(act::Blob &PrivateKeyBlob, act::Blob &PublicKeyBlob)
{
        act::Key key("RSA");   //Schluesselobjekt wird angelegt
        key.Generate();               //Schluessel wird erzeugt
        key.Export(PrivateKeyBlob,act::PRIVATE); //privater
                                                 //Schluessel wird exportiert
        key.Export(PublicKeyBlob,act::PUBLIC);        //oeffentlicher
                                                 //Schluessel wird exportiert

}


void asym_encrypt(const act::Blob &PublicKeyBlob, const act::Blob
&PlainData, act::Blob &CipherData)
{
        act::Key key(PublicKeyBlob); //oeffentliches Schluesselobjekt
                        //wird angelegt, indem der Schluessel aus einem
                        //Blob importiert wird
        act::Algorithm Encrypt(key,act::ENCRYPT);//Verschluesselungs-
                                                 //algorithmus anlegen

        Encrypt << PlainData  << act::final; //die unverschluesselten
                                        // Daten werden an den Algorithmus
uebergeben
        Encrypt >> CipherData;                //die verschluesselten Daten
                                        //werden an den Blob uebergeben

}


void asym_decrypt(const act::Blob &PrivateKeyBlob,
                const act::Blob &CipherData, act::Blob &PlainData)
{
        act::Key key(PrivateKeyBlob); //privates Schluesselobjekt
                                //wird angelegt, indem der Schluessel aus
einem
                                //Blob importiert wird
        act::Algorithm Decrypt(key,act::DECRYPT);//Entschluesselungs-
                                                 //algorithmus anlegen

        Decrypt << CipherData << act::final; //die verschluesselten
                        // Daten werden an den Algorithmus uebergeben

        Decrypt >> PlainData; //die unverschluesselten Daten
                        //werden an den Blob uebergeben

}
```

Die Funktion `generate_key_pair` hat dieselbe Form wie die entsprechende Funktion beim Schlüsselaustausch. Die in der **cv act *library*** enthaltenen asymmetrischen Algorithmen sind

`IES` (also DLIES und ECIES) und `RSA`.

Die nächste Funktion `asym_encrypt` erhält als Parameter Referenzen auf drei verschiedene Blobs: auf einen Blob, der den öffentlichen Schlüssel repräsentiert, auf einen Blob, der den Klartext und auf einen, der den Chiffretext repräsentiert. Dann wird ein öffentliches Schlüsselobjekt angelegt, indem man den Schlüssel aus einem Blob importiert. Anschließend wird ein Verschlüsselungsalgorithmus erzeugt und verschlüsselt. Die dritte Funktion `asym_decrypt` hat dieselbe Form wie die Verschlüsselungsfunktion `asym_encrypt`, nur mit einem privaten Schlüssel und einem Entschlüsselungsalgorithmus.

Beim IES-Verfahren wird sowohl zum Ent- als auch Verschlüsseln der öffentliche Schlüssel der Gegenpartei benötigt. Dazu wird bei Anlegen des Algorithmus der Public Key-Blob als dritter Parameter übergeben:

```
act::Algorithm Encrypt(key,act::ENCRYPT,OtherPubKeyBlob);
```

Zusätzlich müssen die *domain parameter* übereinstimmen (siehe Kapitel 5.2.4).

5. 2. 6 Digitale Signaturen

Eine digitale Signatur ist das Äquivalent zu einer Unterschrift. In der Praxis wird beispielsweise der Hashwert einer Datei mit einem asymmetrischen Verfahren verschlüsselt, wobei die Rollen des öffentlichen und privaten Schlüssels vertauscht werden.

In diesem Beispiel wird also ein asymmetrischer Schlüssel erzeugt und in einer zweiten Funktion werden Daten aus einem Stream ausgelesen, signiert und in einen anderen Stream ausgegeben. In der dritten Funktion werden Daten aus einem Stream ausgelesen und die Signatur überprüft.

```
void generate_key_pair(act::Blob &PrivateKeyBlob, act::Blob &PublicKeyBlob)
{
        act::Key key("ECDSA"); //Schluesselobjekt wird angelegt
        key.Generate(); //Schluessel wird erzeugt
        key.Export(PrivateKeyBlob,act::PRIVATE); //privater
                                                 //Schluessel wird exportiert
        key.Export(PublicKeyBlob,act::PUBLIC); //oeffentlicher
                                                 //Schluessel wird exportiert
}

void sign_from_stream(const act::Blob &PrivateKeyBlob,
                  std::istream &in, act::Blob &Signature);
{
        act::Key key(PrivateKeyBlob); //privates Schluesselobjekt
                        //wird angelegt, indem der Schluessel aus einem
                        //Blob importiert wird
        act::Algorithm Sign(key,act::SIGN); //Signier-
                                        //algorithmus anlegen
        act::Blob InData(BUFFER_SIZE); //Blob fuer Eingabe anlegen
        std::streamsize count;
        do
        {
                in.read(reinterpret_cast<char*>(&PlainData[0]),BUFFER_SIZE);
//Einlesen
                count=in.gcount();
                if(count < BUFFER_SIZE) //Sonderbehandlung des letzten
                                //Blocks, der nicht mehr volle Laenge hat
                {
                        InData.resize(count);
                }
                Sign << InData; // die zu signierenden Daten
                       // werden an den Algorithmus uebergeben
        } while(count == BUFFER_SIZE);

        Sign << act::final; //es kommen keine weiteren Daten, die
                        //signiert werden sollen
        Sign >> Signature; //die signierten Daten werden an einen
                        // Blob uebergeben
}


bool verify_from_stream (const act::Blob &PublicKeyBlob,
                        std::istream &in,const act::Blob &Signature);
{
        act::Key key(PublicKeyBlob); //Schluesselobjekt wird angelegt
        act::Algorithm Verify(key,act::VERIFY,Signature);
                //Verifizierungsalgorithmus wird angelegt
        act::Blob InData(BUFFER_SIZE); //Blob fuer Eingabe anlegen
        std::streamsize count;
        do
        {
```

```
                in.read(reinterpret_cast<char*>(&PlainData[0]),BUFFER_SIZE);
//Einlesen
                count=in.gcount();
                if(count < BUFFER_SIZE) //Sonderbehandlung des letzten
                                //Blocks, der nicht mehr volle Laenge hat
                {
                        InData.resize(count);
                }
                Verify << InData; // die zu verifizierenden Daten
                                // werden an den Algorithmus uebergeben
        } while(count == BUFFER_SIZE);

        Verify << act::final; //es kommen keine weiteren Daten, die
                        //verifiziert werden sollen
        return Verify.GetStatus()==act::SIGNATURE_OK;
}
```

Die Funktion `generate_key_pair` ist dieselbe Funktion wie beim Schlüsselaustausch. Die in der **cv act *library*** enthaltenen Signaturalgorithmen sind

`DSA, ECDSA` und `RSA (mit Appendix).`

Die zweite Funktion `sign_from_stream` erhält als Parameter folgende drei Referenzen: eine auf einen Blob, der den privaten Schlüssel repräsentiert, eine auf einen Stream, von dem die zu signierenden Daten gelesen werden und eine auf einen Blob, in dem die Signatur zurückgegeben wird.

Die dritte Funktion `verify_from_stream` erhält als Parameter folgende drei Referenzen: eine auf einen Blob, der den öffentlichen Schlüssel repräsentiert, eine auf einen Stream, von dem die zu überprüfenden Daten gelesen werden und eine auf einen Blob, in dem die Signatur übergeben wird. Es wird `false` zurückgegeben, wenn die Verifizierung fehlgeschlagen ist und `true`, falls die Signatur gültig ist.

5. 2. 7 Zertifikate

Die Funktionsweise eines Zertifikats kann man mit der eines amtlichen Ausweises vergleichen. Es enthält den Namen des Besitzers, seinen öffentlichen Schlüssel und andere Informationen, wie vor allem die Unterschrift einer Zertifizierungsstelle (CA). Im folgenden Beispiel wird ein Zertifkat überprüft und bei Gültigkeit wird der öffentliche Schlüssel aus dem Zertifikat extrahiert und an einen Blob übergeben.

```
void verify_cert_and_get_public_key(const act::Blob &CAPublicKeyBlob, const
act::Blob &CertBlob, act::Blob &PublicKeyBlob)
{
        act::Key cakey(CAPublicKeyBlob);
        act::Certificate cert("X509",CertBlob);       //Zertifikatsobjekt
wird angelegt,
                                                  // indem das Zertifikat aus
einem Blob importiert wird
        cert.Verify(cakey); //falls das Zertifikat ungueltig ist, wird eine
Exception geworfen
        act::Key key(cert); //oeffentlicher Schluessel wird aus Zertifikat
extrahiert...
        key.Export(PublicKeyBlob,act::PUBLIC);//und in einen Blob
geschrieben

}
```

Die Funktion `verify_Cert_and_get_public_key` erhält als Parameter Referenzen auf einen Blob, der das Zertifikat repräsentiert und auf einen Blob, in den als Rückgabewert der öffentliche Schlüssel des Zertifikats geschrieben wird. Zunächst wird ein Zertifikatsobjekt angelegt, indem man das Zertifikat aus einem Blob importiert. Die Überprüfung wird mit der Methode `Verify` durchgeführt. Falls das Zertifikat gültig ist, wird ein Schlüsselobjekt mit dem öffentlichen Schlüssel erzeugt und mit der Methode `Export` der Schlüssel an einen Blob übergeben.

5.3 Secure Token

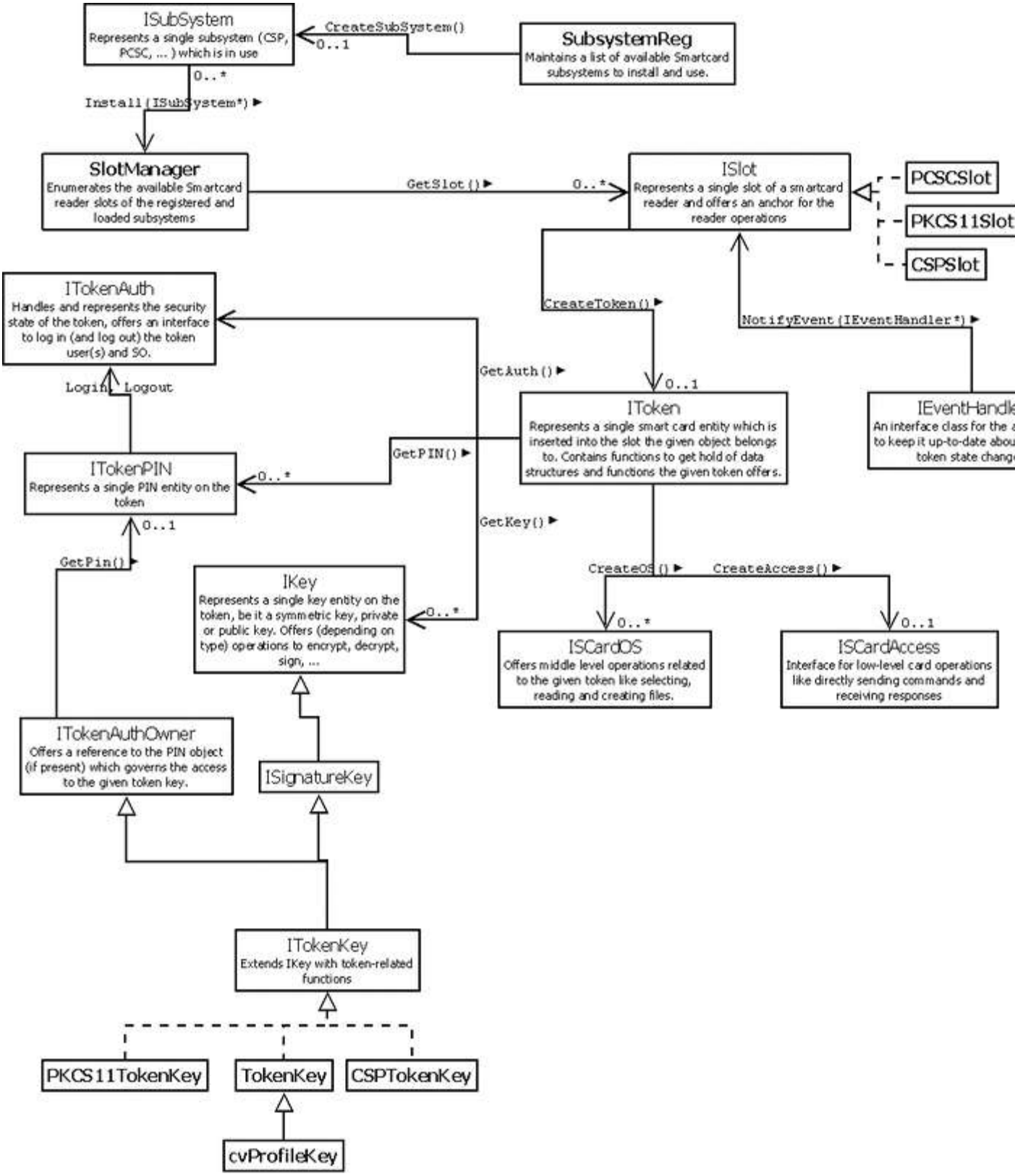
In der Kryptographie müssen geheimzuhaltende Informationen sicher gespeichert werden. Neben dem hochsicheren Speicherort Smartcard gibt es weitere, teilweise gleichwertige Möglichkeiten, private Schlüssel und Zertifikate zu speichern. Diese verschiedenen Methoden bzw. Orte zur Speicherung werden mit dem Oberbegriff „Token“ bzw. „Secure Token“ bezeichnet. Beispiele für Secure Token sind Smartcards, USB-Tokens (Beispiel: Rainbow ikey), CSP-Token und PKCS#11-Token.

Die **cv act *library*** stellt ein allgemeines Interface zur Verfügung, mit dem verschiedene Secure Token über eine einheitliche Schnittstelle verwendet werden können. Diese einheitliche Schnittstelle unterstützt die Verwendung aller Secure Token, die Module für diese Schnittstelle zur Verfügung stellen. Neben den Funktionen, die für die kryptographischen Operationen im engeren Sinne notwendig sind, stehen auch unterstützende Funktionen zur Verfügung, wie z. B. für die automatische Erkennung von Smartcards und Smartcard-Readern oder eine Callback-Funktion, mit der festgestellt werden kann, ob eine Smartcard eingelegt oder entfernt wurde.

Es gibt Smartcard-Reader, die auf die Verwendung bestimmter Smartcards ausgelegt sind, andere Smartcard-Reader können mit beliebigen Smartcards verwendet werden. Eine aktuelle Liste der Smartcard-Reader, die mit den in der **cv act *library*** integrierten Smartcards (siehe 5.3.2) erfolgreich getestet wurden, findet sich in der Datei Liesmich.txt auf der Produkt-CD.

Die Einbindung von Secure Token in die **cv act *library***, die in diesem Kapitel beschrieben wird, wurde gegenüber früheren Versionen der **cv act *library*** weiterentwickelt. Aus Gründen der Kompatibilität ist es aber weiterhin möglich, Smartcards auf dieselbe Art wie bisher einzubinden. Die entsprechende Anleitung befindet sich in Anhang E dieses Handbuches.

Ein Überblick über die wichtigsten Handle- und Interfaceklassen, die mit dem Secure Token Interface in Beziehung stehen, gibt folgendes Diagramm:



Die hier dargestellten Klassen und Interfaces werden im Einzelnen in den folgenden Kapiteln und ab Kapitel 7.3 erklärt.

5. 3. 1 Slot Manager

Der SlotManager wird benutzt, um die verfügbaren Slots zu erkennen und die Erstellung von Slot-Instanzen zu ermöglichen.

In der Klasse `SlotManager` sind die folgenden Funktionen verfügbar:

- `SlotManager(); / ~SlotManager();`
  Konstruktor / Destruktor

- `bool Install(ISubsystem* )`
  Installation eines Subsystems; das angegebene Subsystem wird in eine interne Liste eingetragen.
  In der cv act library werden die folgenden Subsysteme unterstützt:
  `CTAPISystem, PCSCSystem, CSPSystem und PKCS11System`
  An dieser Stelle findet noch keine Kommunikation zwischen der Anwendung und dem Slot statt.

- `void Refresh();`
  Start der automatischen Erkennung der verfügbaren Slots

- `const ISlot* GetSlot(int pos) const;`
  Rückgabe eines Slots der Position pos

- `int GetSlotNumber() const;`
  Rückgabe der Anzahl der verfügbaren Slots

- `ISlot* CreateSlot(const char* name) const;`
  Erzeugung einer Instanz von Slot; der Slot-Name kann mit der Funktion `GetName()` von `ISlot` ermittelt werden.

Im folgenden Beispiel werden alle vier verfügbaren Subsysteme installiert und die Anzahl der verfügbaren Slots wird ausgegeben:

```
        SlotManager manager;
        // Subsysteme installieren
        manager.Install(SubsystemReg::CreateSubsystem("PCSC"));
        manager.Install(SubsystemReg::CreateSubsystem("CSP"));
        manager.Install(SubsystemReg::CreateSubsystem("PKCS11","GDP11S10.dl
l"));
        manager.Install(SubsystemReg::CreateSubsystem("CTAPI","
Ctapiw32.dll"));
        manager.Refresh();
        int n = manager.GetSlotNumber();
        cout << "verfügbare slots: " << endl;
        for (int i = 0; i< n; i++)
        std::cout << std::dec << manager.GetSlot(i)->GetName() <<
std::endl;
```

Bemerkungen:

1. Es ist nicht erforderlich, alle verfügbaren Subsysteme zu installieren. In der Regel müssen nur die Subsysteme installiert werden, die benutzt werden sollen.

2. Bei PCSC und CSP wird die DLL winscard.dll automatisch geladen. Bei CTAPI und PKCS11 muss der DLL-Name angegeben werden.

3. Die erstellten Subsysteme werden von SlotManager automatisch freigegeben, falls sie nicht mehr benötigt werden. Ein expliziter Aufruf von

```
delete subsystem
```

ist nicht erforderlich.

4. Das CTAPI-Protokoll unterstützt nicht in vollem Umfang die automatische Erkennung von Slots. Aus diesem Grund dauert die Verarbeitung der Funktion Refresh() bei manchen Smartcard-Readern sehr

lange. Abhilfe bietet in diesem Fall die manuelle Installation der CTAPI-Slots. Die beiden folgenden Beispiele demonstrieren dieses.

Beispiel 1:

```
        SlotManager manager;
        // CTAPI installieren
        ISubsystem* towitoko = SubsystemReg::CreateSubsystem("CTAPI","
Ctapiw32.dll");
        towitoko->Install(1);  // an COM-Port 1 ist ein Towitoko-Reader
                               // angeschlossen
        towitoko->Install(2);  // an COM-Port 2 ist ein weiterer
                               // Towitoko-Reader angeschlossen

        ISubsystem* kobil =
SubsystemReg::CreateSubsystem("CTAPI","Ct32.dll");
        kobil->Install(3);     // an COM-Port 3 ist ein KOBIL-Reader
angeschlossen

        // CTAPI Reader beim SlotManager registrieren
        manager.Install(towitoko);
        manager.Install(kobil);

        // PCSC Reader können automatisch erkannt werden
        manager.Install(SubsystemReg::CreateSubsystem("PCSC"));

        manager.Refresh();     // Diese Funktion sollte immer aufgerufen
werden
```

Alternativ kann das obige Beispiel wie folgt vereinfacht werden .

Beispiel 2:

```
        SlotManager manager;

        // CTAPI installieren
        manager.Install(SubsystemReg::CreateSubsystem("CTAPI","
Ctapiw32.dll, 1, 2");
        manager.Install(SubsystemReg::CreateSubsystem("CTAPI","Ct32.dll,
3");

        // PCSC Reader können automatisch erkannt werden
        manager.Install(SubsystemReg::CreateSubsystem("PCSC"));

        manager.Refresh();     // Diese Funktion sollte immer aufgerufen
werden
```

5.3.2 Slots

Nachdem der SlotManager die verfügbaren Slots erkannt hat, wird eine Slot-Instanz wie folgt erzeugt:

```
ISlot* slot = manager.CreateSlot(slot_name);
```

Danach kann dieser Slot benutzt werden. Der SlotManager sorgt automatisch dafür, Speicher für die Subsysteme wieder freizugeben.

Die folgenden wichtigen Funktionen der Klasse Slot stehen zur Verfügung:

```
bool b = slot->IsTokenPresent();
if (b) Blob atr1 = slot->GetATR();
```

Abfrage, ob eine Smartcard in den Smartcard-Reader eingelegt wurde

```
IToken* CreateToken();
```

Die Registrierung der unterstützten Smartcards ist in `actInit.h` definiert.

Falls eine Smartcard verwendet wird, können das Betriebssystem der Smartcard (Card Operating System, COS) und die Datei-Struktur der Smartcard (Profil) automatisch erkannt werden.

Folgende Kartensysteme werden unterstützt:

Nativ:	CardOS M4, M4.01(a), V4.2, V4.3b
	ACOS A04, A05
	G&D StarCOS 3.0, 3.1, 3.2
JavaCard:	G&D, JCOP, AustriaCard JCOP, Infineon jTOP, Gemalto

Die aktuelle Liste der unterstützten Smartcards im Detail ist ebenso in der Datei `actInit.h` zu finden.

Darüber hinaus werden alle Smartcards unterstützt, für die ein PKCS#11-Modul oder ein Microsoft CSP-Modul verfügbar ist.

5.3.3 EventHandler

Für einige Anwendungen ist es von Bedeutung, ob eine Smartcard, die als Secure Token verwendet wird, in den Smartcard-Reader gesteckt wird bzw. nach der Erstellung einer Signatur aus dem Smartcard-Reader entfernt wird. Dieses ist z. B. der Fall, wenn eine Smartcard zur Authentifizierung genutzt wird. Die folgende Funktion kann verwendet werden, um festzustellen, ob eine Smartcard entfernt oder in den Smartcard-Reader eingelegt wurde:

```
void NotifyEvent(IEventHandler* cmd);
```

Diese Funktion startet einen Thread und überwacht den Slot ständig. Die folgenden Events werden von der cv act library unterstützt:

`TOKEN_REMOVED, TOKEN_INSERTED.`

Voraussetzung ist, dass die Anwendung, die diese Funktion benutzt, als multithread Code erstellt wurde.

Das folgende Beispiel demonstriert die Benutzung von `IEventHandler`:

```
class SlotStateHandler : public IEventHandler
{
public:
        SlotStateHandler() {}
        virtual ~SlotStateHandler() {}
        virtual void OnEvent(int event)
        {
                if (event & TOKEN_REMOVED) std::cout << "card removed!" <<
std::endl;
                if (event & TOKEN_INSERTED) std::cout << "card inserted!" <<
std::endl;
                if (event & SLOT_REMOVED) std::cout << "reader removed!" <<
std::endl;
        }
};
```

Mit

```
slot->NotifyEvent( new SlotStateHandler());
```

wird ein neuer Thread gestartet, der diesen Slot überwacht. Sobald die Smartcard aus dem Smartcard-Reader entfernt oder eingelegt wird, wird die zugehörige Funktion

```
OnEvent(event)
```

aufgerufen.

Der Slot gibt den Speicher, der für den `SlotStateHandler` reserviert war, automatisch wieder frei, falls er nicht mehr benötigt wird.

5.3.4 Token

Die Klasse Token ist das zentrale Objekt der Komponente Secure Token. Sie beschreibt das Profil eines Token und verwaltet die Objekte, die auf dem Token gespeichert sind. Dazu gehören z. B. die Liste der privaten Schlüssel, die PINs und die Zertifikate.

Der private Schlüssel muss ausgewählt werden, um mit einer Smartcard arbeiten zu können. Mit dieser Funktion wird ein privater Schlüssel anhand eines Zertifikates ausgewählt:

```
IKey* key = token->CreateKey(certificate_blob);
```

Hier dient das Zertifikat `certificate_blob` als Auswahlkriterium. Danach können Daten mit diesem privaten Schlüssel signiert oder entschlüsselt werden.
Falls das zugehörige Zertifikat nicht bekannt ist, kann mit der Funktion

```
IKey* key = token->GetKey(nr)->Clone();
```

ein privater Schlüssel direkt ausgewählt werden.

Die angelegte Instanz `key` muss nach der Anwendung wieder freigegeben werden

```
delete key;
```

5.3.5 Key

Ein Key-Objekt stellt die Verbindung zwischen einer Anwendung und einem Token her. Die Verwendung eines Key-Objektes auf einem Token entspricht der Verwendung anderer Key-Objekte:

```
        key->SetParam(act::PIN,Blob("123456"));       // "123456" ist die
PIN
        act::Blob plaintext("abc");
        Algorithm sign(key, act::SIGN);
        act::Blob signature;
        sign << plaintext << act::final >> signature;
        cout << "signature:  " << std::hex << signature << endl;

        Algorithm verify(key,act::VERIFY,signature);
        verify << plaintext << act::final;
        bool result = (verify.GetStatus()==act::SIGNATURE_OK);
        cout << "verify:  " << result << endl;
```

Ein Key-Objekt auf einem Token unterstützt unterschiedliche Parameter, die mit den Funktionen SetParam/GetParam gesetzt oder gelesen werden können:

- `SetParam(PIN,...)`
legt die PIN fest. Diese PIN wird bei der Verwendung des privaten Schlüssels (Erstellung einer Signatur, Entschlüsselung) geprüft.

- `SetParam(HASH,...)`
legt die Hashfunktion fest, die bei der Erstellung der Signatur verwendet werden soll. Dies setzt voraus, dass das Token die angegebene Hashfunktion unterstützt.

- `GetParam(KEYUSAGE)`
gibt die erlaubte Schlüsselverwendung (key usage) zurück. Die möglichen Werte sind in `actITokenPIN.h` definiert, u. a. `SIGN_DATA, DEC_DATA, AGREE_KEY`.

- `GetParam(KEYSIZE)`
gibt die Länge des Schlüssels zurück, z.B. 1024 für RSA-Key

- `GetParam(CERTIFICATE)`
gibt das Zertifikat zu diesem Key-Objekt zurück, falls das Zertifikat auf der Smartcard vorhanden ist. Sonst wird `Blob()` zurückgegeben.

- `GetParam(MODULO, modulo)`
- `GetParam(PUBLICKEY, publickey)`
Gibt den Modulo und den Public key, zurück falls es sich um einen RSA-Key handelt.

5.3.6 PIN

Die Klasse PIN fasst Operationen im Zusammenhang mit der PIN zusammen. Die verschiedenen Funktionen werden im folgenden Beispiel benutzt:

```
        ITokenPIN* user_pin = token->GetPIN(0)// suppose the first pin is
user pin
        ITokenPIN* so_pin = token->GetPIN(1)  // suppose the second pin is
                                              // system operator pin

        user_pin.ChangePin(Blob("123456"), Blob("87654321"));
        // old PIN 123456, new PIN: 87654321

        user_pin.UnlockPin(so_pin, Blob("11111111"), Blob("654321"));
        // Entsperren: "11111111" ist die so pin. Nachdem Entsperren ist
die
        // neue user pin "654321"
```

5. 4 Ausnahme- und Fehlerbehandlung

In der **cv act *library*** werden Fehler und Ausnahmeereignisse behandelt, indem Exceptions geworfen werden, so dass der Fehlerbehandlungscode explizit vom restlichen Code getrennt ist. Die geworfenen Exceptions können aufgefangen werden, damit Sie als Anwender die Ausnahmeereignisse an den entsprechenden Stellen behandeln können. Die Ausnahmeklasse `Exception` unterteilt sich in die Unterklassen `BadException`, `LogicalException` und `RuntimeException`.

Bei der `LogicalException` handelt es um logische Ausnahmen, die beispielsweise geworfen werden, wenn Inkonsistenzen in Datenstrukturen auftreten. Mit Hilfe der Exception-Hierarchie kann man die Ausnahmen einer bestimmten Art fangen. Die `LogicalException` unterteilt sich in weitere Unterklassen, die sich noch folgendermaßen unterteilen:

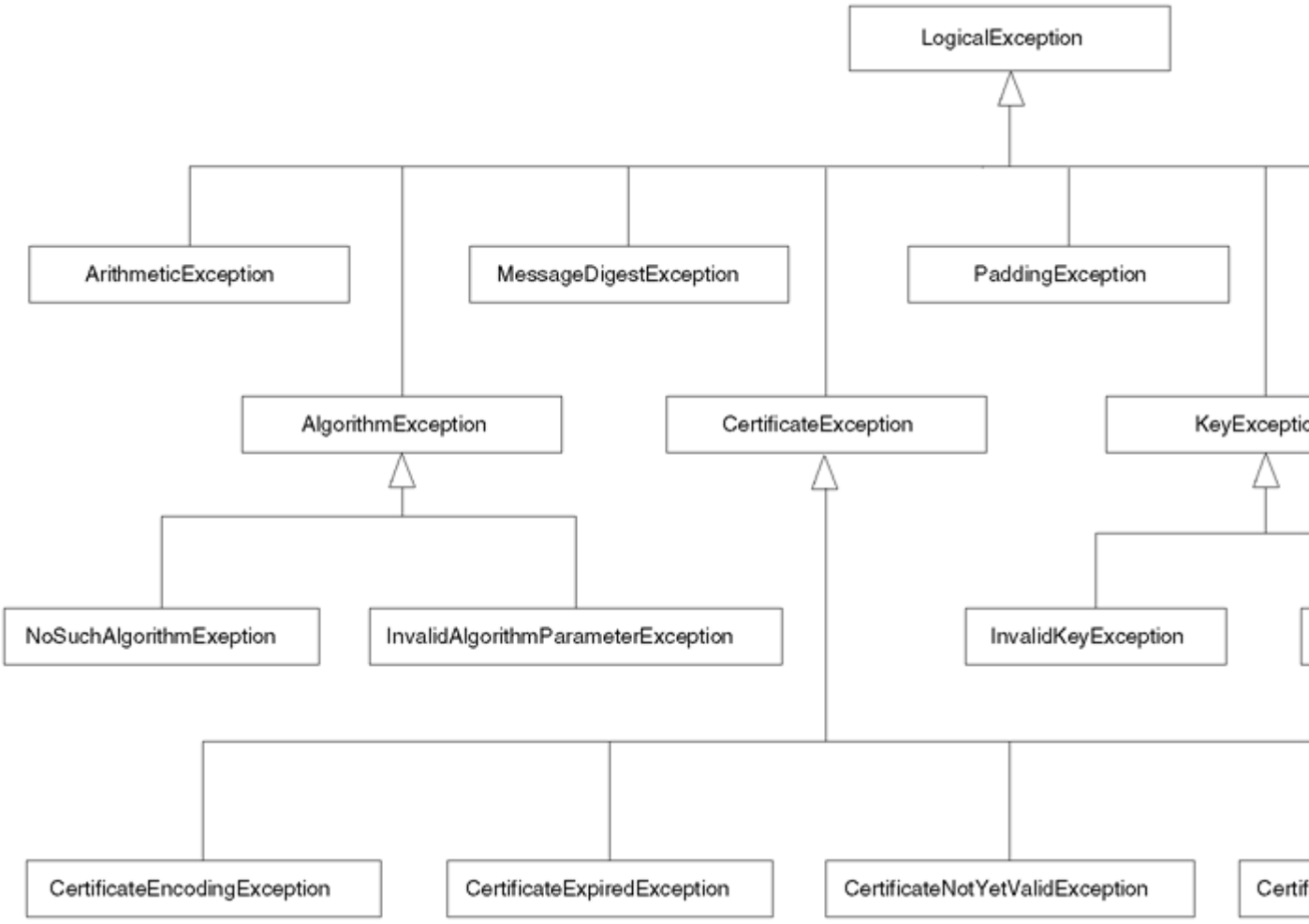


Abbildung: Exception-Hierarchie

In der Klasse `Exception` sind zwei Methoden definiert:

```
const char* what();

const char* where();
```

Die Funktion `where()` gibt an, wo die Exception geworfen wird und die Funktion `what()` gibt eine Fehlermeldung aus.

Am folgenden Beispiel wird nun gezeigt, wie Sie Ausnahmebehandlung implementieren können. Dazu ist das modifizierte Beispiel für digitale Signaturen aus 5. 2. 6 mit Fehlerbehandlung aufgeführt:

```
void generate_key_pair(act::Blob &PrivateKeyBlob,
                           act::Blob &PublicKeyBlob)
{
        try
        {
                act::Key key("ECDSA"); //Schluesselobjekt wird angelegt
                key.Generate();  //Schluessel wird erzeugt
                key.Export(PrivateKeyBlob,act::PRIVATE); //privater
                                //Schluessel wird exportiert
                key.Export(PublicKeyBlob,act::PUBLIC); //oeffentlicher
                                //Schluessel wird exportiert
        }
        catch (act::KeyException& e)
        {
                //Fehlerbehandlung:
                // exit(-1)            //Die Anwendung beenden oder
                // cout << e.what() << endl;//die Fehlermeldung ausgeben
        }
// ggf. weitere Fehlerbehandlungen
// ...
}


void sign_from_stream(const act::Blob &PrivateKeyBlob,
                std::istream &in, act::Blob &Signature)
{
        try
        {
                act::Key key(PrivateKeyBlob); //privates
                                // Schluesselobjekt wird angelegt, indem der
                                // Schluessel aus einem Blob importiert wird
                act::Algorithm Sign(key,act::SIGN); );// Signier-
                                // algorithmus anlegen
                act::Blob InData(BUFFER_SIZE); // Blob fuer Eingabe
                                // anlegen
                std::streamsize count;
                do
                {       //Einlesen
                        in.read(reinterpret_cast < char*>
(&PlainData[0]),BUFFER_SIZE); //Einlesen
                        count=in.gcount();
                        if(count < BUFFER_SIZE) //Sonderbehandlung des
                        // letzten Blocks, der nicht mehr volle Laenge hat
                        {
                                InData.resize(count);
                        }
                        Sign << InData; // die zu signierenden Daten
                       // werden an den Algorithmus uebergeben
                } while(count == BUFFER_SIZE);

                Sign << final; //es kommen keine weiteren Daten, die
                                //signiert werden sollen
                Sign >> Signature; //die signierten Daten werden an
                                // einen Blob uebergeben
        }
```

```
        catch (act::AlgorithmException& e)
        {
                // Fehler innerhalb des Algorithmus
                std::cout << e.what() << std::endl;
                //Fehlermeldung ausgeben

        }
        catch (act::InvalidKeyException& e)
        {
                // Der Schlüssel ist ungültig
                std::cout << e.what() << std::endl;
                //Fehlermeldung ausgeben
        }
        catch (act::Exception& e)
        {
        // ggf. weitere Fehlerbehandlungen
        // ...
        }
}
```

5.5 Die Objekte der **cv act *library***

In den vorherigen Abschnitten sind die Methoden `SetParam()` und `GetParam()` verwendet worden. Damit Sie diese Parameter setzen können, werden hier die möglichen Werte aufgelistet.

Falls mit den Methoden `SetParam()` bzw. `GetParam()` Namen wie z.B. bei `CIPHER` oder `BCMODE` gesetzt bzw. ausgelesen werden, ist der Datentyp ein `Blob`, der den Namen enthält. Bei `SetParam()` lässt sich zusätzlich der Name als NULL-terminierter `const char*` übergeben, damit Sie C/C++ String Literale verwenden können. Die Namen der in der **cv act *library*** enthaltenen elliptischen Kurven zusammen mit ihren Daten finden Sie in Anhang D.

Falls mit den Methoden `SetParam()` bzw. `GetParam()` Langzahlen gesetzt bzw. ausgelesen werden, sind bzw. werden diese folgendermaßen in einen Blob kodiert: für einen Blob `x` mit `n:=x.size()` gilt:

- `Langzahl:=x[0]*256`$^{n-1}$ `+ x[1]*256`$^{n-2}$ `+...+ x[n-1]` für positive Zahlen und
- `Langzahl:=(x[0] XOR 255)*256`$^{n-1}$ `+...+ (x[n-1] XOR 255)+1` für negative Zahlen (Zweierkomplement).

Das höchstwertige Bit von `x[0]` gibt das Vorzeichen an mit 0 für positiv und 1 für negativ.

Bei `SetParam()` lässt sich zusätzlich die Zahl als NULL-terminierter String übergeben, wie z.B. `"128234345"`.

Falls es sich bei einem Parameter um einen Punkt auf der elliptischen Kurve handelt, ist dieser nach ANSI X9.62 Kapitel 4.3.6 Abschnitt 3 kodiert:

` 0x04||x||y`.

Dabei werden `x` und `y` als Langzahlen kodiert. Falls es sich bei dem zugrundeliegenden Körper um GF(p) handelt, erfolgt die Kodierung mit der Bitlänge der Primzahl des zugrundeliegenden Körpers - aufgerundet zum nächsten Vielfachen von 8. Falls es sich bei dem zugrundeliegenden Körper um GF(2^m) handelt, erfolgt die Kodierung mit der Bitlänge von m – aufgerundet zum nächsten Vielfachen von 8.

5. 5. 1 Die `Key` -Objekte

Die folgenden Key-Objekte sind in der **cv act *library*** enthalten: `BlockCipher`, `DSA`, `RSA`, `ECDSA`, `DH`, `ECDH`, `HashMAC`, `CBCMAC` und `IES`. Im folgenden Abschnitt wird aufgeführt, was diese Objekte repräsentieren und welche Parameter man setzen oder auslesen kann. Der Default-Exporttyp ist bei asymmetrischen Schlüsseln `PRIVATE` und bei symmetrischen Schlüsseln `SECRET` und beim `SmartCard`-Objekt `PUBLIC`.

Name: **`BlockCipher`**

repräsentiert den symmetrischen Schlüssel einer Blockchiffre. Mit der Methode `CreateAlgorithm()` können ein Ver- oder ein Entschlüsselungsalgorithmus angelegt werden. Dazu übergibt man dieser Methode entsprechend `ENCRYPT` oder `DECRYPT`. Bei einem BlockCipher-Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

`BCMODE:`

damit lässt sich der Name des verwendeten Betriebsmodus setzen oder auslesen. Defaultwert: "`CBC`"

`CIPHER:`

damit lässt sich der Name der verwendeten Blockchiffre setzen oder auslesen. Beim Verändern des Parameters wird `RAWKEY` ungültig. Defaultwert: "`AES`"

`DERIVATOR:`

damit lässt sich der Name des verwendeten Schlüsselableitungsverfahrens setzen oder auslesen, dass beim Verwenden der Methode `Derive()` benutzt wird. Defaultwert: "`KDF1`"

`IV:`

damit lässt sich ein Initialisierungsvektor setzen oder auslesen, wie er beispielsweise im CBC-Modus verwendet wird. Dieser Wert wird nach jedem Verschlüsseln zurückgesetzt. Wenn kein IV gesetzt wird, wird er zufällig generiert. Der zu verwendende Datentyp ist Blob.

`KEYSIZE:`

damit lässt sich die Schlüsselgröße (in Byte) setzen oder auslesen, die bei der Generierung eines Schlüssels mit den Methoden `Generate()` oder `Derive()` verwendet wird. Wenn die verwendete Chiffre diese Schlüsselgröße nicht unterstützt, wird die nächstliegende unterstützte Größe verwendet. Der zu verwendende Datentyp ist `int`. Defaultwert: 10.

`PADDING:`

damit lässt sich der Name des verwendeten Paddingverfahrens setzen oder auslesen. Defaultwert: "`PKCS5`"

`RAWKEY:`

damit lässt sich der eigentliche Schlüssel der Blockchiffre, d.h. die Bytefolge in Form eines Blobs setzen oder auslesen. Der zu verwendende Datentyp ist Blob.

Interface: **IBlockCipherKey**

Exporttypen: **SECRET**

Name: **DSA**

repräsentiert den asymmetrischen Schlüssel des DSA-Signaturverfahrens. Entweder den privaten Schlüssel mit dem man sowohl signieren als auch verifizieren kann, oder den öffentlichen Schlüssel, mit dem man nur verifi zieren kann. Mit der Methode `CreateAlgorithm()` können ein Siginer- oder ein Verifikationsalgorithmus angelegt werden. Dazu übergibt man dieser Methode entsprechend `SIGN` oder `VERIFY`. Bei einem DSA-Key- Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

`CHECKDOMAINPARAMS:`

damit wird die Konsistenz der *domain parameter* überprüft. Es wird 0 für Inkonsistenz und 1 für Konsistenz zurückgegeben.

`COFACTOR:`

ist der Cofaktor des Ringes (einer der *domain parameter*) und wird als Langzahl gesetzt oder ausgelesen.

`ENCODER:`

damit lässt sich der Name des Kodierungsverfahrens zum Signieren und Verifizieren setzen oder auslesen. Defaultwert: "`EMSA1`"

`MODULOSIZE:`

damit lässt sich die Größe des Modulus (in Bit) setzen oder auslesen, die bei der Generierung eines Schlüssels mit der Methode `Generate()` verwendet wird. Der Wertebereich ist 512 bis 1024 in 64-Bit-Schritten und der zu verwendende Datentyp ist `int`. Defaultwert: 1024

`PRIVATEKEY:`

damit lässt sich der private Schlüssel des Verfahrens als eine Langzahl setzen oder auslesen.

`PUBLICKEY:`

damit lässt sich der öffentliche Schlüssel des Verfahrens setzen oder auslesen. Dieser ist ein Element aus dem Ring und wird als eine Langzahl gesetzt oder ausgelesen.

`RING:`

ist als der zugrundeliegende Ring einer der *domain parameter* und wird als Langzahl gesetzt oder ausgelesen.

RINGGENERATOR:

ist das erzeugende Element des Rings (*domain parameter*) und wird als Langzahl gesetzt oder ausgelesen.

RINGORDER:

ist die Ordnung des Rings (*domain parameter*) und wird als Langzahl gesetzt oder ausgelesen.

Interface: **ISignatureKey**

Exporttypen: **PRIVATE, PUBLIC, DOMAINPARAMS**

Name: **RSA**

repräsentiert den asymmetrischen Schlüssel des RSA-Signaturverfahrens. Dies ist entweder der privaten Schlüssel zum Signieren, Verifizieren, Ver- und Entschlüsseln, oder der öffentliche Schlüssel, mit dem man nur Verifizieren oder Verschlüsseln kann. Mit der Methode `CreateAlgorithm()` können die entsprechenden Algorithmen mit `SIGN`, `VERIFY`, `ENCRYPT` oder `DECRYPT` angelegt werden. Bei einem RSA -Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

ENCODER:

damit lässt sich der Name des Kodierungsverfahrens zum Signieren und Verifizieren setzen oder auslesen. Defaultwert: "`PKCS1V1_5EMSA`"

MODULO:

ist der Modulus n (Produkt zweier großer Primzahlen) und wird als Langzahl gesetzt oder ausgelesen. Durch das Setzen dieses Parameters werden `PRIME_P` und `PRIME_Q` verworfen.

MODULOSIZE:

damit lässt sich die Größe des Modulus in Bit setzen oder auslesen, die beim Erzeugen des Schlüssls mit der Methode `Generate()` verwendet wird. Der zu verwendende Datentyp ist `int`. Defaultwert:1024

PADDING:

damit lässt sich der Name des Paddingverfahrens zum Ent- und Verschlüsseln setzen oder auslesen. Defaultwert: "`RSAES`"

`PRIME_P` und `PRIME_Q`:

sind die Primzahlen im Produkt n=pq und werden als Langzahlen gesetzt oder ausgelesen. Diese Parameter sollten unmittelbar nacheinander gesetzt werden. `MODULO` wird automatisch auf ihr Produkt gesetzt.

PRIVATEKEY:

damit lässt sich der private Exponent des Verfahrens als eine Langzahl setzen oder auslesen.

`PUBLICKEY:`

damit lässt sich der öffentliche Exponent des Verfahrens als eine Langzahl setzen oder auslesen.

`PUBLICKEYSIZE:`

damit kann man die Größe des öffentlichen Exponenten beschränken, der mit der mit der Methode `Generate()` erzeugt wird. Der zu verwendende Datentyp ist `int`. Defaultwert: 1024

Interface: **`IRSAKey`**

Exporttypen: **`PRIVATE, PUBLIC`**

Name: **`ECDSA`**

repräsentiert den asymmetrischen Schlüssel des EC-DSA-Signaturverfahrens: entweder den privaten Schlüssel mit dem man sowohl signieren als auch verifizieren kann, oder den öffentlichen Schlüssel, mit dem man nur verifizieren kann. Mit der Methode `CreateAlgorithm()` können ein Signier- oder ein Verifikationsalgorithmus angelegt werden. Dazu übergibt man dieser Methode entsprechend `SIGN` oder `VERIFY`. Bei einem EC-DSA -Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

`CURVE:`

damit lässt sich der Name der elliptischen Kurve setzen oder auslesen. Defaultwert: "`SECGp160r1`"

`CURVEPARAM:`

damit lässt sich eine nicht in der **cv act *library*** enthaltene Kurve verwenden. Eine Beispielanwendung findet sich in Abschnitt 5.2.4, die Methoden zum Anlegen einer solchen Kurve in Anhang D. "

`ENCODER:`

damit lässt sich der Name des Kodierungsverfahrens zum Signieren und Verifizieren setzen oder auslesen. Defaultwert: "`EMSA1`"

`PRIVATEKEY:`

damit lässt sich der private Schlüssel des Verfahrens als eine Langzahl setzen oder auslesen.

`PUBLICKEY:`

damit lässt sich der öffentliche Schlüssel des Verfahrens als ein Punkt auf der elliptischen Kurve setzen oder auslesen.

`PUBLIC_X:`

damit lässt sich die X-Koordinate des öffentlichen Schlüssels als Langzahl setzen oder auslesen.

`PUBLIC_Y:`

damit lässt sich die Y-Koordinate des öffentlichen Schlüssels als Langzahl setzen oder auslesen.

Interface: **`ISignatureKey`**

Exporttypen: **`PRIVATE, PUBLIC, DOMAINPARAMS`**

Name: **`DH`**

repräsentiert den asymmetrischen Schlüssel des Diffie-Hellman-Austauschverfahrens: und zwar entweder den privaten Schlüssel oder den öffentlichen Schlüssel. Mit der Methode `Derive()` eines privaten DH-Schlüssels wird ein gemeinsames Geheimnis erzeugt, in dem man den exportierten, öffentlichen Schlüssel der anderen Partei als ersten Parameter übergibt. Bei einem DH -Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

`CHECKDOMAINPARAMS:`

damit wird die Konsistenz der *domain parameter* überprüft. Es wird 0 für Inkonsistenz und 1 für Konsistenz zurückgegeben.

`COFACTOR:`

ist der Cofaktor des Ringes (einer der *domain parameter*) und wird als Langzahl gesetzt oder ausgelesen.

`COMPATIBLE:`

damit lässt sich setzen oder auslesen, ob das Verfahren kompatibel zum klassischen DH-Verfahren ist oder zu dem mit Cofaktor-Multiplikation. 0 steht für `false` und 1 für `true` und der zu verwendende Datentyp ist `int`. Defaultwert: `true`

`DERIVATOR:`

damit lässt sich der Name des verwendeten Schlüsselableitungsverfahrens setzen oder auslesen, dass beim Verwenden der Methode `Derive()` benutzt wird. Defaultwert: "`KDF1`"

`MODULOSIZE:`

damit lässt sich die Größe des Modulus (in Bit) setzen oder auslesen, die bei der Generierung eines Schlüssels mit der Methode `Generate()` verwendet wird. Der Wertebereich ist 512 bis 4096 in 64-Bit-Schritten und der zu verwendende Datentyp ist `int`. Defaultwert: 1024

`PRIVATEKEY:`

damit lässt sich der private Schlüssel des Verfahrens als eine Langzahl setzen oder auslesen.

PUBLICKEY:

damit lässt sich der öffentliche Schlüssel des Verfahrens setzen oder auslesen. Dieser ist ein Element aus dem Ring und wird als eine Langzahl gesetzt oder ausgelesen.

RING:

ist als der zugrundeliegende Ring einer der *domain parameter* und wird als Langzahl gesetzt oder ausgelesen.

RINGGENERATOR:

ist das erzeugende Element des Ringes (*domain parameter*) und wird als Langzahl gesetzt oder ausgelesen.

RINGORDER:

ist die Ordnung des Ringes (*domain parameter*) und wird als Langzahl gesetzt oder ausgelesen.

Interface: **IAgreementKey**

Exporttypen: **PRIVATE, PUBLIC, DOMAINPARAMS, SECRET**

Name: **ECDH**

repräsentiert den asymmetrischen Schlüssel des Diffie-Hellman-Austauschverfahrens auf Basis elliptischer Kurven: und zwar entweder den privaten Schlüssel oder den öffentlichen Schlüssel. Mit der Methode `Derive()` eines privaten EC-DH-Schlüssels wird ein gemeinsames Geheimnis erzeugt, in dem man den exportierten, öffentlichen Schlüssel der anderen Partei als ersten Parameter übergibt. Bei einem EC-DH -Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

COMPATIBLE:

damit lässt sich setzen oder auslesen, ob das Verfahren kompatibel zum klassischen EC-DH-Verfahren ist oder zu dem mit Cofaktor-Multiplikation. 0 steht für `false` und 1 für `true` und der zu verwendende Datentyp ist `int`. Defaultwert: `true`

CURVE:

damit lässt sich der Name der elliptischen Kurve setzen oder auslesen. Defaultwert: "`SECGp160r1`"

CURVEPARAM:

damit lässt sich eine nicht in der **cv act *library*** enthaltene Kurve verwenden. Eine Beispielanwendung findet sich in Abschnitt 5.2.4, die Methoden zum Anlegen einer solchen Kurve in Anhang D. "

DERIVATOR:

damit lässt sich der Name des verwendeten Schlüsselableitungsverfahrens setzen oder auslesen, dass beim Verwenden der Methode `Derive()` benutzt wird.

`PUBLICKEY:`

damit lässt sich der öffentliche Schlüssel des Verfahrens setzen oder auslesen. Dieser wird als ein Punkt auf der elliptischen Kurve gesetzt oder ausgelesen.

`PRIVATEKEY:`

damit lässt sich der private Schlüssel des Verfahrens als eine Langzahl setzen oder auslesen.

`PUBLIC_X:`

damit lässt sich die X-Koordinate des öffentlichen Schlüssels als Langzahl setzen oder auslesen.

`PUBLIC_Y:`

damit lässt sich die Y-Koordinate des öffentlichen Schlüssels als Langzahl setzen oder auslesen.

Interface: **`IAgreementKey`**

Exporttypen: **`PRIVATE, PUBLIC, DOMAINPARAMS, SECRET`**

Name: **`HashMAC`**

repräsentiert den symmetrischen Schlüssel eines Hash-MACs. Mit der Methode `CreateAlgorithm()` kann mit der Parameter-ID `MAC` ein MAC-Algorithmus angelegt werden. Bei einem HashMAC-Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

`DERIVATOR:`

damit lässt sich der Name des verwendeten Schlüsselableitungsverfahrens setzen oder auslesen, dass beim Verwenden der Methode `Derive()` benutzt wird. Defaultwert: "`KDF1`"

`HASH:`

damit lässt sich der Name des verwendeten Hash-Algorithmus setzen oder auslesen. Defaultwert: "`SHA1`"

`KEYSIZE:`

damit lässt sich die Schlüsselgröße (in Byte) setzen oder auslesen, die bei der Generierung eines Schlüssels mit den Methoden `Generate()` oder `Derive()` verwendet wird. Der zu verwendende Datentyp ist `int`. Defaultwert: 10

RAWKEY:

damit lässt sich der eigentliche Schlüssel des Hash-MACs, d.h. die Bytefolge in Form eines Blobs setzen oder auslesen. Der zu verwendende Datentyp ist Blob.

Interface: **IHashMACKey**

Exporttypen: **SECRET**

Name: **CBCMAC**

repräsentiert den symmetrischen Schlüssel eines CBC-MACs. Mit der Methode `CreateAlgorithm()` kann mit der Parameter-ID `MAC` ein MAC-Algorithmus angelegt werden. Bei einem CBCMAC-Key-Objekt können folgende Parameter gesetzt bzw. ausgelesen werden:

CIPHER:

damit lässt sich der Name der verwendeten Blockchiffre setzen oder auslesen. Defaultwert: "`TripleDES`"

DERIVATOR:

damit lässt sich der Name des verwendeten Schlüsselableitungsverfahrens setzen oder auslesen, dass beim Verwenden der Methode `Derive()` benutzt wird. Defaultwert: "`KDF1`"

IV:

damit lässt sich ein Initialisierungsvektor setzen oder auslesen. Wenn kein IV gesetzt wird, wird der Nullvektor verwendet. Der zu verwendende Datentyp ist Blob.

KEYSIZE:

damit lässt sich die Schlüsselgröße (in Byte) setzen oder auslesen, die bei der Generierung eines Schlüssels mit den Methoden `Generate()` oder `Derive()` verwendet wird. Der zu verwendende Datentyp ist `int`. Defaultwert: 20

RAWKEY:

damit lässt sich der eigentliche Schlüssel des CBC-MACs, d.h. die Bytefolge in Form eines Blobs setzen oder auslesen. Der zu verwendende Datentyp ist Blob.

Interface: **ICBCMACKey**

Exporttypen: **SECRET**

Name: **IES**

repräsentiert einen asymmetrischen Schlüssel für das IES-Schema. Mit der Methode `CreateAlgorithm()` können ein Ent- oder Verschlüsselungsalgorithmus angelegt werden. Dazu übergibt man dieser Methode entsprechend `ENCRYPT` oder `DECRYPT`. Als zweiten Parameter benötigt die Methode den exportierten, öffentlichen Schlüssel der anderen Partei als Blob. Weitere Parameter als die unten angeführten sind die des verwendeten Schlüsselaustauschverfahrens, wie beispielsweise Diffie-Hellman.

`AGREEMENT:`

damit lässt sich der Name des verwendeten Schlüsselaustauschverfahrens setzen oder auslesen. Defaultwert: "`ECDH`"

`MAC:`

damit lässt sich der Name des verwendeten MAC-Verfahrens setzen oder auslesen. Defaultwert: "`HashMAC`"

Interface: **IIESKey**

Exporttypen: **SECRET, PRIVATE, PUBLIC, DOMAINPARAMS**

5. 5. 2 Die `BlockCipher`-Objekte

Die folgenden BlockCipher-Objekte sind in der **cv act *library*** enthalten: `AES, BlowFish, CAST128, DES, Mars, Rijndael, Serpent, TripleDES` und `TwoFish`.

Interface: **`IBlockCipher`**

5. 4. 3 Die `BlockCipherMode`-Objekte

Die folgenden BlockCipherMode-Objekte sind in der **cv act *library*** enthalten: `ECB, CBC, CFB, CTR und OFB`. Die implementierten Objekte erwarten keine Parameter.

Interface: **`IBlockCipherMode`**

5. 5. 4 Die `Hash`-Objekte

Die folgenden Hash-Objekte sind in der **cv act *library*** enthalten: `MD2, MD4, MD5, RIPEMD128, RIPEMD160, SHA0, SHA1, SHA224, SHA256, SHA384` und `SHA512`. Die implementierten Objekte erwarten keine Parameter.

Interface: **`IHashAlg`**

5. 5. 5 Die EMSA-Objekte

Die folgenden EMSA-Objekte sind in der **cv act *library*** enthalten: `EMSA1` (IEEE 1363) , `PKCS1V1_5EMSA, PKCS1_PSS_EMSA und TLS_EMSA`. Für die lässt sich mit dem Parameter `HASH` der Name des verwendeten Hashalgorithmus setzen oder auslesen. Defaultwert: "`SHA1`"

Interface: **`IEMSAWithHashAlg`**

5. 5. 6 Das `Derivator`-Objekt

Das folgenden Derivator-Objekt ist in der **cv act *library*** enthalten: `KDF1, KDF2(= X963KDF), PBKDF1 und PBKDF2` (IEEE 1363). Dafür lässt sich mit dem Parameter `HASH` der Name des verwendeten Hashalgorithmus setzen oder auslesen. Defaultwert: KDF1/2: "`SHA512`", PBKDF1/2: "`SHA1`"

Interface: **`IDerivatorWithHash`**

5. 4. 7 Die `Padding`-Objekte

Die folgenden Padding-Objekte sind in der **cv act *library*** enthalten: `PKCS5`, `RSAES` (PKCS #1 V1.5), `ISO`, `NOPAD` und `ISO9796`. Die implementierten Objekte erwarten keine Parameter.

Interface: **`IPadding`**

5. 5. 8 Die RNG-Objekte

Die folgenden Zufallszahlengenerator-Objekte sind in der **cv act *library*** enthalten: LCG, BBS und FIPS186(-DES). Diese drei Zufallszahlengeneratoren unterscheiden sich bezüglich ihrer Stärke und der Geschwindigkeit, mit der sie Ergebnisse liefern. Den „besten“ Zufall liefert der BBS, der LCG den schwächsten und FIPS186 liegt dazwischen. Bezüglich der Geschindigkeit verhält es sich umgekehrt, LCG rechnet am schnellsten, FIPS186 am zweitschnellsten und BBS am langsamsten. Sie haben folgende Parameter:

Name: **LCG**

MODULOSIZE:

damit lässt sich die Bit-Größe des Modulus setzen und auslesen. Dieser Wert muss ein Vielfaches von 8 und größer als 32 sein. Der zu verwendende Datentyp ist int. Defaultwert: 512

MULTIPLIER:

damit lässt sich der Multiplikator des LCGs als Langzahl setzen oder auslesen.Defaultwert:
"131791536128275945572846251812980853904873168641191397765156334579182394357949947058157829183842492446913855666129352945677145057094177144520761859586670 77"

X:

damit lässt sich innere Zustand (*seed*) des LCGs als Langzahl setzen oder auslesen. Default: Es wird ein zufälliger seed aus Hardwareinformationen generiert.

Interface: **IRNGAlg**

Name: **BBS**

MODULO:

damit lässt sich Modulo (Produkt zweier großer Primzahlen) als Langzahl setzen und auslesen.

Defaultwert:
879349164056206910316309077479520662396214912908625896295513605950127584463295282715552106176190550277853893035339209834121748750806514082505318362739047841549951496585943783521264976429337231365227168290221071883432550164978533863656768514379636236240538528626992373159884200561253102489775245845432988950 77"

X:

damit lässt sich innere Zustand (*seed*) des BBS als Langzahl setzen oder auslesen.

Interface: **IRNGAlg**

Name: **FIPS186(-DES)**

`X:`

damit lässt sich innere Zustand (*seed*) des FIPS186 als Langzahl setzen oder auslesen. Default: Es wird ein zufälliger seed aus Hardwareinformationen generiert.

Interface: **`IRNGAlg`**

5. 5. 9 Das `Certificate`-Objekt

Die folgenden Zertifikats-Objekte sind in der **cv act *library*** enthalten: `X509` und `CV`.

Name: **`x509`**

`ISSUER:`

damit lässt sich der Name des Zertifikatsausstellers als Klartext nach LDAP-Konvention setzen oder auslesen.

Beispielsweise:"CN=Test CA, o= cv cryptovision gmbh, C=DE"

`ISSUER_DER:`

damit lässt sich der Name des Zertifikatsausstellers als ASN.1-DER-kodierter Directory-Name setzen oder auslesen (siehe [X.509]).

`ISSUER_UID:`

damit lässt sich der Unique-Identifier des Zertifikatsausstellers setzen oder auslesen. Von der Benutzung der Unique-Identifier ist abzuraten, da diese vom IETF missbilligt sind. Falls Sie diese doch verwenden möchten, müssen die Blobs nach ASN.1-Bitstrings formatiert werden, d.h. das erste Byte eines Blobs gibt die Anzahl der nicht genutzten Bits des letzten Bytes des Blobs an.

`SERIALNR:`

damit lässt sich die Nummer des Zertifikats als Langzahl setzen oder auslesen. Der Defaultwert ist eine beim Signieren des Zertifikats zufällig generierte 128-Bit-Zahl.

`SUBJECT:`

damit lässt sich der Name des Benutzers als Klartext nach LDAP-Konvention setzen oder auslesen.

`SUBJECT_DER:`

damit lässt sich der Name des Benutzers als ASN.1-DER-kodierter Directory-Name setzen oder auslesen (siehe [X.509]).

`SUBJECT_UID:`

damit lässt sich der Unique-Identifier des Benutzers setzen oder auslesen. Von der Benutzung der Unique-Identifier ist abzuraten, da diese vom IETF missbilligt sind. Falls Sie diese doch verwenden möchten, lesen Sie bitte den Hinweis unter `ISSUER_UID`.

`VERSION:`

damit lässt sich die Version des Zertifikats auslesen, die in Abhängigkeit vom Vorhandensein von Extensions oder UIDs automatisch gesetzt wird. Dabei steht 0 für v1-Zertifikate, 1 für v2-Zertifiate und 2 für v3-Zertifikate.

`NOTBEFORE_DER:`

damit lässt sich der Gültigkeitsbeginn des Zertifikats setzen oder auslesen, wobei dieses Datum ASN.1-DER-kodiert ist. Der Defaultwert ist der Zeitpunkt des Signierens.

NOTAFTER_DER:

damit lässt sich das Gültigkeitsende des Zertifikats setzen oder auslesen, wobei dieses Datum ASN.1-DER-kodiert ist. Der Defaultwert ist der Zeitpunkt des Signierens + 2 Jahre.

Interface: **X509Certificate**

Name: **CV**

CVCAREFERENCE:

setzt einen Verweis auf den öffentlichen Schlüssel der Certificate Authority.

CVHOLDERREFERENCE:

setzt einen Verweis auf den öffentlichen Schlüssel dieses Zertifikats.

Weist sich die Certificate Authority mit diesem Zertifikat selbst aus, dann müssen CA-Reference und Holder-Reference übereinstimmen.

Die References setzen sich aus einem zweibuchstabigen Ländercode, einer bis zu neunstelligen Kennung und einer fünfstelligen Sequenznummer zusammen.

CVCHATEMPLATE:

setzt das Holder Authorization Template.

CVEFFECTIVEDATE und CVEXPIRATIONDATE:

setzt den ersten oder letzten Tag der Gültigkeit dieses Zertifikats.

CVPUBLICKEY_TLV:

setzt den öffentlichen Schlüssel dieses Zertifikats.

CVPROFILEID:

setzt die Profil-ID dieses Zertifikats.

CVEACTEMPLATE

setzt das EAC ePass-Template für das Certificate Holder Authorization Template.

CURVEPARAM:

setzt die zugrundeliegenden Kurvenparameter für ECC-Schlüssel im Zertifikat.

Interface: **CVCertificate**

5. 5. 10 Das CRL-Objekt

Das folgenden CRL-Objekt ist in der **cv act *library*** enthalten: X509.

Name: **x509**

ISSUER:

damit lässt sich der Name des Zertifikatsausstellers als Klartext nach LDAP-Konvention setzen oder auslesen.

Beispielsweise:"CN=Test CA, o= cv cryptovision gmbh, C=DE"

ISSUER_DER:

damit lässt sich der Name des Zertifikatsausstellers als ASN.1-DER-kodierter Directory-Name setzen oder auslesen (siehe [X.509]).

VERSION:

damit lässt sich die Version des Zertifikats auslesen, die in Abhän-gigkeit vom Vorhandensein von Extensions oder UIDs automatisch gesetzt wird. Dabei steht 0 für v1-Zertifikate und 1 für v2-Zertifiate.

THISUPDATE_DER:

damit lässt sich das Ausstellungsdatum der CRL setzen oder auslesen, wobei dieses Datum ASN.1-DER-kodiert ist. Der Defaultwert ist der Zeitpunkt des Signierens.

NEXTUPDATE_DER:

damit lässt sich das Datum der nächsten Aktualisierung der CRL setzen oder auslesen, wobei dieses Datum ASN.1-DER-kodiert ist. Wenn dieses Feld in der CRL nicht vorhanden ist, wird ein leerer Blob zurückgegeben. Um dieses Feld zu löschen, können Sie einen leeren Blob übergeben. Defaultwert: nicht vorhanden.

Interface: **X509CRL**

5.5.11 Das SCardLock-Objekt

Dieses Objekt dient dazu, konkurrierende Zugriffe auf die Smartcard zu unterbinden. Es folgt dem RAII- (Resource Acquisition Is Initialization) –Paradigma, was heißt, dass allein die Erstellung des Objekts die Sperrung des Zugriffs bewirkt und die Entfernung des Objekts die Freigabe.

```
class SCardLock

{



public:

      SCardLock(ISCardOS* os);

      ~SCardLock();



      void Lock(ISCardOS* os);

      void Unlock();



};
```

So sei folgendes Codestück möglich:

```
{
        SCardLock lock(os);

        // .. Kartenoperationen ..
}
```

Die nach der Erstellung der Variable `lock` erfolgten Operationen auf der Karte erfolgen mit exklusivem Zugriff und die Sperrung wird solange aufrecht erhalten, bis der Programmfluss den Befehlsblock wieder verlässt (ggf. auch in Folge einer Exception)

Kapitel 6 Die Interfaces der **cv act** *library*

Zusätzlich zu der Benutzung der **cv act** ***library*** über Handles können Sie die Library direkt über die Interface-Klassen ansprechen. Diese Interface-Klassen sind rein abstrakte Basisklassen, die die Schnittstelle zu Objekten über einen Satz virtueller Methoden definieren. Durch den Aufruf einer Fabrikmethode erhalten Sie einen Zeiger auf ein Objekt, das eine solche Schnittstelle zur Verfügung stellt (die Fabrikmethode ist ein Standardentwurfsmuster).

Im ersten Unterabschnitt werden die Interface-Klassen erklärt und ihr Zusammenhang anhand von Diagrammen veranschaulicht. In den beiden weiteren Abschnitten wird dann jeweils an Beispielen gezeigt, wie man ein anderes -statt des standardmäßig vorgesehenen- oder ein neues -nicht in der Bibliothek enthaltenes- Verfahren einbinden kann.

6. 1 Das Objektmodell der **cv act *library***

oder

Der große Zusammenhang

Die beiden wichtigsten konzeptionellen Objekte der **cv act *library*** sind `Algorithm` und `Key`, die Sie in Kapitel 5 schon kennengelernt haben. Man kann sich ein `Algorithm`-Objekt als eine Verarbeitungspipeline vorstellen, bei der die einzelnen Verarbeitungsschritte jeweils flexibel kombiniert werden können. So ist z.B. ein symmetrisches Verschlüsselungsverfahren eine Pipeline bestehend aus einer Betriebsart und einer Blockchiffre.

Ein `Key`-Objekt stellt einen Schlüssel oder ein Schlüsselpaar dar. Dabei ist ein Schlüssel mehr als der kryptographische Schlüssel eines Verfahrens, er enthält alle Schlüsselinformationen, wie beispielsweise der verwendete Betriebsmodus bei Blockchiffren oder die *domain parameter* bei asymmetrischen Verfahren. Das `Key`-Objekt legt das kryptographische Verfahren fest.

Zu diesen konzeptionellen Objekten gibt es jeweils zwei Ausprägungen, und zwar das Interface und das Handle. Das Interface ist eine abstrakte Basisklasse, welche die Methoden definiert, die alle zugehörigen Objekte besitzen. Das Handle ist eine konkrete Klasse, welche den Zugriff auf die Interfaces handhabt. Sie enthalten einen Zeiger auf ein Objekt, an das die Methodenaufrufe weitergeleitet werden und übernehmen die Erzeugung und Zerstörung dieses Objekts.

Ein Designaspekt der Library ist, dass die Schnittstellen offen gehalten sind, damit es Ihnen als Anwender möglich ist, neue Funktionalitäten der **cv act *library*** hinzuzufügen. Beispielsweise lassen sich Hardware-Zufallszahlengeneratoren so in die Bibliothek einbinden, dass sie von den vorhandenen Mechanismen verwendet werden.

6. 1. 1 Die **`Key`**-Objekte:

Zur besseren Übersichtlichkeit werden nur die Methodensignaturen aufgeführt, die für das Verständnis des Objektmodells notwendig sind; alle anderen können Sie dem Referenzkapitel entnehmen.

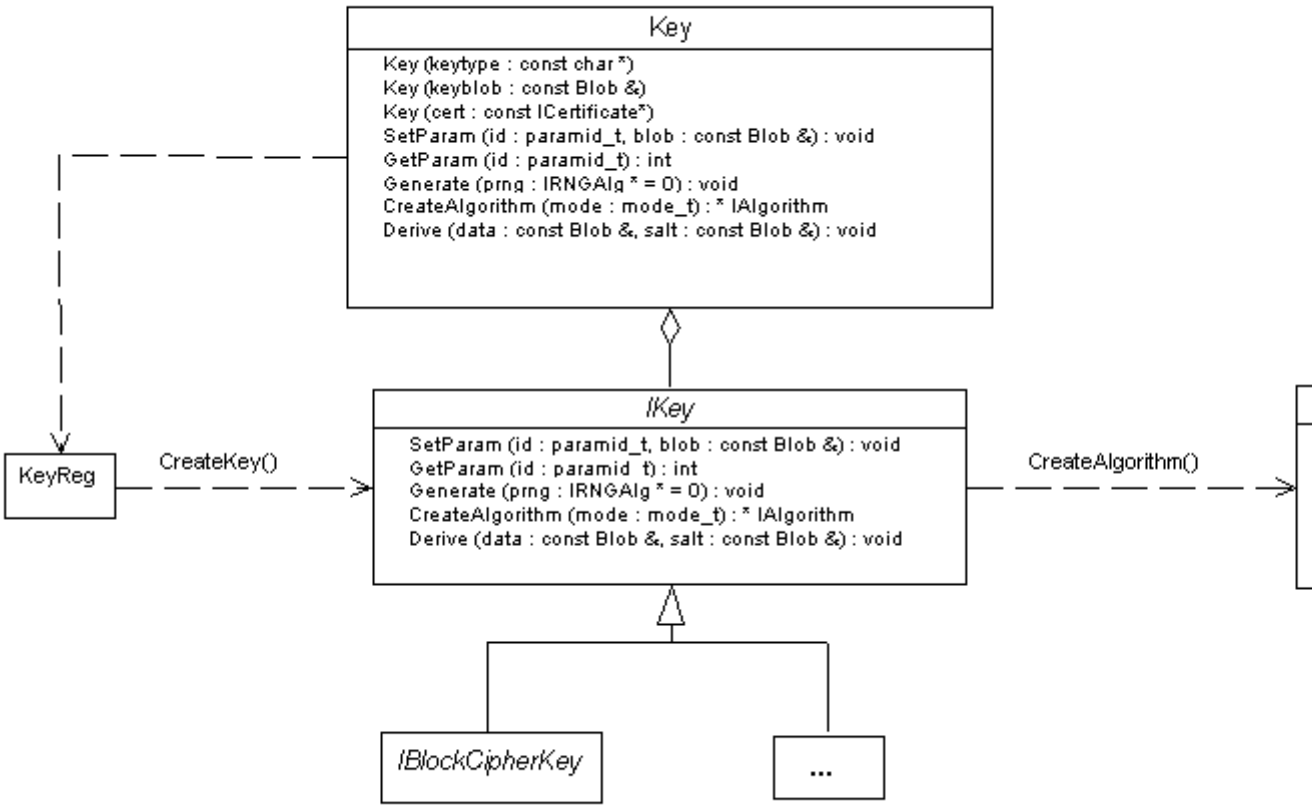


Abbildung: Objektmodell zu `Key`

Wie aus dem obigen Diagramm hervorgeht, stellt die Klasse `Key` das Handle zu Objekten der Interface-Klasse `IKey` dar. Dabei ist `IKey` die abstrakte Basisklasse aller `Key`-Interface-Klassen, wie z.B. `IBlockCipherKey`. Mit `SetParam()` kann man Parameter, wie den zu verwendenden Betriebsmodus setzen und mit `GetParam()` wieder auslesen; `Generate()` generiert einen zufälligen Schlüssel oder ein zufälliges Schlüsselpaar. `Derive()` leitet einen Schlüssel aus dem übergebenen Schlüsselmaterial her, und `CreateAlgorithm()` erzeugt die `IAlgorithm`-Objekte.

`Key` benutzt die Registry-Klasse `KeyReg`, um Instanzen von konkreten Klassen zu erzeugen, die von `IKey` abgeleitet sind. Dazu wird die Fabrikmethode `CreateKey()` aufgerufen, die mittels einer internen Tabelle eine Instanz der gewünschten, konkreten Klasse erzeugt. Sie können mit der Klasse `KeyReg` Schlüssel erzeugen, deren Typ erst zur Laufzeit bekannt wird, indem man `CreateKey()` den Bezeichner übergibt. Dies gilt auch für Schlüsselobjekte, die Sie selbst implementiert haben. Dazu müssen Sie nur eine Klasse erstellen, die das `IKey`-Interface erbt und der Klasse `KeyReg` bekanntgemacht werden (Details entnehmen Sie bitte dem Abschnitt 6. 3).

6. 1. 2 Die **`Algorithm`**-Objekte

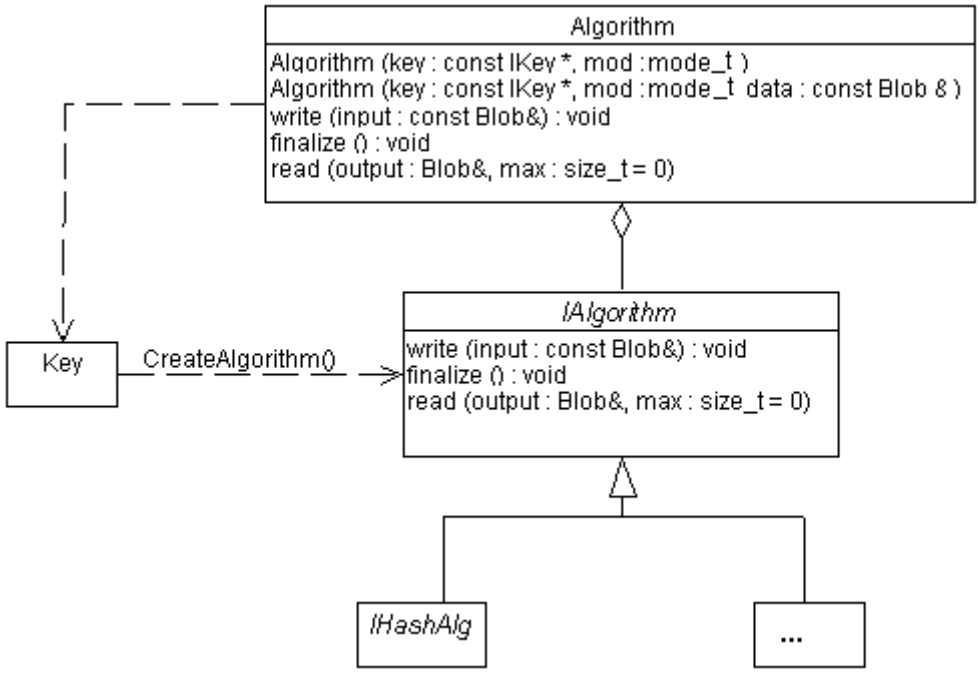


Abbildung: Objektmodell zu `Algorithm`

Wie für die Klasse `Key` existiert auch für Algorithmen eine polymorphe Struktur `IAlgorithm`. Die drei abstrakten Methode `Write()`, `Finalize()` und `Read()` in `IAlgorithm` ermöglichen die Datenverarbeitung im Strommodus. Mit `Write()` werden also Daten dem Algorithmus zur Verarbeitung übergeben und kann beliebig oft aufgerufen werden. Die Ausgabedaten werden solange in dem Objekt zwischengespeichert, bis sie mit `Read()` nach dem FIFO-Prinzip (First In First Out) ausgelesen werden. Wenn keine weiteren Eingabedaten folgen, muss dies mit `Finalize()` dem Algorithmus mitgeteilt werden. Danach kann `Read()` beliebig oft aufgerufen werden, um alle Ausgabedaten auszulesen. Man sollte `Write()` und `Read()` paarweise aufrufen, um nicht zu viel dynamischen Speicher zu belegen. Das folgende Zustandsdiagramm soll diesen Sachverhalt noch mal verdeutlichen:

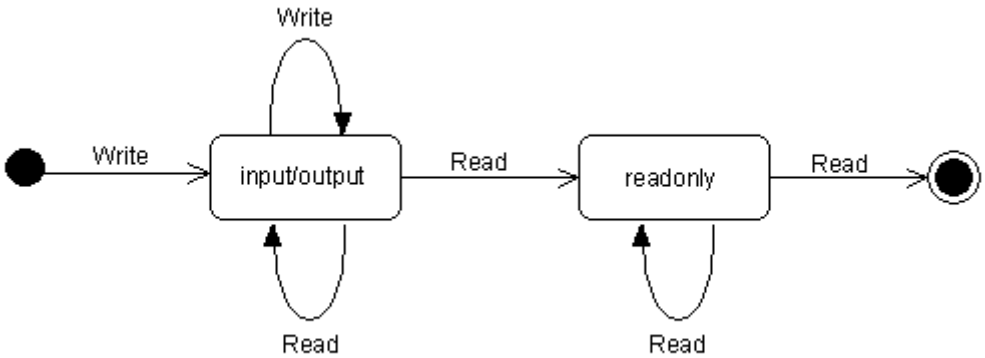


Abbildung: Zustandsdiagramm zu `IAlgorithm`

6. 1. 3 Sonstige Objekte:

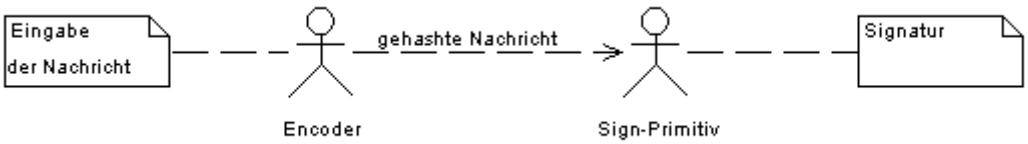
Zwei weitere Objekte mit der Handle/Interface-Ausprägung sind `CRL` und `Certificate`. Eine Beschreibung finden Sie in Kapitel 7 unter den zugehörigen Interfaces.

Alle weiteren Objekte in der **cv act *library*** werden nicht direkt vom Anwender benutzt, sondern stellen Hilfsobjekte dar, die in verschiedenen Key- bzw. Algorithm-Objekten aggregiert sind. Eine vollständige Liste aller Objekte können Sie Kapitel 5. 4 entnehmen.

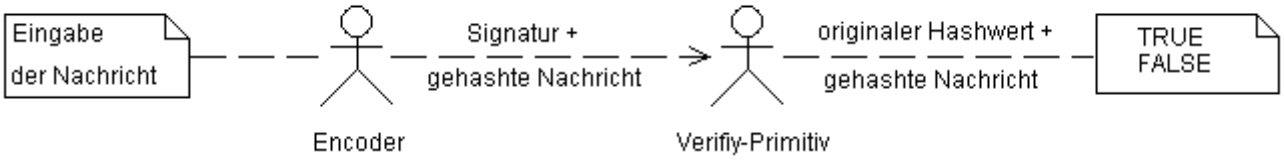
6. 2 So passen Sie die **cv act *library*** an

In diesem Abschnitt wird erklärt, wie man standardmäßig vorgesehene Primitiva durch andere Algorithmen anhand der Interfaces ersetzen kann.

Schauen wir uns das Beispiel der digitalen Signatur an:



– d.h. eine Nachricht wird vor dem eigentlichen Signieren mittels eines Encoders kodiert (genaue Vorgehensweise siehe Kapitel 1. 3. 4) und erst danach anhand eines Signaturprimitivas digital signiert. Das Verifizieren hat dann folgende Form:



Beim Verifizieren wird ebenfalls die Nachricht anhand desselben Encoders gehasht, damit dieser Hashwert mit der empfangenen Signatur mit Hilfe eines Verifizierungsprimitivas verglichen werden kann. Im folgenden Beispiel wird demonstriert, wie man eine Verarbeitungspipeline verändert, indem der Encoder ausgetauscht wird.

Beispiel:

Der Signaturalgorithmus EC-DSA benutzt die Kodierungsmethode EMSA1 (*Encoding Message Signature with Appendix*), die wiederum standartmäßig den Hashalgorithmus SHA-1 verwendet. An diesem Beispiel soll nun gezeigt werden, wie man an Stelle von SHA-1 den Algorithmus RIPEMD-160 benutzen kann.

```
void SignVerify()
{
        Blob message ( "ABCDEFGHABCDEFGHABC" );
        Blob signature;

        IKey *key = KeyReg::CreateKey("ECDSA"); //key wird über die
                                        //angegebene Fabrikmethode erzeugt
        key.Generate();

        //downcast vom Interface IKey nach ISignatureKey, um den
        //Zugriff auf die Methode GetEncoder zu ermoeglichen
        act::ISignatureKey *sigkey= dynamic_cast < act::ISignatureKey  *>
(key);

        //wenn der cast gelungen ist, was bei EC-DSA immer der Fall
        //sein sollte, kann mit GetEncoder ein Pointer auf das
        //aggregierte IEMSAAlg-Objekt beschafft werden. Danach laesst
        //sich der verwendete Hashalgorithmus in IEMSAAlg ersetzen.
```

```
        if( sigkey!=0 )
        {
                sigkey->GetEncoder()->SetParam(act::HASH, "RIPEMD160");
        }

        IAlgorithm *key_alg = key->CreateAlgorithm(act::SIGN);

        for (int i=0; i<100; i++)
                key_alg->Write(message);

        key_alg->Finalize();
        key_alg->Read(signature);

        delete key_alg;

        key_alg = key->CreateAlgorithm(act::VERIFY, signature);
        for (i=0; i < 100; i++)
                key_alg->Write(message);
        key_alg->Finalize();

        cout << endl << "Signatur korrekt: " << boolalpha << (key_alg-
>GetStatus()==act::SIGNATURE_OK);
        delete key_alg;
        delete key;
}
```

6. 3 So erweitern Sie die **cv act *library***

In diesem Unterkapitel wird erklärt, wie man neue Algorithmen durch eigene Implementierungen zur **cv act *library*** hinzufügen kann, so dass diese durch neue Funktionalitäten erweitert wird. So gehören beispielsweise der symmetrische Verschlüsselungsalgorithmus IDEA nicht zum Auslieferungsumfang dieser Library, da sie für gewerbliche Zwecke Restriktionen durch Patente unterworfen sind.

Anhand des Beispiels von IDEA wird mit Quellcode und der entsprechenden Headerdatei nun aufgezeigt, wie man diesen Algorithmus in diese Bibliothek einbinden kann.

Aufgrund der modularen Architektur der **cv act *library*** hat die Headerdatei von IDEA die gleiche Form wie die von alle anderen symmetrischen Algorithmen, da alle von der Klasse `IBlockCipher` abgeleitet sind. Also ist auch die Klasse `Idea` von dieser Klasse abgeleitet, so dass alle Funktionen in der Referenzliste in Punkt 7. 3 nachgeschlagen werden können. Im privaten Teil kann man bei Bedarf eigene Membervariablen und Methoden deklarieren. In unserem Beispiel wurde nur zusätzlich der geheime Schlüssel `mUserKey` deklariert.

```
// ----------------------------------------------------------------
// Dateiname           :Idea.h
// Paket               :blockcipher
//
// ----------------------------------------------------------------

#ifndef ACT_IDEA_H
#define ACT_IDEA_H

#include "IBlockCipher.h"

namespace act
{
        IBlockCipher *CreateIdea();


        class Idea:public IBlockCipher
        {
        public:
        Idea();

                virtual IBlockCipher* Clone() const;
                virtual void Import(const Blob& InData);
                virtual void Export(Blob& OutData) const;

                virtual void SetRawKey(const Blob& key);
                virtual void GetRawKey(Blob& key) const;

                virtual size_t GetKeySize(size_t keysize=0) const;
                virtual size_t GetMinKeySize() const;
                virtual size_t GetMaxKeySize() const;
                virtual size_t GetNextKeySize(size_t prevsize) const;

                virtual void Encrypt(const byte* in, byte* out) const;
                virtual void Decrypt(const byte* in, byte* out) const;

                virtual size_t GetBlockSize() const;

                virtual void* GetCreatePointer() const;

                virtual ~Idea();
```

```
                private:
                        Blob mUserKey;
        };

} // namespace act
#endif
```

Der Sourcecode Idea.cpp ist angelehnt an die Standardimplementierung von Alain Beuchat und Daniel Zimmermann:

```
// ------------------------------------------------------------------
// Dateiname        : Idea.cpp
// Paket            : blockcipher
//
//      Beschreibung
// Zur Implementierung dieser Klasse wurden die Funktionen von
// WWW.ASCOM.COM/INFOSEC benutzt. Aus Gründen der Übersichtlichkeit
// wurde auf Code-Optimierung und das Überschreiben freigegebenen
// Speichers verzichtet. Eine vollständige Implementierung dieser
// Wrapper-Klasse finden Sie auf der CD.
//
// ------------------------------------------------------------------

#include "Ideaplus.h" // Implementation from WWW.ASCOM.COM/INFOSEC

#include "Idea.h"

#define  KEYSIZE              16      // bytes
#define  BLOCKSIZE            8       // bytes

namespace act
        {

        // Diese globale Funktion wird für die Registration in der
        // BlockCipherReg benötigt.
        IBlockCipher *CreateIdea()
        {
                return new Idea;
        }

        Idea::Idea()
        {
        }

        Idea::~Idea()
        {
        }

        IBlockCipher* Idea::Clone() const
        {
                return new Idea(*this);
        }

        void Idea::Import(const Blob& indata)
        {
                SetRawKey(indata);
        }

        void Idea::Export(Blob& outdata) const
        {
                GetRawKey(outdata);
        }
```

```
void Idea::SetRawKey(const Blob& userkey)
{
        if(userkey.size() != KEYSIZE)
                throw InvalidKeyException("wrong key
                                        size","Idea::SetRawKey");
        mUserKey=userkey;
}

void Idea::GetRawKey(Blob& userkey) const
{
        userkey=mUserKey;
}


size_t Idea::GetKeySize(size_t keysize) const
{
        return KEYSIZE;
}

size_t Idea::GetMinKeySize() const
{
        return KEYSIZE;
}

size_t Idea::GetMaxKeySize() const
{
        return KEYSIZE;
}

size_t Idea::GetNextKeySize(size_t prevsize) const
{
        return KEYSIZE;
}

size_t Idea::GetBlockSize() const
{
        return BLOCKSIZE;
}

void* Idea::GetCreatePointer() const
{
        return CreateIdea;
}

void Idea::Encrypt(const byte* in, byte* out) const
{

        idea_key_t key;
        idea_subkeys_t enkey;
        idea_block_t in16, out16;
        int i;

        // Datentyp von Key konvertieren
        for (i=0;i < KEYSIZE/2;i++) {
            key[i]=(mUserKey[i*2]<<8)|(mUserKey[i*2+1]);
        }

        // Die Funktion von Alain Beuchat/Daniel
        // Zimmermann aufrufen
        idea_encrypt_subkeys(key,enkey);

        // Inputdaten konvertieren
```

```
                for (i = 0;i < BLOCKSIZE/2;i++) {
                        in16[i]=(in[i*2]<<8) | (in[i*2+1]);
                }

                // Die Funktion von Alain Beuchat/Daniel
                // Zimmermann aufrufen
                idea_cipher(in16,out16,enkey);

                // Outputdaten konvertieren
                for(i=0;i < BLOCKSIZE/2;i++) {
                        out[i*2+1] = byte(out16[i]&0xff);
                        out[i*2] = byte(out16[i]>>8);
                }
        }

        void Idea::Decrypt(const byte* in, byte* out) const
        {

                idea_key_t key;
                idea_subkeys_t enkey;
                idea_subkeys_t dekey;
                idea_block_t in16, out16;
                int i;

                // Datentyp von Key konvertieren
                for (i=0;i < KEYSIZE/2;i++) {
                    key[i]=(mUserKey[i*2]<<8)|(mUserKey[i*2+1]);
                }

                // Die Funktion von Alain Beuchat/Daniel
                // Zimmermann aufrufen
                idea_encrypt_subkeys(key,enkey);
                idea_decrypt_subkeys(enkey,dekey);

                // Inputdaten konvertieren
                for (i = 0;i < BLOCKSIZE/2;i++) {
                        in16[i]=(in[i*2]<<8) | (in[i*2+1]);
                }

                // Die Funktion von Alain Beuchat/Daniel
                // Zimmermann aufrufen
                idea_cipher(in16,out16,dekey);

                // Outputdaten konvertieren
                for(i=0;i < BLOCKSIZE/2;i++) {
                        out[i*2+1] = byte(out16[i]&0xff);
                        out[i*2] = byte(out16[i]>>8);
                }
        }

} // namespace act
```

Der erste Teil der Implementierung besteht aus den Deklarationen der benötigten Funktionen, deren Bedeutung Sie wieder in der Referenzliste in Kapitel 7 nachlesen können. Danach werden die Funktionen `Encrypt` und `Decrypt` aus der Implementierung von Beuchat/Zimmermann aufgerufen. Um diese ansprechen zu können, müssen die Eingabedaten vorher und die Ausgabedaten hinterher konvertiert werden.

Um der Library die Erweiterung mitzuteilen, muss Idea als zusätzlicher Schritt der Klasse `BlockCipherReg` hinzugefügt werden. Dies geschieht durch den Methodenaufruf:
`BlockCipherReg::Insert("Idea",CreateIdea);`

Zusätzlich zu den Blockchiffres gibt es „Registries“ für andere Objektklassen in der **cv act *library***, und zwar im Folgenden:

JCApplet:	Für die Registrierung von Applet-AIDs von JavaCards für JavaCardOS
SCardOS:	Für die Handhabung einzelner Kartenbetriebssysteme
SCardToken:	Für die Handhabung einzelner Profilarten, auch unterschiedlicher Profiltypen auf Karten desselben Betriebssystems
SlotMonitor:	Stellt einen Eventsender für ein einzelnes Subsystem
Subsystem:	Ist die Schnittstelle zwischen den Karten-OS-Klassen und der Kommunikationsschicht (CSP, PKCS11, PCSC, ...) zum Kartenleser und Token
TokenExtension:	Stellt Erweiterungen zu ‚Standard’-Profilen dar.
Certificate:	Beinhaltet Zertifikats- und CRL-Klassen
EMSA:	Für die Implementierung neuer EMSA-Algorithmen
Hash:	Algorithmen zum Generieren von Hash-Werten
Derivator:	Algorithmen für die Schlüsselableitung
Padding:	Methoden um Auffüllen (und Zurechtschneiden) von Datenblöcken
StreamCipher:	Stromchiffre-Methoden
PKCS15Behavior:	Implementationen des PKCS15-Standards auf unterschiedlichen Kartenbetriebssystemen und -Profilen

Die entsprechenden Interfaces sind in den Include-Dateien mit den Namen ‚act<Name>Reg.h’ (z.B. actSCardOSReg.h für SCardOS) verfügbar. Die Include-Dateien mit den Namen ‚act<Name>Kit.h’ beinhalten u.a. die Prototypen der Klassen und Funktionen, die bereits durch `act::Init()` registriert werden.

Kapitel 7 Referenzliste der **cv act** *library*

In diesem Kapitel werden alle, für die Verwendung der **cv act** ***library*** notwendigen Funktionalitäten in einer Referenz aufgelistet. Zuerst wird der verwendete Datentyp beschrieben und dann die Handles für die High-Level-Programmierung. Danach werden die Interfaces erklärt, die bei der Low-Level-Programmierung benutzt werden können oder von denen beim Erstellen eigener Klassen abgeleitet wird, wie dies zum Erweitern der Bibliothek notwendig ist.

7. 1. Konkrete Datentypen

Blob

Date

X.509Extension

7. 1. 1 Blob

Der konkrete Datentyp Blob (Binary Large OBjekt) wird als universeller Typ für beliebige Daten verwendet. Diese Klasse `Blob` verhält sich wie `std::vector<unsigned char>` aus der C++ Standardbibliothek mit dem Unterschied, dass Blob den belegten Speicher vor der Freigabe mit 0 überschreibt. Dadurch werden sensitive Informationen, wie geheime, private Schlüssel, vor unberechtigten Zugriffen nach Freigabe des Speichers geschützt. Dieses ist notwendig, da solche sensitiven Informationen nach deren Benutzung vollständig aus dem System entfernt werden müssen, weil sonst andere Programme darauf zugreifen können. Desweiteren ist gewährleistet, dass der von einem Blob verwaltete Speicher zusammenhängend ist.

7. 1. 2 Date

Diese Datumsklasse wird bei der Benutzung von Zertifikaten verwendet. Die Methoden dieser Klasse finden Sie in der Headerdatei "`actDate.h`" auf der CD. Besonders hervorzuheben sind folgende Methoden:

`Date()`: Defaultkonstruktor setzt das aktuelle Datum und die Zeit des Aufrufs.

`Date(const Blob&)`: setzt Datum und Zeit auf den im Blob übergebenen ASN.1-DER-kodierten Wert. Solche Blobs werden in X.509-Zertifikaten und -CRLs verwendet.

`Blob Encode()const`: konvertiert das Datum (mit Zeit) in den übergebenen ASN.1-DER-kodierten Wert.

`operator Blob () const`: ruft Encode auf.

7. 1. 3 X509Extension

Dieser Datentyp repräsentiert Extensions in X.509-Zertifikaten und -CRLs.

```
class X509Extension
{
public:
        X509Extension();
        X509Extension(const Blob &oid, const Blob &value ,
                      bool critflag);
        X509Extension(const char* oid, const Blob &value ,
                      bool critflag);
        void GetOID(Blob &oid) const;
        void GetOIDString(Blob &oid) const;
        void SetOID(const Blob &oid);
        void SetOIDString(const char* oid);
        void GetValue(Blob &value) const;
        void SetValue(const Blob &value);
        bool IsCritical() const;
        void SetCritical(bool crit=true);
        void GetDER(Blob &der) const;
        void SetDER(const Blob &der);
};
```

Mit `GetOID` und `SetOID` lässt sich der ASN.1-DER-kodierte Wert des Object Identifier-Feldes modifizieren (der kodierte Wert ohne Typ und Längenkodierung). Mit `SetOIDString` und `GetOIDString` lässt sich die OID String-kodiert modifizieren, wie beispielsweise `SetOIDString("2.5.29.19")`, um den Identifier für `basicConstraints` zu setzen.

Mit `GetDER` und `SetDER` lässt sich die gesamte Extension ASN.1-DER de/kodieren.

7. 2 Handles

Der Zugriff auf die Funktionalitäten der **cv act *library*** wird über Handles bereitgestellt, die die entsprechenden Schnittstellen zur Verfügung stellen. Im folgenden werden die Handles `Algorithm`, `Certificate`, `CRL` und `Key` im Detail beschrieben.

7. 2. 1 Das Handle **`Algorithm`**

Die Klasse `Algorithm` erzeugt den Algorithmus zum Verschlüsseln oder Signieren, der vorher durch die Klasse `Key` festgelegt wurde. Benötigte Funktionalitäten, wie Verifizierungsalgorithmen oder Hashalgorithmen können ebenfalls über dieses Handle erzeugt werden.

```
class Algorithm {
public:
        Algorithm(IAlgorithm *alg);
        Algorithm(const IKey *key, mode_t mode);
        Algorithm(const IKey *key, mode_t mode, const Blob &data);
        Algorithm(const char* name);

        void Write(const Blob& indata);
        void Write(const byte* indata, size_t insize);

        void Finalize();

        size_t Read(Blob& outdata, size_t max = 0);
        size_t Read(byte* outbuffer, size_t buffersize );

        size_t GetAvailableSize() const ;
        status_t GetStatus() const ;

        Algorithm& operator<<(const Blob& indata)
        Algorithm& operator<<(Algorithm& (*Manipulator)
                                (Algorithm &alg))
        Algorithm& operator>>(Blob& outdata)

        IAlgorithm *GetPointer();
        const IAlgorithm *GetPointer() const;
        IAlgorithm *ReleasePointer();

        ~Algorithm();
};
```

`Algorithm(IAlgorithm *alg)`

Erzeugt ein Algorithm-Handle, indem der Besitz von `alg` übernommen wird. Die Exception `NoSuchAlgorithmException` wird geworfen, falls `alg` den Wert `0` hat.

`Algorithm(const IKey *key, mode_t mode)`

erzeugt einen Algorithmus zu einem Key. Dabei kann `mode` in Abhängigkeit von dem Typ `Key` verschiedene Werte annehmen.

`Algorithm(const IKey *key, mode_t mod, const Blob &data)`

erzeugt einen Algorithmus zu einem Key, der zusätzlich mit dem Blob `data` initialisiert wird. Dies ist z.B. beim Erzeugen Algorithmus zur Verfizierung einer Signatur notwendig.

`Algorithm(const char* name)`

erzeugt einen Hashalgorithmus.

`void Write(const Blob& indata)`

übergibt Daten an den Algorithmus. In dem Fall einer Hashfunktion werden die Daten sofort gehasht und nachdem `Finalize()` aufgerufen wurde, kann mit `Read()` der Hashwert ausgelesen werden. Im Falle eines symmetrischen Verfahrens können `Write()` und `Read()` abwechselnd aufgerufen werden, um größere Datenmengen bearbeiten zu können.

`void Finalize()`

teilt dem Algorithmus mit, dass keine weiteren Eingabedaten mehr folgen. Z.B. wird nach Aufruf dieser Methode bei einem Hashalgorithmus der endgültige Hashwert berechnet und steht von da an zum Abholen mittels der Methode `Read()` bereit

`size_t GetAvailableSize() const`

gibt die Datenmenge in Bytes an, die zum Lesen mit `Read()` zur Verfügung stehen.

`size_t Read(Blob& outdata,size_t Max = 0)`

schreibt maximal `Max` Byte Daten in `outdata`, falls `Max` > 0. Sonst wird der gesamte zur Verfügung stehende Output in `outdata` geschrieben.

`status_t GetStatus() const`

gibt den Status des Objekts an. Mögliche Rückgabewerte sind: `READY`, `SIGNATURE_OK` oder `IS_FINALIZED`.

`Algorithm &operator<<(const Blob& indata)`

übergibt Daten mit dem Stream-In-Operator, d.h. es wird `Write(indata)` aufgerufen.

`Algorithm &operator<<(Algorithm &(*Manipulator)(Algorithm &Alg))`

Stream-In-Operator für Manipulatoren. In der Library ist der Manipulator `final` enthalten, der zum Aufruf der Methode `Finalize()` führt.

`Algorithm &operator>>(Blob& outdata)`

liest alle zur Verfügung stehenden Daten aus.

`IAlgorithm *GetPointer()` `oder` **`const IAlgorithm *GetPointer() const`**

gibt den intern gespeicherten `IAlgorithm`-Pointer zurück. Die Handle-Klasse `Algorithm` behält aber den Besitz des Pointers.

`IAlgorithm *ReleasePointer()`

gibt den intern gespeicherten `IAlgorithm`-Pointer zurück und den Besitz frei.

7. 2. 2 Das Handle **`Certificate`**

Mit Hilfe dieser Klasse kann man Zertifikate ansprechen, also z.B. den darin enthaltenen öffentlichen Schlüssel oder dessen Gültigkeit.

```
class Certificate
{
public:
        Certificate(const Certificate& cert);
        Certificate(const char* certtype);
        Certificate(const char* certtype, const Blob &certblob);

        void Import(const Blob &certblob);
        void Export(Blob &certblob) const;
        void SetParam(paramid_t id, const Blob &blob);
        void SetParam(paramid_t id, int val);
        void SetParam(paramid_t id, const char *cstr);
        int GetParam(paramid_t id) const;
        void GetParam(paramid_t id, Blob& blob) const;

        void SetPublicKey(const IKey* pubkey);
        IKey* CreatePublicKey() const;
        void Sign(const IKey* privkey);
        void Verify(const IKey* pubkey) const;

        ICertificate* GetPointer();
        const ICertificate* GetPointer() const;
        operator ICertificate* ();
        operator const ICertificate* () const;

        ICertificate* ReleasePointer();

        Certificate& operator= (const Certificate& cert);

        ~Certificate();
};
```

`Certificate(const Certificate& cert)`

Kopiekonstruktor.

`Certificate(const char* certtype)`

erzeugt ein Zertifikat vom angegebenen Typ. In der Library ist der Typ "`X509`" enthalten.

`Certificate(const char* certtype, const Blob &certblob)`

erzeugt ein Zertifikat vom Typ `certtype` und importiert den Inhalt des Zertifikats aus `certblob`.

`void Import(const Blob &certblob)`

importiert ein Zertifikat aus einem Blob.

void Export(Blob &certblob) const

exportiert ein Zertifikat in einen Blob.

void SetParam(paramid_t id, const Blob &blob) oder **void SetParam(paramid_t id, int val)** oder **void SetParam(paramid_t id, const char *cstr)**

setzt Zertifikatsparameter.

int GetParam(paramid_t id) const oder **void GetParam(paramid_t id, Blob& blob) const**

liest Zertifikatsparameter aus.

void SetPublicKey(const IKey* pubkey)

fügt den öffentlichen Schlüssel `pubkey` in das Zertifikat ein.

IKey* CreatePublicKey() const

erzeugt ein Schlüsselobjekt mit dem öffentlichen Schlüssel aus dem Zertifikat.

void Sign(const IKey* privkey)

signiert das Zertifikat mit dem privaten Schlüssel `privkey`.

void Verify(IKey* pubkey) const

überprüft, ob das Zertifikat mit dem zu `pubkey` gehörendem, privaten Schlüssel signiert worden ist. Wenn dies nicht der Fall ist, wird eine Exception geworfen.

ICertificate* GetPointer() oder **const ICertificate* GetPointer() const**

gibt den intern gespeicherten `ICertificate`-Pointer zurück. Die Handle-Klasse `Certificate` behält aber den Besitz des Pointers.

operator ICertificate* () oder **operator const ICertificate* () const**

Diese Konvertierungsoperatoren führen zu einem impliziten Aufruf von `GetPointer()`.

ICertificate* ReleasePointer()

gibt den intern gespeicherten `ICertificate`-Pointer zurück und den Besitz frei.

Certificate& operator= (const Certificate& cert)

Zuweisungsoperator.

7. 2. 3 Das Handle **CRL**

Mit Hilfe dieser Klasse kann man eine *Certificate Revocation List* ansprechen.

```
class CRL
{
public:
        CRL(const CRL& crl);
        CRL(const char* crltype);
        CRL(const char* crltype,const Blob &crlblob);
        void Import(const Blob &crlblob);
        void Export(Blob &crlblob) const;
        void SetParam(paramid_t id, const Blob &blob);
        void SetParam(paramid_t id, int val);
        void SetParam(paramid_t id, const char *cstr);
        int GetParam(paramid_t id) const;
        void GetParam(paramid_t id, Blob& blob) const;

        void Revoke(const ICertificate* cert);
        bool IsRevoked( const ICertificate* cert) const;
        void RemoveRevokation(const ICertificate* cert);
        void Sign( const IKey* privkey);
        void Verify( const IKey* pubkey) const;

        ICRL* GetPointer();
        const ICRL* GetPointer() const;
        operator ICRL* ();
        operator const ICRL* () const;

        ICRL* ReleasePointer();

        CRL& operator= (const CRL& crl);
        ~CRL();
};
```

CRL(const CRL& crl)

Kopiekonstruktor.

CRL(const char* crltype)

erzeugt eine CRL vom angegebenen Typ. In der Library ist der Typ "`X509CRL`" enthalten.

CRL(const char* crltype,const Blob &crlblob)

erzeugt eine CRL vom Typ `crltype` und importiert den Inhalt der CRL aus `crlblob`.

void Import(const Blob &crlblob)

importiert eine CRL aus einem Blob.

void Export(Blob & crlblob) const

exportiert eine CRL in einen Blob.

`void SetParam(paramid_t id, const Blob &blob)` oder **`void SetParam(paramid_t id, int val)`** oder **`void SetParam(paramid_t id, const char *cstr)`**

setzt CRL-Parameter.

`int GetParam(paramid_t id) const` oder **`void GetParam(paramid_t id, Blob& blob) const`**

liest CRL-Parameter aus.

`void Revoke(const ICertificate* cert)`

setzt ein Zertifikat auf die CRL.

`bool IsRevoked( const ICertificate* Cert) const`

überprüft, ob ein Zertifikat auf der CRL steht.

`void RemoveRevokation(const ICertificate* cert)`

entfernt ein Zertifikat von der CRL.

`void Sign( const IKey* privkey)`

signiert die CRL mit dem privaten Schlüssel `privkey`.

`void Verify( const IKey* pubkey) const`

überprüft, ob die CRL mit dem zu `pubkey` gehörendem, privaten Schlüssel signiert worden ist. Wenn dies nicht der Fall ist, wird eine Exception geworfen.

`ICRL* GetPointer()` oder **`const ICRL* GetPointer() const`**

gibt den intern gespeicherten ICRL-Pointer zurück. Die Handle-Klasse `CRL` behält aber den Besitz des Pointers.

`operator ICRL* ()` oder **`operator const ICRL* () const`**

Diese Konvertierungsoperatoren führen zu einem impliziten Aufruf von `GetPointer()`.

`ICRL* ReleasePointer()`

gibt den intern gespeicherten ICRL-Pointer zurück und den Besitz frei.

`CRL& operator= (const CRL& crl)`

Zuweisungsoperator.

7. 2. 4 Das Handle **`Key`**

Mit Hilfe der Handle-Klasse `Key` werden Schlüsselinformationen verwaltet. Durch dieses Handle kann man trotz unterschiedlicher Anforderungen einheitlich mit diesen Schlüsselinformationen umgehen.

```
class Key {
public:
        Key();
        Key(const Key& key);
        Key(IKey* keyptr);
        Key(const char *keytype);
        Key(const Blob &keyblob);
        Key(const ICertificate* cert);
        void SetParam(paramid_t id,const Blob &blob);
        void SetParam(paramid_t id,size_t len);
        void SetParam(paramid_t id,const char *cstr);
        int GetParam(paramid_t id) const;
        void GetParam(paramid_t id, Blob &blob) const;
        void Generate(IRNGAlg *prng=0);
        void Derive(const Blob &data, const Blob &salt=Blob() );
        void Import(const Blob &keyblob);
        void Export(Blob &keyblob,export_t type=DEFAULT) const;
        IAlgorithm* CreateAlgorithm( mode_t mode) const;
        IAlgorithm* CreateAlgorithm( mode_t mode, const Blob &data) const;

        IKey* GetPointer();
        const IKey* GetPointer() const;
        operator IKey* ();
        operator const IKey* () const;

        IKey* ReleasePointer();

        Key& operator= (const Key& key);

        ~Key();
};
```

`Key(const char *keytype)`

erzeugt ein Key-Objekt, wobei `keytype` einer der folgenden C-Strings sein kann: "`BlockCipher`", "`DH`", "`DSA`", "`ECDSA`", "`RSA`". Die weiteren Möglichkeiten siehe in Kapitel 5.4.

`Key(const Blob &keyblob)`

Konstruktor zum Erzeugen eines Key-Objekts, wobei die Key-Daten aus `keyblob` importiert werden.

`Key(const ICertificate* cert)`

Konstruktor zum Erzeugen eines Key-Objekts, wobei die Key-Daten aus `cert` importiert werden.

`void SetParam(paramid_t id,const Blob &blob)` oder **`void SetParam(paramid_t id,int val)`** oder **`void SetParam(paramid_t id,const char *cstr)`**

setzt Key-Attribute. Zulässige Werte sind abhängig vom verwendeten Key-Objekt. Eine `InvalidKeyException` wird geworfen, falls `id` nicht unterstützt wird.

int GetParam(paramid_t id) const oder **void GetParam(paramid_t id, Blob &blob) const**

liest Key-Attribute aus. Zulässige Werte entsprechen denen von `SetParam()`.

void Generate(IRNGAlg *prng=0)

erzeugt einen zufälligen Schlüssel. Bei `prng!=0` wird der Zufallszahlengenerator `prng` benutzt. Ansonsten verwendet `Generate()` den Default-Zufallszahlengenerator.Außerdem werden *domain parameter* für asymmetrische Schlüssel mit dieser Methode erzeugt, falls diese nicht vorher importiert oder gesetzt wurden. Die Schlüsselgröße kann abhängig vom benutzten Schlüsselobjekt vorher mit `SetParam()()` gesetzt werden (siehe Kapitel 5.4.1).

void Derive(const Blob &data, const Blob &salt=Blob())

leitet Schlüssel von `data` ab. Es kann ein optionaler `salt` -Parameter übergeben werden. Diese Methode findet bei symmetrischen Schlüsseln und bei Schlüsselaustauschverfahren Verwendung.

void Import(const Blob &keyblob)

importiert einen Schlüssel und alle dazugehörigen Parameter und aggregierte Objekte aus `keyblob`.

void Export(Blob &keyblob,export_t type=DEFAULT) const

exportiert einen Schlüssel und alle dazugehörigen Parameter und aggregierte Objekte in `keyblob`. Der Parameter `type` kann abhängig vom verwendeten Key-Objekt variieren: für symmetrische Schlüssel verwendet man den Wert SECRET, für asymmetrische Schlüssel PUBLIC, PRIVATE oder DOMAINPARAMS (außer bei RSA).

IAlgorithm* CreateAlgorithm(mode_t mode) const oder **IAlgorithm* CreateAlgorithm(mode_t mode, const Blob &data) const**

erzeugt einen Algorithmus, der das angeforderte kryptographische Verfahren durchführt. `mode` kann abhängig vom Schlüssel-Objekt einen der folgenden Werte annehmen: ENCRYPT, DECRYPT, SIGN, VERIFY und MAC. Die zweite Methode findet Verwendung, wenn zum Initialisieren des Algorithmus weitere Daten notwendig sind, wie beispielsweise im Falle von VERIFY die zu überprüfende Signatur oder beim Erzeugen eines IES-Ver- oder Entschlüsselungsalgorithmus der Public Key-Blob der anderen Partei.

IKey* GetPointer() oder **const IKey* GetPointer() const**

gibt den intern gespeicherten IKey-Pointer zurück. Die Handle-Klasse `Key` behält aber den Besitz des Pointers.

operator IKey* () oder **operator const IKey* () const**

Diese Konvertierungsoperatoren führen zu einem impliziten Aufruf von `GetPointer()`.

IKey* ReleasePointer()

gibt den intern gespeicherten IKey-Pointer zurück und den Besitz frei.

Key& operator= (const Key& key)

Zuweisungsoperator.

7. 3 Interfaces

In vielen der folgenden Interface-Klassen tauchen immer wieder die Methoden `Import()`, `Export()`, `SetParam()` und `GetParam()` auf. Die Bedeutung der Methoden liegt auf der Hand: es werden Informationen aus einem Blob importiert, exportiert und Parameter gesetzt oder zurückgegeben.

Bei Interfaces, die den Zugriff auf aggregierte Objekte ermöglichen, ist der Sachverhalt allerdings komplizierter: falls auf ein umfassendes Objekte zugegriffen wird, wird sukzessive auf dessen aggregierte Objekte zugegriffen. Wird z.B. ein umfassendes Objekt exportiert, müssen auch die in dem aggregierten Objekt enthaltenen Informationen exportiert werden. Diese Aufgabe, die in den aggregierten Objekten enthaltenen Informationen zu integrieren, liegt bei dem umfassenden Objekt.

Beispiel:

- Als Interface für ein umfassendes Objekt wählen wir `IBlockCipherKey`, also das Interface, dass den Zugriff auf Schlüssel-Objekte für Blockchiffren ermöglicht.
- Als Interface für ein aggregiertes Objekt wählen wir `IBlockCipherMode`, dass den Zugriff auf den Betriebsmodus von Schlüssel-Objekten für Blockchiffren ermöglicht.

  Mit `SetParam()` setzt man Parameter, wie z.B. den Betriebsmodus. Dieses ist eine Information des aggregierten Objektes. Soll nun der Betriebsmodus exportiert werden, ist es die Aufgabe des umfassenden Objekts, also der Klasse, die das Interface `IBlockCipherKey` besitzt, sukzessive das aggregierte Objekt aufzurufen und die Information zu integrieren. Will man also eine Export-Methode für ein Objekt, dass das Interface `IBlockCipherKey` besitzt, implementieren, obliegt das Datenformat allein bei diesem Objekt. Es muss dafür sorgen, dass auch der Betriebsmodus exportiert wird, also das Objekt, dass das Interface `IBlockCipherMode` besitzt.

Wenn Sie eine eigene von IKey-Klasse abgeleitete Klasse implementieren und zu dem Handle Key kompatibel sein wollen, ist bei der Implementierung der Import- und Export-Methoden folgendes zu beachten: Die Handle-Klasse Key erwartet, dass der von einem IKey-Objekt exportierte Blob in der Form

```
SEQUENCE
{
        Name IA5String,
        ...
}
```

nach ASN.1-BER-kodiert ist. Dabei muss `Name` der Bezeichner des Objektes sein, wie er in der Key-Registry eingetragen ist.

Desweiteren wird oft mit Registries, Erzeugerfunktionen und der Methode `GetCreatePointer()` gearbeitet. Die Methodik, die dahinter steht, wird nun erklärt Zu jeder konkreten Klasse, die implementiert wird, muss auch eine Erzeugerfunktion bereitgestellt werden:

```
IXXX CreateYYY()
{
        return new YYY;
}
```

wobei `XXX` der Name einer Interface-Klasse und `YYY` der Name der konkreten Klasse ist. Hat man einen Pointer auf eine Erzeugerfunktion, kommt man zum Begriff der Registry. Eine Registry ist eine

Tabelle, in der ein Bezeichner-String mit einem Pointer auf eine Erzeugerfunktion assoziiert wird. Die Registry enthält Methoden, um mit Hilfe eines Bezeichner-Strings Objekte der zugehörigen Klasse zu erzeugen, mittels des Zeigers auf die Erzeugerfunktion den Bezeichner-String zu bekommen und zum Durchsuchen der Registry nach registrierten Klassen. In dem Fall, dass man ein Objekt hat, kann mit der Methode `GetCreatePointer()` den Zeiger auf die Erzeugerfunktion erhalten und mit diesem über die Registry den eingetragenen Bezeichner-String seiner Klasse. Den Bezeichner-String kann man z.B. innerhalb einer Export-Funktion benutzen, um aggregierte Objekte zu identifizieren.

Beispiel:

> In der Export-Methode von `BlockCipherKey` wird die Export-Methode des aggregierten BlockCipher-Objekts aufgerufen. Der zurückgegebene Datenblob des BlockCipher-Objekts enthält den symmetrischen Schlüssel und kann somit von `BlockCipherKey` mitexportiert werden. Es wird `GetCreatePointer()` aufgerufen, um herauszufinden, um welche Blockchiffre es sich handelt, und um über die BlockCipher-Registry den Bezeichner-String der Blockchiffre zu ermitteln. Dieser Bezeichner-String wird mitexportiert.

7.3.1 Algorithm-Interfaces

Interfaces aus dieser Kategorie behandeln die Verwaltung und Benutzung der verschiedenen Algorithmen, die an die Schlüssel verknüpft sind.

7.3.1.1 Das Interface **`IAlgorithm`**

Dieses ist die abstrakte Klasse, von der alle anderen konkreten `Algorithm`-Klassen abgeleitet sind.

```
class IAlgorithm
{
public:
        virtual void Write(const Blob& input ) = 0;
        virtual void Write(const byte* input, size_t insize ) = 0;
        virtual void Finalize() = 0;
        virtual size_t GetAvailableSize() const = 0;
        virtual size_t Read(Blob& output, size_t max = 0 ) = 0;
        virtual size_t Read(byte* outbuffer, size_t buffersize ) = 0;
        virtual status_t GetStatus() const = 0;
        virtual ~IAlgorithm() {}
};
```

Die verschiedenen Methoden haben die dieselbe Bedeutung wie unter der Handle-Klasse `Algorithm` aufgeführt.

7.3.2 Key-Interfaces

Interfaces aus dieser Kategorie offerieren das Anlegen, Exportieren, Importieren und Benutzen der verschiedensten Schlüssel. Je nach Art des Schlüssels können unterschiedliche Funktionen zur Verfügung stehen; so mag z.B. ein Export eines privaten Schlüssels auf einer Smartcard nicht möglich sein.

7.3.2.1 Das Interface **`IKey`**

Dieses ist die abstrakte Klasse, von der alle anderen konkreten Key-Klassen abgeleitet sind. Die Methoden, die der Handle-Klasse `Key` entsprechen, werden nicht noch einmal aufgeführt.

```
class IKey
{
public:
        virtual IKey* Clone() const =0;
        virtual void Import(const Blob &keyblob) =0;
        virtual void Export(Blob &keyblob, export_t type=DEFAULT) const =0;
        virtual void SetParam(paramid_t id, const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id, const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;
        virtual void Generate(IRNGAlg* prng=0)=0;
        virtual void Derive(const Blob &data, const Blob &salt=Blob())=0;
        virtual IAlgorithm* CreateAlgorithm(mode_t Mode) const =0;
        virtual IAlgorithm* CreateAlgorithm(mode_t Mode, const Blob &data)
const =0;
        virtual void* GetCreatePointer() const=0;

        virtual ~IKey() {}
};
```

`IKey* Clone() const`

erzeugt eine Kopie der aktuellen Key-Instanz.

`void* GetCreatePointer() const`

gibt einen Funktionspointer zum Erzeugen einer Key-Instanz zurück.

7.3.2.2 Das Interface **`IAgreementKey`**

Dies ist das Interface für alle *key agreement*-Schlüssel. Hierbei ist *key agreement* der englische Begriff für Schlüsselaustausch.

```
class IAgreementKey : public IKey
{
public:
        virtual void SetDerivator(IDerivator* derivator) =0;
        virtual const IDerivator* GetDerivator() const =0;
        virtual IDerivator* GetDerivator() =0;
};
```

`void SetDerivator(IDerivator* derivator)`

setzt das Objekt zur Schlüsselableitung.

`const IDerivator* GetDerivator() const` oder **`IDerivator* GetDerivator()`**

liefert einen Zeiger auf das verwendete Schlüsselableitungsobjekt zurück.

7.3.2.3 Das Interface **IIESKey**

Dieses Interface ermöglicht den Zugriff auf die aggregierten Objekte eines IES-Schlüssels, wie DL-IES oder EC-IES.

```
class IIESKey : public IKey
{
public:
        virtual void SetMAC(IMACKey* mac) =0;
        virtual const IMACKey* GetMAC() const =0;
        virtual IMACKey* GetMAC() =0;
        virtual void SetAgreement(IAgreementKey* derivator) =0;
        virtual const  IAgreementKey* GetAgreement() const =0;
        virtual IAgreementKey* GetAgreement() =0;

        virtual ~IIESKey() {}

};
```

void SetMAC(IMACKey* mac)

setzt den zu verwendenden *Message Authentication Code*.

const IMACKey* GetMAC() const oder **IMACKey* GetMAC()**

liefert den Zeiger auf das verwendete MACKey-Objekt zurück.

void SetAgreement(IAgreementKey* derivator)

setzt das Objekt zur Schlüsselableitung.

const IAgreementKey* GetAgreement() const oder **IAgreementKey* GetAgreement()**

liefert einen Zeiger auf das verwendete Schlüsselableitungsobjekt zurück.

7.3.2.4 Das Interface **IRSAKey**

Dieses Interface ermöglicht den Zugriff auf das aggregierte Padding-Objekt des RSAKeys.

```
class IRSAKey : public ISignatureKey
{
public:
        virtual void SetPadding(IPadding* padding) =0;
        virtual const IPadding* GetPadding()const =0;
        virtual IPadding* GetPadding() =0;
};
```

void SetPadding(IPadding* padding)

konfiguriert das Key-Objekt mit einem Paddingalgorithmus.

const IPadding* GetPadding() const oder **IPadding* GetPadding()**

liefert einen Zeiger auf das Padding-Objekt zurück.

7.3.2.5 Das Interface **`ISignatureKey`**

Dieses Interface ermöglicht den Zugriff auf das aggregierte EMSA-Objekt des SignatureKeys.

```
class ISignatureKey : public IKey
{
public:
        virtual void SetEncoder(IEMSAAlg *encoder)=0;
        virtual const IEMSAAlg *GetEncoder() const =0;
        virtual IEMSAAlg *GetEncoder() =0;
};
```

`void SetEncoder(IEMSAAlg *encoder)`

setzt das EMSA-Objekt.

`const IEMSAAlg *GetEncoder() const` oder **`IEMSAAlg *GetEncoder()`**

liefert einen Zeiger auf das EMSA-Objekt zurück.

7.3.3 BlockCipher-Interfaces

Interfaces aus dieser Kategorie handhaben alle Aspekte der Benutzung von Blockchiffren, z.B. die Auswahl des Arbeitsmodus und anderer Parameter.

7.3.3.1 Das Interface **`IBlockCipher`**

Diese Klasse repräsentiert symmetrische Blockchiffren und stellt die Low-Level-Verschlüsselungsoperationen `Encrypt` und `Decrypt` auf Blockebene zur Verfügung. Sie wird von `IBlockCipherKey` verwendet und kann dazu benutzt werden, neue Blockchiffren zu implementieren.

```
class IBlockCipher
{
public:
        virtual IBlockCipher* Clone() const =0;

        virtual void Import( const Blob& indata ) =0;
        virtual void Export( Blob& outdata ) const =0;
        virtual void SetRawKey( const Blob& keyblob ) =0;
        virtual void GetRawKey( Blob& keyblob ) const =0;

        virtual void Encrypt(const byte* in,byte* out) const =0;
        virtual void Decrypt(const byte* in,byte* out) const =0;

        virtual size_t GetKeySize(size_t keysize=0) const =0;
        virtual size_t GetMinKeySize() const =0;
        virtual size_t GetMaxKeySize() const =0;
        virtual size_t GetNextKeySize(size_t prevsize) const =0;
        virtual size_t GetBlockSize() const =0;

        virtual void* GetCreatePointer() const =0;
        virtual ~IBlockCipher() {}
};
```

`IBlockCipher* Clone() const`

erzeugt eine Kopie der aktuellen `BlockCipher`-Instanz.

`void Import( const Blob& indata )`

importiert den symmetrischen Schlüssel und gegebenenfalls weitere Parameter aus `indata`.

`void Export( Blob& outdata ) const`

exportiert den symmetrischen Schlüssel und gegebenenfalls weitere Parameter in `outdata`.

`void SetRawKey( const Blob& keyblob )`

setzt den symmetrischen Schlüssel der Chiffre, der sich in `keyblob` befindet.

`void GetRawKey( Blob& keyblob ) const`

schreibt den symmetrische Schlüssel in `keyblob`.

`void Encrypt(const byte* in,byte* out) const`

verschlüsselt den Speicherbereich, auf den der Zeiger `in` zeigt und schreibt ihn in den Speicherbereich, auf den der Zeiger `out` zeigt. Diese Methode erwartet, dass sowohl der `in`- als auch der `out`-Speicherbereich eine Größe von `GetBlockSize()` hat. Dabei wird der intern gespeicherte Schlüssel verwendet.

void Decrypt(const byte* in,byte* out) const

wie `Encrypt`, nur zum Entschlüsseln.

size_t GetKeySize(size_t keysize=0) const

Wenn `GetKeySize()` mit 0 aufgerufen wird, wird die Schlüssellänge des aktuell gespeicherten Schlüssels zurückgegeben. Bei Werten größer 0 wird die mögliche Schlüssellänge, die dem übergebenen Wert am nächsten liegt, zurückgegeben. Sowohl der Eingabe- als auch der Rückgabewert ist jeweils die Schlüssellänge in Byte.

size_t GetMinKeySize() const

gibt die Mindestschlüssellänge in Byte zurück.

size_t GetMaxKeySize() const

gibt die maximale Schlüssellänge in Byte zurück.

size_t GetNextKeySize(size_t prevsize) const

gibt die nächst größere, zulässige Schlüssellänge in Byte zurück. Wenn `prevsize` größer oder gleich der maximale Schlüssellänge ist, wird die maximale Schlüssellänge zurückgegeben, so dass der Rückgabewert von `GetNextKeySize()` immer einer gültigen Schlüssellänge entspricht.

size_t GetBlockSize() const

gibt die Blocklänge in Byte zurück.

void* GetCreatePointer() const

gibt den Funktionspointer auf die zu dieser Instanz gehörende Erzeugerfunktion zurück.

7.3.3.2 Das Interface **IBlockCipherKey**

Dieses Interface ermöglicht den Zugriff auf die aggregierten Objekte eines Schlüssels für eine Blockchiffre. Diese sind Padding-Objekte, Cipher-Objekte, BlockCipherMode-Objekte und Derivator-Objekte.

```
class IBlockCipherKey : public IKey
{
public:
        virtual void SetPadding(IPadding* padding) =0;
        virtual const IPadding* GetPadding() const =0;
        virtual IPadding* GetPadding()  =0;

        virtual void SetCipher(IBlockCipher* cipher) =0;
        virtual const IBlockCipher* GetCipher() const =0;
        virtual IBlockCipher* GetCipher() =0;


        virtual void SetMode(IBlockCipherMode* mode) =0;
        virtual const IBlockCipherMode* GetMode() const =0;
        virtual IBlockCipherMode* GetMode() =0;

        virtual void SetDerivator(IDerivator* Derive) =0;
        virtual const IDerivator* GetDerivator() const =0;
        virtual IDerivator* GetDerivator()  =0;

};
```

void SetPadding(IPadding* padding)

konfiguriert das Key-Objekt mit einem Paddingalgorithmus.

const IPadding* GetPadding() const oder **IPadding* GetPadding()**

liefert einen Zeiger auf das Padding-Objekt zurück.

void SetCipher(IBlockCipher* cipher)

übergibt dem Key-Objekt den Zeiger auf ein Cipher-Objekt.

const IBlockCipher* GetCipher() const oder **IBlockCipher* GetCipher()**

liefert einen Zeiger auf das Cipher-Objekt zurück.

void SetMode(IBlockCipherMode* mode)

konfiguriert das Key-Objekt mit einem anderen Verschlüsselungsmodus.

const IBlockCipherMode* GetMode() const oder **IBlockCipherMode* GetMode()**

liefert einen Zeiger auf das Modus-Objekt zurück.

void SetDerivator(IDerivator* Derive)

konfiguriert das Key-Objekt mit einem anderen Schlüsselableitungsalgorithmus.

const IDerivator* GetDerivator() const oder **IDerivator* GetDerivator()**

liefert einen Zeiger auf das Schlüsselableitungs-Objekt.

7.3.3.3 Das Interface **IBlockCipherMode**

Mit Hilfe dieses Interfaces kann jeder Blockalgorithmus in den Modi CBC, ECB oder CFB betrieben werden. Weitere Modi wie z.B. OFB können von diesem Interface unterstützt werden.

```
class IBlockCipherMode
{
public:
        virtual IBlockCipherMode* Clone() const=0;
        virtual void Import(const Blob& indata) =0;
        virtual void Export(Blob& outdata) const =0;
        virtual void SetParam(paramid_t id, const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id, const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual IAlgorithm* CreateEncAlgorithm( IBlockCipher* cipher,
IPadding* padding, const Blob& iv) const=0;
        virtual IAlgorithm* CreateDecAlgorithm( IBlockCipher* cipher,
IPadding* padding) const=0;
        virtual void* GetCreatePointer() const=0;

        virtual ~IBlockCipherMode() {}
};
```

IBlockCipherMode* Clone() const

erzeugt eine Kopie der aktuellen `BlockCipherMode`-Instanz.

void Import(const Blob& indata)

importiert Konfigurationsparameter aus `indata`.

void Export(Blob& outdata) const

exportiert Konfigurationsparameter in `outdata`.

void SetParam(paramid_t id, const Blob &blob) oder **void SetParam(paramid_t id,int val)** oder **void SetParam(paramid_t id, const char *cstr)**

setzt Konfigurationsparameter.

int GetParam(paramid_t id) const oder **void GetParam(paramid_t id, Blob& blob) const**

liest Konfigurationsparameter aus.

IAlgorithm* CreateEncAlgorithm(IBlockCipher* cipher, IPadding* padding, const Blob& iv) const

erzeugt eine `IAlgorithm`-Instanz zur Verschlüsselung eines Datenstroms unter Verwendung der übergebenen Chiffre, des Paddings und des Initialisierungsvektors.

IAlgorithm* CreateDecAlgorithm(IBlockCipher* cipher, IPadding* padding) const

erzeugt eine `IAlgorithm`-Instanz zur Entschlüsselung eines Datenstroms unter Verwendung der übergebenen Chiffre, des Paddings und des Initialisierungsvektors.

void* GetCreatePointer() const

gibt den Funktionspointer auf die zu dieser Instanz gehörende Erzeugerfunktion zurück.

7.3.4 Hash-Interfaces

Interfaces aus dieser Kategorie bieten die Benutzung von Hashes an.

7.3.4.1 Das Interface **IHashAlg**

Dieses ist das Interface für alle Hashalgorithmen.

```
class IHashAlg : public IAlgorithm
{
public:
        virtual IHashAlg* Clone() const =0;
        virtual void Import(const Blob& indata) =0;
        virtual void Export(Blob& outdata) const =0;
        virtual void SetParam(paramid_t id, const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id, const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual size_t GetBlockSize() const = 0;
        virtual size_t GetHashSize() const = 0;
        virtual void Reset() = 0;
        virtual void* GetCreatePointer() const =0;
        virtual ~IHashAlg() {}
};
```

IHashAlg* Clone() const

erzeugt eine Kopie der aktuellen Hash-Instanz.

void Import(const Blob& indata)

importiert Konfigurationsparameter aus `indata`.

void Export(Blob& outdata) const

exportiert Konfigurationsparameter in `outdata`.

void SetParam(paramid_t id, const Blob &blob) oder **void SetParam(paramid_t id,int val)** oder **void SetParam(paramid_t id, const char *cstr)**

setzt Konfigurationsparameter.

int GetParam(paramid_t id) const oder **void GetParam(paramid_t id, Blob& blob) const**

liest Konfigurationsparameter aus.

size_t GetBlockSize() const

gibt die Blocklänge in Byte zurück.

size_t GetHashSize() const

gibt die Länge des Hashwertes in Byte zurück.

void Reset()

verwirft die bisherigen Eingaben des Hashalgorithmus und stellt den Startzustand wieder her.

`void* GetCreatePointer() const`

gibt den Funktionspointer auf die zu dieser Instanz gehörende Erzeugerfunktion zurück.

7.3.4.2 Das Interface **`IHashMACKey`**

Dieses Interface ermöglicht den Zugriff auf das aggregierte Hashobjekt des HashMACs.

```
class IHashMACKey : public IMACKey
{
public:
        virtual void SetHash(IHashAlg* hash) = 0;
        virtual const IHashAlg* GetHash() const = 0;
        virtual IHashAlg* GetHash() = 0;

};
```

`void SetHash(IHashAlg* hash)`

konfiguriert das HashMAC-Objekt mit dem Hashalgorithmus `hash`.

`const IHashAlg* GetHash() const` oder **`IHashAlg GetHash()`**

liefert einen Zeiger auf das Hash-Objekt zurück.

7.3.5 EMSA-Interfaces

Interfaces aus dieser Kategorie leiten von den ‚Standard'-Algorithmus-Interfaces ab und bieten zusätzliche Funktionalitäten, die die Konfiguration der Benutzung der verwendeten Kodier- und Signaturalgorithmen ermöglichen.

7.3.5.2 Das Interface **`IEMSAAlg`**

Dies ist die Interface-Klasse für die *Encoding Method for Signature with Appendix* -Algorithmen. Diese Algorithmen bereiten zu signierende Daten auf. Dazu werden mit der `Write`-Methode die zu signierenden Daten an den Algorithmus übergeben und nach einem `Finalize()` mit `Read()` die kodierten Daten abgeholt und einem Signierprimitiva übergeben.

```
class IEMSAAlg : public IAlgorithm
{
public:
        virtual IEMSAAlg* Clone() const =0;

        virtual void Import(const Blob &blob)=0;
        virtual void Export(Blob &blob) const =0;
        virtual void SetParam(paramid_t id, const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id, const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual void Reset(size_t sizeinbits) =0;

        virtual void* GetCreatePointer() const =0;
};
```

`IEMSAAlg* Clone() const`

erzeugt eine Kopie der EMSA-Instanz.

`void Import(const Blob &blob)`

importiert Konfigurationsparameter aus einem Blob.

`void Export(Blob &blob) const`

exportiert Konfigurationsparameter in einen Blob.

`void SetParam(paramid_t id, const Blob &blob)` oder **`void SetParam(paramid_t id,int val)`** oder **`void SetParam(paramid_t id, const char *cstr)`**

setzt Konfigurationsparameter.

`int GetParam(paramid_t id) const` oder **`void GetParam(paramid_t id, Blob& blob) const`**

liest Konfigurationsparameter aus.

`void Reset(size_t sizeinbits)`

verwirft die bisherigen Eingaben des EMSA-Algorithmus und stellt den Startzustand wieder her. `sizeinbits` gibt dabei die Größe der Ausgabedaten an, die nach einem `Finalize()` zur Verfügung stehen.

`void* GetCreatePointer() const`

gibt den Funktionspointer auf die zu dieser Instanz gehörende Erzeugerfunktion zurück.

7.3.5.2 Das Interface **`IEMSAWithHashAlg`**

Dieses Interface ermöglicht den Zugriff auf das aggregierte Hashobjekt eines EMSA-Objektes. Dabei steht die Abkürzung EMSA für *Encoding Method for Signature with Appendix.*

```
class IEMSAWithHashAlg : public IEMSAAlg
{
public:
        virtual void SetHash(IHashAlg* hash) =0;
        virtual const IHashAlg* GetHash() const =0;
        virtual IHashAlg* GetHash() =0;
};
```

`void SetHash(IHashAlg* hash)`

konfiguriert das EMSA-Objekt mit dem Hashalgorithmus `hash`.

`const IHashAlg* GetHash() const` oder **`IHashAlg* GetHash()`**

liefert einen Zeiger auf das Hash-Objekt zurück.

7.3.6 Derivator-Interfaces

Die Derivator-Interfaces erlauben Zugriff auf Objekte zur Schlüsselableitung.

7.3.6.1 Das Interface **`IDerivator`**

Dieses Interface ermöglicht den Zugriff auf Objekte zur Schlüsselableitung.

```
class IDerivator
{
public:
        virtual IDerivator* Clone() const=0;

        virtual void Import(const Blob &blob)=0;
        virtual void Export(Blob &blob) const =0;
        virtual void SetParam(paramid_t id, const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id, const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual void Derive(const Blob& secret, const Blob& salt, Blob&
keymat) const=0;
        virtual void* GetCreatePointer() const =0;
        virtual ~IDerivator() {}
};
```

`IDerivator* Clone() const`

erzeugt eine Kopie der `Derivator`-Instanz.

`void Import(const Blob &blob)`

importiert Konfigurationsparameter aus einem Blob.

`void Export(Blob &blob) const`

exportiert Konfigurationsparameter in einen Blob.

`void SetParam(paramid_t id, const Blob &blob)` oder **`void SetParam(paramid_t id,int val)`** oder **`void SetParam(paramid_t id, const char *cstr)`**

setzt Konfigurationsparameter.

`int GetParam(paramid_t id) const` oder **`void GetParam(paramid_t id, Blob& blob) const`**

liest Konfigurationsparameter aus.

`void Derive(const Blob& secret, const Blob& salt,Blob& keymat) const`

`secret` und `salt` werden zusammengemischt (meistens durch eine Hashfunktion) und das Ergebnis in `keymat` geschrieben.

`void* GetCreatePointer() const =0`

gibt einen Funktionspointer zum Erzeugen einer `IDerivator`-Instanz zurück. Dieser Funktionspointer zeigt auf eine globale Funktion mit der Signatur `IDerivator* create()`.

7.3.6.2 Das Interface **IDerivatorWithHash**

Dieses Interface ermöglicht den Zugriff auf das aggregierte Hashobjekt des Schlüsselableitungsobjekts.

```
class IDerivatorWithHash: public IDerivator
{
public:
        virtual void SetHash( IHashAlg* hash) =0;
        virtual const IHashAlg* GetHash() const =0;
        virtual IHashAlg GetHash() =0;
}
```

void SetHash(IHashAlg* hash)

konfiguriert das Derivator-Objekt mit dem Hashalgorithmus hash.

const IHashAlg* GetHash() const oder **IHashAlg GetHash()**

liefert einen Zeiger auf das Hash-Objekt zurück.

7.3.7 Padding-Interfaces

Die Padding-Interfaces erlauben Zugriff auf Objekte, die das Auffüllen von Daten auf eine bestimmte Länge und das ‚Zuschneiden' aufgefüllter (gepaddeter) Daten auf die Originallänge erlauben.

7.3.7.1 Das Interface **`IPadding`**

Das Padding-Interface ermöglicht das Auffüllen von Daten auf eine bestimmte Länge und die Umkehrung dessen.

```
class IPadding
{
public:
        virtual IPadding* Clone() const =0;

        virtual void Import(const Blob& indata) =0;
        virtual void Export(Blob& outdata) const =0;
        virtual void SetParam(paramid_t id,const Blob &blob) =0;
        virtual void SetParam(paramid_t id,int val) =0;
        virtual void SetParam(paramid_t id,const char *cstr) =0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual void Pad(Blob& Buffer, size_t len) const =0;
        virtual size_t Unpad(Blob& Buffer) const =0;

        virtual void* GetCreatePointer() const =0;

        virtual ~IPadding() {}
};
```

`IPadding* Clone() const`

erzeugt eine Kopie der Padding-Instanz.

`void Import(const Blob& indata)`

`indata` stellt das in `pad()` verwendete Füllmuster zur Verfügung.

`void Export(Blob& outdata) const`

schreibt das Füllmuster in `outdata`.

`void SetParam(paramid_t id, const Blob &blob` oder **`void SetParam(paramid_t id,int val)`** oder **`void SetParam(paramid_t id, const char *cstr)`**

setzt Konfigurationsparameter.

`int GetParam(paramid_t id) const` oder **`void GetParam(paramid_t id, Blob& blob) const`**

liest Konfigurationsparameter aus.

`void Pad(Blob& buffer, size_t len) const`

füllt (in der Regel mit Hilfe eines Füllmusters) den Inhalt von `Buffer` auf eine Länge von `len` auf. Falls `len` < `buffer.size()`, wird die Exception `PaddingException` geworfen.

`size_t Unpad(Blob& buffer) const`

führt die Umkehrung von `pad()` aus und die neue Größe von `buffer` wird zurückgegeben. Falls die Umkehrung nicht möglich ist, wird die Exception `PaddingException` geworfen.

`void* GetCreatePointer() const`

gibt einen Funktionspointer zum Erzeugen einer Padding-Instanz zurück.

7.3.8 Random-Interfaces

Interfaces aus dieser Gruppe ermöglichen den Zugriff auf Objekte, mit denen sich Zufallszahlengeneratoren (Random Number Generators) auslesen lassen. Hierbei ist zu beachten, dass eine *Schreib*funktionalität auf einem Zufallszahlengenerator den Entropiepool von diesem ändern kann.

7.3.8.1 Das Interface **IRNGAlg**

Dieses ist das Interface für Zufallszahlengeneratoren, wobei RNG die Abkürzung von *Random Number Generator* ist. Mit `Read()` werden Zufallszahlen abgeholt und mit `Write()` Entropie zum Zufallszahlengenerator hinzugefügt. Die Methode `Finalize()` hat keine Auswirkung. Im Gegensatz zu allen anderen Algorithm-Objekten führt der Aufruf der Methode `Read(blob)` nicht dazu, dass alle zur Verfügung stehenden Daten abgeholt werden, da bei einem Zufallszahlengeneratoren unendlich viele Daten zur Verfügung stehen. Aus diesem Grund liefert die Methode `GetAvailableSize()` auch stets 0 zurück.

Zu diesem Interface gibt es keine Registry-Klasse. Statt dessen verwenden die Objekte, die Zufallszahlengenerator benötigen, die Erzeugerfunktion, deren Zeiger in zwei globalen Variablen gespeichert ist: `CreateStrongRNGAlg` für Zufallszahlen zur Schlüsselerzeugung und `CreateFastRNGAlg` für zufälliges Padding und Initialisierungsvektoren. Standardmäßig liefert `CreateStrongRNGAlg` einen Zufallszahlengenerator auf Basis des FIPS einen linearen Kongruenzgenerator.

```
class IRNGAlg:public IAlgorithm
{
public:
        virtual IRNGAlg* Clone() const =0;

        virtual void Import(const Blob &indata) =0;
        virtual void Export(Blob &outdata ) const =0;
        virtual void SetParam(paramid_t id,const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id,const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual void* GetCreatePointer() const =0;
};
```

`IRNGAlg* Clone() const`

erzeugt eine Kopie der aktuellen `RNGAlg`-Instanz.

`void Import(const Blob &indata)`

importiert Konfigurationsparameter aus `indata`.

`void Export(Blob &outdata) const`

exportiert Konfigurationsparameter in `outdata`.

`void SetParam(paramid_t id, const Blob &blob)` oder **`void SetParam(paramid_t id,int val)`** oder **`void SetParam(paramid_t id, const char *cstr)`**

setzt Konfigurationsparameter.

`int GetParam(paramid_t id) const` oder **`void GetParam(paramid_t id, Blob& blob) const`**

liest Konfigurationsparameter aus.

`void* GetCreatePointer() const`

gibt den Funktionspointer auf die zu dieser Instanz gehörende Erzeugerfunktion zurück.

7.3.9 Certificate-Interfaces

Die Zertifikats-Interfaces erlauben ein Auslesen von Zertifikaten sowie eine Bearbeitung von Zertifikatsanforderungen. Darunter zählen u.a. das Extrahieren des Zertifikats selbst, des öffentlichen Schlüssels, zu dem das Zertifikat gehört, Prüfung der Gültigkeitsdauer sowie anderer Attribute, die zu dem Zertifikat gehören.

7.3.9.1 Das Interface **`ICertificate`**

Mit Hilfe dieser Klasse kann man Zertifikate ansprechen, also z.B. den darin enthaltenen öffentlichen Schlüssel oder dessen Gültigkeit.

```
class ICertificate
{
public:
        virtual ICertificate* Clone() const=0;
        virtual void Import(const Blob &certblob)=0;
        virtual void Export(Blob &certblob) const =0;
        virtual void SetParam(paramid_t id,
                                    const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id,
                                    const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob) const =0;

        virtual void SetPublicKey(const IKey* pubkey) =0;
        virtual IKey* CreatePublicKey() const =0;
        virtual void Sign(const IKey* privkey)=0;
        virtual void Verify(const IKey* pubkey) const =0;
        virtual void* GetCreatePointer() const=0;

        virtual ~ICertificate() {}
};
```

`ICRL* Clone() const`

erzeugt eine Kopie der `Certificate`-Instanz.

`void* GetCreatePointer() const`

gibt einen Funktionspointer zum Erzeugen einer Zertifikatsinstanz zurück.

Die weiteren Methoden haben die dieselbe Bedeutung wie unter der Handle-Klasse `Certificate` aufgeführt.

7.3.9.2 Das Interface IX509Certificate

Dieses Interface ermöglicht den Zugriff auf das X.509-Zertifikats-Objekt.

```
class IX509Certificate : public ICertificate
{
public:
        virtual void SetValidityNotBefore( const Date &vnbefore ) =0;
        virtual void SetValidityNotAfter( const Date &vnafter ) =0;
        virtual void GetValidityNotBefore(Date &vnbefore ) const =0;
        virtual void GetValidityNotAfter(Date &vnafter ) const =0;

        virtual bool CheckValidity(const Date& date=Date()) const =0;

        virtual X509Extension GetExtension( const char* oid) const=0;
        virtual void SetExtension( const X509Extension &ext)=0;
        virtual X509Extension GetNextExtension( const X509Extension& ext)
const=0;
        virtual void RemoveExtension(const char* oid)=0;
};
```

`void SetValidityNotBefore( const Date &vnbefore )` oder **`void SetValidityNotAfter( const Date &vnafter )`** oder **`void GetValidityNotBefore(Date &vnbefore ) const`** oder **`void GetValidityNotAfter(Date &vnafter ) const`**

setzen bzw. lesen den Gültigkeitszeitraum des Zertifikats.

`bool CheckValidity(const Date& date=Date()) const`

überprüpft, ob das übergebene Datum im Gültigkeitszeitraum liegt.

`X509Extension GetExtension( const char* oid) const`

gibt die X.509-Extension mit der angegebenen OID zurück. Die OID wird String-kodiert; siehe Kapitel 7.1.3.

`void SetExtension( const X509Extension &ext)`

fügt die übergebene Extension in das Zertifikat ein.

`X509Extension GetNextExtension(const X509Extension& ext)const`

gibt die auf `ext` folgende Extension in der Extensionliste des Zertifikats zurück. Falls `ext==X509Extension()` wird die erste Extension zurückgegeben. Falls `ext` nicht vorhanden oder die letzte Extension in der Liste ist, wird `X509Extension()` zurückgegeben.

`void RemoveExtension(const char* oid)`

entfernt die Extension mit der angegebenen OID aus dem Zertifikat.

7.3.9.3 Das Interface ICVCertificate

Dieses Interface ermöglicht den Zugriff auf das CV-Zertifikats-Objekt. Im Wesentlichen wird hiermit die Funktionalität vom Certificate-Objekt auf einzelne Funktionen abgebildet.

```
class ICVCertificate : public ICertificate

{

public:

        virtual ICVCertificate* Clone() const=0;


        virtual void SetEffectiveDate( const Date &effective ) =0;

        virtual void SetExpirationDate( const Date &expiration ) =0;

        virtual void GetEffectiveDate(Date &effective ) const =0;

        virtual void GetExpirationDate(Date &vnafter ) const =0;


        virtual bool CheckValidity(const Date& date = Date() ) const =0;


        virtual void SetProfileIdentifier(int pi) =0;

        virtual int GetProfileIdentifier() const =0;


        virtual void SetCAReference( const char* issuer ) =0;

        virtual void SetCAReference( const Blob& issuer ) =0;

        virtual void GetCAReference( Blob &issuer ) const =0;


        virtual void SetHolderReference( const char* subject ) =0;

        virtual void SetHolderReference( const Blob& subject ) =0;

        virtual void GetHolderReference( Blob &subject ) const =0;


        virtual void SetHolderAuthorizationTemplate(const Blob& chat) =0;
```

```
        virtual void GetHolderAuthorizationTemplate(Blob& chat) const =0;


        virtual void SetEACePassportTemplate(byte mask) =0;

        virtual byte GetEACePassportTemplate() const =0;


        virtual void SetPublicKeyTLV(const Blob& pktlv) =0;

        virtual void GetPublicKeyTLV(Blob& pktlv) const =0;


        virtual void GetSignature(Blob &sig) const =0;

        virtual void SetSignature(const Blob &sig) =0;


        virtual void GetTBSCertificate( Blob &tbscert) const =0;


};
```

`void SetEffectiveDate( const Date &effective )` oder **`void SetExpirationDate( const Date &expiration )`** oder **`void GetEffectiveDate(Date &effective ) const`** oder **`void GetExpirationDate(Date &expiration ) const`**

setzen bzw. lesen den Gültigkeitszeitraum des Zertifikats. Die angegebenen Tage geben den jeweils ersten und letzten Tag an, an denen das Zertifikat (noch) gültig ist.

`bool CheckValidity(const Date& date=Date()) const`

überprüpft, ob das übergebene Datum im Gültigkeitszeitraum liegt.

`virtual void SetCAReference( const char* issuer ) =0;`

`virtual void SetCAReference( const Blob& issuer ) =0;`

`virtual void GetCAReference( Blob &issuer ) const =0;`

setzt/liest den Verweis auf den öffentlichen Schlüssel für die Überprüfung der Signatur der Certificate Authority

`virtual void SetProfileIdentifier(int pi) =0;`

`virtual int GetProfileIdentifier() const =0;`

setzt/liest den Profile Identifier für das Zertifikat

```
virtual void SetHolderReference( const char* subject ) =0;

virtual void SetHolderReference( const Blob& subject ) =0;

virtual void GetHolderReference( Blob &subject ) const =0;
```

setzt/liest den Verweis auf den öffentlichen Schlüssel dieses Zertifikats.

```
virtual void SetHolderAuthorizationTemplate(const Blob& chat) =0;

virtual void GetHolderAuthorizationTemplate(Blob& chat) const =0;
```

setzt/liest die Authentisierungsinformationen, die die Rollen und Authentisierungsmethoden der Zertifikatsinhaber berschreiben.

```
virtual void SetEACePassportTemplate(byte mask) =0;

virtual byte GetEACePassportTemplate() const =0;
```

setzt/liest das Holder Auhorization Template nach den Vorgaben für EAC-geschützte ePässe

```
virtual void SetPublicKeyTLV(const Blob& pktlv) =0;

virtual void GetPublicKeyTLV(Blob& pktlv) const =0;
```

setzt/liest den öffentlichen Schlüssel dieses Zertifikats

```
virtual void GetSignature(Blob &sig) const =0;

virtual void SetSignature(const Blob &sig) =0;
```

setzt/liest die Signatur dieses Zertifikats

7.3.9.4 Das Interface ICVCertRequest

Dieses Interface ermöglicht den Zugriff auf das CV-Zertifikatsanforderungs-Objekt.

```
class ICVCertRequest
{
public:
	virtual ICVCertificate* GetReducedCVCertificate() =0;
	virtual const ICVCertificate* GetReducedCVCertificate() const
=0;

	virtual void SetCAReference( const char* issuer ) =0;
	virtual void SetCAReference( const Blob& issuer ) =0;
	virtual void GetCAReference( Blob &issuer ) const =0;

	virtual void GetSignature(Blob &sig) const =0;
	virtual void SetSignature(const Blob &sig) =0;
	virtual bool IsAuthenticated() const =0;

	virtual void Import(const Blob &certblob) =0;
	virtual void Export(Blob &certblob) const =0;

	virtual void Sign(const IKey* privkey) =0;
	virtual int Verify(const IKey* pubkey) const =0;

	virtual ICVCertRequest* Clone() const=0;
	virtual ~ICVCertRequest() {}
	virtual void* GetCreatePointer() const=0;
};
```

```
virtual ICVCertificate* GetReducedCVCertificate() =0;

virtual const ICVCertificate* GetReducedCVCertificate() const =0;
```

Gibt das reduzierte CV-Zertifikat zurück, zu dem diese Anforderung ausgestellt wird.

void GetSignature(Blob &sig) const oder **void SetSignature(const Blob &sig)**

liest/setzt die Signatur zu der Zertifikatsanforderung

void SetCAReference(const Blob& issuer) oder **void GetCAReference(Blob& issuer) const**

setzt/liest den Verweis auf die Certificate Authority

```
virtual void SetCAReference( const char* issuer ) =0;

virtual void SetCAReference( const Blob& issuer ) =0;

virtual void GetCAReference( Blob &issuer ) const =0;
```

setzt/liest den Verweis auf die Certificate Authority

```
virtual void GetSignature(Blob &sig) const =0;

virtual void SetSignature(const Blob &sig) =0;
```

liest/setzt die Signatur zu der Zertifikatsanforderung

```
virtual bool IsAuthenticated() const =0;
```

True, wenn eine äußere Signatur vorhanden ist. Eine äußere Signatur ist für die Anforderung eines Folgezertifikats nötig und wird mit dem Schlüssel erstellt, das zu dem Zertifikat gehört, zu dem das Folgezertifikat angefragt wird.

HINWEIS: Es wird nur auf das Vorhandensein der Signatur geprüft, nicht auf deren Korrektheit.

```
virtual void Import(const Blob &certblob) =0;

virtual void Export(Blob &certblob) const =0;
```

importiert/exportiert die Zertifikatsanforderung

```
virtual void Sign(const IKey* privkey) =0;

virtual int Verify(const IKey* pubkey) const =0;
```

Unterzeichnet die komplette Zertifikatsanforderung mit den übergebenen privaten Schlüssel / prüft die Korrektheit der Signatur mit den öffentlichen Schlüssel

7.3.10 CRL-Interfaces

Die CRL (Certificate Revocation List, Zertifikatssperrlisten) –Interfaces bieten Funktionen zum Verwalten von Zertifikatssperrlisten, wie die Aufnahme (oder Entfernung) von Zertifikaten in die Sperrliste, das Unterzeichnen und die Gültigkeitsprüfung derselben.

7.3.10.1 Das Interface **ICRL**

Mit Hilfe dieser Klasse kann man CRLs benutzen, also z.B. Überprüfen, ob ein Zertifikat zurückgerufen wurde.

```
class ICRL
{
public:
        virtual ICRL* Clone() const=0;
        virtual void Import(const Blob &crlblob)=0;
        virtual void Export(Blob &crlblob) const =0;
        virtual void SetParam(paramid_t id, const Blob &blob)=0;
        virtual void SetParam(paramid_t id,int val)=0;
        virtual void SetParam(paramid_t id, const char *cstr)=0;
        virtual int GetParam(paramid_t id) const =0;
        virtual void GetParam(paramid_t id, Blob& blob)       const =0;

        virtual void Revoke(const ICertificate* cert) =0;
        virtual bool IsRevoked( const ICertificate* cert) const =0;
        virtual void RemoveRevokation(const ICertificate* cert) =0;
        virtual void Sign( const IKey* privkey) =0;
        virtual void Verify( const IKey* pubkey) const =0;
        virtual void* GetCreatePointer() const=0;

        virtual ~ICRL() {}
};
```

ICRL* Clone() const

erzeugt eine Kopie der `CRL`-Instanz.

void* GetCreatePointer() const

gibt einen Funktionspointer zum Erzeugen einer CRL-Instanz zurück.

Die weiteren Methoden haben die dieselbe Bedeutung wie unter der Handle-Klasse `CRL` aufgeführt.

7.3.10.2 Das Interface IX509CRL

Dieses Interface ermöglicht den Zugriff auf das X.509-CRL-Objekt.

```
class IX509CRL: public ICRL
{
public:
        virtual void SetThisUpdate( const Date &date )=0;
        virtual void SetNextUpdate( const Date &date )=0;
        virtual void GetThisUpdate(Date &date ) const=0;
        virtual void GetNextUpdate(Date &date ) const=0;

        virtual X509Extension GetExtension( const char* oid) const=0;
        virtual void SetExtension( const X509Extension &ext)=0;
        virtual X509Extension GetNextExtension(const X509Extension& ext)
const=0;
        virtual void RemoveExtension(const char* oid)=0;

        virtual Blob GetNextEntry(const Blob& certsernr) const=0;

        virtual void SetEntryRevocationDate(const Blob& certsn, const Date
&vnbefore )=0;
        virtual void GetEntryRevocationDate(const Blob& certsn, Date
&vnbefore ) const=0;

        virtual X509Extension GetEntryExtension(const Blob& certsn, const
char* oid) const=0;
        virtual void SetEntryExtension(const Blob& certsn, const
X509Extension &ext)=0;
        virtual X509Extension GetNextEntryExtension(const Blob& certsn,
const X509Extension& ext) const=0;
        virtual void RemoveEntryExtension(const Blob& certsn, const char*
oid)=0;

};
```

`void SetThisUpdate( const Date &date )` oder **`void SetNextUpdate( const Date &date )`** oder **`void GetThisUpdate(Date &date ) const`** oder **`void GetNextUpdate(Date &date ) const`**

setzt bzw. liest das aktuelle bw. nächste Aktualisierungsdatum der CRL.

`X509Extension GetExtension( const char* oid) const`

gibt die X.509-Extension mit der angegebenen OID zurück. Die OID wird String-kodiert; siehe Kapitel 7.1.3.

`void SetExtension( const X509Extension &ext)`

fügt die übergebene Extension in die CRL ein.

`X509Extension GetNextExtension(const X509Extension& ext)const`

gibt die auf `ext` folgende Extension in der Extensionliste der CRL zurück. Falls `ext==X509Extension()` wird die erste Extension zurückgegeben. Falls `ext`

nicht vorhanden oder die letzte Extension in der Liste ist, wird `X509Extension()` zurückgegeben.

`void RemoveExtension(const char* oid)`

entfernt die Extension mit der angegebenen OID aus der Extensionliste der CRL.

`Blob GetNextEntry(const Blob& certsernr) const`

gibt die Seriennummer des nächsten auf `certsernr` folgenden Eintrags in der CRL zurück. Falls `certsernr` der leere Blob ist, wird die Seriennummer des ersten Eintrags zurückgegeben. Falls `certsernr` nicht vorhanden oder der letzte Eintrag in der Liste ist, wird ein leerer Blob zurückgegeben.

`void SetEntryRevocationDate(const Blob& certsn,const Date &vnbefore)`

setzt das Rückrufdatum des CRL-Eintrags mit der Zertifikatsseriennummer `certsn`. Defaultmäßig wird das aktuelle Datum und der aktuelle Zeitpunkt beim Aufruf der Methode `Revoke()` gesetzt.

`void GetEntryRevocationDate(const Blob& certsn, Date &vnbefore ) const`

gibt das Rückrufdatum des CRL-Eintrags mit der Zertifikatsseriennummer `certsn` zurück.

`X509Extension GetEntryExtension(const Blob& certsn, const char* oid) const`

gibt die Extension in einem CRL-Eintrag mit der OID `oid` zurück. Der Eintrag wird über die Zertifikatsseriennummer `certsn` bestimmt.

`void SetEntryExtension(const Blob& certsn, const X509Extension &ext)`

fügt die übergebene Extension in den CRL-Eintrag ein. Der Eintrag wird über die Zertifikatsseriennummer `certsn` bestimmt.

`X509Extension GetNextEntryExtension(const Blob& certsn,const X509Extension& ext) const`

gibt die auf `ext` folgende Extension in den durch `certsn` bestimmten CRL-Eintrag zurück. Falls `ext==X509Extension()` wird die erste Extension zurückgegeben. Falls `ext` nicht vorhanden oder die letzte Extension in der Liste ist, wird `X509Extension()` zurückgegeben.

`void RemoveEntryExtension(const Blob& certsn, const char* oid)`

entfernt die Extension mit der angegebenen OID aus der Extensionliste des mit `certsn` bestimmten CRL-Eintrags.

7.3.11 MAC-Interfaces

Die MAC (Message Authentication Codes) –Interfaces bieten Funktionen, die für die Errechnung und Benutzung von MACs nötig sind.

7.3.11.1 Das Interface **ICBCMACKey**

Dieses Interface ermöglicht den Zugriff auf das aggregierte Blockchiffre-Objekt des CBCMACs.

```
class ICBCMACKey : public IMACKey
{
public:
        virtual void SetCipher(IBlockCipher* cipher) = 0;
        virtual const IBlockCipher* GetCipher() const = 0;
        virtual IBlockCipher* GetCipher() = 0;

};
```

`void SetCipher(IBlockCipher* cipher)`

übergibt dem Key-Objekt den Zeiger auf ein Cipher-Objekt.

`const IBlockCipher* GetCipher() const` oder **`IBlockCipher* GetCipher()`**

liefert einen Zeiger auf das Cipher-Objekt zurück.

7.3.12 SecureToken-Interfaces

Die SecureToken-Interfaces sind zum einen eine Erweiterung bereits bestehender Interfaces auf die Erfordernisse von Smartcards, zum anderen bieten sie selbst Zugriff auf Objekte, die allein zur Handhabung von Smartcards, Daten auf Smartcards und Kartenlesern vorbehalten sind.

7.3.12.1 ISubsystem

Diese Klasse enthält Funktionen, mit denen die automatische Erkennung von Slots gestartet werden können und ein Slot-Objekt erstellt werden können.

Die Klasse ISubsystem arbeitet intern mit einem Referenz-Counter, um nicht mehr benötigten Speicher automatisch freigegeben zu können. Im folgenden werden zunächst alle Funktionen aufgelistet und danach zu einigen Funktionen ergänzende Kommentare gegeben.

```
class ISubsystem : public IRefCounted
{
public:
        virtual void Refresh() = 0;
        virtual const char* GetName() const = 0;
        virtual void Install(int port)

        virtual const ISlot* GetSlot(int pos) const = 0;
        virtual int GetSlotNumber() const = 0;
        virtual ISlot* CreateSlot(const char* name) = 0;

        virtual void SetTokenConfig(ITokenConfig* tkcfg) = 0;

        virtual void NotifyEvent(IEventHandler* cmd) = 0;

};
```

Ergänzende Anmerkungen zu einzelnen Funktionen:

`virtual void Refresh() = 0;`

Erzwingt eine Neuauflistung aller durch diesem Subsystem verwalteten Slots.

`virtual const char* GetName() const = 0;`

Gibt den Namen dieses Subsystems (CSP, PCSC, ...) zurück.

`virtual void Install(int port)`

Diese Funktion ist nur beim CTAPI-Subsystem verwendet und erlaubt das manuelle Installieren eines Slots, weil die automatische Erkennung mittels `Refesh()` zu lange dauern könnte.

`virtual const ISlot* GetSlot(int pos) const = 0;`

Gibt das nte Slot-Objekt dieses Subsystems zurück.

`virtual int GetSlotNumber() const = 0;`

Gibt die Anzahl der Slots zurück, die dieses Subsystem verwaltet.

```
virtual void NotifyEvent(IEventHandler* cmd) = 0;
```

Registriert einen Event-Handler, der ein `SLOT_INSERTED`-Event übermittelt bekommt, wenn dieses Subsystem die Anwesenheit eines neuen Slots erkannt hat (z.B. ein neu angeschlossener Kartenleser)

7.3.12.2 ISlot

Diese Klasse ermöglicht den Zugriff auf einen Slot.

Die Klasse ISlot arbeitet intern mit einem Referenz-Counter, um nicht mehr benötigten Speicher automatisch freigeben zu können. Im folgenden werden zunächst alle Funktionen aufgelistet und danach zu einigen Funktionen ergänzende Kommentare gegeben.

```
        class ISlot : public IRefCounted
        {
        public:
                virtual ISlot* Clone() const = 0;

                virtual bool IsTokenPresent() const throw() = 0;
                virtual const char* GetName() const = 0;
                virtual Blob GetATR() const = 0;
                virtual ISubsystem* GetSystem() const = 0;

                virtual ISCardOS* CreateOS(const Blob& historical_bytes =
Blob()) = 0;
                virtual IToken* CreateToken(ISCardOS* os = 0) = 0;
                virtual ISCardAccess* CreateAccess() = 0;

                virtual void NotifyEvent(IEventHandler* cmd) = 0;

                virtual Blob GetHistoricalBytes() const = 0;

                virtual void SetTokenConfig(ITokenConfig* tkcfg) = 0;
                virtual bool HasSecurePinEntry(ISCardAccess* ac = 0) const =
0;

                virtual void SetParam(paramid_t id,int val) = 0;
                virtual int  GetParam(paramid_t id) const = 0;

        protected:
                virtual ~ISlot() { }
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual ISlot* Clone() const = 0;`

Erzeugt eine Kopie des Slot-Objekts.

`virtual bool IsTokenPresent() const throw() = 0;`

`True`, wenn ein Token im Slot eingelegt ist. Hier wird nur auf die Präsenz einer entsprechenden Hardware überprüft, sodass auch ein nicht unterstütztes Token als ‚vorhanden' im Sinne dieser Funktion erkannt wird.

`virtual const char* GetName() const = 0;`

Liest den Namen des Slots (d.h. des Kartenlesers) aus. Dieser kann z.B. für die Reselektion des Slots mit einer anderen Zugriffsart verwendet werden.

virtual Blob GetATR() const = 0;

Gibt – falls verfügbar – die ATR des eingelegten Tokens zurück.

NOTIZ: Da die ATR nur beim Reset des Tokens übermittelt wird werden etwaige Änderungen in dieser (z.B. durch Operationen wie das Anlegen eines neuen Profils) nur sichtbar, wenn das Token wieder zurückgesetzt wurde.

virtual ISubsystem* GetSystem() const = 0;

Liefert einen Rückverweis auf das Subsystem (CSP, PCSC, PKCS11, ...) zu das dieser Slot gehört.

virtual ISCardOS* CreateOS(const Blob& historical_bytes = Blob()) = 0;

Erzeugt ein OS-Objekt, um das eingelegte Token anzusprechen. Ist `historical_bytes` ein nichtleerer Blob, so wird dessen Inhalt als Entscheidungsgrundlage verwendet, welches OS für die Tokenzugriffe zugrundegelegt wird.

virtual IToken* CreateToken(ISCardOS* os = 0) = 0;

Erzeugt ein Token-Objekt für den Zugriff auf das eingelegte Token. Diese Funktion gibt eine Null zurück, wenn das eingelegte Token nicht unterstützt wird.

virtual ISCardAccess* CreateAccess() = 0;

Erzeugt ein Access-Objekt, um auf das eingelegte Token auf APDU-Ebene zugreifen zu können.

virtual void NotifyEvent(IEventHandler* cmd) = 0;

Trägt ein von `IEventHandler` abgeleitetes Objekt als Listener für Slot- und Tokenevents ein. Übermittelte Events sind `SLOT_REMOVED`, `TOKEN_REMOVED` und `TOKEN_INSERTED`.

virtual Blob GetHistoricalBytes() const = 0;

Gibt die Historical Bytes aus der ATR des eingelegten Tokens zurück. Diese Historical Bytes werden auch verwendet, wenn `CreateOS()` ohne Parameter aufgerufen wird.

virtual bool HasSecurePinEntry(ISCardAccess* ac = 0) const = 0;

True, wenn der Slot eine eigene PIN-Eingabemöglichkeit besitzt (PinPad). Zur Zeit sind nur PinPad-Kartenleser unterstützt, die das VERIFY_PIN_DIRECT-Protokoll anbieten.

7.3.12.3 ISCardAccess

Diese Klasse ermöglicht den Zugriff auf eine Smartcard auf der Protokoll-Ebene. Unterstützt werden das CTAPI- und PCSC-Protokoll. CSP- und PKCS11-Systeme haben eigene Zugriffsmechanismen auf eine Smartcard. Eine `SCardAccess`-Klasse ist in dieser Situation nicht erforderlich. Im folgenden werden zunächst alle Funktionen aufgelistet und danach zu einigen Funktionen ergänzende Kommentare gegeben.

```
        class ISCardAccess : public IRefCounted
        {
        public:
                virtual long GetProtocol() const = 0;
                virtual void SetProtocol(long protocol) = 0;

                virtual ulong GetTimeout() const throw() = 0;
                virtual bool SetTimeout(ulong timeout_msec) = 0;

                virtual void Open() = 0;
                virtual void Close() throw() = 0;
                virtual void ResetCard() = 0;
                virtual long CancelCardOperation() = 0;

                virtual void BeginTransaction() = 0;
                virtual void EndTransaction(DispositionType disposition =
LEAVE_CARD) = 0;
                virtual ulong GetTransactionDepth() const throw() = 0;

                virtual word SendCard(const Blob& cmd, Blob& response) = 0;

                virtual const ISlot* GetSlot() const = 0;
                virtual Blob GetResponse() const = 0;

                virtual word Send(const Blob& cmd) = 0;
                virtual word Send(const Blob& header, const Blob& data) = 0;
                virtual word Send(const Blob& header, const Blob& data, byte
le) = 0;

                virtual word Send(const std::string& cmd) = 0;
                virtual word Send(const std::string& header, const
std::string& data) = 0;
                virtual word Send(const std::string& header, const
std::string& data, byte le) = 0;

                // Extended APDU
                virtual word SendX(const Blob& header, const Blob& data) =
0;
                virtual word SendX(const Blob& header, const Blob& data,
unsigned short le) = 0;
                virtual word SendX(const std::string& header, const
std::string& data) = 0;
                virtual word SendX(const std::string& header, const
std::string& data, unsigned short le) = 0;

                // Secure Pin Entry
                virtual word SendVerifyToReader(const Blob& apdu, byte
min_pinsize, byte max_pinsize) = 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

```
virtual long GetProtocol() const = 0;

virtual void SetProtocol(long protocol) = 0;
```

Liest/setzt das zu verwendende Übertragungsprotokoll.

```
virtual ulong GetTimeout() const throw() = 0;

virtual bool SetTimeout(ulong timeout_msec) = 0;
```

Liest/setzt den Timeout für Tokenoperationen

```
virtual void Open() = 0;

virtual void Close() throw() = 0;

virtual void ResetCard() = 0;

virtual long CancelCardOperation() = 0;
```

```
virtual void BeginTransaction() = 0;

virtual void EndTransaction(DispositionType disposition = LEAVE_CARD) = 0;

virtual ulong GetTransactionDepth() const throw() = 0;
```

Eröffnet/beendet eine Tokentransaktion. Solange eine Transaktion offen ist, werden konkurrierende Zugriffe auf das Token angehalten.

Transaktionen lassen sich schachteln; `GetTransactionDepth()` gibt die derzeitige Schachtelungstiefe wieder.

```
virtual word SendCard(const Blob& cmd, Blob& response) = 0;
```

Sendet einen Befehl an das Token und liest die Rückgabe aus.

```
virtual const ISlot* GetSlot() const = 0;
```

Gibt einen Verweis auf das Slot-Objekt zurück, zu dem dieses Objekt gehört.

```
virtual Blob GetResponse() const = 0;
```

Liefert das Ergebnis des letzten Befehls zurück.

```
virtual word Send(const Blob& cmd) = 0;

virtual word Send(const Blob& header, const Blob& data) = 0;

virtual word Send(const Blob& header, const Blob& data, byte le) = 0;

virtual word Send(const std::string& cmd) = 0;

virtual word Send(const std::string& header, const std::string& data) = 0;

virtual word Send(const std::string& header, const std::string& data, byte
le) = 0;
```

Sendet eine APDU an das Token, je nach Aufrufform mit zusätzlichen Daten und einer ‚length expected'-Information, einer Angabe, wieviele Bytes als Rückgabe erwartet werden.

```
virtual word SendX(const Blob& header, const Blob& data) = 0;

virtual word SendX(const Blob& header, const Blob& data, unsigned short le)
= 0;

virtual word SendX(const std::string& header, const std::string& data) = 0;

virtual word SendX(const std::string& header, const std::string& data,
unsigned short le) = 0;
```

Sendet eine Extended-APDU an das Token, äquivalent zu Send(...)

NOTIZ: Je nach Karte, Leser und Treiber werden Extended APDU's u.U. nicht unterstützt und der Versuch, eine solche zu senden wird mit einer „Send error"-Smartcard-Exception quittiert!

```
virtual word SendVerifyToReader(const Blob& apdu, byte min_pinsize, byte
max_pinsize) = 0;
```

Schickt einen (vorbereiteten) Befehl zum Einloggen auf das Token mittels PinPad an den Kartenleser.

7.3.12.4 ISCardOS

Diese Klasse ermöglicht den Zugriff auf das COS einer Smartcard, basierend auf Befehlen, die ISO7816-4 bzw. ISO7816-8 konform sind.

CSP- und PKCS11-Systeme haben eigene Zugriffsmechanismen auf eine Smartcard. Eine SCardOS-Klasse ist in diesem Fall nicht erforderlich.

Im folgenden werden zunächst alle Funktionen aufgelistet und danach zu einigen Funktionen ergänzende Kommentare gegeben.

```
        class ISCardOS
                : public IRefCounted
                , public IAcConverter
        {
        public:
                virtual bool IsMoC() = 0;
                virtual const char* GetName() const = 0;
                virtual Blob GetSerialNumber() const = 0;
                virtual Blob GetCardId(unsigned short fid_card_id) throw() =
0;
                virtual Blob GetCardCF(unsigned short fid_card_cf) throw() =
0;

                virtual ISCardAccess* GetAccess() const = 0;

                virtual Blob ComputeDigitalSignature(const Blob& data) = 0;
                virtual Blob ComputeDigitalSignature(IEMSAAlg* emsa) = 0;
                virtual Blob Encipher(const Blob& plaintext) = 0;
                virtual Blob Decipher(const Blob& encrypted_data) = 0;
                virtual Blob AgreeKey(const Blob& pubkey) = 0;

                virtual Blob SelectFile(const int filetype, const char*
filepath) = 0;
                virtual Blob SelectFile(const int filetype, const Blob&
filepath) = 0;
                virtual Blob SelectFile(const int filetype, unsigned short
fid) = 0;
                virtual void SelectApp(const Blob& aid) = 0;
                virtual void SelectAID(int nr = 1) = 0;
                virtual Blob GetAID(int nr = 1) = 0;

                virtual int GetEFSize(unsigned short fid) = 0;
                virtual int GetEFSize(const Blob& filepath, Blob& fci) = 0;
                virtual void GetFCI(unsigned short fid, Blob& fci) = 0;
                virtual Blob GetFCIValue(const Blob& fci, const byte tag) =
0;

                virtual Blob ReadBinary(const char* filepath) = 0;
                virtual Blob ReadBinary(const Blob& filepath) = 0;
                virtual Blob ReadBinary(unsigned short fid) = 0;

                // read/write binary, file is selected, file size is known
                virtual Blob ReadSelectedBinary(size_t file_size, unsigned
short offset = 0) = 0;
                virtual void UpdateSelectedBinary(const Blob& data, unsigned
short offset = 0) = 0;

                // binary write (includes resize)
                virtual void UpdateBinary(const Blob& filepath, const Blob&
data, unsigned short offset = 0) = 0;
```

```
                virtual void UpdateBinary(unsigned short fid, const Blob&
data, unsigned short offset = 0) = 0;

                // record read/write (file must be selected)
                virtual Blob ReadRecord(byte mode) = 0;
                virtual Blob ReadRecord(byte rec_id, byte mode) = 0;
                virtual void UpdateRecord(const Blob& data, byte rec_id,
byte mode) = 0;
                virtual void AppendRecord(const Blob& data, byte ef_id = 0)
= 0;

                // record write (includes file selection and resize)
                virtual void UpdateRecord(const Blob& filepath, const Blob&
data, byte rec_id, byte mode) = 0;
                virtual void AppendRecord(const Blob& filepath, const Blob&
data) = 0;

                virtual void DeleteEF(const Blob& fid) = 0;
                virtual void DeleteEF(unsigned short fid) = 0;

                virtual void CreateEF(const Blob& fci) = 0;
                virtual void CreateEF(const Blob& fid, unsigned short len,
bool secure = false) = 0;
                virtual void CreateEF(unsigned short fid, unsigned short
len, bool secure = false) = 0;

                virtual unsigned short GetOSFileOffset() const throw() = 0;

                virtual Blob GetChallenge(byte le = byte(8)) const = 0;
                virtual Blob GetRandom(const unsigned long count) const = 0;

                virtual int GetTokenProfileType() = 0;

                virtual void SetTokenConfig(ITokenConfig* tkcfg) = 0;
                virtual ITokenConfig* GetTokenConfig() const = 0;

                // set behavior
                virtual void SetBehavior(ISCardOSBehavior* behave) = 0;

                // access rights
                virtual void SetAccessRights(const Blob& access_rights) = 0;
                virtual void SetExplicitRights(IAccessCondition* ac, size_t
ac_cnt = 1) = 0;

                virtual bool HasExplicitRights() const = 0;

                virtual Blob GetAccessRights() = 0;
                virtual IAccessCondition* GetAccessRights(const Blob& fci) =
0;

                // selected path
                virtual void SetSelectedPath(ITokenFile* selected_path)
const throw() = 0;
                virtual ITokenFile* GetSelectedPath() const throw() = 0;

                // key handling
                virtual void GenerateRSAKeyPair(int key_id,   size_t
key_size /* in bits */, Blob& pubkey) = 0;

                virtual void ImportRSAKeyPair(int key_id, const Blob&
prime_p, const Blob& prime_q,
                        const Blob& priv_exp, Blob& pubkey) = 0;
```

```
                virtual void ImportDESKey(int key_id, const Blob& keyblob) =
0;

                virtual void DeletePrivateKeyObj(unsigned short fid) = 0;
                virtual void DeleteBlockCipherKeyObj(unsigned short fid) =
0;
                virtual void SelectKey(int key_id) = 0;

                // authentication functions
                virtual Blob ComputeResponse(const Blob& auth_key, const
Blob& challenge) const = 0;
                virtual void ExternalAuthenticate(byte key_id, const Blob&
response) = 0;
                virtual void UpdateExternalAuthenticateKey(byte key_id,
const Blob& auth_key) = 0;

                virtual void ResetSecurityStatus() = 0;

                // token initialization
                virtual void SetDefaultLabel(const char* label) = 0;
                virtual ITokenInitializer* GetTokenInitializer() = 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual bool IsMoC() = 0;`

True, wenn das Token Biometrie unterstützt.

`virtual const char* GetName() const = 0;`

`virtual Blob GetSerialNumber() const = 0;`

`virtual Blob GetCardId(unsigned short fid_card_id) throw() = 0;`

`virtual Blob GetCardCF(unsigned short fid_card_cf) throw() = 0;`

Liest die grundlegenden Eckdaten vom Token aus.

`virtual ISCardAccess* GetAccess() const = 0;`

Gibt ein ScardAccess-Objekt zurück, was den Zugriff auf das Token auf APDU-Ebene ermöglicht.

`virtual Blob ComputeDigitalSignature(const Blob& data) = 0;`

`virtual Blob ComputeDigitalSignature(IEMSAAlg* emsa) = 0;`

`virtual Blob Encipher(const Blob& plaintext) = 0;`

`virtual Blob Decipher(const Blob& encrypted_data) = 0;`

`virtual Blob AgreeKey(const Blob& pubkey) = 0;`

Grundelegende kryptographische Operationen.

```
virtual Blob SelectFile(const int filetype, const char* filepath) = 0;

virtual Blob SelectFile(const int filetype, const Blob& filepath) = 0;

virtual Blob SelectFile(const int filetype, unsigned short fid) = 0;

virtual void SelectApp(const Blob& aid) = 0;

virtual void SelectAID(int nr = 1) = 0;

virtual Blob GetAID(int nr = 1) = 0;
```

Selektiert eine Datei oder Applikation anhand der gegebenen Informationen (Typ, Pfad, AID oder Index in der Liste von Applikationen)

```
virtual int GetEFSize(unsigned short fid) = 0;

virtual int GetEFSize(const Blob& filepath, Blob& fci) = 0;

virtual void GetFCI(unsigned short fid, Blob& fci) = 0;

virtual Blob GetFCIValue(const Blob& fci, const byte tag) = 0;
```

Gibt die kompletten File Control Information einer Datei oder das gewählte Attribut aus diesen zurück.

```
virtual Blob ReadBinary(const char* filepath) = 0;

virtual Blob ReadBinary(const Blob& filepath) = 0;

virtual Blob ReadBinary(unsigned short fid) = 0;
```

Liest eine komplette (transparente) Datei vom Token und gibt sie als Blob zurück

```
virtual Blob ReadSelectedBinary(size_t file_size, unsigned short offset =
0) = 0;

virtual void UpdateSelectedBinary(const Blob& data, unsigned short offset =
0) = 0;
```

Liest von/schreibt in eine selektierte (s.o.) Datei.

```
virtual void UpdateBinary(const Blob& filepath, const Blob& data, unsigned
short offset = 0) = 0;

virtual void UpdateBinary(unsigned short fid, const Blob& data, unsigned
short offset = 0) = 0;
```

Schreibt in eine Datei, die mit Namen oder mit FID spezifiziert ist.

```
virtual Blob ReadRecord(byte mode) = 0;
```

```
virtual Blob ReadRecord(byte rec_id, byte mode) = 0;
```

```
virtual void UpdateRecord(const Blob& data, byte rec_id, byte mode) = 0;
```

```
virtual void AppendRecord(const Blob& data, byte ef_id = 0) = 0;
```

```
virtual void UpdateRecord(const Blob& filepath, const Blob& data, byte
rec_id, byte mode) = 0;
```

```
virtual void AppendRecord(const Blob& filepath, const Blob& data) = 0;
```

Liest/schreibt Datensätze in eine in Records unterteilte Datei, die (je nach Aufruf) entweder selektiert sein muss oder durch Filepath spezifiziert wird.

```
virtual void DeleteEF(const Blob& fid) = 0;
```

```
virtual void DeleteEF(unsigned short fid) = 0;
```

Löscht eine Datei von dem Token.

```
virtual void CreateEF(const Blob& fci) = 0;
```

```
virtual void CreateEF(const Blob& fid, unsigned short len, bool secure =
false) = 0;
```

```
virtual void CreateEF(unsigned short fid, unsigned short len, bool secure =
false) = 0;
```

Erstellt eine Datei auf dem Token.

```
virtual void SetAccessRights(const Blob& access_rights) = 0;
```

Setzt die grundlegende Maske an Zugriffsrechten.

```
virtual void SetExplicitRights(IAccessCondition* ac, size_t ac_cnt = 1) =
0;
```

Setzt ein AccessCondition-Objekt, dass die Zugriffsrechte für die nächsten `ac_cnt` geschriebenen Objekte festlegt.

```
virtual IAccessCondition* GetAccessRights(const Blob& fci) = 0;
```

Gibt ein AccessCondition-Objekt zurück, dass die Zugriffsrechte auf die Datei, dessen File Control Information übergeben wurden, beschreibt.

```
virtual void SetSelectedPath(ITokenFile* selected_path) const throw() = 0;

virtual ITokenFile* GetSelectedPath() const throw() = 0;
```

Setzt/liest den selektierten Pfad aus.

```
virtual void SetDefaultLabel(const char* label) = 0;
```

Setzt einen Standardlabel für neu initialisierte Tokens

```
virtual ITokenInitializer* GetTokenInitializer() = 0;
```

Gibt ein TokenInitializer-Objekt zurück, mit dem ein Token neu initialisiert, d.h. ein neues Profil auf dem Token installiert werden kann.

7.3.12.5 IToken

Diese Klasse beschreibt das Profil eines Token und verwaltet die Objekte auf dem Token, insbesondere die Liste der privaten Schlüssel, PINs und Zertifikate.

Die Klasse `IToken` arbeitet intern mit einem Referenz-Zähler, um nicht mehr benötigten Speicher automatisch wieder freigeben zu können. Im folgenden werden zunächst alle Funktionen aufgelistet und danach zu einigen Funktionen ergänzende Kommentare gegeben.

```
class IToken
      : public IRefCounted
      , public IAcConverter
{
protected:
      virtual ~IToken() { }

public:
      virtual ITokenConfig* GetConfig() const = 0;
      virtual ITokenFile* GetTokenFile(const Blob& path) = 0;

      virtual int GetStatus() const = 0;
      virtual int GetType() const = 0;
      virtual const char* GetName() const = 0;
      virtual Blob GetSerialNumber() const = 0;
      virtual Blob GetCardholderName() const = 0;
      virtual Blob GetRandom(const unsigned long count) const = 0;
      virtual void SetSeed(const Blob& seed) = 0;

      virtual IKey* NewKey(int key_type = KEY_RSA) = 0;
      virtual IKey* WriteKey(const IKey* key) = 0;

      virtual IKey* CreateKey(const Blob& cert) const = 0;
```

```
            virtual IKey* GetKey(int pos) const = 0;

            virtual int GetKeyNumber() const = 0;


            virtual CertEntry GetCertificate(int pos) const = 0;

            virtual int GetCertificateNumber() const = 0;


            virtual void WriteCertificate(const Blob& cert, const IKey* key
= 0) = 0;

            virtual void DeleteCertificate(const Blob& cert, const IKey*
key = 0) = 0;


            virtual ITokenAuth* GetAuth() const = 0;

            virtual ITokenAuthManager* GetAuthManager(int authManagerType)
= 0;

            virtual ITokenAuthIterator* GetAuthIterator(int type, const
Blob& authId = Blob()) = 0;


            virtual ITokenPIN* GetPIN(const Blob& authId) const = 0;

            virtual ITokenPIN* GetPIN(int pos) const = 0;

            virtual int GetPINNumber() const = 0;


            virtual ITokenPIN* GetUserPin() const = 0;

            virtual ITokenPIN* GetSOPin() const = 0;

            virtual void VerifyUserPin(const Blob& pinvalue) = 0;

            virtual void ChangeUserPin(const Blob& oldpin, const Blob&
newpin) = 0;

            virtual void UnlockUserPin(const Blob& so_pin, const Blob&
new_userpin) = 0;

            virtual void VerifySOPin(const Blob& pinvalue) = 0;

            virtual void ChangeSOPin(const Blob& oldpin, const Blob&
newpin) = 0;
```

```
        virtual bool ResetSecurityState() const = 0;


        virtual ISCardOS* GetOS() const = 0;


        virtual void DeleteKeyPair(const IKey* key) = 0;

        virtual void DeleteKeyPair(const Blob& cert) = 0;


        virtual bool IsPKCS15() const throw() = 0;

        virtual bool IsReadOnly() const throw() = 0;

        virtual bool LoginRequired() const throw() = 0;


        virtual bool IsLocked() const throw() = 0;


        virtual Blob GetCardId() const throw() = 0;

        virtual Blob GetCardCF() const throw() = 0;

    };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:
(NOTIZ: Je nach Art des Tokens können diverse Funktionen nicht unterstützt sein)

`virtual int GetStatus() const = 0;`

Gibt den Produktionszustand des Tokens (leer, initialisiert, personalisiert, …) an.

`virtual int GetType() const = 0;`

`virtual const char* GetName() const = 0;`

`virtual Blob GetSerialNumber() const = 0;`

`virtual Blob GetCardholderName() const = 0;`

Gibt diverse Eckdaten (Typ, Seriennummer, Name, ...) des Tokens zurück, soweit verfügbar und im Tokenprofil unterstützt.

```
virtual Blob GetRandom(const unsigned long count) const = 0;

virtual void SetSeed(const Blob& seed) = 0;
```

Initialisiert den tokeneigenen Zufallszahlengenerator (sofern das Token es vorsieht) und liest `count` Zufallszahlen aus diesem aus.

```
virtual IKey* NewKey(int key_type = KEY_RSA) = 0;

virtual IKey* WriteKey(const IKey* key) = 0;

virtual IKey* CreateKey(const Blob& cert) const = 0;
```

Generiert einen neuen Schlüssel (`NewKey`) auf dem Token, importiert einen Schlüssel (`WriteKey`) auf dem Token oder erstellt diesen anhand des gegebenen Zertifikats (`CreateKey`).

Die Art und die Länge des zu erstellenden/importierenden Schlüssels muss vom Token unterstützt werden-

```
virtual IKey* GetKey(int pos) const = 0;

virtual int GetKeyNumber() const = 0;
```

Liest die Anzahl der existierenden Schlüssel aus (`GetKeyNumber`) und gibt den nten Schlüssel (`GetKey`) zurück, wenn die Zugriffsrechte dieses erlauben.

```
virtual CertEntry GetCertificate(int pos) const = 0;

virtual int GetCertificateNumber() const = 0;
```

Liest die Anzahl der existierenden Zertifikate aus (`GetCertificateNumber`) und gibt das nte Zertifikat (`GetCertificate`) zurück, wenn die Zugriffsrechte dieses erlauben.

```
virtual void WriteCertificate(const Blob& cert, const IKey* key = 0) = 0;

virtual void DeleteCertificate(const Blob& cert, const IKey* key = 0) = 0;
```

Schreibt / löscht ein Zertifikat auf dem Token.

```
virtual ITokenAuth* GetAuth() const = 0;
```

Gibt das TokenAuth-Objekt zum angegebenen Token zurück.

```
virtual ITokenAuthManager* GetAuthManager(int authManagerType) = 0;
```

Gibt das TokenAuthManager-Objekt zum angegebenen Token zurück.

```
virtual ITokenAuthIterator* GetAuthIterator(int type, const Blob& authId =
Blob()) = 0;
```

Gibt einen Iterator zurück, der über die PINs mit den angegebenen Kriterien zählt.

```
virtual ITokenPIN* GetPIN(int pos) const = 0;

virtual int GetPINNumber() const = 0;
```

Gibt die Anzahl aller PINs / die nte PIN zurück.

```
virtual ITokenPIN* GetPIN(const Blob& authId) const = 0;

virtual ITokenPIN* GetUserPin() const = 0;

virtual ITokenPIN* GetSOPin() const = 0;

virtual void VerifyUserPin(const Blob& pinvalue) = 0;

virtual void ChangeUserPin(const Blob& oldpin, const Blob& newpin) = 0;

virtual void UnlockUserPin(const Blob& so_pin, const Blob& new_userpin) =
0;

virtual void VerifySOPin(const Blob& pinvalue) = 0;

virtual void ChangeSOPin(const Blob& oldpin, const Blob& newpin) = 0;

virtual bool ResetSecurityState() const = 0;
```

Diese Funktionen sind aus Kompatiblitätsgründen im Interface enthalten und sollten in neuem Programmcode nicht mehr verwendet werden. Bitte verwenden Sie hierzu das TokenAuth-Objekt (s.o.) und die damit verbundenen TokenPIN-Objekte.

```
virtual ISCardOS* GetOS() const = 0;
```

Gibt das zum Token und dessen Profil passende OS-Objekt zurück, mit dem Zugriffe auf der Dateisystemebene des Tokens möglich sind.

```
virtual void DeleteKeyPair(const IKey* key) = 0;
```

```
virtual void DeleteKeyPair(const Blob& cert) = 0;
```

Löscht das angegebene (oder zum Zertifikat gehörende) Schlüsselpaar vom Token.

```
virtual bool IsPKCS15() const throw() = 0;

virtual bool IsReadOnly() const throw() = 0;

virtual bool LoginRequired() const throw() = 0;

virtual bool IsLocked() const throw() = 0;
```

Gibt an, ob das Token ein PKCS15-Profil besitzt / schreibgeschützt ist / ob eine Authentisierung nötig ist / das Token gesperrt ist.

7.3.12.6 ITokenPIN

Mit Hilfe dieser Klasse können die Operationen im Zusammenhang mit den PINs durchgeführt werden. Im folgenden werden zunächst alle Funktionen aufgelistet und danach zu einigen Funktionen ergänzende Kommentare gegeben.

```
        class ITokenPIN : public IRefCounted
        {
        public:
                virtual ITokenPIN* Clone() const = 0;
                virtual bool Equals(const ITokenPIN* other) const throw() =
0;

                virtual const char* GetName() const = 0;
                virtual int GetUsage() const = 0;
                virtual int GetType() const = 0;

                virtual bool IsPinInited() const = 0;
                virtual bool NeedsUpdate() const = 0;
                virtual bool GetLastChange(Date& date, bool& supported)
const = 0;
                virtual bool GetLengthInfo(int& min_pinlen, int& max_pinlen,
int& stored_pinlen) const = 0;
                virtual bool CheckPinLength(const Blob& pin) const = 0;

                virtual void VerifyPin(const Blob& pin) const = 0;

                virtual void ChangePin(const Blob& oldpin, const Blob&
newpin) const = 0;

                virtual void UnlockPin(const ITokenPIN* so, const Blob& pin,
const Blob& newpin) const = 0;

                virtual void SetPinValue(const ITokenPIN* so, const Blob&
so_pin, const Blob& newpin) const = 0;
                virtual bool NeedsPINValue() const = 0;

                virtual ITokenPIN* GetParentPin() const = 0;

                virtual IToken* GetToken() const throw() = 0;
                virtual ITokenFile* GetPath() const throw() = 0;

                virtual ITokenAuthRef* GetAuthRef() const throw() = 0;
                virtual ITokenAuthRef* GetParentAuthRef() const throw() = 0;

                // Bio extensions
                virtual byte GetFinger() const throw() = 0;
                virtual bool GetBioHeader(Blob& bioheader) = 0;

                // ExternalAuth Key extensions
                virtual Blob GetChallenge() const = 0;
                virtual void ResetChallenge() const = 0;
                virtual Blob ComputeResponse(const Blob& auth_key, const
Blob& challenge) const = 0;

        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

```
virtual ITokenPIN* Clone() const = 0;
```

Erzeugt eine Kopie des TokenPIN-Objekts.

```
virtual bool Equals(const ITokenPIN* other) const throw() = 0;
```

Prüft auf Gleichheit dieses und des anderen TokenPIN-Objekts

```
virtual const char* GetName() const = 0;
```

Gibt einen Bezeichner (‚User PIN', ‚SO Pin', ...) zum PIN-Objekt zurück

```
virtual int GetUsage() const = 0;
```

Gibt zurück, wofür die PIN eingesetzt wird.

```
virtual int GetType() const = 0;
```

Gibt ein Bitfield mit den entsprechenden Typkennzeichnungen (User-PIN, SO-PIN, External Auth PIN, ...) der PIN zurück.

```
virtual bool IsPinInited() const = 0;

virtual bool NeedsUpdate() const = 0;
```

Gibt an, ob die PIN bereits initialisiert ist / geändert werden muß

```
virtual bool GetLastChange(Date& date, bool& supported) const = 0;
```

Gibt das Datum der letzten PIN-Änderung zurück (und ob diese Information überhaupt verfügbar ist)

```
virtual bool GetLengthInfo(int& min_pinlen, int& max_pinlen, int&
stored_pinlen) const = 0;
```

Gibt die minimale, maximale und aktuelle PIN-Länge zurück, wenn verfügbar.

```
virtual bool CheckPinLength(const Blob& pin) const = 0;
```

Prüft die Länge der übergebenen PIN anhand der o.a. Eckdaten.

virtual void VerifyPin(const Blob& pin) const = 0;

Prüft die übergebene PIN. Der Versuchszähler des Tokens wird herruntergezählt, wenn die PIN falsch ist.

virtual void ChangePin(const Blob& oldpin, const Blob& newpin) const = 0;

Ändert die PIN dieses TokenPIN-Objekts.

virtual void UnlockPin(const ITokenPIN* so, const Blob& pin, const Blob& newpin) const = 0;

Hebt die Sperrung der PIN dieses TokenPIN-Objekts mit Hilfe der Entsperr-PIN (üblicherweise das TokenPIN-Objekts zur SO-Pin) auf.

Die Funktion `GetParentPIN()` liefert das für die Entsperrung nötige TokenPIN-Objekt zurück.

virtual void SetPinValue(const ITokenPIN* so, const Blob& so_pin, const Blob& newpin) const = 0;

Setzt mit Hilfe des authentisierenden Objekts (üblicherweise TokenPIN-Objekt der SO-PIN sowie die SO-PIN selbst, s.o.) eine neue PIN für das aktuelle TokenPIN-Objekt.

virtual bool NeedsPINValue() const = 0;

Gibt an, ob zur Authentisierung deses TokenPIN-Objekts eine PIN vom Programm übergeben werden muss. So ist es z.B. bei Tokens in einem PinPad-Kartenleser nicht erforderlich und i.d.R. auch nicht wünschenswert.

virtual ITokenPIN* GetParentPin() const = 0;

Gibt das TokenPIN-Objekt zurück, das (Schreib-)Zugriff auf dieses PIN-Objekt gewährt.

virtual IToken* GetToken() const throw() = 0;

Liefert einen Rückverweis auf das entsprechende Token-Objekt.

7.3.12.7 IEventHandler

Diese Klasse definiert die Callback-Funktionen, die in der Funktion

```
   void NotifyEvent(IEventHandler* cmd);
```

von `ISlot` und `ISubSystem` verwendet werden.

Folgende Funktionen sind definiert:

```
class IEventHandler
{
public:
        virtual ~IEventHandler() {}

        virtual void OnEvent(int event) = 0;
};
```

Ergänzende Anmerkungen zu einer Funktion:

- `void OnEvent(int event)`
implementiert die Aktion bei den Ereignissen SLOT_REMOVED, SLOT_INSERTED, TOKEN_REMOVED und TOKEN_INSERTED.

7.3.12.8 ISlotMonitor

Diese Klasse überwacht einen Slot ständig. In der Regel wird ein weiterer Thread gestartet, der den Status des Slot regelmäßig abfragt. Selbstverständlich setzt dieses voraus, dass die Anwendung, die diese Funktion benutzt, als multithread Code erstellt wurde.

Folgende Funktionen sind definiert:

```
class ISlotMonitor
{
public:
        virtual ~ISlotMonitor() {}
        virtual void Start(IEventHandler* cmd) = 0;
        virtual void Stop() = 0;
        virtual bool Interrupted() const = 0;
};
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

- `void Start(IEventHandler* cmd)`

startet die Überwachung.

- `void Stop()`

stoppt die Überwachung.

- `bool Interrupted() const`

prüft, ob der Thread beendet werden soll.

7.3.12.9 ITokenAuth

Klassen mit diesem Interface dienen zur Benutzung existierender Zugriffsrechte des Tokens.

```
        class ITokenAuth : public IRefCounted
        {
        public:
                virtual bool Login(const ITokenPIN* token_pin, const Blob&
pin) = 0;

                virtual bool Logout() = 0;
                virtual bool Logout(const ITokenPIN* token_pin) = 0;
                virtual bool LogoutEx(bool reset_security_state) = 0;

                virtual Blob GetChallenge(const ITokenPIN* token_pin) = 0;

                virtual IToken* GetToken() const throw() = 0;
                virtual const ITokenPIN* GetPin() const throw() = 0;
                virtual const ITokenPIN* GetPin(size_t index) const throw()
= 0;

                virtual bool NeedsPINValue() const throw() = 0;
                virtual bool HasChallenge() const throw() = 0;
                virtual bool HasChallenge(const ITokenPIN* token_pin) const
throw() = 0;
                virtual bool IsAuthenticated() const throw() = 0;
                virtual bool IsAuthenticated(const ITokenPIN* token_pin)
const throw() = 0;

                virtual bool ClearAuthState(const ITokenPIN* token_pin) = 0;
                virtual bool SelectPin(const ITokenPIN* token_pin, bool
bForce) = 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual bool Login(const ITokenPIN* token_pin, const Blob& pin) = 0;`

Benutzt ein TokenPIN-Objekt und die angegebene PIN (alphanumerisch, Biometrie-Template, Response,...) um einen Benutzer gegenüber dem Token zu authentisieren.

`virtual bool Logout() = 0;`

Nimmt alle erteilten Nutzungsberechtigungen auf dem Token zurück.

`virtual bool Logout(const ITokenPIN* token_pin) = 0;`

Versucht allein die Nutzungsberechtigungen, die durch das angegebene TokenPIN-Objekt gegeben wurden, wieder zurückzunehmen.

virtual Blob GetChallenge(const ITokenPIN* token_pin) = 0;

Holt sich ein zum TokenPIN-Objekt gehörendes Challenge von der Karte. Das Programm muß mit Hilfe des geteilten Geheimnisses daraus den Response errechnen und mit Login(token_pin, response) den Einlogg-Vorgang komplettieren.

virtual IToken* GetToken() const throw() = 0;

Liefert ein Rückverweis zum Token-Objekt, zu dem dieses TokenAuth-Objekt gehört.

virtual const ITokenPIN* GetPin() const throw() = 0;

virtual const ITokenPIN* GetPin(size_t index) const throw() = 0;

Gibt das (xte) Pin-Objekt zurück, das zu diesem TokenAuth-Objekt (und damit Troken) gehört.

virtual bool NeedsPINValue() const throw() = 0;

Gibt an, ob zum Einloggen auf diesem Token eine (alphanumerische) PIN übergeben werden muss. Ist das Token z.B. in einem PinPad-Kartenleser eingesteckt und die Benutzung desselben gestattet und möglich, liefert diese Funktion ein false zurück, damit das Programm nicht selbst nach einer PIN fragen muss. In diesem Fall wird bei Login(...) ein leeres Blob übergeben.

virtual bool HasChallenge() const throw() = 0;

virtual bool HasChallenge(const ITokenPIN* token_pin) const throw() = 0;

Gibt an, ob (zum TokenPIN-Objekt oder Token) ein Challenge vorhanden ist, d.h. ob der Rückgabewert von GetChallenge() gültig ist und die Challenge-Response-Authentisierung verwendet werden kann.

virtual bool IsAuthenticated() const throw() = 0;

virtual bool IsAuthenticated(const ITokenPIN* token_pin) const throw() = 0;

Gibt den Authentisierungsstatus (des Tokens oder der jeweiligen PIN) zurück.

virtual bool ClearAuthState(const ITokenPIN* token_pin) = 0;

Löscht den Authentisierungszustand des TokenAuth-Objekts.

HINWEIS: Ändert nicht den Authentisierungszustand des Tokens selbst!

7.3.12.10 ITokenAuthManager

Diese Interfaces dienen zur Handhabung der PIN-Objekte selbst.

```
        class ITokenAuthManager : public IRefCounted
        {
        public:
                virtual AuthManagerType getType() const = 0;
                virtual void deleteAuthObj(ITokenPIN* authObj) = 0;
        };

        class ITokenAuthPINManager : public ITokenAuthManager
        {
        public:
                virtual ITokenPIN* createAuthObj(ITokenAuthRef* authRef,
ITokenAuthRef* parentAuthRef,
                        int retryCount, const Blob& pin) = 0;
        };

        class ITokenAuthBioManager : public ITokenAuthManager
        {
        public:
                virtual ITokenPIN* createAuthObj(ITokenAuthRef* authRef,
ITokenAuthRef* parentAuthRef,
                        int retryCount, int bioFinger) = 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual AuthManagerType getType() const = 0;`

Gibt den Typ des AuthManagers (PIN, Biometrie, ...) zurück. Der Typ entscheidet, welches der abgeleiteten Interfaces zu benutzen ist.

`virtual void deleteAuthObj(ITokenPIN* authObj) = 0;`

Löscht ein TokenPIN-Objekt. Ein Einloggen mit der im TokenPIN-Objekt festgelegten Methode (z.B. der Fingerabdruck, der darin gespeichert ist) ist daraufhin nicht mehr möglich.

`virtual ITokenPIN* createAuthObj( . . . ) = 0;`

Erstellt ein neues TokenPIN-Objekt. Dabei ‚erbt' das neue TokenPIN-Objekt die Zugriffsrechte von TokenPINs, auf die mit authref verwiesen wird. Parentauthref wird für die PINsverwendet, die erforderlich ist, um das neue TokenPIN-Objekt zu entsperren, wenn es denn mal gesperrt ist.

7.3.12.11 IAccessCondition

Dieses Interface dient zur Handhabung von Zugriffsberechtigungen auf einzelne Objekte auf dem Token.

```
        class IAccessCondition : public IRefCounted
        {
        public:
                virtual IAccessCondition* Clone() const = 0;

                virtual const char* GetName() const throw() = 0;
                virtual ACType GetAcType() const throw() = 0;

                virtual Blob& GetAccessRights() throw() = 0;
                virtual const Blob& GetAccessRights() const throw() = 0;

                virtual IAccessCondition* ConvertAc(const IToken* token,
ACType acdest) const = 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual IAccessCondition* Clone() const = 0;`

Erstellt eine Kopie des AccessCondition-Objekts.

`virtual const char* GetName() const throw() = 0;`

Gibt (sofern vorhanden) einen symbolischen Namen für den angegebenen Satz Zugriffsrechte zurück

`virtual ACType GetAcType() const throw() = 0;`

Gibt zurück, in welcher Art die Zugriffsrechte in diesem Objekt gespeichert werden

`virtual Blob& GetAccessRights() throw() = 0;`

`virtual const Blob& GetAccessRights() const throw() = 0;`

Gibt ein Blob mit den Zugriffsrechten selbst zurück. Das Format wird durch `GetAcType()` bestimmt.

`virtual IAccessCondition* ConvertAc(const IToken* token, ACType acdest) const = 0;`

Konvertiert die Zugriffsrechte von eine Darstellung in die andere. Wählbar sind `AC_TYPE_RAW` für eine Tokenspezifische Repräsentation und `AC_TYPE_GENERIC` für eine allgemeine Repräsentation.

7.3.12.12 ITokenInitializer

Diese Klasse bietet Funktionen, um das Profil eines Tokens komplett zu löschen und ein neues Profil gemäß den gewählten Einstellungen zu installieren.

```
        class ITokenInitializer : public IRefCounted
        {
        public:
                virtual const TITokenInfo& getTokenInfo() const throw() = 0;
                virtual const TokenProfileType* getSupportedProfileList()
const throw() = 0;
                virtual bool supportsProfile(TokenProfileType profile_type)
const throw() = 0;

                virtual bool eraseProfile(const Blob& card_pin) = 0;
                virtual bool existProfile(bool& is_pkcs15, bool&
requires_adminpin) = 0;
                virtual void createProfile(TIData& data, TokenProfileType
profile_type) = 0;
                virtual void finalizeProfile(TIData& data) = 0;

                virtual void pinChangeReminder(TIData& data, bool enable) =
0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual const TITokenInfo& getTokenInfo() const throw() = 0;`

Gibt eine Struktur zurück, die die Eckdaten des Tokens (PIN-Längen, ...) beschreibt.

`virtual const TokenProfileType* getSupportedProfileList() const throw() = 0;`

Gibt eine Liste aller unterstützten Profile für das jeweilige Token zurück.

`virtual bool supportsProfile(TokenProfileType profile_type) const throw() = 0;`

`True`, wenn das Token das übergebene Profil unterstützt.

`virtual bool eraseProfile(const Blob& card_pin) = 0;`

Löscht das existierende Profil auf dem Token (soweit möglich). Benötigt ggf. die Token-PIN (nicht SO-PIN, nicht User-PIN) um diese Operation durchzuführen.

`virtual bool existProfile(bool& is_pkcs15, bool& requires_adminpin) = 0;`

Gibt `true` zurück, wenn ein Profil auf dem Token installiert ist (ein `eraseProfile()` vor dem Installieren eines neuen Profils nötig sei). Gibt weiterhin zurück, ob es sich um ein PKCS15-Profil handelt und ob zum Löschen eine Token-PIN gebraucht werden würde.

```
virtual void createProfile(TIData& data, TokenProfileType profile_type) =
0;
```

Initialisiert anhand der übergebenen Daten ein neues Profil. Ein eventuell vorhandenes Profil muss vorher mit `eraseProfile()` gelöscht worden sein.

Die Erstellung muss mit `finalizeProfile()` abgeschlosssen werden.

```
virtual void finalizeProfile(TIData& data) = 0;
```

Beendet das Erstellen eines neuen Profils. Zwischen `createProfile()` und `finalizeProfile()` dürfen – je nach Profil und Token – weitere privilegierte Operationen getätigt werden, wie z.B. das Anlegen eines grundlegenden Dateisystems für die jeweilige Applikation.

```
virtual void pinChangeReminder(TIData& data, bool enable) = 0;
```

Setzt/löscht ein Flag, welches an das Ändern der Benutzer-PIN erinnern soll.

NOTIZ: Dies dient rein informativen Zwecken; es ist stark abhängig vom Token, ob dieser Information noch weitere Bedeutung beigemessen wird oder nicht.

7.3.12.13 ITokenFile

Objekte dieser Klasse repräsentieren Dateien auf den gegebenen Token.

```
        class ITokenFile : public IRefCounted
        {
        public:
                virtual bool IsValid() const throw() = 0;
                virtual const Blob& GetAID() const throw() = 0;

                virtual int GetFileType() const throw() = 0;
                virtual const Blob& GetFilePath() const throw() = 0;

                virtual bool Equals(const ISCardOS* os) const throw() = 0;
                virtual bool Equals(const ITokenFile* other) const throw() =
0;

                virtual void SelectFilePath(ISCardOS* os) = 0;
                virtual void SetFilePath(int filetype, const Blob& filepath)
= 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual bool IsValid() const throw() = 0;`

True, wenn der Inhalt des Objekts gültig ist.

`virtual const Blob& GetAID() const throw() = 0;`

Liefert die Application ID (AID) der Datei.

`virtual int GetFileType() const throw() = 0;`

Liefert den Typ der Datei

`virtual const Blob& GetFilePath() const throw() = 0;`

Gibt den Pfad der Datei als Blob zurück.

`virtual bool Equals(const ISCardOS* os) const throw() = 0;`

`virtual bool Equals(const ITokenFile* other) const throw() = 0;`

Vergleicht zwei Dateien oder ob die gegebene Datei zum OS gehört.

```
virtual void SelectFilePath(ISCardOS* os) = 0;
```

Selektiert die entsprechende Datei.

```
virtual void SetFilePath(int filetype, const Blob& filepath) = 0;
```

Setzt einen neuen Dateipfad.

7.3.12.14 ITokenConfig

Dieses Interface bietet Zugang zu einer Klasse von Hilfsobjekten, die bei der Initialisierung eines neuen Token-Objekts im Speicher gebraucht werden.

```
        class ITokenConfig
                : public IRefCounted
                , public IParam
        {
        public:
                virtual ITokenConfig* Clone() const = 0;

                // Configures the token, if fails throws an Exception.
                virtual void ConfigureToken(IToken* token) const = 0;

                virtual ITokenAuth* CreateTokenAuth(IToken* token) = 0;
                virtual ITokenFileCache* CreateTokenCache(IToken* token) =
0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual ITokenConfig* Clone() const = 0;`

Generiert eine Kopie des aktuellen Objekts.

`virtual void ConfigureToken(IToken* token) const = 0;`

Konfiguriert das übergebene Token.

`virtual ITokenAuth* CreateTokenAuth(IToken* token) = 0;`

Erstellt für das übergebene Token ein TokenAuth-Objekt.

`virtual ITokenFileCache* CreateTokenCache(IToken* token) = 0;`

Erstellt für das übergebene Token einen Dateicache

7.3.12.15 ITokenFileCache

Dieses Inteface wird intern zur Handhabung von Dateien auf dem Token verwendet.

```
        class ITokenFileCache : public IRefCounted
        {
        public:
                virtual bool InitCache(IToken* token) = 0;
                virtual Blob GetCache(const Blob& file_path) const = 0;
                virtual void PutCache(const Blob& file_path, const Blob&
memory) const = 0;
        };
```

7.3.12.16 IPKCS15Behavior

Objekte müssen dieses Interface benutzen, wenn sie tokenspezifische Eigenschaften zur Umsetzung eines PKCS15-Profils auf das jeweilige Token implementieren.

```
        class IPKCS15Behavior : public ISCardOSBehavior
        {
        public:
                virtual int GetTokenType() const throw() = 0;
                virtual void GetProfileCapabilities(ITokenConfig* tkcfg)
const = 0;

                virtual bool IsReadOnly() const throw() = 0;
                virtual const Blob& GetMID() const throw() = 0;
                virtual const Blob& GetRootPath() const throw() = 0;

                virtual ITokenFile* GetTokenFile(const Blob& p15path) = 0;
                virtual ITokenFile* GetTokenFile(const PKCS15Path& p15path)
= 0;
                virtual ITokenPIN* GetPIN(IToken* token, const PKCS15Object&
p15object, bool use_obj_ref = false) = 0;

                // NOTE: If one of these functions return false, the object
gets destroyed
                // and is not added to the corresponding collection,
therefore not accessable!
                virtual bool InitTokenPIN(ITokenPIN* token_pin, const
PKCS15Object& p15object) = 0;
                virtual bool InitTokenKey(ITokenKey* token_key, int usage,
int key_nr, const PKCS15Object& p15object) = 0;
                virtual bool InitTokenKey(ITokenBlockCipherKey* token_key,
int key_nr, const PKCS15Object& p15object) = 0;

                virtual FIDType GetFirstFID(byte obj_type, size_t key_size =
0) const = 0;
                virtual void GetObjectInfo(FIDType fid, Blob& object_path,
byte* obj_ref, byte obj_type, size_t key_size = 0) const = 0;

                virtual FIDType TransformKeyReference(FIDType id, bool
to_on_card_reference = false, bool* bReference = 0) const = 0;

                virtual FIDType GetNativeFID(byte obj_type, const FIDType
fid, byte object_reference = 0) const = 0;

                virtual void GetRelativeFilePath(Blob& absolute_filepath)
const = 0;

                virtual const Blob& GetACTokenRights() const = 0;
                virtual Blob GetAccessRights(const ITokenPIN* token_pin,
byte obj_type) const = 0;
                virtual Blob GetAccessRights(ITokenAuthRef* authRef,
ITokenAuthRef* parentAuthRef, byte obj_type) const = 0;

                virtual ITokenAuthManager* GetAuthManager(IToken* token, int
authManagerType) const = 0;
                virtual void CreateAuthObj(ITokenPIN* token_pin, byte
obj_type) const = 0;

                virtual bool GetNativePubKeySupport() const = 0;
                virtual Blob ReadPublicKey(ISCardOS* os, FIDType fid) const
= 0;
```

```
                virtual FIDType GetNativePubKeyInfo(FIDType priv_key_fid,
const PublicKeyInfo& info,
                        PKCS15Object& p15object) const = 0;
        };
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual int GetTokenType() const throw() = 0;`

Gibt den Typ des Token zurück.

`virtual void GetProfileCapabilities(ITokenConfig* tkcfg) const = 0;`

Gibt eine Struktur zurück, die die Features des Profils beschreibt.

`virtual bool IsReadOnly() const throw() = 0;`

`True`, wenn das Token nur lesbar ist. ‚Nur lesbar' im Sinne dieser Funktion heißt, dass sämtliche Funktionen, die den Inhalt des Token verändern, nicht gestattet sind, selbst diese unabhängig vom System oder Programm durchgeführt werden können.

`virtual const Blob& GetMID() const throw() = 0;`

Gibt die Hersteller-ID (manufacturer id) des Tokens zurück.

NOTIZ: Die Hersteller-ID wird beim Einlesen des Tokens im Speicher dazu verwendet, aus den registrierten ‚Behaviors' das Passende auszusuchen.

Auf der anderen Seite ist es möglich, dasselbe ‚Behavior' unter mehreren Hersteller-IDs zu vermerken, z.B. wenn mehrere kompatible Kartenmodelle abgedeckt werden.

`virtual const Blob& GetRootPath() const throw() = 0;`

Gibt den Wurzelpfad zu den PKCS15-Objekten in Relation zum Wurzelverzeichnis des Tokens selbst zurück.

`virtual ITokenFile* GetTokenFile(const Blob& p15path) = 0;`

`virtual ITokenFile* GetTokenFile(const PKCS15Path& p15path) = 0;`

Holt sich eine Datei vom Token, die mit dem PKCS15-Objektpfad referenziert ist.

```
virtual ITokenPIN* GetPIN(IToken* token, const PKCS15Object& p15object,
bool use_obj_ref = false) = 0;
```

Gibt ein TokenPIN-Objekt zu dem Objekt zurück, das mit pkcs15object und dem Rückverweis auf das Token selbst referenziert wird.

```
virtual bool InitTokenPIN(ITokenPIN* token_pin, const PKCS15Object&
p15object) = 0;
```

Initialisiert das TokenPIN-Objekt mit den Inhalten aus dem PKCS15-Objekt.

```
virtual bool InitTokenKey(ITokenKey* token_key, int usage, int key_nr,
const PKCS15Object& p15object) = 0;

virtual bool InitTokenKey(ITokenBlockCipherKey* token_key, int key_nr,
const PKCS15Object& p15object) = 0;
```

Initialisiert das Schlüsselobjekt (asymmetrischer Schlüssel oder Blockchiffre) anhand den übergebenen Daten.

```
virtual FIDType GetFirstFID(byte obj_type, size_t key_size = 0) const = 0;
```

Holt die erste FID aus den für die übergebenen Daten festgelegten ‚Adressraum'.

```
virtual void GetObjectInfo(FIDType fid, Blob& object_path, byte* obj_ref,
byte obj_type, size_t key_size = 0) const = 0;
```

Gibt die Objektinformationen zum angegebenen Objekt zurück.

```
virtual FIDType TransformKeyReference(FIDType id, bool to_on_card_reference
= false, bool* bReference = 0) const = 0;
```

```
virtual FIDType GetNativeFID(byte obj_type, const FIDType fid, byte
object_reference = 0) const = 0;
```

Übersetzt die übergebene FID in die tokenspezifische FID.

```
virtual void GetRelativeFilePath(Blob& absolute_filepath) const = 0;
```

```
virtual const Blob& GetACTokenRights() const = 0;
```

Gibt die grundelegenden Zugriffsrechte für das jeweilige Token zurück.

```
virtual Blob GetAccessRights(const ITokenPIN* token_pin, byte obj_type)
const = 0;

virtual Blob GetAccessRights(ITokenAuthRef* authRef, ITokenAuthRef*
parentAuthRef, byte obj_type) const = 0;
```

Gibt die Zugriffsrechte für die jeweilige Authentisierung und dem Objekttyp zurück.

```
virtual ITokenAuthManager* GetAuthManager(IToken* token, int
authManagerType) const = 0;

virtual void CreateAuthObj(ITokenPIN* token_pin, byte obj_type) const = 0;

virtual bool GetNativePubKeySupport() const = 0;

virtual Blob ReadPublicKey(ISCardOS* os, FIDType fid) const = 0;

virtual FIDType GetNativePubKeyInfo(FIDType priv_key_fid, const
PublicKeyInfo& info, PKCS15Object& p15object) const = 0;
```

7.3.12.17 ITokenAuthOwner

Dieses Interface wird in der Regel nicht für sich alleine benutzt; andere Interfaces leiten hiervon ab. Der Zweck dieses Interfaces liegt darin, einen Rückverweis auf das `TokenAuth`- und `TokenPIN`Objekt zu liefern, das den Zugriff auf das Objekt mit diesem Interface regelt.

```
class ITokenAuthOwner
{
public:
        virtual void SetPin(const ITokenPIN* pin) = 0;
        virtual const ITokenPIN* GetPin() const = 0;
        virtual ITokenAuthRef* GetAuthRef() const throw() = 0;

protected:
        virtual ~ITokenAuthOwner() { }
};
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual void SetPin(const ITokenPIN* pin) = 0;`

`virtual const ITokenPIN* GetPin() const = 0;`

Setzt/Liest das PIN-Objekt, was mit diesem Objekt verknüpft wird.

`virtual ITokenAuthRef* GetAuthRef() const throw() = 0;`

Gibt ein Verweis auf das `TokenAuth`-Objekt zurück.

7.3.12.18 ITokenKey

Dieses Interface ist eine Spezialisierung von IKey und berückstichtigt die Besonderheiten, die sich daraus ergeben, dass der Schlüssel auf einem Token gespeichert ist.

```
class ITokenKey
        : public ISignatureKey
        , public ITokenAuthOwner
{
public:
        virtual void Configure(ITokenConfig* tkcfg) = 0;
        virtual IToken* GetToken() const throw() = 0;

        virtual Blob GetCertificate() const = 0;
        virtual IKey* GetPublicKey() const = 0;

        virtual void VerifyPin(const Blob& pinvalue) const = 0;
};
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual void Configure(ITokenConfig* tkcfg) = 0;`

Konfiguriert den Schlüssel.

`virtual IToken* GetToken() const throw() = 0;`

Gibt das Token zurück, zu das dieser Schlüssel gehört.

`virtual Blob GetCertificate() const = 0;`

Gibt das zum Schlüssel gehörende Zertifikat zurück.

`virtual IKey* GetPublicKey() const = 0;`

Gibt, soweit vorhanden, den zu diesem Schlüssel gehörenden öffentlichen Schlüssel zurück.

Eine NULL kann zurückgegeben werden wenn

- dieser Schlüssel bereits ein öffentlicher Schlüssel ist
- der öffentliche Schlüssel nicht separat auf dem Token gespeichert ist, sondern z.B. nur innerhalb eines Zertifikats

`virtual void VerifyPin(const Blob& pinvalue) const = 0;`

Versucht den Zugriff auf diesen Schlüssel mit Hilfe der übergebenen PIN zu authentisieren. Hierbei wird davon ausgegangen, dass die übergebene PIN zur benötigten Authentisierungsmethode passt. Um genauere Informationen über die zu verwendende PIN zu erhalten (und ob überhaupt eine PIN nötig ist) kann man hierzu auf das Interface `ITokenAuthOwner` zurückgreifen, von dem dieses Interface abgeleitet ist.

7.3.12.19 ITokenBlockCipherKey

Dieses Interface ist eine Spezialisierung von IBlockCipherKey und berückstichtigt die Besonderheiten, die sich daraus ergeben, dass der Schlüssel auf einem Token gespeichert ist.

```
class ITokenBlockCipherKey
        : public IBlockCipherKey
        , public ITokenAuthOwner
{
public:
        virtual void Configure(ITokenConfig* tkcfg) = 0;
        virtual IToken* GetToken() const throw() = 0;

        virtual void SetRawKey(const Blob& rawkey) = 0;
        virtual void GetRawKey(Blob& rawkey) const = 0;

        virtual void SetMode(const char* modename) = 0;
        virtual void SetPadding(const char* padname) = 0;
};
```

Ergänzende Anmerkungen zu den einzelnen Funktionen:

`virtual void Configure(ITokenConfig* tkcfg) = 0;`

Konfiguriert den Schlüssel.

`virtual IToken* GetToken() const throw() = 0;`

Gibt das Token zurück, zu das dieser Schlüssel gehört.

`virtual void SetRawKey(const Blob& rawkey) = 0;`

`virtual void SetMode(const char* modename) = 0;`

`virtual void SetPadding(const char* padname) = 0;`

Setzt den Schlüssel / den Betriebsmodus / die Paddingmethode.

Modi sind u.a. ECB und CBC, Paddingmethoden sind u.a. ISO, PKCS5 und RSAES.

Wichtig ist hierbei zu beachten, dass die Länge und der Inhalt des Schlüssels sowie auch die Auswahl der zulässigen Betriebsmodi stark von der Art des Schlüssels und des Tokens selbst abhängig sind.

`virtual void GetRawKey(Blob& rawkey) const = 0;`

Gibt den Schlüssel zurück.

7.3.13 StreamCipher-Interfaces

Die StreamCipher-Interfaces bieten Zugriff auf die Stromchiffre-Algorithmen.

7.3.13.1 Das Interface **IStreamCipher**

Diese Klasse repräsentiert symmetrische Stromchiffren und stellt die Low-Level-Verschlüsselungsoperation `Process` auf Stromebene zur Verfügung. Sie wird von `IStreamCipherKey` verwendet und kann dazu benutzt werden, neue Stromchiffren zu implementieren.

```
class IStreamCipher
{
public:

        virtual IStreamCipher* Clone() const =0;
        virtual void Import( const Blob& indata ) =0;
        virtual void Export( Blob& outdata ) const =0;

        virtual void SetRawKey( const Blob& keyblob ) =0;
        virtual void GetRawKey( Blob& keyblob ) const =0;

        virtual void Process(const byte* input, byte* output, size_t
input_len) const =0;

        virtual size_t GetKeySize(size_t keysize=0) const =0;
        virtual size_t GetMinKeySize() const =0;
        virtual size_t GetMaxKeySize() const =0;
        virtual size_t GetNextKeySize(size_t prevsize) const =0;

        virtual void* GetCreatePointer() const =0;

        virtual ~IStreamCipher() {};
};
```

IStreamCipher* Clone() const

erzeugt eine Kopie der aktuellen `StreamCipher`-Instanz.

`void Import( const Blob& indata )`

importiert den symmetrischen Schlüssel und gegebenenfalls weitere Parameter aus `indata`.

`void Export( Blob& outdata ) const`

exportiert den symmetrischen Schlüssel und gegebenenfalls weitere Parameter in `outdata`.

`void SetRawKey( const Blob& keyblob )`

setzt den symmetrischen Schlüssel der Chiffre, der sich in `keyblob` befindet.

`void GetRawKey( Blob& keyblob ) const`

schreibt den symmetrische Schlüssel in `keyblob`.

`void Process(const byte* input, byte* output, size_t input_len) const`

Verschlüsselt / Entschlüsselt `input_len` Bytes im Speicherbereich beginnend ab `input` und schreibt das Resultat in den Puffer, auf den `output` zeigt. `Output` muss hierbei auf einen Speicherbereich zeigen, in dem mindestens `input_len` Bytes verfügbar sind.

`size_t GetKeySize(size_t keysize=0) const`

Wenn `GetKeySize()` mit 0 aufgerufen wird, wird die Schlüssellänge des aktuell gespeicherten Schlüssels zurückgegeben. Bei Werten größer 0 wird die mögliche Schlüssellänge, die dem übergebenen Wert am nächsten liegt, zurückgegeben. Sowohl der Eingabe- als auch der Rückgabewert ist jeweils die Schlüssellänge in Byte.

`size_t GetMinKeySize() const`

gibt die Mindestschlüssellänge in Byte zurück.

`size_t GetMaxKeySize() const`

gibt die maximale Schlüssellänge in Byte zurück.

`size_t GetNextKeySize(size_t prevsize) const`

gibt die nächst größere, zulässige Schlüssellänge in Byte zurück. Wenn `prevsize` größer oder gleich der maximale Schlüssellänge ist, wird die maximale Schlüssellänge zurückgegeben, so dass der Rückgabewert von `GetNextKeySize()` immer einer gültigen Schlüssellänge entspricht.

`void* GetCreatePointer() const`

gibt den Funktionspointer auf die zu dieser Instanz gehörende Erzeugerfunktion zurück.

7.3.13.2 Das Interface **IStreamCipherKey**

Dieses Interface ermöglicht den Zugriff auf die aggregierten Objekte eines Schlüssels für eine Stromchiffre. Diese sind Cipher-Objekte und Derivator-Objekte.

```
class IStreamCipherKey : public IKey
{
public:

        virtual void SetCipher(IStreamCipher* cipher) =0;
        virtual const IStreamCipher* GetCipher() const =0;
        virtual IStreamCipher* GetCipher() =0;

        virtual void SetDerivator(IDerivator* derive) =0;
        virtual const IDerivator* GetDerivator() const =0;
        virtual IDerivator* GetDerivator()  =0;
};
```

void SetCipher(IStreamCipher* cipher)

übergibt dem Key-Objekt den Zeiger auf ein Cipher-Objekt.

const IStreamCipher* GetCipher() const oder **IStreamCipher* GetCipher()**

liefert einen Zeiger auf das Cipher-Objekt zurück.

void SetDerivator(IDerivator* Derive)

konfiguriert das Key-Objekt mit einem anderen Schlüsselableitungsalgorithmus.

const IDerivator* GetDerivator() const oder **IDerivator* GetDerivator()**

liefert einen Zeiger auf das Schlüsselableitungs-Objekt.

7. 4 Registries

Wie schon in Abschnitt 7. 3 ausführlicher erwähnt, verwendet die **cv act *library*** Registry-Klassen, um Objekte zu erzeugen. Als erstes Beispiel wird die Key-Registry `KeyReg` aufgeführt. Für die anderen Registry-Klassen sind die Methoden analog, so dass sie nicht nochmal dargestellt werden. Alle anderen Registries sind analog, außer den beiden die unten aufgeführten Sonderfällen. Die Registries werden mit der Funktion `act::Init()` initialisiert. Diese Inline-Funktion ist in der Datei `actInit.h` definiert und kann von Ihnen angepasst werden.

```
typedef IKey* (*CreateKeyPtr)();

struct KeyMapEntry {
        const char* Name;
        CreateKeyPtr CreatePtr;
};

class KeyReg
{
public:
        static IKey* CreateKey(const char* name);
        static CreateKeyPtr GetCreatePointer(const char* name);
        static const char* GetName(void *createptr);
        static const char* GetNextName(const char* name);
        static void Insert(const char* name, CreateKeyPtr createptr);
        static void Insert(const KeyMapEntry* keymap);
};
```

`IKey* CreateKey(const char* name)`

erzeugt ein Schlüssel-Objekt vom angegebenem Typ.

`CreateKeyPtr GetCreatePointer(const char* name)`

gibt den Zeiger auf die Erzeugerfunktion des Schlüssel-Objekts vom angegebenem Typ zurück.

`const char* GetName(void *createptr)`

gibt den Bezeichner-String zu dem übergegebenen Erzeugerfunktionszeiger zurück.

`const char* GetNextName(const char* name)`

gibt den in lexikalisch aufsteigender Reihenfolge nächsten Bezeichner-String zurück, der in der Registry enthalten ist.

`void Insert(const char* name, CreateKeyPtr createptr)`

fügt eine neue Klasse in die Registry ein mit dem Bezeichenr-String `name` und dem Zeiger auf die Erzeugerfunktion `createptr`.

`void Insert(const KeyMapEntry* keymap)`

mit dieser Methode lassen sich mehrere Klassen gleichzeitig registrieren. `keymap` zeigt dabei auf ein Array von `KeyMapEntry`–Werten, wobei für jeden Eintrag `X` des

Arrays die Methode `Insert(X.Name, X.CreatePtr)` aufgerufen wird. Dieses Array muss mit dem Wert `{0,0}` terminiert sein.

Als zweites Beispiel wird die Registry-Klasse`X509KeyReg` aufgeführt. Diese Klasse stellt die Verbindung zwischen den, in X.509-Zertifkaten enthaltenen, öffentlichen Schlüsseln und den Key-Objekten der **cv act *library*** her. Analog dazu ist die Registry-Klasse `X509SignReg`, die die Verbindung zwischen den in X.509-Zertifkaten enthaltenen Signaturalgorithmen-Parameter und den Key-Objekten darstellt.

```
typedef IKey* (*CreateKeyPtr)();
typedef void (*X509ToKeyPtr)(const Blob &pkinfo, IKey* key);
typedef void (*KeyToX509Ptr)(const IKey* key, Blob &pkinfo );

struct X509KeyMapEntry {
        const char* OID;
        CreateKeyPtr CreatePtr;
        X509ToKeyPtr X509ToKey;
        KeyToX509Ptr KeyToX509;
};

class X509KeyReg
{
public:
        static IKey* CreateX509Key(const Blob& pkInfo);
        static void GetX509KeyInfo(const IKey* key, Blob& pkInfo);

        static CreateKeyPtr GetCreatePointer(const char* oid);
        static const char* GetOID(void* createptr);
        static const char* GetNextOID(const char* oid);

        static void Insert(const char* oid, CreateKeyPtr createptr,
X509ToKeyPtr x509tokey, KeyToX509Ptr keytox509 );
        static void Insert(const X509KeyMapEntry* keymap);
};
```

`IKey* CreateX509Key(const Blob& pkInfo)`

erzeugt aus einem ASN.1-DER-kodierten PublicKeyInfo-Eintrags eines X.509-Zertifkats ein Schlüsselobjekt.

`void GetX509KeyInfo(const IKey* key, Blob& pkInfo)`

erzeugt aus einem Schlüsselobjekt einen ASN.1-DER-kodierten PublicKeyInfo-Eintrags eines X.509-Zertifkats.

`CreateKeyPtr GetCreatePointer(const char* oid)`

gibt den Zeiger auf die Erzeugerfunktion des Schlüssel-Objekts zur übergebenen OID zurück.

`const char* GetOID(void* createptr)`

gibt die OID zu dem übergegebenen Erzeugerfunktionszeiger zurück.

`const char* GetNextOID(const char* oid)`

gibt den in lexikalisch aufsteigender Reihenfolge nächsten Bezeichner-String zurück, der in der Registry enthalten ist. Falls 0 übergeben wird, wird die erste OID zurückgeliefert.

`void Insert(const char* oid,CreateKeyPtr creteptr,X509ToKeyPtr x509tokey, KeyToX509Ptr keytox509 )`

fügt eine neue Klasse in die Registry ein mit dem OID `oid` und dem Zeiger auf die Erzeugerfunktion `createptr`. Zusätzlich werden Zeiger auf die Konvertierungsfunktionen `X509ToKey` und `KeyToX509` eingetragen.

`void Insert(const X509KeyMapEntry* keymap)`

mit dieser Methode lassen sich mehrere Klassen gleichzeitig registrieren. `keymap` zeigt dabei auf ein Array von `KeyMapEntry`–Werten, wobei für jeden Eintrag `X` des Arrays die Methode `Insert(X.OID, X.CreatePtr, X.X509ToKey, X.X509ToKey)` aufgerufen wird. Dieses Array muss mit dem Wert `{0,0,0,0}` terminiert sein.

Kleines abc

der

Kryptographie

Hinweise zum Lesen dieses Wörterbuchs

Die cv cryptovision möchte Ihnen mit diesem Buch eine kleine Hilfe auf den Weg geben, damit Sie auch in der Begriffswelt der Kryptographie Sicherheit erlangen.

Dieses Wörterbuch ist aufgebaut wie ein Glossar: die Begriffe sind alphabetisch geordnet. Außerdem tauchen im Text Begriffe auf, die eine grüne Farbe haben. Diese sagen Ihnen, dass dieses Wort an der entsprechenden Stelle (nochmals) erklärt wird.

Ansonsten sind englische Begriffe, die als allgemein bekannt gelten oder an dieser Stelle definiert werden, kursiv dargestellt.

Wir wünschen viel Spaß beim Lesen!

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

A5

Stromchiffre welche innerhalb des GSM-Standards, also im Mobilfunkbereich verwendet wird.

AES

Der AES (*Advanced Encryption Standard*) ist als Standard für symmetrische Verschlüsselungsverfahren festgelegt worden; im Oktober 2000 wurde Rijndael als AES gewählt. Das NIST hat 1997 einen formalen Aufruf für diesen Nachfolger des DES veröffentlicht.

Vorgaben waren unter anderem, das es sich um eine symmetrische Blockchiffre mit einer Blocklänge von 128 und Schlüssellängen von 128, 192 und 256 Bit handeln soll. Nachdem anfangs ca. 15 Kandidaten vorgestellt wurden, waren Anfang 2000 noch 5 Algorithmen im Rennen. Die anderen Finalisten waren MARS, RC6, Serpent und Twofish.

Alice und Bob

Wohl die beiden bekanntesten Anwender von Kryptographie. Werden in technischen Beschreibungen die Endpunkte oft mit A und B bezeichnet, hat es sich im Bereich der Kryptographie eingebürgert, diese Namen zu verwenden.

Weitere, öfters anzutreffende Personen sind: die Lauscherin Eve (von e*avesdropper*) und der Angreifer Mallory (von m*alicious*).

ANSI

Abkürzung für *American National Standards Institute.*

ASN.1

Abkürzung für *Abstract Syntax Notation One* und ist ein verbreiteter Standard zur Darstellung abstrakter Objekte. Bei der Kodierung (den Regeln wie solche Objekte als String darzustellen sind) unterscheidet man zwischen den *Basic Encoding Rules* (BER) und den *Dinstinguished Encoding Rules* (DER).

Asymmetrische Chiffre

Verschlüsselungsverfahren, bei dem (im Gegensatz zur symmetrischen Chiffre) zwei verschiedene Schlüssel eingesetzt werden: Ein öffentlich bekannter - der *public key* - zum Verschlüsseln von Daten und ein nur dem Empfänger von Nachrichten bekannter geheimer Schlüssel - der *private key* - zum Entschlüsseln.

Authentifizierung

Authentifizierung bezeichnet ein Verfahren, das sicherstellt, dass ein Kommunikationspartner der ist, für den er sich ausgibt. Die eleganteste Methode basiert auf dem Einsatz sogenannter digitaler Signaturen.

Blockchiffre

Ein Algorithmus, der den Klartext in Bitgruppen, die Blöcke genannt werden, bearbeitet, heißt Blockalgorithmus oder Blockchiffre. Eine Alternative dazu sind die sogenannten Stromchiffren.

Blowfish

Blowfish ist eine symmetrische Blockchiffre mit einer Blocklänge von 64 Bit und einer Schlüssellänge von bis zu 448 Bit. Er wurde von dem Kryptographen Bruce Schneier entwickelt und 1993 vorgestellt.

Blum-Blum-Shub-Generator

Dies ist ein nach seinen Entwicklern benannter Algorithmus zur Erzeugung von Pseudozufallszahlen. Er gilt als sehr sicher, ist aber (da er auf der Arithmetik mit langen Zahlen beruht) auch sehr langsam.

Brute Force-Attacke

Ein Angriff auf einen kryptographischen Algorithmus, bei dem man den gesamten Schlüsselraum systematisch absucht.

BSI

> Das Bundesamt für Sicherheit in der Informationstechnik entstand 1991 und hat die Aufgabe, die IT-Sicherheit in unterschiedlichen Bereichen zu fördern und Unterstützungsleistung zu erbringen.

CAST

CAST ist eine symmetrische Blockchiffre mit 64 Bit Blocklänge und einer Schlüssellänge von 40-128 Bit. Er wurde nach seinen Entwicklern C. Adams und S. Tavares benannt und 1996 zum Patent angemeldet. Der Nachfolger CAST-256 war ebenfalls ein AES-Kandidat.

CBC-MAC

Dies ist ein Konstruktionsmechanismus, welcher einen MAC mit Hilfe einer Blockchiffre im CBC-Modus konstruiert.

CBC-Modus

Bei diesem kryptographischen Modus (dem *Cipher Block Chaining*) wird ein Block vor der eigentlichen Verschlüsselung mit dem vorherigen, chiffrierten Block XOR-verknüpft.

CCITT

Standardisierungsgremium, dessen Abkürzung von *Comite Consultatif International Telephonique et Telegraphique* kommt; seit 1993: ITU-T

wird.

CFB-Modus

Der *Cipher Feedback Modus* ist ein kryptographischer Modus, bei dem eine Blockchiffre quasi als Stromchiffre betrieben wird; dies ermöglicht die Chiffrierung von Klartextteilen, welche kleiner als die Blockgröße sind.

Chiffre

Eine Chiffre ist das Verfahren, bzw. der Algorithmus zum Ver- und Entschlüsseln von Daten. Es wird weiter unterschieden in asymmetrische und symmetrische Chiffre, diese unterteilen sich wiederum in Block- und Stromchiffren.

Chosen plaintext

Bezeichnung für die Situation, dass ein Angreifer beim Angriff auf eine Chiffre in der Lage ist, den zu verschlüsselnden Text frei zu wählen. Ist natürlich bei Public Key-Verfahren der normale Fall; siehe auch known plaintext.

DES

Der DES-Algorithmus (*Data Encryption Standard* oder auch *Data Encryption Algorithm,* DEA) ist eine symmetrische 64 Bit Blockchiffre welche (zuerst unter dem Namen Lucifer) bei IBM entwickelt wurde. Die Schlüssellänge beträgt 64 Bit, wovon 8 Bit einer Paritätsprüfung dienen. Dies ist der Klassiker unter den Verschlüsselungsalgorithmen, der aber aufgrund der zu geringen Schlüssellänge nicht mehr sicher ist. Alternativen sind der Triple-DES oder der Nachfolger AES.

Differentielle Kryptoanalyse

Angriffsmethode auf eine Chiffre, gehört zur Klasse der *chosen plaintext* Attacken. Man versucht Abhängigkeiten zwischen Chiffre- und Klartext zu finden und durch gezielte Abänderungen des Klartextes Informationen zu gewinnen.

Diffie-Hellman (DH)

Bezeichnung für das wohl bekannteste kryptographische Protokoll; Ziel ist der Schlüsselaustausch über einen unsicheren Kanal. Mit Vorstellung dieser Idee im Jahre 1976 entwickelte sich die Public Key-Kryptographie. Das Verfahren basiert auf dem Problem des diskreten Logarithmus.

Diffusion

Neben der Konfusion eines der beiden grundlegenden Prinzipien beim Design von Verschlüsselungsalgorithmen. Man versucht (z.B. durch Permutation, d.h. Vertauschen von Klartextteilen) die Redundanz des Klartexts über den Chiffretext zu verteilen.

Digitale Signatur

Das Gegenstück zu einer handschriftlichen Unterschrift für Dokumente, die in digitaler Form vorliegen; diese soll Sicherheit bei den folgenden Fragestellungen erbringen:

- Die Authentifizierung, d.h. die Sicherheit über den Absender des Dokumentes
- Die Integrität des Dokumentes soll gewährleistet werden
- Die Verbindlichkeit; d.h. der Absender soll die Erstellung nicht bestreiten können

Diese Eigenschaften können mit Hilfe asymmetrischer Verfahren erreicht werden: Mit Hilfe des geheimen Schlüssels werden Informationen erzeugt, anhand derer eine dritte Person sich unter Kenntnis des zugehörigen öffentlichen Schlüssels von der Korrektheit überzeugen kann.

Für die bekannten Public Key-Verfahren, wie RSA, existieren Protokolle, um sie im Rahmen von Digitalen Signaturen einzusetzen. Für DL-basierte Verfahren haben sich Verfahren vom ElGamal-Typ etabliert.

Diskreter Logarithmus (DL)

Die Potenzierung ist eine aus der Schulmathematik bekannte Rechnung. Für die Berechnung von X^n ist die vorgegebene Zahl n-mal mit sich selbst zu multiplizieren. Das entsprechend inverse Problem ist die Bestimmung des Logarithmus einer Zahl: Für gegebenes X und Y ist eine Zahl n zu suchen, so dass $Y = X^n$ gilt.

Was im Falle der bekannten (reellen) Zahlen auf vergleichbar einfache Probleme hinaus zu laufen scheint, erweist sich in bestimmten Mengen als erstaunlich schwierig: Man hat zwar stets einen effizienten Algorithmus zur Potenzierung von Elementen zur Verfügung, die Umkehrung dieser Operation (eben die Berechnung des Logarithmus eines gegebenen Elementes) ist aber unter bestimmten Umständen nicht praktikabel.

Einsatzmöglichkeiten für dieses Basisproblem sind Protokolle zum Schlüsselaustausch (z.B. Diffie-Hellman), Verschlüsselung von Daten oder aber auch für den Einsatz innerhalb von digitalen Signaturen. Der DSA und die Algorithmen auf Basis elliptischer Kurven beruhen auf diesem Problem.

DLIES

> Ein komplexes asymmetrisches Verschlüsselungsschema, dass aus einem symmetrischen Verfahren, einer Hashfunktion und einem MAC besteht.

DSA

Der DSA-Algorithmus (*Digital Signature Algorithm*) ist eine Variante zur Erstellung digitaler Signaturen basierend auf dem DL-Problem in endlichen Körpern. Da in diesem Fall für einen Angriff auf das zugrunde liegende Problem im wesentlichen die gleichen Methoden wie beim Faktorisieren von Zahlen verwendet werden, sind auch die Parameter in vergleichbaren Größenordnungen anzusetzen. Für Anwendungen finden also (wie auch im Fall des RSA-Algorithmus) gegenwärtig Zahlen mit 1024 Bit Verwendung.

ECB-Modus

Beim ECB-Modus (nach *Electronic Code Book*) einer Blockchiffre wird jeder Klartextblock einfach in den entsprechenden Chiffreblock verschlüsselt. Die Konsequenz ist, dass identische Nachrichtenblöcke auch in gleiche Chiffretexte überführt werden, weshalb diese Vorgehensweise unter bestimmten Umständen als sicher gelten kann.

ECC

Der Einsatz von elliptischen Kurven in der Kryptographie wird ECC (*Elliptic Curve Cryptography*) genannt. Diese Klasse von Verfahren stellen eine attraktive Alternative zum wohl bekanntesten asymmetrischen Verfahren, dem RSA-Algorithmus dar. Das zugrundeliegende mathematische Problem ist - ähnlich wie beim DSA-Algorithmus - die Berechnung des diskreten Logarithmus in endlichen Mengen. Die Menge der hier betrachteten Elemente ist die Menge von Punkten, welche eine bestimmte mathematische Gleichung, nämlich die einer elliptischen Kurve erfüllen.

Der entscheidende Vorteil dieses Verfahrens besteht darin, dass die bislang bekannten schnellen Algorithmen zur Lösung des DL-Problems in endlichen Körpern in diesem Fall nicht anwendbar sind. Da für das DL-Problem in der Punktegruppe elliptischer Kurven nur sehr allgemeine Verfahren

vorhanden sind, kommt man mit deutlich geringeren Schlüssel- und Parameterlängen aus, ohne Abstriche hinsichtlich der Sicherheit in Kauf nehmen zu müssen. Dies wirkt sich besonders beim Einsatz in Situationen aus, wo Speicher- oder Rechenkapazität knapp sind, wie z.B. bei Smartcards und anderen Small Devices.

EC-DH

Eine auf elliptischen Kurven basierende Variante des Diffie-Hellman Protokolls.

EC-DSA

Die Umsetzung des DSA unter Verwendung von elliptischen Kurven.

EC-IES

Eine Variante des DLIES-Schematas unter Verwendung elliptischer Kurven.

EC-MQV

Die Entsprechung des MQV-Protokolls die als zugrunde liegende Struktur eine elliptische Kurve verwendet.

EC-NR

Die Variante des Nyberg-Rueppel-Protokolls (NR) unter Verwendung elliptischer Kurven.

ElGamal

Eine bestimmte Klasse von Signatur- und Verschlüsselungsprotokollen basierend auf dem Problem des diskreten Logarithmus. Ist benannt nach seinem Entwickler T. ElGamal.

Elliptische Kurven

Ein mathematisches Konstrukt, in dem ein Teil der üblichen Rechenregeln gilt und das seit ca. 1985 erfolgreich in der Kryptographie eingesetzt wird. Falls es sich bei dem zugrundeliegenden Körper um GF(p) (p prim) handelt, ist ein Element (oder Punkt) einer elliptischen Kurve (mit den Parametern A und B) gegeben durch ein Tupel (x,y), welches eine Gleichung der folgenden Form löst:

$$y^2 = x^3 + Ax + B$$

Hat der endliche Körper die Charakteristik 2, so lautet die entsprechende Gleichung

$$y^2 + xy = x^3 + Ax^2 + B$$

Elliptische Kurven lassen sich über beliebigen Körper definieren; Verwendung in der Kryptographie finden aber nur Kurven über endlichen Körpern. Erfüllt die elliptische Kurve und der zugrunde gelegte Körper geeignete Bedingungen, so lässt sich das Problem des diskreten Logarithmus in diesem Fall nicht effizient lösen.

Endlicher Körper

Eine mathematische Struktur mit endlich vielen Elementen, in denen die üblicherweise bekannten Rechenregeln für Addition, Subtraktion, Multiplikation und Division gelten. Eine gebräuchliche Bezeichnung ist GF (für *Galois Field*). Verwendung innerhalb der Kryptographie finden vor allem GF(p), d.h. die Anzahl der Elemente ist eine Primzahl p, oder GF(2^m), d.h. die Anzahl ist eine Potenz von 2.

Entropie

Die Entropie einer Quelle misst die Informationen, die man durch Beobachten der Quelle durchschnittlich bekommen kann, oder umgekehrt die Unbestimmtheit, die über die erzeugte Nachricht herrscht, wenn man die Quelle nicht beobachten kann.

Faktorisierung ▶

Faktorisierung ist das Problem, für eine gegebene Zahl die Primfaktoren zu bestimmen. Für den Fall n=pq (d.h. n ist das Produkt zweier Primzahlen p und q) ist dies das dem RSA-Algorithmus zugrundeliegende Problem. Die Schwierigkeit, dies zu lösen, steigt mit zunehmender Größe von p und q an; üblicherweise nimmt man für diese Faktoren Zahlen in der Größenordnung von 512 Bit (n hat dann 1024 Bit, also mehr als 300 Dezimalstellen).

Falltürfunktion

Eine Falltürfunktion (trap door function) ist eine Funktion f, die leicht zu berechnen ist, bei der es aber ohne Kenntnis des geheimen Schlüssels praktisch nicht möglich ist, zu gegebenem Funktionswert y ein Argument x mit f(x)=y zu berechnen. Hat man erst einmal eine solche Funktion gefunden, so sieht man, dass man mit dieser in vielen Fällen ein asymmetrisches Kryptosystem konstruieren kann.

Feistel-Chiffre

Ein nicht unwesentliches Problem bei der Chiffrierung von Daten ist, dass die Verschlüsselungsfunktion umkehrbar sein muss, um eine korrekte Entschlüsselung von Texten zu ermöglichen. Bei Feistel-Chiffren wird diese Anforderung durch ein bestimmtes Design sichergestellt. Wahrscheinlich bekanntester Vertreter mit diesem Aufbau ist der DES.

FIPS

Federal Information Processing Standard, Standard in den Vereinigten Staaten für die Verwendung innerhalb von Behörden.

GF ▶

Galois Field, s. endlicher Körper

Hashfunktion ▶

Eine Funktion, die für eine beliebige Eingabe einen Funktionswert (den Hashwert) mit fest vorgegebener Länge berechnet. Diese Funktionen werden dazu benutzt, das elektronische Gegenstück eines Fingerabdrucks zu erzeugen. Wesentlich ist dabei, dass es nicht möglich sein darf, zwei Eingaben zu erzeugen, welche auf denselben Hashwert führen (sogenannte Kollisionen) oder aber gar zu einem gegebenen Hashwert eine passende Nachricht zu erzeugen. Gängige Hashfunktionen sind RIPEMD-160 und SHA-1 jeweils mit Hashwerten von 160 Bit Länge sowie der auch heute noch oft verwendete MD5 mit einer Hashwertlänge von 128 Bit.

Hybrid-Verfahren

Die Kombination von asymmetrischen und symmetrischen Chiffren: Auf Basis eines asymmetrischen Verfahrens wird ein Schlüssel ausgetauscht, mit dessen Hilfe nachfolgend die symmetrische Verschlüsselung durchgeführt wird. Auf diese Weise versucht man die Vorteile beider Verfahren zu erhalten, nämlich das unproblematische Schlüsselmanagement für asymmetrische und die hohe Geschwindigkeit des symmetrischen Verfahrens.

IDEA ▶

IDEA (*International Data Encryption Algorithm*) ist eine symmetrische Blockchiffre mit 64 Bit Blocklänge und einer Schlüssellänge von 128 Bit. Er wurde Anfang der 90er Jahre von X. Lai und J. Massey entwickelt; der Algorithmus ist patentiert, die Rechte liegen bei der Schweizer Firma Ascom. Bekannt geworden ist dieser Algorithmus vor allem durch die Verwendung innerhalb von PGP.

IEEE

Standardisierungsgremium, Abkürzung von *Institute of Electrical and Electronics Engineers*. Seit einigen Jahren arbeitet man in der Arbeitsgruppe P1363 an der Standardisierung von Algorithmen für die Public Key- Kryptographie.

Integrität

Der Test auf die Integrität von Daten ist die Überprüfung einer Nachricht durch den Empfänger auf Veränderungen während der Übermittlung. Übliche Prüfungsverfahren sind der Einsatz von Hashfunktionen, MACs (*Message Authentication Codes*) oder - mit zusätzlicher Funktionalität - der Einsatz von digitalen Signaturen.

ISO

Internationale Standardisierungsorganisation (*International Organization for Standardization*); beschäftigt sich in zahlreichen Arbeitsgruppen mit Algorithmen aus allen Bereichen der Kryptographie.

Kerckhoffs´ Maxime

Ein wichtiges Prinzip bei der Beurteilung von kryptographischen Algorithmen: Die Sicherheit des Verfahrens sollte **nicht** darauf beruhen, dass dieses geheim gehalten wird, sondern einzig und allein auf dem verwendeten Schlüssel.

key escrow

Damit bezeichnet man die sogenannte Schlüsselhinterlegung, d.h. die Möglichkeit für eine übergeordnete Instanz, an den privaten Schlüssel eines Benutzers zu gelangen. Dieses ist im allgemeinen für den privaten Bereich unerwünscht, aber für den firmeninternen Einsatz von Kryptographie sinnvoll.

Knapsack-Problem

Das sogenannte Rucksack- (oder *knapsack*) Problem war eines der ersten Probleme, welches für Public Key-Verfahren eingesetzt wurde. Die Beschreibung ist einfach: Gegeben ist ein Rucksack, mit dem man ein gewisses Gewicht tragen kann und eine große Anzahl von Objekten mit unterschiedlichen Massen. Das Problem ist, auszuwählen, was man in den Rucksack packen muss, um diesen optimal zu füllen. Für eine große Anzahl von Objekten ist dies wirklich ein hartes Problem, die darauf aufbauenden Algorithmen haben sich inzwischen aber alle als unsicher erwiesen.

Known plaintext

Bezeichnung für die Situation, dass ein Angreifer beim Angriff auf eine Chiffre den zugehörigen Klartext kennt. Ist nicht zu vernachlässigen, da viele chiffrierte Nachrichten ein bestimmtes Format aufweisen.

Kollision

Damit bezeichnet man den Umstand, dass bei einer Hashfunktion zwei verschiedene Nachrichten auf ein und denselben Hashwert führen. Ist es für eine gegebene Funktion nicht möglich, solche Kollisionen zu erzeugen, so bezeichnet man diese als kollisionsresistent.

Konfusion

Neben der Diffusion eines der beiden grundlegenden Prinzipien beim Design von Verschlüsselungsalgorithmen. Ziel ist das Verbergen von Redundanzen im Klartext. Eine Möglichkeit ist beispielsweise die Substitution von verschiedenen Zeichen im Klartextes durch andere. Ein einfaches Beispiel ist die sogenannte Cäsar-Chiffre, bei der jeder Buchstabe im Klartext durch einen bestimmten anderen Buchstaben ersetzt wird.

Kryptographischer Modus

Dies ist der kryptographische Modus, in dem eine Blockchiffre betrieben wird. Er verknüpft die durch die Grundchiffrierung behandelten Blöcke miteinander. Beispiele sind ECB, CBC, CFB oder OFB.

Lawineneffekt

Der Lawineneffekt (engl. *avalanche effect*) bezeichnet den Umstand, dass sich bei einer guten Chiffre Änderungen im Klartext möglichst schnell (innerhalb der Chiffrierfunktion) auf den Chiffretext auswirken.

Lineare Kongruenzgeneratoren

Dies ist ein schnelles Verfahren zur Erzeugung von Pseudozufallszahlen, das aber für kryptographische Zwecke ungeeignet ist (da die Ergebnisse des Generators schnell vorhersehbar werden).

Lineare Kryptoanalyse

Angriffsmethode auf eine Chiffre, gehört zur Klasse der *known plaintext* Attacken. Man versucht, einfache („lineare“) Abhängigkeiten zwischen den Bits des Klartextes und des Chiffretextes zu entdecken und auszunutzen, um Informationen über den Schlüssel zu erhalten.

MAC

Ein *Message Authentication Code* erweitert eine Nachricht mit Hilfe eines geheimen Schlüssels um spezielle redundante Informationen, welche zusammen mit der Nachricht gespeichert bzw. übertragen werden, um im nachhinein die Authentifizierung der Nachricht zu ermöglichen. Damit ein Angreifer die angefügte Redundanz nicht gezielt modifizieren kann, muss diese in geeigneter Weise geschützt sein.

MailTrusT

Dies ist der Versuch einen Standard für die Verschlüsselung und für die Erstellung digitaler Signaturen bei E-Mails zu etablieren; Gremium ist der Verein TeleTrusT. Eine Alternative dazu ist S/MIME.

MARS

Ein AES-Kandidat der letzten Runde. Wird entwickelt bei IBM (unter anderem von Leuten die schon bei der Entwicklung von DES beteiligt waren).

Massnahmenkatalog

Die (hauptsächlich durch das BSI) vorgenommene Interpretation des deutschen Signaturgesetzes (SigG/SigV) welche festlegt, welche Punkte bei der Umsetzung zu beachten sind, um mit einer Applikation rechtsgültige digitale Signaturen erzeugen zu können.

MD2

MD2 ist eine Hashfunktion, die von R. Rivest für den Einsatz auf 8-Bit-Rechnern konzipiert und 1992 im RFC 1319 veröffentlicht wurde. Der Algorithmus erzeugt Hashwerte von 128 Bit Länge und ist relativ langsam.

MD4

MD4 ist eine Hashfunktion, die 1990 von Ron Rivest speziell für den Einsatz auf 32-Bit-Rechnern konzipiert wurde. Die Blocklänge beträgt 512 und der Hashwert 128 Bit. Gegen diesen Algorithmus kennt man seit einigen Jahren erfolgreiche Angriffe, so dass man von seiner Verwendung abraten muss. Auf Designprinzipien dieses Verfahren beruhen aber eine ganze Reihe erfolgreicher Hashfunktionen, ein bekannter ist der MD5.

MD5

MD5 ist eine Hashfunktion der MD4-Familie, ebenfalls entwickelt von R. Rivest. Die Blocklänge beträgt 512 und der Hashwert 128 Bit. Der Algorithmus ist nicht direkt geknackt; allerdings ist eine Hashwertlänge von 128 Bit für heutige Verhältnisse eindeutig zu niedrig.

Message Recovery

Die Möglichkeit bei Protokollen für digitale Signaturen, die ursprüngliche Nachricht aus der Signatur zurück zu gewinnen, bezeichnet man als *message recovery*. Vorteile solcher Vorgehensweise ist die bessere Ausnutzung der Bandbreite, da die Signatur nicht zusätzlich zur eigentlichen Nachricht übertragen werden muss. Im Falle der Verwendung des RSA-Algorithmus ist die Vorgehensweise naheliegend, aber auch bei der Verwendung von DL-basierten Verfahren gibt es entsprechende Protokolle (z.B. NR).

Miller-Rabin-Test

Ein probabilistischer Primzahltest. Diese Verfahren werden verwendet, da ein echter Beweis, ob eine vorliegende Zahl eine Primzahl ist, zu aufwendig ist. Bei einem Test wie dem Miller-Rabin-Test ist das Ergebnis einer Runde (d.h. die Aussage, ob die zu prüfende Zahl prim ist oder nicht) mit einer gewissen Fehlerwahrscheinlichkeit behaftet; mehrmalige Ausführung reduziert diese Fehlerwahrscheinlichkeit auf einen Wert welcher vernachlässigbar ist.

MQV

Dieses Protokoll für den Schlüsselaustausch wurde 1995 von A. Menezes, M. Qu und S. Vanstone entwickelt. Es basiert im Wesentlichen (wie fast alle Schlüsselaustauschprotokolle) auf dem DH-Protokoll, hat aber den Vorteil, dass die entstehenden Schlüssel gleichzeitig authentifiziert werden.

NIST

Das *National Institute for Standards and Technology* - vorher NBS *(National Bureau of Standards)* - ist eine Abteilung des US-Handelsministeriums, das unter anderem kryptographische Standards festlegt.

NR

Dieser Algorithmus ist ein Verfahren für digitale Signaturen und wurde nach seinen Entwicklern K. Nyberg und R. Rueppel benannt. Er basiert auf dem DL-Problem über endlichen Körpern und ermöglicht die Wiederherstellung von Nachrichten aus der eigentlichen Signatur (das sogenannte *message recovery*).

NSA

National Security Agency, Sicherheitsorgan der US-Regierung.

OFB-Modus

Der *Output Feedback Modus* ist (ähnlich dem CFB) ein kryptographischer Modus für Blockchiffren, bei dem man die Vorteile von Stromchiffren zu erreichen sucht, nämlich die Möglichkeit, Nachrichten mit kürzerer Länge als die Blocklänge zu verschlüsseln.

One Time Pad

Die einzig beweisbar sichere Art der Verschlüsselung von Daten. Es gibt aber einen gewaltigen Nachteil: Der verwendete Schlüssel muss die gleiche Länge wie der zu verschlüsselnde Text besitzen und darf auch nur einmal verwendet werden. Dafür ist die eigentliche Verschlüsselung denkbar einfach: Die Bits des Klartextes werden mit den entsprechenden Bits des Schlüssels XOR-verknüpft, um den Chiffretext zu erhalten; nochmalige XOR-Verknüpfung kehrt diesen Prozess um und ergibt die Entschlüsselung.

Padding

Die meisten Nachrichten lassen sich nicht problemlos in Blöcke mit einer bestimmten Länge von beispielsweise 64 Bit unterteilen. Es kann ein unvollständiger Block übrigbleiben, der dann mit der Methode des Auffüllens (*Padding*) vervollständigt wird. Solche Verfahren existieren für alle blockorientierten Algorithmen.

Passphrase

Eine lange, aber einprägsame Zeichenfolge (wie z.B. kurze Sätze mit Interpunktion), die Passwörter ersetzen sollten, da sie mehr Sicherheit bieten.

Passwort

Eine geheime Zeichenkette, deren Kenntnis als der Ersatz für die Authentifizierung eines Teilnehmers dienen soll. Üblicherweise ist ein Passwort zu kurz, um wirklich die Sicherheit zu gewährleisten, dass ein Angreifer es nicht durch Ausprobieren erraten kann.

PGP

Pretty Good Privacy ist ein Programm von P. Zimmermann zum Verschlüsseln und Signieren von E-Mails; vor allem durch die Verbreitung dieses Programms wurde ab ungefähr 1994 der Gebrauch von Public Key-Verfahren populär.

PKCS

Abkürzung für *Public Key Cryptography Standard*. Er wurde ausgegeben und betreut von den RSA Laboratories und ist ein Firmenstandard, welcher bei dem anfangs schwierigen Problem der Kompatibilität von Produkten helfen sollte. Die Bezeichnung umfasst eine Reihen verschiedener Dokumente, Beispiele sind PKCS#1 (für den RSA-Algorithmus), PKCS#7 (für die innerhalb der Kryptographie verwendeten Formate) oder PKCS#11 (für eine generische Schnittstelle zu kryptographischen Token wie beispielsweise Smartcards).

Primzahl

Eine Zahl, die nur durch 1 und sich selber ohne Rest teilbar ist. Die Bedeutung von Primzahlen für die Kryptographie ist z.B. durch das Problem der Faktorisierung gegeben.

Private key

Dies ist der geheime, nur dem Empfänger einer Nachricht bekannte Schlüssel, der bei asymmetrischen Chiffren zur Entschlüsselung bzw. zur Erstellung digitaler Signaturen verwendet wird.

Pseudoprimzahl

Eine Zahl, von der nicht bewiesen wurde, dass sie eine Primzahl ist, von der aber mit Hilfe bestimmter Verfahren (z.B. dem sogenannten Miller-Rabin-Test) gezeigt wurde, dass es sich nur mit sehr geringer Wahrscheinlichkeit um eine zusammengesetzte Zahl handelt. Die z.B. im RSA-Algorithmus eingesetzten Zahlen sind in der Regel immer Pseudoprimzahlen. Eine humorvolle Umschreibung dieser Bezeichnung ist „Primzahl von industrieller Qualität“.

Pseudozufallszahl

In vielen kryptographischen Mechanismen werden Zufallszahlen benötigt (z.B. bei der Schlüsselerzeugung); das Problem dabei ist allerdings, dass man Zufälle nur schwer in Software implementieren kann. Deshalb greift man auf sogenannte Pseudozufalls-zahlengeneratoren zurück, die dann allerdings mit einem echten Zufallselement (dem sogenannten *seed*) initialisiert werden sollten.

Public key

Dies ist der öffentlich bekannte Schlüssel bei einer asymmetrischen Chiffre, der zum Verschlüsseln und zum Überprüfen von digitalen Signaturen verwendet wird.

Public Key-Infrastruktur (PKI)

Das größte Problem beim Einsatz von Public Key-Verfahren stellt die Authentizität von Schlüsseln dar. Dahinter verbirgt sich die Frage, wie gewährleistet werden kann, dass der vorliegende Schlüssel wirklich vom gewünschten Kommunikationspartner stammt. Eine PKI ist eine Kombination aus Hardware- und Software-Komponenten, Policen und verschiedenen Prozeduren. Sie basiert hauptsächlich auf sogenannten Zertifikaten; diese sind ihrerseits durch digitale Signaturen von einer vertrauenswürdigen Instanz beglaubigte Schlüssel der Kommunikationspartner.

Public Key-Verfahren

Bezeichnet eine Klasse von Algorithmen, bei denen der erforderliche Schlüssel in einen öffentlichen, bekannten (den *public key*) und einen geheim zu haltenden Teil (den *private key*) aufgespalten wird. Aufgrund dieser Aufspaltung werden diese Verfahren auch asymmetrische Chiffren genannt; der bekannteste Vertreter ist der RSA-Algorithmus.

RC2

64-Bit-Blockchiffre mit variabler Schlüssellänge, entwickelt bei RSA Security Inc. von R. Rivest. Nur bei Lizensierung erhältlich (aber nicht patentiert).

RC4

Stromchiffre mit variabler Schlüssellänge, entwickelt bei RSA Security Inc. von R. Rivest. Nur bei Lizensierung erhältlich (aber nicht patentiert).

RC5

Blockchiffre mit variabler Schlüssel- und Blocklänge, entwickelt bei RSA Security Inc. von R. Rivest. Nur bei Lizensierung erhältlich.

RC6

Ein AES-Kandidat der letzten Runde. Entwickelt bei RSA Security Inc., unter anderem von R. Rivest.

RFC

Die im Internet eingesetzten Verfahren werden in *Request For Comment* genannten Dokumenten vorgestellt und festgehalten.

Rijndael

Ursprünglicher Name des AES. Wurde entwickelt von einer Forschergruppe aus Belgien (J. Daemen, V. Rijmen).

RIPEMD-128

RIPEMD-128 ist eine Hashfunktion welche im Rahmen des RACE-Projektes (*Research in Advanced Communication Technologies*) entwickelt wurde. Die Blocklänge beträgt 512 und der Hashwert 128 Bit; ein Hashwert dieser Länge ist allerdings für heutige Verhältnisse eindeutig zu niedrig.

RIPEMD-160

Ein Algorithmus ähnlich dem RIPEMD-128. Die Blocklänge beträgt ebenfalls 512, die Hashwertlänge ist hier aber 160 Bit. Dieser Algorithmus ist (zusammen mit SHA-1) einer der beiden als sicher angesehenen und verbreiteten Algorithmen.

RSA

Dieser nach den Erfindern Rivest, Shamir und Adleman benannte und 1978 vorgestellte Algorithmus basiert auf der Erfahrung, dass das Faktorisieren großer Zahlen (in realen Anwendungen oberhalb von 300 Dezimalstellen) ein praktisch nicht zu lösendes Problem darstellt. Es ist der bekannteste und am weitesten verbreitete Algorithmus aus der Klasse der asymmetrischen Chiffren.

Außerdem ist es sehr einfach, den Algorithmus zur Erstellung digitaler Signaturen einzusetzen, indem die Rollen von geheimen und öffentlichem Schlüssel vertauscht werden.

S/MIME

Erweiterung von MIME (*Multipurpose Internet Mail Extensions*) um die Funktionalitäten wie Verschlüsseln und Signieren von Nachrichten; hauptsächlich unter Verwendung der PKCS-Spezifikationen.

S-Box

Eine S-Box ist eine Matrix mit 4 Zeilen und 16 Spalten

SAFER

Eine 64-Bit-Blockchiffre mit 64 Bit Schlüssellänge, entwickelt von J. Massey. Es gibt auch Varianten mit längeren Schlüsseln.

Schlüsselaustausch

Der Einsatz symmetrischer Chiffrierverfahren verlangt, dass sich zwei Kommunikationspartner auf einen gemeinsamen, nur ihnen bekannten Schlüssel einigen. Die Schwierigkeit dabei besteht darin, dass für den Austausch solcher Informationen heutzutage meist nur bedingt sichere Kanäle vorhanden sind.

Protokolle für den Schlüsselaustausch müssen also so verfasst sein, dass nur Informationen ausgetauscht werden, aus deren Kenntnis kein Wissen über das eigentliche Geheimnis (den Schlüssel) resultiert.

Das bekannteste derartiger Protokolle ist Diffie-Hellman, dessen Vorstellung 1976 als Geburtsstunde der Public Key-Kryptographie bezeichnet werden kann.

Secret Sharing

Ein kryptographisches Protokoll, welches zum Ziel hat, eine vertrauliche Information so auf eine Reihe von Teilnehmern zu verteilen, dass nicht einer von diesen allein, sondern nur eine bestimmte Anzahl von Personen in der Lage ist, das ursprüngliche Geheimnis zu rekonstruieren.

Seed

Der Wert mit dem ein Pseudozufallszahlengenerator initialisiert werden muss, damit die entstehende Folge für einen Angreifer nicht vorhersehbar ist.

Serpent

Ein AES-Kandidat der letzten Runde. Entwickelt von E. Biham, R. Anderson und L. Knudsen.

SHA

Mit SHA (*Secure Hash Algorithm*) wird die innerhalb des DSA verwende Hashfunktion bezeichnet. Die Blocklänge beträgt 512 und der Hashwert 160 Bit. Der ursprüngliche Algorithmus wurde noch nach seiner Veröffentlichung modifiziert, so dass neben diesem Namen auch noch die Begriffe SHA-0 und SHA-1 verwendet werden.

SHA-0

Bezeichnung für die ursprüngliche Version des SHA, da mit diesem Begriff nun normalerweise die korrigierte Version SHA-1 gemeint ist.

SHA-1

Die modifizierte und in die Standards (z.B. DSA) eingegangene Version des SHA.

SigG/SigV

Abkürzung für die Begriffe Signaturgesetz und -verordnung; diese regeln seit 1997 in Deutschland die Verwendung von digitalen Signaturen. Siehe auch unter Massnahmenkatalog.

Skipjack

Von der NSA entwickelte 80-Bit-Blockchiffre mit 80 Bit Schlüssellänge für den Einsatz innerhalb der Clipper-Initiative. Der Algorithmus wurde nicht veröffentlicht, es existieren aber Referenzimplementationen.

Smartcards

Eine Smartcard ist eine Chipkarte, die intern einen Mikrokontroller enthält, also im Gegensatz zu einer reinen Speicherkarte auch eigene Berechnungen durchführen kann. Für kryptographische Anwendungen werden Smartcards verwendet, deren Funktionalität auf einem einzigen Chip untergebracht ist (im Gegensatz zu Chipkarten, die mehrere miteinander verdrahtete Chips enthalten). Damit ist solch eine Smartcard prädestiniert für den Einsatz in der Kryptographie, da es fast unmöglich ist, die inneren Vorgänge zu manipulieren.

Small Devices

Dies sind kleine tragbare Geräte, die einen Mikrokontroller oder vergleichbares beinhalten. Typische Small Devices sind (neben Smartcards) Palmtops, Organizer oder Handys, also Geräte zur individuellen Datenkommunikation.

Steganographie

Ein neues Einsatzgebiet für kryptographische Anwendungen: Man versucht die Existenz von (in der Regel verschlüsselten) Nachrichten in digitalen Objekten wie z.B. Bildern zu verbergen, um so unerkannt Nachrichten zu übermitteln.

Stromchiffre

Ein symmetrischer Verschlüsselungsalgorithmus, welcher den Klartext Bit- oder Byteweise bearbeitet, heißt Stromchiffre; die andere üblicherweise verwendete Klasse von Verfahren umfasst die sogenannten Blockchiffren.

Symmetrische Chiffre

Verschlüsselungsverfahren bei dem zur Chiffrierung und zur Dechiffrierung derselbe Schlüssel verwendet wird (oder bei dem diese zwei Schlüssel einfach voneinander ableitbar sind). Es wird unterschieden zwischen Blockchiffren, welche den Klartext in Blöcken fester Länge bearbeiten (meist 64 oder 128 Bit), und Stromchiffren, welche auf Ebene einzelner Zeichen arbeiten.

TeleTrusT

Gemeinnütziger Verein, der sich zum Ziel gesetzt hat, die Akzeptanz der digitalen Signatur als Instrument zur Rechtssicherheit einer Transaktion zu erreichen. Hat zur Zeit (Anfang 2000) ungefähr 100 Mitglieder, größtenteils Firmen, inklusive der **cv cryptovision**, und Behörden.

Triple-DES

Die Bezeichnung für Varianten des DES, welche die Probleme aufgrund der zu geringen Schlüssellänge beheben sollen; der DES-Algorithmus wird dabei dreimal hintereinander mit verschiedenen Schlüssel angewendet. Es gibt verschiedene Varianten welche sich z.B. in der Anzahl der verwendeten Schlüssel unterscheiden. Die gebräuchlichste (und auch durch ANSI standardisierte Methode) ist das EDE-Verfahren: Dabei wird mit dem 1. Schlüssel chiffriert (*encrypted*), mit dem 2. Schlüssel dechiffriert (*decrypted*) und noch einmal mit dem 1. verschlüsselt (*encrypted*). Die effektive Schlüssellänge beträgt in diesem Fall also 112 Bit; es gibt aber auch Varianten mit 3 verschiedenen Schlüsseln, also 168 Bit Schlüssellänge.

Trust Center

Mit Trust Center (manchmal auch *Trusted Third Party* genannt) wird ein ausgezeichneter Teil einer PKI bezeichnet. In der Regel unterteilt man die Aufgabenbereiche eines Trust Centers in drei verschiedene Bereiche:

- die eigentliche Zertifizierungsstelle (*Certification Authority*, CA) übernimmt die eigentliche Beglaubigung der vorliegenden Informationen
- die Registrierungsstelle (*Registration Authority*, RA) ist für die Identifikation der Teilnehmer und die Übergabe der Zertifikate zuständig
- der Verzeichnisdienst stellt die zur Erstellung und zur Überprüfung ausgestellter Zertifikate und Signaturen notwendigen Informationen (wie z.B. Zeitstempel oder Listen mit gesperrten Zertifikaten) zur Verfügung.

Twofish

Ein AES-Kandidat der letzten Runde. Wurde von B. Schneier und seiner Firma Counterpane entwickelt und vorgestellt.

Verbindlichkeit

Eines der Ziele beim Einsatz von digitalen Signaturen. Es bezeichnet den Umstand, dass der Sender einer Nachricht im nachhinein nicht leugnen können soll, diese erstellt zu haben; die englische Bezeichnung hierfür ist *non repudiation*. Das Problem ist allein mit kryptographischen Routinen nicht zu lösen; statt dessen muss das gesamte Umfeld betrachtet werden und durch entsprechende Gesetze der Rahmen geschaffen werden.

Vertraulichkeit

Das Ziel, welches durch Verschlüsselung von Daten erreicht werden soll: Nur berechtigte Kommunikationspartner sollen bestimmte Informationen erhalten können.

XOR

Eine wichtige Funktion der Kryptographie ist das bitweise XOR. Es entspricht im binären Zahlraum der Addition ohne Überlauf, d.h. es gilt 0+0=0, 1+0=1, 0+1=1 und 1+1=0. Solche bitweise orientierten Funktionen sind interessant, da sie sich besonders schnell in Hardware implementieren lassen.

Zeitstempel

Für einige Anwendung innerhalb der Kryptographie (z.B. verbindliche digitale Signaturen) ist die Information wesentlich, zu welchem Zeitpunkt genau bestimmte Daten vorlagen. Dazu dient das Anhängen (und entsprechendes Signieren) eines Zeitstempels an die entsprechenden Nachrichten.

Zertifikat

Ein digitales Zertifikat ist ein elektronisches Dokument, das mit dem öffentlichen Schlüssel verbunden ist und eine vertrauenswürdigen Instanz (wie eine PKI) beglaubigt, dass der Schlüssel z.B. zu einer bestimmten Person gehört und nicht verändert wurde. Vorteile solcher Vorgehensweisen sind, dass man zur vollständigen Prüfung später nur noch den öffentlichen Schlüssel der sogenannten Wurzel-Instanz der PKI (und nicht von jedem Teilnehmer) benötigt.

Zero Knowledge

Ein Protokoll mit der Eigenschaft, dass nur ein einziges Bit übertragen wird: Der Empfänger soll überzeugt werden, dass der Absender eine bestimmte Information besitzt, über die Information selber soll er aber keinerlei Kenntnis (deshalb *zero knowledge* Protokoll) erhalten.

Zufallszahlen

Viele kryptographische Algorithmen oder Protokolle benötigen ein Zufallselement, meist in Form einer Zufallszahl, die jeweils neu erzeugt wird. In diesen Fällen hängt die Sicherheit des Verfahrens zum Teil von der Eignung dieser Zufallszahlen ab.

Da die Erzeugung echter Zufallszahlen innerhalb von Computern auch heute noch ein Problem darstellt (eine Quelle für echte Zufallsereignisse kann eigentlich nur die genaue Beobachtung physikalischer Ereignisse sein, was für eine Software nicht leicht realisierbar ist), behilft man sich statt dessen mit

sogenannten Pseudozufallszahlen.

Literaturverzeichnis

[AES99] http://csrc.nist.gov/encryption/aes

[ABK] R. Anderson, E. Biham, L. Knudson: „Serpent: A Proposal for Advanced Encryption Standard“, Paper, verfügbar unter: http://www.cl.cam.ac.uk./~rja14/serpent.html

[ADH94] L. Adleman, J. DeMarrais and M. Huang: "A subexponential algorithm for discrete logarithms over the rational subgroup of the jacobians of large genus hyperelliptic curves over finite fields", Algorithmic Number Theory, Lecture Notes in Computer Science, 877, Springer-Verlag, 1994.

[ANSI X9.9] (Revision) „American National Standard for Financial Institution Message Authentication (Wholesale)“, American Bankers Association, 1986

[ANSI X9.17] (Revision) „American National Standard for Financial Institution Key Management (Wholesale)“, American Bankers Association, 1985

[ANSI X9.19] (Revision) „American National Standard for Retail Message Authentication“, American Bankers Association, 1985

[ANSI X9.62] „Public Key Cryptography for the Financial Service Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)“, 1998

[ANSI X9.63] „Public Key Cryptography for the Financial Service Industry: The “Key Agreement and Key Transport Using Elliptic Curve Cryptography", 2001

[BBS86] L. Blum, M. Blum, M. Shub: „A Simple Unpredictable Pseudo-Random Number Generator“, SIAM Journal on Computing, 1986

[BK] R. Balasubramanian and N. Koblitz: "The improbability that an elliptic curve has subexponential discrete log problem under the Menezes-Okamoto Vanstone algorithm," wird im Journal of Cryptology erscheinen

[BL92] D. Boneh and R. Lipton: "Algorithms for black-box fields and their applications to cryptography", Advances in Cryptology - CRYPTO '96, Lecture Notes in Computer Science, 1109, Springer-Verlag, 1992.

[CCITT509] ISO/CCITT Recommendation X.509-„The Directory Authentification Framework“, 1989. -Identisch mit ISO 9594-8

[DBP96] H. Dobbertin, A. Bosselaers, B. Preneel: „RIPEMD-160: A strengthened version of RIPEMD.“ Fast Software Encryption, Cambridge Workshop, Lecture Notes in Computer Science, Bd. 1039, Springer-Verlag, 1996, S.71-82. (Eine korrigierte Fassung, die verbindliche Beschreibung von RIPEMD-160, sowie eine Referenzimplementierung in C findet man im Internet unter ftp://ftp.esat.kuleuven.ac.be/ im Directory /pub/COSIC/bosselae/ripemd/. Ein Internet Draft ist in Vorbereitung.)

[DH76] W. Diffie, M.E. Hellman: „New directions in cryptography.“ IEEE Transactions on Information Theory, IT-22, 1976.

[Dob96a] H. Dobbertin: „Cryptanalysis of MD4. Fast Software Encryption“, Cambridge Workshop, Lecture Notes in Computer Science, Bd. 1039, SpringerVerlag, 1996

[Dob96b] H. Dobbertin: „Cryptanalysis of MD5 compress“, Rump Session der Eurocrypt '96

[Dob97] Dobbertin, H.: „RIPEMD with tworound compress function is not collisionfree“, Journal of Cryptology, erscheint

[DR99] J. Daemon, V. Rijmen: “AES Proposal: Rijndael“, Version 2, Paper, November 1999

[ElG84] T. Gamal: „Cryptography and logarithms over finite fields", Lecture Notes in Computer Science, Advances in Cryptology: Proc. Crypto '84, 1984.

[ElG85] T. ElGamal: „A public key cryptosystem and signature scheme based in discrete logarithm“, IEEE Transaction on Information Theory, 31, 1985

[FIPS 46] FIPS PUBS 46-3: „Federal Information Processing Standards Publication 46-3, Data Encryption Standard (DES), National Institute for Standards and Technology“, Oktober 1999 (reaffirmed)

[FIPS180] FIPS PUBS 1801: „Federal Information Processing Standards Publication 1801, Secure Hash Standard (SHS)“, National Institute for Standards and Technology, April 1995

[FIPS 186] FIPS PUBS 186: „Digital Signature Standard“, May 1994

[FP97] R. Flassenberg and S. Paulus, „Sieving in function fields“, Preprint, 1997

[For96] O. Forster, „Algorithmische Zahlentheorie“, Vieweg-Verlag, 1996

[FR94] G. Frey and H. Rück: "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves", Mathematics of Computation, 62, 1994

[Gor98] W. Gora, „ASN.1 Abstract Syntax Notation One“, Fossil-Verlag, 1998

[HMP94] P. Horster, M. Michels, H. Petersen: "MetaElGamal signature schemes", Proc. 2. ACM conference on Computer and Communications security Fairfax. Virginia. 2.4., 1994

[HMP95] P. Horster, M. Michels, H. Petersen: "Digital signature schemes based on Lucas functions", Communication and Multimedia Security IT Security '95, 1995

[IBM99] IBM: „The MARS Encryption Algorithm“, Abstract, August 1999

[IEEE P1363] M. Abdalla, M. Bellare, P. Rogaway: "DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem", Paper, September 1998, erhältlich unter: http://grouper.ieee.org/groups/1363/index.html

[IEEE P1363E] IEEE P1363/Editorial Contribution (Draft), erhältlich unter: http://stdsbbs.ieee.org/groups/1363/edcont.html

[ISO99] ISO/IEC: „Information Technology – Security Techniques – Cryptographic Techniques based on elliptic Curves: Part 2 – Digital Signatures"

[ISO 8731] „Banking -- Approved algorithms for message authentication", 1987

[ISO 8732] „Banking-Key Management (Wholesale)", 1988

[ISO 9796] (Revision von 1991) ISO/IEC 9796: „Digital Signature Scheme giving message recovery", 1998

[ISO 9797] ISO/IEC 9797: „Information technology -- Security techniqes -- Data integrity mechanism using a cryptographic check function employing a block cipher algorithm", 1993

[ISO 14888] ISO/IEC FDIS 14888: „Information technology - Security techniques – Digital signatures with appendix", 1998

[ISO 15946] ISO/IEC WD 15946: „Cryptographic techniques based on elliptic curves", 1998

[Knu81] D. Knuth: „The Art of Computer Programming: Volume 2, Seminumerical Algorithm", 2nd edition, Addison Wessley, 1981

[Kob94] N. Koblitz: „A Course in Number Theory and Cryptography", Springer-Verlag, 1994

[KSF] J. Kelsey, B. Schneier, N. Ferguson: „Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" verfügbar unter: http://www.counterpane.com/yarrow.html

[Lec90] P. L`Ecuyer: „Random numbers for Simulation", Communications of the ACM, 1990

[LM90] X.Lai, J.L.Massey: „Proposal for a New Block Encryption Standard", Proceedings Eurocrypt '90, SpringerVerlag, LNCS 473 1990

[LM91] X. Lai and J.L. Massey: „Proposal for a New Block Encryption Standard", Advances in Cryptology, Proceedings of Eurocrypt '90, SpringerVerlag, Berlin 1991

[Mil86] V. Miller: "Uses of elliptic curves in cryptography", Advances in Cryptology CRYPTO '85, Lecture Notes in Computer Science, 218, Springer-Verlag, 1986

[MH78] R. Merkle, M. Hellman: „Hiding information and signatures in trapdoor knapsacks“, IEEE Transaction on Information Theory, 24, 1978

[MQV95] A. J. Menezes, M. Qu, and S. A. Vanstone: „Some key agreement protocols providing implicit authentication“, Presented in the 2nd Workshop on Selected Areas in Cryptography, 1995

[NR96] K. Nyberg and R. Rueppel: "Message recovery for signature schemes based on the discrete logarithm problem", Designs, Codes and Cryptography, volume 7, 1996

[PKCS #1] RSA Laboratories: „PKCS #1: RSA Encryption Standard“, Version 1.5, November 1993

[PKCS #3] RSA Laboratories: „PKCS #3: Diffie-Hellman Key-Agreement Standard“, Version 1.4, November 1993

[PKCS #5] RSA Laboratories: „PKCS #5: Password-Based Encryption Standard“, Version 1.5, November 1993

[PKCS #6] RSA Laboratories: „PKCS #6: Extended-Certificate Syntax Standard“, Version 1.5, November 1993

[PKCS #7] RSA Laboratories: „PKCS #7: Cryptographic Message Syntax Standard“, Version 1.5, November 1993

[PKCS #8] RSA Laboratories: „PKCS #8: Private-Key Information Syntax Standard“, Version 1.2, November 1993

[PKCS #9] RSA Laboratories: „PKCS #9: Selected Attribute Types“, Version 1.1, November 1993

[PKCS #10] RSA Laboratories: „PKCS #10: Certification Request Syntax Standard“, Version 1.0, November 1993

[PKCS #11] RSA Laboratories: „PKCS #11: Cryptographic Token Interface Standard”, Version 2.01, December 1997

[RC95] N. Rogier and P. Chauvaud: „The compression function of MD2 is not collision free“, Presented at Selected Areas in Cryptography '95, Ottawa, Canada, 1995

[RFC 1319] B.S. Jr. Kaliski: „The MD2 Message-Digest Algorithm“, RSA Laboratories,1992

[RFC 1320] R. Rivest: „The MD4 messagedigest algorithm.“ Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992

[RFC 1321] R. Rivest: „The MD5 messagedigest algorithm.“ Request for Comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, April 1992

[RFC 2119] S. Bradner: „Key words for use in RFCs to Indicate Requirement

Levels", BCP 14, Request for Comments (RFC) 2119, March 1997

[RFC 2144] C. Adams: „The CAST-128 Encryption Algorithm“, Request for Comments (RFC) 2144, May 1997

[RFC 2247] S. Kille, M. Wahl, A. Grimstad, R. Huber, and S. Sataluri. „Using Domains in LDAP/X.500 Distinguished Names", Request for Comments (RFC) 2247, January 1998

[RFC 2313] B. Kaliski: „PKCS #1: RSA Encryption Version 1.5", Request for Comments (RFC) 2313, March 1998

[RRFY98] R. Rivest, M. Robshaw, R. Sidney, Y. Yin: „The RC6™ Block Cipher“, Version 1.1, Paper, RSA Labs, August 1998

[Rüc97] H. Rück: „On the discrete logarithm in the divisor class group of curves“, Preprint 1997

[RSA78] R. L. Rivest, A. Shamir, L. M. Adleman: „A Method for Obtaining Digital Signatures and Public-Key Cryptosystems“, Communications of the ACM, v. 21, n. 2, Feb. 1978

[RSA79] R. L. Rivest, A. Shamir, L. M. Adleman: „On Digital Signatures and Public Key Cryptosystems“, MIT Laboratory for Computer Science, Technocal Report, MIT/LCS/TR-212, Jan 1979

[SA97] T. Satoh, K. Araki: „Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves“, Preprint, 1997

[Sch94] B. Schneier: „Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)“, Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994

[Sch97] B. Schneier: „Angewandte Kryptographie“, Addison-Wesley-Verlag, 1997

[Schn91] C. P. Schnorr: „Method for identifying Subscribers and for Generating and Verifying electronic Signatures in a Data Exchange System“, U.S. Patent #4,995,082, 19.02.1991

[Sec92] Proceedings of Securicom 92, an exceptional tutorial day on: "Electronic signature algorithms and protocols" by Ernest F. Brickell and Chris Holloway, CNIT Paris La Defense France, March 17, 1992.

[Sem] I. Semaev: „Evaluation of discrete logarithms on some elliptic curves“, wird in Mathematics of Computation erscheinen

[Sil86] J. H. Silverman: „The Arithmetic of Elliptic Curves“, Springer-Verlag, 1994

[SKW98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson: „Twofish: A 128-Bit Block Chipher“, Paper, Juni 1998

[Sma97] N. Smart: „Announcement of an attack on the ECDLP for anomalous

elliptic curves“, 1997

[SN77] J. Simmons, J.N. Norris: „Preliminary Comments on the M.I.T. Public Key Cryptosystem“, Cryptologica, Vol.1(4), 1977

[Ste96] A. Stein: "Equivalences between elliptic curves and real quadratic congruence function fields", presented at Pragocrypt '96.

[Stro98] B. Stroustrup: „Die C++ Programmiersprache“, Addison-Wesley-Verlag, 1998

[Tsu92] G. Tsudik: „Message Authentication with One-Way Hash Functions“, ACM Computer Communications Reviewv.22, n.5, 1992

[Vol97] J. Voloch: „The discrete logarithm problem on elliptic curves and descents“, Preprint, 1997

[VV83] U.V. Vazirani, V.V. Vazirani: „Trapdoor Pseudo-Random Number“, Proceedings of the 24th IEEE Symposium on the Foundations of Computer Science, 1983

[VV84] U.V. Vazirani, V.V. Vazirani: „Efficient and Secure Pseudo-Random Number Generator with Applications to Protocol Design“, Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science,1983

[VV85] U.V. Vazirani, V.V. Vazirani: „Efficient and Secure Pseudo-Random Number Generator“, Advances in Cryptography: Proceedings of CRYPTO ´84, Springer-Verlag, 1985

[X.501] ITU-T Recommendation X.501: „Information Technology - Open Systems Interconnection - The Directory: Models“,1993

[X.509] ITU-T Recommendation X.509: „Information Technology - Open Systems Interconnection-The Directory: Authentication Framework“, August 1997

[X.208] CITT Recommendation X.208: „Specification of Abstract Syntax Notation One (ASN.1)“, 1988

[Zuc96] R. Zuccherato, "The equivalence between elliptic curve and quadratic function field discrete logarithms in characteristic 2", Preprint, 1996.

Anhang A Quadratische Reste

Der Pseudozufallszahlengenerator von Blum, Blum und Shub basiert auf der Theorie der quadratischen Reste.

Ist p eine Primzahl und 0<a<p, so heißt a quadratischer Rest modulo p, falls es ein x gibt mit

$$x^2 \equiv a \bmod p.$$

Beispiel:

Die quadratischen Reste modulo 7 sind 1,2 und 4:

$1^2 \equiv 1 \bmod 7$
$2^2 \equiv 4 \bmod 7$
$3^2 \equiv 9 \bmod 7 \equiv 2 \bmod 7$
$4^2 \equiv 16 \bmod 7 \equiv 2 \bmod 7$
$5^2 \equiv 25 \bmod 7 \equiv 4 \bmod 7$
$6^2 \equiv 36 \bmod 7 \equiv 1 \bmod 7$

Denn:

Für a=1 gibt es ein x=1, so dass $1^2 \equiv 1 \bmod 7$
Für a=2 gibt es ein x=3, so dass $3^2 \equiv 2 \bmod 7$ oder ein x=4..
Für a=4 gibt es ein x=2, so dass $2^2 \equiv 4 \bmod 7$ oder ein x=5..

Man sollte beachten, dass jeder quadratische Rest zweimal in dieser Liste erscheint.

Allgemeiner gilt für eine ungerade Primzahl p:

Für ein p gibt es genau (p-1)/2 quadratische Reste modulo p.

Anhang B ElGamal Kryptographie

Das Verschlüsselungsverfahren nach ElGamal

ElGamal hat beschrieben, wie man die Annahmen von Diffie und Hellman für ein asymmetrisches Verschlüsselungsverfahren nutzen kann:

Alice sendet Bob als verschlüsselte Nachricht das Paar

$$(g^a , M\, g^{a\,b}),$$

wobei M die Klartext-Nachricht ist.

Bob hat den Wert b und kann damit $g^{a\,b}$ und auch $(g^{ab})^{-1}$ berechnen und so die Nachricht durch

$$(M\, g^{a\,b}) * (g^{a\,b})^{-1} = M$$

dechiffrieren. Dieser Algorithmus bildet eigentlich eine Klasse von Verfahren. Dazu gehört z.B. EC-DSA oder DSA.

Das Signaturschemas von ElGamal

Das Signaturverfahren ist ebenfalls eine Variation des Protokolls von Diffie und Hellman zum Schlüsselaustausch. Der Algorithmus wurde ursprünglich für das DL-Problem in endlichen Körpern formuliert, ist aber auch auf andere Gruppen übertragbar.

Für das Signatur-Schema von ElGamal wählt man eine Primzahl p und ein g prim zu p. Es besteht aus folgenden Teilen:

- Schlüsselgenerierung:

Jeder Benutzer wählt sich seinen geheimen Schlüssel x, der kleiner als p sein sollte. Dann ist der zugehörige öffentliche Schlüssel $y=g^x$.

- Erzeugen einer Signatur

Um eine Unterschrift zu einer Nachricht m zu erzeugen, wählt man eine zufällige, zu p-1 teilerfremde Zahl k und berechnet damit

$$r \equiv g^k \bmod p$$

$$s \equiv (m-xr)\, k^{-1} \bmod (p-1)$$

Die vollständige Signatur besteht dann aus dem Tupel (m,r,s).

- Überprüfen einer Signatur

Mit Hilfe des öffentlichen bekannten Schlüssels y erfolgt die Überprüfung dadurch, indem man nachrechnet, ob

$$g^m \equiv r^s y^r \bmod p$$

für die empfangenen Werte erfüllt ist.

Sicherheit und Patente

Die Sicherheit dieser Verfahren beruht auf der Schwierigkeit des DL-Problems.

Ein Vorteil von den ElGamal-Verfahren ist, dass es nicht patentiert ist. Nach Ansicht eines amerikanischen Patentverwalters ist es zwar vom Diffie-Hellman-Patents abgedeckt, jedoch lief dieses Patent 1997 aus.

 [ElG85]

Anhang C ASN.1

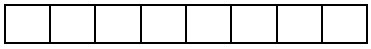
Die Methode, um Zertifikate in einer maschinenunabhängigen Form zu erzeugen, ist ASN.1 und wird von der ISO als „abstrakte Syntaxnotation“ (deshalb auch die Abkürzung) bezeichnet. Mit dieser Syntax kann eine Menge von Datentypen definiert werden.

Als Kodierung wird die bitweise Repräsentation für einen beliebigem Wert eines Datentyps bezeichnet. Dies geschieht in Oktetten, die Darstellung eines Wertes besteht also aus einer Folge von 8 Bit.

Die *Basic Encoding Rules* (kurz BER) ordnen der abstrakten Syntax von ASN.1 konkrete Kodierungs- und Dekodierungsregeln zu. Es werden ein oder mehrere Wege beschrieben, einen ASN.1-Wert als einen Oktett-String zu repräsentieren.

Die Numerierung von Bit- und Oktettfolgen hat folgende Form:

1. Oktett



8 7 6 5 4 3 2 1 := Bitpositionen

1.Oktett 2. Oktett 3. Oktett .....n. Oktett
[][][][][][][][] [][][][][][][][] [][][][][][][][] .....[][][][][][][][]

- Ein Oktett besteht aus 8 Bits, die Zählweise innerhalb beginnt rechts mit der 1

- Bit 1 ist das niedrigstwertige (2^0) Bit und Bit 8 führt den höchsten Wert (2^7) eines Oktetts mit sich

- Die Numerierung einer Oktettfolge erfolgt von links nach rechts

Die Kodierung eines Datenwertes gliedert sich bei den BER in maximal vier Komponenten auf:

Kodierungsmöglichkeit 1 (ohne EOC= End of Content):

Bezeichnerfeld	Längenfeld	Inhaltsfeld

Kodierungsmöglichkeit 2 (mit EOC = End of Content):

Bezeichnerfeld	Längenfeld	Inhaltsfeld	EOC-Oktette

Die *Distinguished Encoding Rules* for ASN.1 (kurz DER) sind eine Untermenge der BER und beschreiben einen eindeutigen Weg, einen ASN.1-Wert als Oktett-String zu repräsentieren. DER wurde für den Fall eingeführt, dass eine eindeutige Darstellung benötigt wird, z.B. bei der Berechnung einer digitalen Signatur auf einem ASN.1- Wert.

Anhang D Elliptische Kurven Parameter

In diesem Anhang finden Sie die elliptischen Kurven, die in der **cv act *library*** implementiert sind. Es wird zwischen Kurven mit GF(p) (p prim) als zugrundeliegendem Körper und Kurven mit GF(2^m) als zugrundeliegendem Körper unterschieden. Dabei ist z.B. `ANSIp192r1` der Name der elliptische Kurve.

Darüber hinaus werden die Funktionen beschrieben, mit denen eigene Kurven mit der **cv act *library*** verwendet werden können. Ein Beispiel für die Verwendung dieser Funktionen findet sich in Abschnitt 5. 2. 4.

D1 Kurven über GF(p)

Für eine elliptische Kurve mit der Gleichung

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

gilt folgende Notation:

p	Primzahl mit GF(p) zugrundeliegender Körper
a	Koeffizient der elliptischen Kurve
b	Koeffizient der elliptischen Kurve
x	x-Koordinate des Basispunktes
y	y-Koordinate des Basispunktes
n	Ordnung des Basispunktes
h	Cofactor

Kurven über GF(p), die nicht in der **cv act *library*** implementiert sind, können mit den folgenden Methoden angelegt werden. Alle Parameter der Kurve werden mit der Methode `Blob EncodeNumber (const char* number)` in 2er-Komplement-Darstellung umgewandelt. Die weiteren Methoden erwarten alle Parameter in dieser Darstellung. Die Bedeutung der Variablen p, a, b, n und h entspricht der obigen Tabelle. Bezüglich der Angabe des Basispunktes unterscheiden sich die beiden Methoden. Der Basispunkt wird als Blob ausgegeben. Mit `output_type` wird das Format der Ausgabe festgelegt, zur Auswahl stehen komprimierter, unkomprimierter (Standard) oder hybrider Oktetstring gemäß ANSI X9.62/63.

```
Blob EncodePrimeCurveParam (const Blob& p, const Blob& a, const Blob& b, const
Blob& Gx, const Blob& Gy, const Blob& n, const Blob& h, int
output_type=UNCOMPRESSED);
```

Gx bzw. Gy bezeichnen die x- bzw. y-Koordinate des Basispunktes.

```
Blob EncodePrimeCurveParam (const Blob& p, const Blob& a, const Blob& b, const
Blob& G, const Blob& n, const Blob& h, int output_type=UNCOMPRESSED);
```

In diesem Fall wird der Basispunkt G gemäß ANSI X9.62/63 als komprimierter, unkomprimierter oder hybrider Oktetstring angegeben. In der **cv act *library*** sind folgende elliptische Kurven über GF(p) implementiert:

```
ANSIp192r1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF
```

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC

b = 0x64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1

x = 0x188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012

y = 0x7192B95FFC8DA78631011ED6B24CDD573F977A11E794811

n = 0xFFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831

h = 0x1

ANSIp192r2

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC

b = 0xCC22D6DFB95C6B25E49C0D6364A4E5980C393AA21668D953

x = 0xEEA2BAE7E1497842F2DE7769CFE9C989C072AD696F48034A

y = 0x6574D11D69B6EC7A672BB82A083DF2F2B0847DE970B2DE15

n = 0xFFFFFFFFFFFFFFFFFFFFFFFE5FB1A724DC80418648D8DD31

h = 0x1

ANSIp192r3

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC

b = 0x22123DC2395A05CAA7423DAECCC94760A7D462256BD56916

x = 0x7D29778100C65A1DA1783716588DCE2B8B4AEE8E228F1896

y = 0x38A90F22637337334B49DCB66A6DC8F9978ACA7648A943B0

n = 0xFFFFFFFFFFFFFFFFFFFFFFFF7A62D031C83F4294F640EC13

h = 0x1

ANSIp239r1

p = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFFFFFFFF8000000000007FFFFFFFFFF

a = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFFFFFFFF8000000000007FFFFFFFFFC

b = 0x6B016C3BDCF18941D0D654921475CA71A9DB2FB27D1D37796185C2942C0A

x = 0xFFA963CDCA8816CCC33B8642BEDF905C3D358573D3F27FBBD3B3CB9AAAF

y = 0x7DEBE8E4E90A5DAE6E4054CA530BA04654B36818CE226B39FCCB7B02F1AE

n = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFF9E5E9A9F5D9071FBD1522688909D0B

h = 0x1

ANSIp239r2

p = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFFFFFFFF8000000000007FFFFFFFFFF

a = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFFFFFFFF8000000000007FFFFFFFFFC

b = 0x617FAB6832576CBBFED50D99F0249C3FEE58B94BA0038C7AE84C8C832F2C

x = 0x38AF09D98727705120C921BB5E9E26296A3CDCF2F35757A0EAFD87B830E7

y = 0x5B0125E4DBEA0EC7206DA0FC01D9B081329FB555DE6EF460237DFF8BE4BA

n = 0x7FFFFFFFFFFFFFFFFFFFFFFF800000CFA7E8594377D414C03821BC582063

h = 0x1

ANSIp239r3

p = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFFFFFFFF8000000000007FFFFFFFFFF

a = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFFFFFFFF8000000000007FFFFFFFFFC

b = 0x255705FA2A306654B1F4CB03D6A750A30C250102D4988717D9BA15AB6D3E

x = 0x6768AE8E18BB92CFCF005C949AA2C6D94853D0E660BBF854B1C9505FE95A

y = 0x1607E6898F390C06BC1D552BAD226F3B6FCFE48B6E818499AF18E3ED6CF3

n = 0x7FFFFFFFFFFFFFFFFFFFFFF7FFFFF975DEB41B3A6057C3C432146526551

h = 0x1

ANSIp256r1

p = 0xFFFFFFFF000000010000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF

a = 0xFFFFFFFF000000010000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC

b = 0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B

x = 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296

y = 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5

n = 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551

h = 0x1

NISTp192r1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC

b = 0x64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1

x = 0x188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012

y = 0x7192B95FFC8DA78631011ED6B24CDD573F977A11E794811

n = 0xFFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831

h = 0x1

NISTp224r1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFE

b = 0xB4050A850C04B3ABF54132565044B0B7D7BFD8BA270B39432355FFB4

x = 0xB70E0CBD6BB4BF7F321390B94A03C1D356C21122343280D6115C1D21

y = 0xBD376388B5F723FB4C22DFE6CD4375A05A07476444D5819985007E34

n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF16A2E0B8F03E13DD29455C5C2A3D

h = 0x1

NISTp256r1

p = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF

a = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC

b = 0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B

x = 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296

y = 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5

n = 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551

h = 0x1

NISTp384r1

p=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FEFFFFFFFF0000000000000000FFFFFFFF

a=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFEFFFFFFFF0000000000000000FFFFFFFC

b=0xB3312FA7E23EE7E4988E056BE3F82D19181D9C6EFE8141120314088F5013875
AC656398D8A2ED19D2A85C8EDD3EC2AEF

x=0xAA87CA22BE8B05378EB1C71EF320AD746E1D3B628BA79B9859F741E082542A3
85502F25DBF55296C3A545E3872760AB7

y=0x3617DE4A96262C6F5D9E98BF9292DC29F8F41DBD289A147CE9DA3113B5F0B8C
00A60B1CE1D7E819D7A431D7C90EA0E5F

n=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC7634D81F4372DD
F581A0DB248B0A77AECEC196ACCC52973

h=0x1

NISTp521r1

p=0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

a=0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC

b=0x51953EB9618E1C9A1F929A21A0B68540EEA2DA725B99B315F3B8B489918EF109E
156193951EC7E937B1652C0BD3BB1BF073573DF883D2C34F1EF451FD46B503F00

x=0xC6858E06B70404E9CD9E3ECB662395B4429C648139053FB521F828AF606B4D3DBA
A14B5E77EFE75928FE1DC127A2FFA8DE3348B3C1856A429BF97E7E31C2E5BD66

y=0x11839296A789A3BC0045C8A5FB42C7D1BD998F54449579B446817AFBD17273E662C
97EE72995EF42640C550B9013FAD0761353C7086A272C24088BE94769FD16650

n=0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA5
1868783BF2F966B7FCC0148F709A5D03BB5C9B8899C47AEBB6FB71E91386409

h=0x1

SECGp112r1

p = 0xDB7C2ABF62E35E668076BEAD208B

a = 0xDB7C2ABF62E35E668076BEAD2088

b = 0x659EF8BA043916EEDE8911702B22

x = 0x9487239995A5EE76B55F9C2F098

y = 0xA89CE5AF8724C0A23E0E0FF77500

n = 0xDB7C2ABF62E35E7628DFAC6561C5

h = 0x1

SECGp112r2

p = 0xDB7C2ABF62E35E668076BEAD208B

a = 0x6127C24C05F38A0AAAF65C0EF02C

b = 0x51DEF1815DB5ED74FCC34C85D709

x = 0x4BA30AB5E892B4E1649DD0928643

y = 0xADCD46F5882E3747DEF36E956E97

n = 0x36DF0AAFD8B8D7597CA10520D04B

h = 0x4

SECGp128r1

p = 0xFFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFF

a = 0xFFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFC

b = 0xE87579C11079F43DD824993C2CEE5ED3

x = 0x161FF7528B899B2D0C28607CA52C5B86

y = 0xCF5AC8395BAFEB13C02DA292DDED7A83

n = 0xFFFFFFFE0000000075A30D1B9038A115

h = 0x1

SECGp128r2

p = 0xFFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFF

a = 0xD6031998D1B3BBFEBF59CC9BBFF9AEE1

b = 0x5EEEFCA380D02919DC2C6558BB6D8A5D

x = 0x7B6AA5D85E572983E6FB32A7CDEBC140

y = 0x27B6916A894D3AEE7106FE805FC34B44

n = 0x3FFFFFFF7FFFFFFFBE0024720613B5A3

h = 0x4

SECGp160k1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFAC73

a = 0x0

b = 0x7

x = 0x3B4C382CE37AA192A4019E763036F4F5DD4D7EBB

y = 0x938CF935318FDCED6BC28286531733C3F03C4FEE

n = 0x100000000000000000001B8FA16DFAB9ACA16B6B3

h = 0x1

SECGp160r1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFF

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFC

b = 0x1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45

x = 0x4A96B5688EF573284664698968C38BB913CBFC82

y = 0x23A628553168947D59DCC912042351377AC5FB32

n = 0x100000000000000000001F4C8F927AED3CA752257

h = 0x1

SECGp192k1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFEE37

a = 0x0

b = 0x3

x = 0xDB4FF10EC057E9AE26B07D0280B7F4341DA5D1B1EAE06C7D

y = 0x9B2F2F6D9C5628A7844163D015BE86344082AA88D95E2F9D

n = 0xFFFFFFFFFFFFFFFFFFFFFFFE26F2FC170F69466A74DEFD8D

h = 0x1

SECGp192r1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC

b = 0x64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1

x = 0x188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012

y = 0x7192B95FFC8DA78631011ED6B24CDD573F977A11E794811

n = 0xFFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831

h = 0x1

SECGp224k1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFE56D

a = 0x0

b = 0x5

x = 0xA1455B334DF099DF30FC28A169A467E9E47075A90F7E650EB6B7A45C

y = 0x7E089FED7FBA344282CAFBD6F7E319F7C0B0BD59E2CA4BDB556D61A5

n = 0x10000000000000000000000001DCE8D2EC6184CAF0A971769FB1F7

h = 0x1

SECGp224r1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFBFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

a = 0xFFFFFFFFFFFFFFFFFFFFFFFFBFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC

b = 0x2AC38C3B71A514ADF2181EBCCFBB4D09551C93049D15AA8969E04497

x = 0xFAA3234E2F84A96B568466469844FA91A32E197198A12314

y = 0x21AFFACB1CB4815B948F9811D75228AACAF91ECA44CE79B0F747FB53

n = 0xFFFFFFFFFFFFFFFFFFFFFFFFC00008DFC64C6909629C138D91C844D3D

h = 0x1

SECGp256k1

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2F

a = 0x0

b = 0x7

x = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798

y = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8

n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

h = 0x1

SECGp256r1

p = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF

a = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC

b = 0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B

x = 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296

y = 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5

n = 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551

h = 0x1

D2 Kurven über GF(2^m)

Für eine elliptische Kurve mit der Gleichung

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

gilt folgende Notation:

f	erzeugendes Polynom mit GF(2^m) als zugrundeliegender Körper
a	Koeffizient der elliptischen Kurve
b	Koeffizient der elliptischen Kurve
x	x-Koordinate des Basispunktes
y	y-Koordinate des Basispunktes
n	Ordnung des Basispunktes
h	Cofactor

Kurven über GF(2^m), die nicht in der **cv act *library*** implementiert sind, können mit den folgenden Methoden angelegt werden. Alle Parameter der Kurve werden mit der Methode `Blob EncodeNumber (const char* number)` in 2er-Komplement-Darstellung umgewandelt. Die weiteren Methoden erwarten alle Parameter in dieser Darstellung. Die Bedeutung der Variablen p, a, b, n und h entspricht der obigen Tabelle. Bezüglich der Angabe des Basispunktes unterscheiden sich die beiden Methoden. Der Basispunkt wird als Blob ausgegeben. Mit `output_type` wird das Format der Ausgabe festgelegt, zur Auswahl stehen komprimierter, unkomprimierter (Standard) oder hybrider Oktetstring gemäß ANSI X9.62/63.
Für Kurven mit Trinomialbasis hat das erzeugende Polynom die Form $f = x^m + x^k + 1$. Es stehen folgende Methoden zur Verfügung:

```
Blob EncodeTrinomialCurveParam (const int m, const int k, const Blob& a, const
Blob& b, const Blob& G, const Blob& n, const Blob& h, int
output_type=UNCOMPRESSED);
```

In diesem Fall wird der Basispunkt gemäß ANSI X9.62/63 als komprimierter, unkomprimierter (Standard) oder hybrider Oktetstring angegeben.

Für Kurven mit Pentanomialbasis hat das erzeugende Polynom die Form $f = x^m + x^{k3} + x^{k2} + x^{k1} + 1$. Es stehen folgende Methoden zur Verfügung:

```
Blob EncodePentanomialCurveParam (const int m, const int k3, const int k2,
const int k1, const Blob& a, const Blob& b, const Blob& Gx, const Blob& Gy,
const Blob& n, const Blob& h, int output_type=UNCOMPRESSED);
```

Gx bzw. Gy bezeichnen die x- bzw. y-Koordinate des Basispunktes. `Blob Blob EncodePentanomialCurveParam (const int m, const int k3, const int k2, const int k1, const Blob& a, const Blob& b, const Blob& G, const Blob& n, const Blob& h, int output_type=UNCOMPRESSED);`

In diesem Fall wird der Basispunkt G gemäß ANSI X9.62/63 als komprimierter, unkomprimierter oder hybrider Oktetstring angegeben. In der **cv act *library*** sind folgende elliptische Kurven über GF(p) implementiert:

ANSIz163r1

$f = x^{163} + x^8 + x^2 + x + 1$

a = 0x72546B5435234A422E0789675F432C89435DE5242

b = 0x0C9517D06D5240D3CFF38C74B20B6CD4D6F9DD4D9

x = 0x7AF69989546103D79329FCC3D74880F33BBE803CB

y = 0x1EC23211B5966ADEA1D3F87F7EA5848AEF0B7CA9F

n = 0x400000000000000000001E60FC8821CC74DAEAFC1

h = 0x2

ANSIz163r2

$f = x^{163} + x^8 + x^2 + x + 1$

a = 0x108B39E77C4B108BED981ED0E890E117C511CF072

b = 0x667ACEB38AF4E488C407433FFAE4F1C811638DF20

x = 0x024266E4EB5106D0A964D92C4860E2671DB9B6CC5

y = 0x79F684DDF6684C5CD258B3890021B2386DFD19FC5

n = 0x3FFFFFFFFFFFFFFFFFFDF64DE1151ADBB78F10A7

h = 0x2

ANSIz163r3

$f = x^{163} + x^8 + x^2 + x + 1$

a = 0x7A526C63D3E25A256A007699F5447E32AE456B50E

b = 0x3F7061798EB99E238FD6F1BF95B48FEEB4854252B

x = 0x2F9F87B7C574D0BDECF8A22E6524775F98CDEBDCB

y = 0x5B935590C155E17EA48EB3FF3718B893DF59A05D0

n = 0x3FFFFFFFFFFFFFFFFFFFE1AEE140F110AFF961309

h = 0x2

ANSIz176w1

f = $x^{176} + x^{43} + x^2 + x + 1$

a = 0x0E4E6DB2995065C407D9D39B8D0967B96704BA8E9C90B

b = 0x05DDA470ABE6414DE8EC133AE28E9BBD7FCEC0AE0FFF2

x = 0x08D16C2866798B600F9F08BB4A8E860F3298CE04A5798

y = 0x06FA4539C2DADDDD6BAB5167D61B436E1D92BB16A562C

n = 0x10092537397ECA4F6145799D62B0A19CE06FE26AD

h = 0xFF6E

ANSIz191r1

f = $x^{191} + x^9 + 1$

a = 0x2866537B676752636A68F56554E12640276B649EF7526267

b = 0x2E45EF571F00786F67B0081B9495A3D95462F5DE0AA185EC

x = 0x36B3DAF8A23206F9C4F299D7B21A9C369137F2C84AE1AA0D

y = 0x765BE73433B3F95E332932E70EA245CA2418EA0EF98018FB

n = 0x40000000000000000000000004A20E90C39067C893BBB9A5

h = 0x2

ANSIz191r2

f = $x^{191} + x^9 + 1$

a = 0x401028774D7777C7B7666D1366EA432071274F89FF01E718

b = 0x0620048D28BCBD03B6249C99182B7C8CD19700C362C46A01

x = 0x3809B2B7CC1B28CC5A87926AAD83FD28789E81E2C9E3BF10

y = 0x17434386626D14F3DBF01760D9213A3E1CF37AEC437D668A

n = 0x2000000000000000000000050508CB89F652824E06B8173

h = 0x4

ANSIz191r3

f = $x^{191} + x^{9} + 1$

a = 0x6C010747560991222210569 11C77D77E77A777E7E7E77FCB

b = 0x71FE1AF926CF847989EFEF8DB459F66394D90F32AD3F15E8

x = 0x375D4CE24FDE434489DE8746E71786015009E66E38A926DD

y = 0x545A39176196575D985999366E6AD34CE0A77CD7127B06BE

n = 0x155555555555555555555555610C0B196812BFB6288A3EA3

h = 0x6

ANSIz208w1

f = $x^{208} + x^{83} + x^{2} + x + 1$

a = 0x0

b = 0x0C8619ED45A62E6212E1160349E2BFA844439FAFC2A3FD1638F9E

x = 0x089FDFBE4ABE193DF9559ECF07AC0CE78554E2784EB8C1ED1A57A

y = 0x00F55B51A06E78E9AC38A035FF520D8B01781BEB1A6BB08617DE3

n = 0x101BAF95C9723C57B6C21DA2EFF2D5ED588BDD5717E212F9D

h = 0xFE48

ANSIz239r1

f = $x^{239} + x^{36} + 1$

a = 0x32010857077C5431123A46B808906756F543423E8D27877578125778AC76

b = 0x790408F2EEDAF392B012EDEFB3392F30F4327C0CA3F31FC383C422AA8C16

x = 0x57927098FA932E7C0A96D3FD5B706EF7E5F5C156E16B7E7C86038552E91D

y = 0x61D8EE5077C33FECF6F1A16B268DE469C3C7744EA9A971649FC7A9616305

n = 0x2000000000000000000000000000000F4D42FFE1492A4993F1CAD666E447

h = 0x4

ANSIz239r2

f = $x^{239} + x^{36} + 1$

a = 0x4230017757A767FAE42398569B746325D45313AF0766266479B75654E65F

b = 0x5037EA654196CFF0CD82B2C14A2FCF2E3FF8775285B545722F03EACDB74B

x = 0x28F9D04E900069C8DC47A08534FE76D2B900B7D7EF31F5709F200C4CA205

y = 0x5667334C45AFF3B5A03BAD9DD75E2C71A99362567D5453F7FA6E227EC833

n = 0x155555555555555555555555555553C6F2885259C31E3FCDF154624522D

h = 0x6

ANSIz239r3

f = $x^{239} + x^{36} + 1$

a = 0x01238774666A67766D6676F778E676B66999176666E687666D8766C66A9F

b = 0x6A941977BA9F6A435199ACFC51067ED587F519C5ECB541B8E44111DE1D40

x = 0x70F6E9D04D289C4E89913CE3530BFDE903977D42B146D539BF1BDE4E9C92

y = 0x2E5A0EAF6E5E1305B9004DCE5C0ED7FE59A35608F33837C816D80B79F461

n = 0xCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAC4912D2D9DF903EF9888B8A0E4CFF

h = 0xA

ANSIz272w1

f = $x^{272} + x^{56} + x^{3} + x + 1$

a = 0x091A091F03B5FBA4AB2CCF49C4EDD220FB028712D42BE752B2C40094DBACDB586FB20

b = 0x07167EFC92BB2E3CE7C8AAAFF34E12A9C557003D7C73A6FAF003F99F6CC8482E540F7

x = 0x06108BABB2CEEBCF787058A056CBE0CFE622D7723A289E08A07AE13EF0D10D171DD8D

y = 0x010C7695716851EEF6BA7F6872E6142FBD241B830FF5EFCACECCAB05E02005DDE9D23

n = 0x100FAF51354E0E39E4892DF6E319C72C8161603FA45AA7B998A167B8F1E629521

h = 0xFF06

ANSIz304w1

f = $x^{304} + x^{11} + x^2 + x + 1$

a = 0x0FD0D693149A118F651E6DCE6802085377E5F882D1B510B44160074C1288078365A039
6C8E681

b = 0x0BDDB97E555A50A908E43B01C798EA5DAA6788F1EA2794EFCF57166B8C14039601E5582
7340BE

x = 0x0197B07845E9BE2D96ADB0F5F3C7F2CFFBD7A3EB8B6FEC35C7FD67F26DDF6285A644F7
40A2614

y = 0x0E19FBEB76E0DA171517ECF401B50289BF014103288527A9B416A105E80260B549FDC
1B92C03B

n = 0x101D556572AABAC800101D556572AABAC8001022D5C91DD173F8FB561DA6899
164443051D

h = 0xFE2E

ANSIz359r1

f = $x^{359} + x^{68} + 1$

a = 0x5667676A654B20754F356EA92017D946567C46675556F19556A04616B567D223A5E05
656FB549016A96656A557

b = 0x2472E2D0197C49363F1FE7F5B6DB075D52B6947D135D8CA445805D39BC345626089687
742B6329E70680231988

x = 0x3C258EF3047767E7EDE0F1FDAA79DAEE3841366A132E163ACED4ED2401DF9C6BDCDE98E
8E707C07A2239B1B097

y = 0x53D7E08529547048121E9C95F3791DD804963948F34FAE7BF44EA82365DC7868FE57E4A
E2DE211305A407104BD

n = 0x1AF286BCA1AF286BCA1AF286BCA1AF286BCA1AF286BC9FB8F6B85C556892C20A7EB964F
E7719E74F490758D3B

h = 0x4C

ANSIz368w1

$f = x^{368} + x^{85} + x^2 + x + 1$

a = 0x0E0D2EE25095206F5E2A4F9ED229F1F256E79A0E2B455970D8D0D865BD94778C576D62F0
AB7519CCD2A1A906AE30D

b = 0x0FC1217D4320A90452C760A58EDCD30C8DD069B3C34453837A34ED50CB54917E1C2112D
84D164F444F8F74786046A

x = 0x01085E2755381DCCCE3C1557AFA10C2F0C0C2825646C5B34A394CBCFA8BC16B22E7E78
9E927BE216F02E1FB136A5F

y = 0x07B3EB1BDDCBA62D5D8B2059B525797FC73822C59059C623A45FF3843CEE8F87CD1855A
DAA81E2A0750B80FDA2310

n = 0x10090512DA9AF72B08349D98A5DD4C7B0532ECA51CE03E2D10F3B7AC579BD87E909AE40
A6F131E9CFCE5BD967

h = 0xFF70

ANSIz431r1

$f = x^{431} + x^{120} + 1$

a = 0x1A827EF00DD6FC0E234CAF046C6A5D8A85395B236CC4AD2CF32A0CADBDC9DDF620B0EB99
06D0957F6C6FEACD615468DF104DE296CD8F

b =
0x10D9B4A3D9047D8B154359ABFB1B7F5485B04CEB868237DDC9DEDA982A679A5A919B626D4
E50A8DD731B107A9962381FB5D807BF2618

x = 0x120FC05D3C67A99DE161D2F4092622FECA701BE4F50F4758714E8A87BBF2A658EF8C21E7
C5EFE965361F6C2999C0C247B0DBD70CE6B7

y = 0x20D0AF8903A96F8D5FA2C255745D3C451B302C9346D9B7E485E7BCE41F6B591F3E8F6ADD
CBB0BC4C2F947A7DE1A89B625D6A598B3760

n = 0x3403403403403403403403403403403403403403403403403403403403403403403403
B5EC68D3587FEC60D161CC149C1AD4A91

h = 0x2760

NISTz163r1

$f = x^{163} + x^7 + x^6 + x^3 + 1$

a = 0x1

b = 0x20A601907B8C953CA1481EB10512F78744A3205FD

x = 0x3F0EBA16286A2D57EA0991168D4994637E8343E36

y = 0x0D51FBC6C71A0094FA2CDD545B11C5C0C797324F1

n = 0x40000000000000000000292FE77E70C12A4234C33

h = 0x2

NISTz163k1

$f = x^{163} + x^7 + x^6 + x^3 + 1$

a = 0x1

b = 0x1

x = 0x2FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8

y = 0x289070FB05D38FF58321F2E800536D538CCDAA3D9

n = 0x4000000000000000000020108A2E0CC0D99F8A5EF

h = 0x2

NISTz233r1

$f = x^{233} + x^{74} + 1$

a = 0x1

b = 0x066647EDE6C332C7F8C0923BB58213B333B20E9CE4281FE115F7D8F90AD

x = 0x0FAC9DFCBAC8313BB2139F1BB755FEF65BC391F8B36F8F8EB7371FD558B

y = 0x1006A08A41903350678E58528BEBF8A0BEFF867A7CA36716F7E01F81052

n = 0x1000000000000000000000000000013E974E72F8A6922031D2603CFE0D7

h = 0x2

NISTz233k1

$f = x^{233} + x^{74} + 1$

a = 0x0

b = 0x1

x = 0x17232BA853A7E731AF129F22FF4149563A419C26BF50A4C9D6EEFAD6126

y = 0x1DB537DECE819B7F70F555A67C427A8CD9BF18AEB9B56E0C11056FAE6A3

n = 0x8000000000000000000000000000069D5BB915BCD46EFB1AD5F173ABDF

h = 0x4

NISTz283r1

$f = x^{283} + x^{12} + x^{7} + x^{5} + 1$

a = 0x1

b = 0x27B680AC8B8596DA5A4AF8A19A0303FCA97FD7645309FA2A581485AF6263E313B79A2F5

x = 0x5F939258DB7DD90E1934F8C70B0DFEC2EED25B8557EAC9C80E2E198F8CDBECD86B12053

y = 0x3676854FE24141CB98FE6D4B20D02B4516FF702350EDDB0826779C813F0DF45BE8112F4

n = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEF90399660FC938A90165B042A7CEFADB307

h = 0x2

NISTz283k1

$f = x^{283} + x^{12} + x^{7} + x^{5} + 1$

a = 0x0

b = 0x1

x = 0x503213F78CA44883F1A3B8162F188E553CD265F23C1567A16876913B0C2AC24
58492836

y = 0x1CCDA380F1C9E318D90F95D07E5426FE87E45C0E8184698E45962364E34116
177DD2259

n = 0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE9AE2ED07577265DFF7F94451E0
61E163C61

h = 0x4

NISTz409r1

f = $x^{409} + x^{87} + 1$

a = 0x1

b = 0x021A5C2C8EE9FEB5C4B9A753B7B476B7FD6422EF1F3DD674761FA99D6AC27C8A9A
197B272822F6CD57A55AA4F50AE317B13545F

x = 0x15D4860D088DDB3496B0C6064756260441CDE4AF1771D4DB01FFE5B34E59703DC2
55A868A1180515603AEAB60794E54BB7996A7

y = 0x061B1CFAB6BE5F32BBFA78324ED106A7636B9C5A7BD198D0158AA4F5488D08F385
14F1FDF4B4F40D2181B3681C364BA0273C706

n = 0x10000000000000000000000000000000000000000000000001E2AAD6A612F33
307BE5FA47C3C9E052F838164CD37D9A21173

h = 0x2

NISTz409k1

f = $x^{409} + x^{87} + 1$

a = 0x0

b = 0x1

x = 0x060F05F658F49C1AD3AB1890F7184210EFD0987E307C84C27ACCFB8F9F67CC2C46
0189EB5AAAA62EE222EB1B35540CFE9023746

y = 0x1E369050B7C4E42ACBA1DACBF04299C3460782F918EA427E6325165E9EA10E3DA5
F6C42E9C55215AA9CA27A5863EC48D8E0286B

n = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE5F83B2D4EA2040
0EC4557D5ED3E3E7CA5B4B5C83B8E01E5FCF

h = 0x4

NISTz571r1

$f = x^{571} + x^{10} + x^5 + x^2 + 1$

a = 0x1

b = 0x2F40E7E2221F295DE297117B7F3D62F5C6A97FFCB8CEFF1CD6BA8CE4A9A18AD84FFABBD8
EFA59332BE7AD6756A66E294AFD185A78FF12AA520E4DE739BACA0C7FFEFF7F2955727A

x = 0x303001D34B856296C16C0D40D3CD7750A93D1D2955FA80AA5F40FC8DB7B2ABDBDE5395
0F4C0D293CDD711A35B67FB1499AE60038614F1394ABFA3B4C850D927E1E7769C8EEC2D19

y = 0x37BF27342DA639B6DCCFFFEB73D69D78C6C27A6009CBBCA1980F8533921E8A684423E4
3BAB08A576291AF8F461BB2A8B3531D2F0485C19B16E2F1516E23DD3C1A4827AF1B8AC15B

n = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
E661CE18FF55987308059B186823851EC7DD9CA1161DE93D5174D66E8382E9BB2FE84E47

h = 0x2

NISTz571k1

$f = x^{571} + x^{10} + x^5 + x^2 + 1$

a = 0x0

b = 0x1

x = 0x26EB7A859923FBC82189631F8103FE4AC9CA2970012D5D46024804801841CA4437095
8493B205E647DA304DB4CEB08CBBD1BA39494776FB988B47174DCA88C7E2945283A01C8972

y = 0x349DC807F4FBF374F4AEADE3BCA95314DD58CEC9F307A54FFC61EFC006D8A2C9D4979C
0AC44AEA74FBEBBB9F772AEDCB620B01A7BA7AF1B320430C8591984F601CD4C143EF1C7A3

n = 0x2000000000000000000000000000000000000000000000000000000000000000000000
131850E1F19A63E4B391A8DB917F4138B630D84BE5D639381E91DEB45CFE778F637C1001

h = 0x4

SECGz113r1

$f = x^{113} + x^9 + 1$

a = 0x03088250CA6E7C7FE649CE85820F7

b = 0x0E8BEE4D3E2260744188BE0E9C723

x = 0x09D73616F35F4AB1407D73562C10F

y = 0x0A52830277958EE84D1315ED31886

n = 0x100000000000000D9CCEC8A39E56F

h = 0x2

SECGz113r2

f = $x^{113} + x^9 + 1$

a = 0x0689918DBEC7E5A0DD6DFC0AA55C7

b = 0x095E9A9EC9B297BD4BF36E059184F

x = 0x1A57A6A7B26CA5EF52FCDB8164797

y = 0x0B3ADC94ED1FE674C06E695BABA1D

n = 0x10000000000000108789B2496AF93

h = 0x2

SECGz131r1

f = $x^{131} + x^8 + x^3 + x^2 + 1$

a = 0x7A11B09A76B562144418FF3FF8C2570B8

b = 0x217C05610884B63B9C6C7291678F9D341

x = 0x081BAF91FDF9833C40F9C181343638399

y = 0x78C6E7EA38C001F73C8134B1B4EF9E150

n = 0x400000000000000023123953A9464B54D

h = 0x2

SECGz131r2

f = $x^{131} + x^8 + x^3 + x^2 + 1$

a = 0x3E5A88919D7CAFCBF415F07C2176573B2

b = 0x4B8266A46C55657AC734CE38F018F2192

x = 0x356DCD8F2F95031AD652D23951BB366A8

y = 0x648F06D867940A5366D9E265DE9EB240F

n = 0x400000000000000016954A233049BA98F

h = 0x2

SECGz163r1

f = $x^{163} + x^7 + x^6 + x^3 + 1$

a = 0x7B6882CAAEFA84F9554FF8428BD88E246D2782AE2

b = 0x713612DCDDCB40AAB946BDA29CA91F73AF958AFD9

x = 0x369979697AB43897789566789567F787A7876A654

y = 0x0435EDB42EFAFB2989D51FEFCE3C80988F41FF883

n = 0x3FFFFFFFFFFFFFFFFFFF48AAB689C29CA710279B

h = 0x2

SECGz163r2

f = $x^{163} + x^7 + x^6 + x^3 + 1$

a = 0x1

b = 0x20A601907B8C953CA1481EB10512F78744A3205FD

x = 0x3F0EBA16286A2D57EA0991168D4994637E8343E36

y = 0x0D51FBC6C71A0094FA2CDD545B11C5C0C797324F1

n = 0x40000000000000000000292FE77E70C12A4234C33

h = 0x2

SECGz163k1

f = $x^{163} + x^7 + x^6 + x^3 + 1$

a = 0x1

b = 0x1

x = 0x2FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8

y = 0x289070FB05D38FF58321F2E800536D538CCDAA3D9

n = 0x4000000000000000000020108A2E0CC0D99F8A5EF

h = 0x2

SECGz193r1

f = $x^{193} + x^{15} + 1$

a = 0x017858FEB7A98975169E171F77B4087DE098AC8A911DF7B01

b = 0x0FDFB49BFE6C3A89FACADAA7A1E5BBC7CC1C2E5D831478814

x = 0x1F481BC5F0FF84A74AD6CDF6FDEF4BF6179625372D8C0C5E1

y = 0x025E399F2903712CCF3EA9E3A1AD17FB0B3201B6AF7CE1B05

n = 0x1000000000000000000000000C7F34A778F443ACC920EBA49

h = 0x2

SECGz193r2

f = $x^{193} + x^{15} + 1$

a = 0x163F35A5137C2CE3EA6ED8667190B0BC43ECD69977702709B

b = 0x0C9BB9E8927D4D64C377E2AB2856A5B16E3EFB7F61D4316AE

x = 0x0D9B67D192E0367C803F39E1A7E82CA14A651350AAE617E8F

y = 0x1CE94335607C304AC29E7DEFBD9CA01F596F927224CDECF6C

n = 0x1000000000000000000000015AAB561B005413CCD4EE99D5

h = 0x2

SECGz233r1

f = $x^{233} + x^{74} + 1$

a = 0x1

b = 0x066647EDE6C332C7F8C0923BB58213B333B20E9CE4281FE115F7D8F90AD

x = 0x0FAC9DFCBAC8313BB2139F1BB755FEF65BC391F8B36F8F8EB7371FD558B

y = 0x1006A08A41903350678E58528BEBF8A0BEFF867A7CA36716F7E01F81052

n = 0x1000000000000000000000000000013E974E72F8A6922031D2603CFE0D7

h = 0x2

SECGz233k1

f = $x^{233} + x^{74} + 1$

a = 0x0

b = 0x1

x = 0x17232BA853A7E731AF129F22FF4149563A419C26BF50A4C9D6EEFAD6126

y = 0x1DB537DECE819B7F70F555A67C427A8CD9BF18AEB9B56E0C11056FAE6A3

n = 0x8000000000000000000000000000069D5BB915BCD46EFB1AD5F173ABDF

h = 0x4

SECGz239k1

f = $x^{239} + x^{36} + 1$

a = 0x0

b = 0x1

x = 0x29A0B6A887A983E9730988A68727A8B2D126C44CC2CC7B2A6555193035DC

y = 0x76310804F12E549BDB011C103089E73510ACB275FC312A5DC6B76553F0CA

n = 0x2000000000000000000000000000005A79FEC67CB6E91F1C1DA800E478A5

h = 0x4

SECGz283r1

f = $x^{283} + x^{12} + x^{7} + x^{5} + 1$

a = 0x1

b = 0x27B680AC8B8596DA5A4AF8A19A0303FCA97FD7645309FA2A581485AF626
3E313B79A2F5

x = 0x5F939258DB7DD90E1934F8C70B0DFEC2EED25B8557EAC9C80E2E198F8CD
BECD86B12053

y = 0x3676854FE24141CB98FE6D4B20D02B4516FF702350EDDB0826779C813F0
DF45BE8112F4

n = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEF90399660FC938A90165B04
2A7CEFADB307

h = 0x2

SECGz283k1

$f = x^{283} + x^{12} + x^{7} + x^{5} + 1$

a = 0x0

b = 0x1

x = 0x503213F78CA44883F1A3B8162F188E553CD265F23C1567A16876913B0C2
AC2458492836

y = 0x1CCDA380F1C9E318D90F95D07E5426FE87E45C0E8184698E45962364E34
116177DD2259

n = 0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE9AE2ED07577265DFF7F9445
1E061E163C61

h = 0x4

SECGz409r1

$f = x^{409} + x^{87} + 1$

a = 0x1

b = 0x021A5C2C8EE9FEB5C4B9A753B7B476B7FD6422EF1F3DD674761FA99D6AC27C
8A9A197B272822F6CD57A55AA4F50AE317B13545F

x = 0x15D4860D088DDB3496B0C6064756260441CDE4AF1771D4DB01FFE5B34E59703
DC255A868A1180515603AEAB60794E54BB7996A7

y = 0x061B1CFAB6BE5F32BBFA78324ED106A7636B9C5A7BD198D0158AA4F5488D08F
38514F1FDF4B4F40D2181B3681C364BA0273C706

n = 0x10000000000000000000000000000000000000000000000001E2AAD6A612
F33307BE5FA47C3C9E052F838164CD37D9A21173

h = 0x2

SECGz409k1

f = $x^{409} + x^{87} + 1$

0000000080000000000000000000001

a = 0x0

b = 0x1

x = 0x060F05F658F49C1AD3AB1890F7184210EFD0987E307C84C27ACCFB8F9F67CC2C4
60189EB5AAAA62EE222EB1B35540CFE9023746

y = 0x1E369050B7C4E42ACBA1DACBF04299C3460782F918EA427E6325165E9EA10E3DA
5F6C42E9C55215AA9CA27A5863EC48D8E0286B

n = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE5F83B2D4EA204
00EC4557D5ED3E3E7CA5B4B5C83B8E01E5FCF

h = 0x4

SECGz571r1

f = $x^{571} + x^{10} + x^{5} + x^{2} + 1$

a = 0x1

b = 0x2F40E7E2221F295DE297117B7F3D62F5C6A97FFCB8CEFF1CD6BA8CE4A9A18AD84FFABBD
8EFA59332BE7AD6756A66E294AFD185A78FF12AA520E4DE739BACA0C7FFEFF7F2955727A

x = 0x303001D34B856296C16C0D40D3CD7750A93D1D2955FA80AA5F40FC8DB7B2ABDBDE53950F
4C0D293CDD711A35B67FB1499AE60038614F1394ABFA3B4C850D927E1E7769C8EEC2D19

y = 0x37BF27342DA639B6DCCFFFEB73D69D78C6C27A6009CBBCA1980F8533921E8A684423E43B
AB08A576291AF8F461BB2A8B3531D2F0485C19B16E2F1516E23DD3C1A4827AF1B8AC15B

n =
0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE6
61CE18FF55987308059B186823851EC7DD9CA1161DE93D5174D66E8382E9BB2FE84E47

h = 0x2

SECGz571k1

$f = x^{571} + x^{10} + x^5 + x^2 + 1$

a = 0x0

b = 0x1

x =
0x26EB7A859923FBC82189631F8103FE4AC9CA2970012D5D46024804801841CA44370958493B2
05E647DA304DB4CEB08CBBD1BA39494776FB988B47174DCA88C7E2945283A01C8972

y =
0x349DC807F4FBF374F4AEADE3BCA95314DD58CEC9F307A54FFC61EFC006D8A2C9D4979C0AC4
4AEA74FBEBBB9F772AEDCB620B01A7BA7AF1B320430C8591984F601CD4C143EF1C7A3

n =
0x2000000000000000000000000000000000000000000000000000000000000001318
50E1F19A63E4B391A8DB917F4138B630D84BE5D639381E91DEB45CFE778F637C1001

h = 0x4

Anhang E Smartcards

Mit der **cv act *library*** kann man eine Smartcard zum Signieren oder Entschlüsseln verwenden und die zugehörige PIN ändern oder die Smartcard personalisieren. Um auch große Dokumente signieren zu können, werden die Daten mit SHA-1 gehasht und anstatt des gesamten Dokuments der Hashwert zum Signieren zur Karte geschickt.

Es werden folgende Kartentypen unterstützt:

- `CARDOS_CARD`: CardOS/M3.0 (von Siemens)
- `CARDOSM4_CARD`: CardOS/M4.0 (von Siemens)
- `CVACT_CARD`: **cv act *smartcard*™** (Prototyp/Testkarte)
- `PKCS11_CARD`: Smartcards oder andere Token mit PKCS#11 Schnittstelle, z.B. Rainbow iKey. Es werden Token unterstützt, die den Mechanismus `CKM_RSA_PKCS` bereitstellen.
- `TCOS_CARD`: TCOS min. (noch nicht personalisierte Karte mit TCOS 2.0), NetKey Card, PKS Card.
- `MICARDOEC_CARD`: Micardo 2.0 elliptic
- `MICARDOP21_CARD`: Micardo 2.1 public
- `STARCOSPK_CARD`: STARCOS

Hierbei kann man jedoch die NetKey Card, die PKS Card und die PKCS#11 Token nicht personalisieren. Die CardOS und die TCOS min. kann man mit der Library nur ansprechen, wenn diese auch mit der Library personalisiert wurden. Mit der Micardo 2.0 elliptic und mit der **cv act *smartcard*™** können Signaturen auf Basis von EC-DSA erstellt werden. Diese bietet darüberhinaus das Schlüsselaustauschverfahren EC-DH an. Die anderen Karten können mit RSA verschlüsseln und signieren.

Die unterstützten Kartenleser-Typen (Voraussetzung ist die korrekte Installation der Treiber) sind:

- `CTAPI_PORT`: Kartenleser mit CT-API Treibern. Diese benötigen folgenden Angaben: der DLL-Name, bzw. der Pfad der DLL (der vom Kartenleserhersteller mitgeliefert wurde und die CT-API Befehle enthält) und die Nummer des COM-Ports, an den der Leser angeschlossen ist.
- `PCSC_PORT`: Kartenleser mit PC/SC Treibern. Der `READERNAME` ist ein String, den das Setup-Programm des Kartenlesers in die Windows-Registry einfügt. Wenn `READERNAME` nicht gesetzt wird, wird der erste Leser aus der internen Liste der PC/SC Leser benutzt (PC/SC-Funktion "`SCardListReaders`"). Falls man keinen DLL- Namen angibt, wird die Windows Standard PC/SC DLL benutzt (`winscard.dll`). Diese muss sich im Suchpfad befinden.

Bei Karten mit einer PKCS#11 Schnittstelle muss der `CARDPORTTYPE` nicht gesetzt werden. Der Pfad zur DLL, die vom Kartenhersteller geliefert wird, muss angegeben werden. Wenn `READERNAME` nicht gesetzt wird, wird der erste Leser aus der internen Liste der PKCS#11 Slots benutzt (PKCS#11 Funktion "`C_GetSlotList`").

Das erste Beispiel soll nun verdeutlichen, wie man eine Smartcard mit Hilfe der **cv act *library*** zum Signieren verwenden kann.

```
void sign_from_stream_with_smartcard (const act::Blob &Pin, std::istream &in,
act::Blob &Signature)
{
        act::Key key("SmartCard"); //ein Smartcard-Key wird angelegt

        // MICARDOEC_CARD auswählen
        key.SetParam(act::CARDTYPE,act::MICARDOEC_CARD);

        // CTAPI_PORT auswählen
        key.SetParam(act::CARDPORTTYPE, act::CTAPI_PORT);

        // DLL-Name angeben:
#ifdef linux
        // Wir nutzen die MUSCLE Treiber für towitoko von Cardos Prodos
(cprados@yahoo.com)
        key.SetParam(act::DLLNAME,"/usr/local/lib/libtowitoko.so");
#else
        // towitoko Treiber für WinNT
        key.SetParam(act::DLLNAME,"c:/winnt/ctapiw32.dll");
#endif

        // Die Port-Nummer (PORTNR) muss angegeben werden, falls ein CT-API
Reader benutzt wird:
        key.SetParam(act::PORTNR,1); // Der Kartenleser ist an COM Port 1
angeschlossen

        // PIN zur Authentifizierung wird
          // über Tastatur des Kartenlesers eingegeben
        key.SetParam(act::PIN,Pin);

        act::Algorithm Sign(key,act::SIGN); // Signieralgorithmus anlegen
        act::Blob InData(BUFFER_SIZE); //Blob fuer Eingabe anlegen
        std::streamsize count;
        do      {
                in.read(reinterpret_cast<char*>(&InData[0]),BUFFER_SIZE);
//Einlesen
                count=in.gcount();
                if(count<< InData; // die zu signierenden Daten
                        // werden an den Algorithmus uebergeben
        } while(count == BUFFER_SIZE);
        Sign <>Signature;             //und die Signatur von der Smartcard
geholt.
}
```

Dieses Beispielprogramm ist vergleichbar mit der Signaturfunktion in Abschnitt 5. 2. 6. Dennoch wird in diesem Beispiel das Signaturprimitiva und damit der Zugriff auf den privaten Schlüssel nicht auf dem Computer durchgeführt, sondern auf einer Smartcard. Dadurch besteht nicht die Gefahr, dass der private Schlüssel von einem trojanischen Pferd im Speicher des Computers ausgespäht werden kann. Durch diese einheitliche Bedienbarkeit der `Key`- und `Algorithm`-Objekte haben Sie die Möglichkeit, Code zu schreiben, der sich leicht (gegebenenfalls zur Laufzeit ihrer Anwendung) von den software-

implementierten Verfahren auf Smartcards umstellen lässt. Die Verwendung einer Smartcard zum Entschlüsseln mit Hilfe der **cv act *library*** ist analog zu diesem Beispiel des Signierens. Es wird anstatt des Befehls `Sign` die Methode `Decrypt` verwendet.

Das zweite Beispiel soll verdeutlichen, wie man mit Hilfe der **cv act *library*** die PIN einer Smartcard ändern kann.

```
void change_pin_of_smartcard (const act::Blob &oldpin,
                              const act::Blob &newpin)
{
        act::Key key("SmartCard"); //ein Smartcard-Key wird angelegt

        // MICARDOEC_CARD auswählen
        key.SetParam(act::CARDTYPE,act::MICARDOEC_CARD);

        // PCSC_PORT auswählen
        key.SetParam(act::CARDPORTTYPE, act::PCSC_PORT);

#ifdef linux
        // towitoko Treiber für linux
        key.SetParam(act::READERNAME,"Towitoko");
        key.SetParam(act::DLLNAME,"/usr/local/pcsc/lib/libpcsclite.so");
#else
        // towitoko Treiber für WinNT
        key.SetParam(act::READERNAME,"TOWITOKO CHIPDRIVE 0");
        key.SetParam(act::DLLNAME,"winscard.dll");
#endif

        key.SetParam(act::PIN, oldpin);       // Angabe der alten PIN
        key.SetParam(act::CHANGEPIN, newpin); // Angabe der neuen PIN

}
```

Das dritte Beispiel soll verdeutlichen, wie man eine Smartcard mit Hilfe der **cv act *library*** personalisieren kann. TCOS_CARDs und CARDOS_CARDs erwarten zwei 1024 Bit RSA-Schlüssel und die dazugehörigen Zertifikate und außerdem ein Root-CA Zertifikat (Zertifikate ≤ 1024 Byte). Die CardOS/M3.0 und die TCOS min. müssen, um mit der Library angesprochen werden zu können, auch mit der **cv act *library*** personalisiert werden. Bei der Personalisierung der Karten wurde die PKS Card als Vorbild genommen. Das bedeutet, dass die PIN sechs Stellen hat und nach dem Personalisieren auf 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 (das sogenannte Null-PIN-Verfahren) gesetzt ist. Vor der ersten Benutzung der Karte muss die PIN geändert werden, wobei die Anzahl der Stellen nicht verändert werden kann. Falls dreimal eine falsche PIN eingegeben wurde, wird diese Karte gesperrt.

Das Personalisieren der Karten kann ca. 1 Minute dauern. **Der Benutzer darf auf keinen Fall die Karte während des Personalisierens aus dem Kartenleser herausziehen!** Wenn er es doch tut, darf die Karte nicht benutzt werden, da unter Umständen der private Schlüssel auslesbar ist, weil die Karte noch nicht "abgeschlossen" wurde.

```
void personalize_cardos_card ( const act::Blob &signkey,
                        const act::Blob &enckey,
                        const act::Blob &signcertificate,
```

```
                        const act::Blob &enccertificate,
                        const act::Blob &rootcertificate)
{
        act::Key key("SmartCard"); // anlegen eines Smartcard-Schlüssel

        // Auswahl CARDOS-Karte
        key.SetParam(act::CARDTYPE,act::CARDOS_CARD);

        // Auswahl PCSC_PORT
        key.SetParam(act::CARDPORTTYPE, act::PCSC_PORT);

#ifdef linux
        // towitoko Treiber für linux
        key.SetParam(act::READERNAME,"Towitoko");
        key.SetParam(act::DLLNAME,"/usr/local/pcsc/lib/libpcsclite.so");
#else
        // towitoko Treiber für WinNT
        key.SetParam(act::READERNAME,"TOWITOKO CHIPDRIVE 0");
        key.SetParam(act::DLLNAME,"winscard.dll");
#endif


        // Speicherung der folgenden Daten auf der CardOS-Karte
        key.SetParam(act::SIGNKEY,signkey);
        key.SetParam(act::ENCKEY,enckey);
        key.SetParam(act::SIGNCERTIFICATE,signcertificate);
        key.SetParam(act::ENCCERTIFICATE,enccertificate);
        key.SetParam(act::ROOTCERTIFICATE,rootcertificate);
        key.Generate();
}


void personalize_micardoec_card(const act::Blob &masterpin,
                        const act::Blob &userpin,
                        const act::Blob &signprivkey,         // privater
Schlüssel für Signatur
                        const act::Blob &dhprivkey,           // privater
Schlüssel für Schlüsselaustausch
                        const act::Blob &signcertificate,     // Zertifikat für
Signatur
                        const act::Blob &dhcertificate)               //
Zertifikat für Schlüsselaustausch
{
        act::Key key("SmartCard"); // ein Smartcard-Schlüssel wird angelegt

        // Auswahl MICARDOEC_CARD
        key.SetParam(act::CARDTYPE,act::MICARDOEC_CARD);

        // Auswahl PCSC_PORT
        key.SetParam(act::CARDPORTTYPE, act::PCSC_PORT);

#ifdef linux
        // towitoko Treiber für linux
        key.SetParam(act::READERNAME,"Towitoko");
        key.SetParam(act::DLLNAME,"/usr/local/pcsc/lib/libpcsclite.so");
#else
        // towitoko Treiber für WinNT
        key.SetParam(act::READERNAME,"TOWITOKO CHIPDRIVE 0");
```

```
        key.SetParam(act::DLLNAME,"winscard.dll");
#endif

        // Setzen der Parameter
        key.SetParam(act::MASTERPIN, masterpin);
        key.SetParam(act::USERPIN, userpin);
        key.SetParam(act::SIGNKEY, signprivkey);
        key.SetParam(act::SIGNCERTIFICATE, signcertificate);
        key.SetParam(act::DHKEY, dhprivkey);
        key.SetParam(act::DHCERTIFICATE, dhcertificate);
        key.Generate();
}

void personalize_cvact_card(const act::Blob &masterpin,
                        const act::Blob &userpin,
                        const act::Blob &username,
                        const act::Blob &infotext)
{
        act::Key key("SmartCard"); // ein Smartcard-Schlüssel wird angelegt

        // Auswahl CVACT_CARD
        key.SetParam(act::CARDTYPE,act::CVACT_CARD);

        // Auswahl PCSC_PORT
        key.SetParam(act::CARDPORTTYPE, act::PCSC_PORT);

#ifdef linux
        // towitoko Treiber für linux
        key.SetParam(act::READERNAME,"Towitoko");
        key.SetParam(act::DLLNAME,"/usr/local/pcsc/lib/libpcsclite.so");
#else
        // towitoko Treiber für WinNT
        key.SetParam(act::READERNAME,"TOWITOKO CHIPDRIVE 0");
        key.SetParam(act::DLLNAME,"winscard.dll");
#endif

        // Setzen der Parameter
        key.SetParam(act::MASTERPIN, masterpin);
        key.SetParam(act::USERPIN, userpin);
        key.SetParam(act::USERNAME, username);
        key.SetParam(act::INFOTEXT, infotext);
        key.Generate();
}
```

Release Version, März 2007

Gewährleistung

Änderungen in dieser Veröffentlichung sind vorbehalten.

cryptovision übernimmt keinerlei Gewährleistung auf die in diesem Material enthaltenen Informationen. Dies gilt insbesondere für die stillschweigende Garantie auf Markttauglichkeit sowie Eignung für einen bestimmten Einsatzzweck.

cryptovision übernimmt keine Haftung für mittelbare, unmittelbare, Neben-, Folge- oder andere Schäden, die mit der Auslieferung, Bereitstellung oder Nutzung dieses Materials im Zusammenhang stehen.

cv cryptovision gmbh

http://www.cryptovision.com/