¿$\frac{1}{2}$=fl $\frac{1}{2}$=fl $\frac{1}{2}$=fl $\frac{1}{2}$=fl $\frac{1}{2}$=fl $\frac{1}{2}$=fl $\frac{1}{2}$=fl

Ada Crypto Lib (ACL) aka libadacrypt-dev
Version 0.1.2 (beta)
Benutzer-Dokumentation

Christian Forler

24. Dezember 2009

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bei der Ada Crypto Library (ACL) handelt es sich um eine freie kryptographische Bibliothek für Ada. Eines der beiden Hauptziele bei der Entwicklung dieser Bibliothek war das Design einer **selbst sprechenden und sauberen API**. Das andere Hauptziel war es, einen möglichst sauberen Programmierstil zu verwenden, der die formale Verifikation des Codes erleichtert. Aus diesem Grund wurde bei der ACL komplett auf folgende "Features" verzichtet:

- access types (Zeiger)
- inline Assembler
- tagged types (Objekte)
- goto statements

Auf goto statements und access-types wurde verzichtet, da diese den Quelltext unübersichtlicher machen und Probleme bei der Verifikation des Codes auftreten können. Auf tagged types wurde verzichtet, da hier Prozeduren und Methoden überschrieben werden können, und erst zur Laufzeit dynamisch ermittelt wird welche Methode verwendet wird (dynamisches dispatching). Hierdurch treten massive Probleme bei Verifikation des Codes auf. Auf inline Assembler wurde verzichtet, da dies ein sehr unsauberer Programmierstil ist und dies die starke Typisierung von ADA umgehen kann. Was nicht gerade den wahren Ada-Weg entspricht. Durch diese Einschränkungen erhöht sie zwar die Qualität des Quelltextes im Bezug auf Verifizierbarkeit und Sicherheit, führt aber zu einer schlechteren Performance. Wenn Sie also mehr Wert auf Performance als auf ein sauberes Design legen, dann sollten Sie sich vielleicht nach einer anderen kryptographischen Bibliothek umsehen. Leider sind mir keine weiteren freie kryptographische Bibliothek für Ada bekannt auf die ich sie jetzt verweisen könnte.
Dies ACL hat mehr den Status eines „Proof of Concept". Es fand weder ein "Code-Review" noch ein "Code-Audit" statt. Bis jetzt ist der Autor der ACL

der Einzige, der sich mit dem Code etwas näher befasst hat. Es besteht z.B. durchaus die Möglichkeit, daß sich sicherheitskritisches Material wie der Schlüssel sich nach der Ausführung eines Programms das die ACL verwendet im RAM "überlebt". Ein Storage-Pool der genau diese Aufgabe übernimmt ist in Planung. Die meisten kryptographische Bibliotheken die auch dieses Sicherheitsproblem haben weisen nicht darauf hin. Von solchen Bibliotheken sollte man Abstand nehmen, da diese nicht mit offenen Karten spielen.
Im folgendem habe ich noch einmal alle mir bekannten Nachteile der ACL aufgelistet.

- fehlende Säuberung des Stacks und Heaps
- kein "big endian" Support
- schlechte Performance

In dieser Dokumentation wird kurz auf die Installation und den topologischen Aufbau eingegangen. Danach widmet sich diese Dokumentation ausschließlich der API. Jedes Paket und dessen API wird in einem separaten Kapitel vorgestellt. Am Ende jedes Kapitels finden Sie ein Anwendungsbeispiel.
Falls sie noch Fragen zur ACL haben, einen Fehler finden oder die ACL um ein bzw. mehrere Pakete erweitern wollen, dann können sie mich per E-Mail unter folgender Adresse kontaktieren: crypto@shortie.inka.de.

TODO-Liste

- Optimierung
- Eigener Storage Pool
- Erweiterung (Tiger2, RSA-PSS, Poly1305-AES usw.)
- Fallunterscheidung für die Anwendung des RNG (Windows/Linux)

1.1 Installation HOWTO

1.1.1 libadacrpyt-dev

Abhängigkeiten

Unter Linux benötigen Sie folgende Pakete um die ACL zu installieren:

- tar
- gnat
- make

- bunzip2
- binutils

Kompilieren

Mit der folgenden Befehlssequenzen entpacken und kompilieren sie die ACL sowie den beiliegenden Regressionstest.

- `tar cjf acl.tar.bz2`
- `cd crypto`
- `make`
- `make acltest`

Testen

Bevor Sie die ACL installieren sollten Sie unbedingt den Regressionstest durchführen um sicherzustellen, das die ACL auf ihrem System einwandfrei arbeitet. Der Regressionstest dauert auf einem PII 450 ungefähr 30 Sekunden. Mit dem folgender Befehlssequenz führen Sie den Regressiontest aus.

- `cd test`
- `./acltest`
- `cd ..`

Installieren

Wenn bei dem Test kein Fehler auftrat können sie die ACL mit folgendem Befehl installieren:

```
su -c ‘‘make install’’
```

Deinstallieren

Mit dem folgenden Befehl können sie ACL wieder deinstallieren:

```
su -c ‘‘make uninstall’’
```

Rekompilieren

Mit der folgenden Befehlssequenzen können sie ACL neu kompilieren:

- `make clean`
- `make clean-acltest`
- `make`
- `make acltest`

1.1.2 libadacrypt

Installieren

Sie können die ACL auch mit den folgenden Befehlen als shared library (libacl.so) installieren:

- `make shared`
- `make install-shared`

deinstallieren

Mit dem folgenden Befehl können sie ACL wieder deinstallieren:

```
su -c ‘‘make uninstall-shared’’
```

1.1.3 Dokumentation

Um diese Dokumentation zu erstellen wird das Paket tetex-bin (latex) und tetex-extra benötigt. Wenn nötig sollte sie diese beiden Pakete installieren. Mit dem folgenden beiden Befehlen wird die komplette Dokumentation (de+en) erstellt und nach `/usr/local/share/doc/libadacrypt-dev` kopiert.

- `make doc`
- `su -c ‘‘make install-doc’’`

Mit den folgenden beiden Befehlen können sie die Dokumentationen unter `/usr/local/share/doc/libadacrypt-dev` deinstallieren und die Dokumentation „löschen”.

- `make clean-doc`
- `su -c ‘‘make uninstall-doc’’`

1.1.4 Anpassungen

Im Unterverzeichnis `src` befindet sich eine Datei `Makefile`. In dieser können Sie bei den folgende Variablen Anpassungen vornehmen:

- LIBDIR : Installationspfad der shared library.
- INSTDIR : Installationspfad der ACL.

Ausserdem können sie noch bei den Makefiles in den Unterverzeichnissen von `doc` durch Anpassung der Variablen `DOCPATH` vornemen. Diese Variable enthält den Installationspfad der entsprechenden Dokumentation.

1.1.5 Benchmark

In dem Unterverzeichnis `bench` befindet sich ein Benchmark, der in einer späteren Version Aussagen über die Performance der ACL auf dem eigenen System ermöglichen soll. Der Aufbau orientiert sich an dem des Regressionstests. Bisher wurde eine Möglichkeit zum Messen der Multiplikationsalgorithmen aus dem Package `acl.crypto.types.big-numbers` umgesetzt. Der Benchmark generiert dazu Zufallszahlen, aus denen die Faktoren bestimmt werden. Das Produkt wird durch die jeweils durch alle implementierten Algorithmen bestimmt und die zur Berechnung benötigte Zeit ausgegeben.
Kompilieren und Ausführen lässt sich der Benchmark durch

- `cd bench`
- `make`
- `./aclbench`

Durch einen Aufruf des Benchmarks mit Kommandzeilenparameter `-CSV` lassen sich die Messungen kommasepariert ausgeben. Dadurch kann man die Ergebnisse einfacher in ein Tabellenkalkulationsprogramm importieren und auswerten.
Unter Linux:

- `./aclbench -CSV > Dateiname.txt`

Im Unterverzeichnis `bench` befindet sich eine Datei `Makefile`. In dieser können Sie bei den folgende Variablen Anpassungen vornehmen:

Kompilieren des Benchmarks ohne Optimierung:

```
aclbench.o :
    $(CC) $(CFLAGS) aclbench
    $(BB) $(BFLAGS) aclbench.ali
    $(LL) $(LFLAGS) aclbench.ali
```

Kompilieren des Benchmarks mit O3 Optimierung:

```
aclbench.o :
    $(CC) $(CFLAGS_O3) aclbench
    $(BB) $(BFLAGS_O3) aclbench.ali
    $(LL) $(LFLAGS_O3) aclbench.ali
```

Kompilieren des Benchmarks mit Symbolen für `gdb` oder `gnatgdb` :

```
aclbench.o :
    $(CC) $(CFLAGS_DBUG) aclbench
    $(BB) $(BFLAGS_DBUG) aclbench.ali
    $(LL) $(LFLAGS_DBUG) aclbench.ali
```

Kompilieren des Benchmarks mit Symbolen für `gprof` oder `kprof`:

```
aclbench.o :
    $(CC) $(CFLAGS_GPROF) aclbench
    $(BB) $(BFLAGS_GPROF) aclbench.ali
    $(LL) $(LFLAGS_GPROF) aclbench.ali
```

Kompilieren des Benchmarks mit Symbolen für `gcov` oder `ggcov`:

```
aclbench.o :
    $(CC) $(CFLAGS_GCOV) aclbench
    $(BB) $(BFLAGS_GCOV) aclbench.ali
    $(LL) $(LFLAGS_GCOV) aclbench.ali
```

1.2 Verzeichnis- und Paket-Struktur

1.2.1 Verzeichnis-Struktur

- doc : Dokumentation
- ref : Referenzen und Spezifikationen
- src : Quellcode
- test : Regressionstest
- bench : Benchmark

1.2.2 Paket-Struktur

Die Ada Crypto Library (ACL) besteht aus den folgenden Paketen (Komponenten).

Crypto:
Dies ist das Wurzelpaket der ACL. Alle anderen Pakete der ACL beginnen mit dem Präfix *Crypto.*

Crypto.Types:
In diesem Paket befinden sich grundlegende Typen der ACL (z.B. Byte) und deren Basisfunktionen. Ein Einsatz der ACL ohne Einbindung dieses Paketes ist nur sehr begrenzt möglich.

Crypto.Types.Big_Numbers:
Dieses generische Packet unterstützt die Arithmetik in Z_p oder $GF(2^m)$. Der Basistyp hierführ ist eine modulare n-Bit Zahl. Dieses Packet wird für die asymetrische Kryptrographie benötigt.

Crypto.Types.Elliptic_Curves:
Dieses generischen Pakete ist das Wurzelpacket für elliptische Kurven.

Crypto.Random:
Dieses Paket ist eine Schnittstelle zwischen einem externen Pseudozufallsbitgenerator und der ACL.

Crypto.Symmetric:
Die ist das Wurzelpaket für den symmetrische Zweig.

Crypto.Symmetric.Algorithm:
In diesem Zweig des ACL-Baums befinden sich die symmetrische Algorithmen für symmetrische Blockchiffren und Hashfunktionen.

Crypto.Symmetric.Algorithm.Oneway:
Jeder Algorithmus verfügt über einen Oneway-Algorithmus. Symmetrisch Oneway-Algorithmen sind einfach symmetrische Algorithmen die nur in eine Richtung, entweder Ver- oder Entschlüsseln arbeiten.

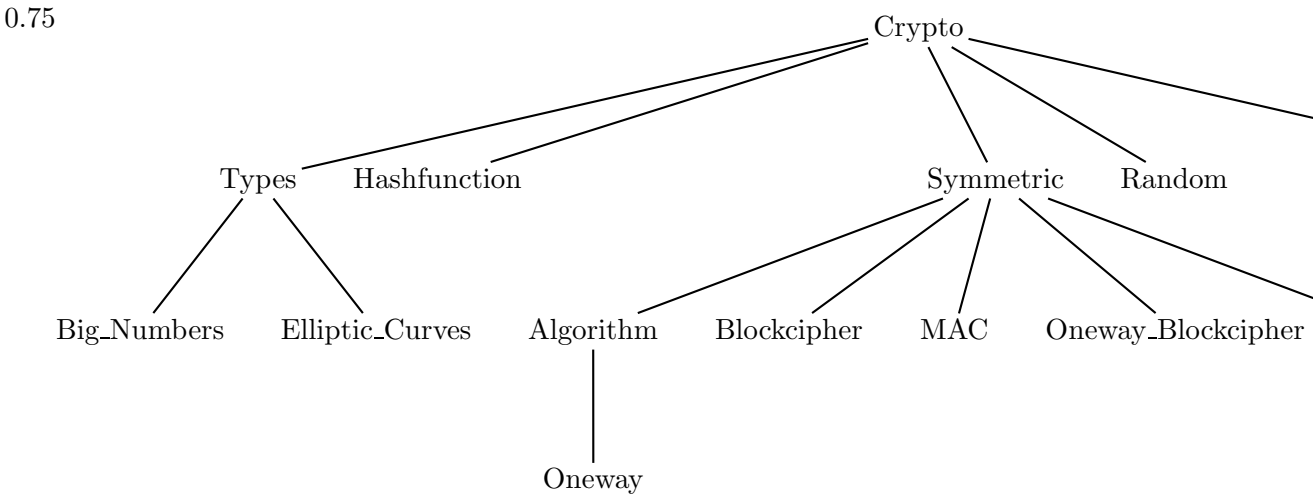
Crypto.Symmetric.Blockcipher:
Mit Hilfe dieses generische Paketes können Sie aus einem symmetrischen Algorithmus eine Blockchiffre generieren.

Crypto.Symmetric.Oneway_Blockcipher:
Mit Hilfe dieses generische Paketes können Sie aus einem symmetrischen Oneway-Algorithmus eine Einweg-Blockchiffre generieren.

Crypto.Symmetric.MAC: Dies ist das Wurzelpacket für MACs.

Crypto.Symmetric.Mode:
In diesem Zweig befinden sich verschiedene Betriebsmodi für Blockchiffren.

Crypto.Symmetric.Mode.Oneway:
In diesem Zweig befinden sich verschiedene Betriebsmodi für Einweg-Blockchiffren.



Crypto.Asymmetric:
Dies ist das Wurzelpaket des asymmetrischen Zweigs.

Crypto.Asymmetric.DSA:
Mit Hilfe dieses Pakets können digitale Unterschriften erstellt und verifiziert werden.

Crypto.Asymmetric.RSA:
Mit Hilfe dieses Paketes lassen sich Daten asymmetrisch ver- bzw. entschlüsseln

Crypto.Hashfunction: Mit Hilfe dieses generische Paketes können Sie aus einem entsprechenden symmetrischen Algorithmus eine Hashfunktion generieren.

Kapitel 2

Crypto

Crypto ist ein leeres Wurzel-Paket. Dieses Paket enthält also keinen Code. Der einzigste Zweck dieses Paketes ist es, einen eindeutigen Namensraum für alle ACL Pakete zu bilden. Da alle Pakete der ACL Kinder dieses Paketes sind, beginnt der Name jedes Paketes mit dem Präfix Crypto (z.B. Crypto.Foo.Bar).

Kapitel 3

Crypto.Types

Dieses Paket stellt die elementaren Typen der ACL und dessen Basisfunktionen zur Verfügung.

WICHTIG:
Bei der Verwendung der ACL sollten Sie unbedingt dieses Paket mittels

```
with Crypto.Types;
```

importieren

3.1 Elementare Typen

Bei den elementaren bzw. primären Typen handelt es sich ausschließlich um modulare Typen. D.h. ein Variablenüberlauf bzw -unterlauf führt nicht zu einer Exception. Wenn das Ergebnis einer Operation nicht im Wertebereich des modularen Types liegt, dann wird $n := 2 ** Type'Size = Type'Last + 1$ solange addiert bzw. subtrahiert, bis das Ergebnis wieder im Wertebereich des modularen Types liegt.

```
type Bit is mod 2 ** 1;
for Bit'Size use 1;

type Byte is mod 2 ** 8;
for Byte'Size use  8;

type Word is mod 2 ** 32;
for Word'Size use 32;

type DWord is mod 2 ** 64;
for DWord'Size use 64;
```

Beispiel

```
with Crypto.Types;
with Ada.Text_IO;
use Crypto.Types;

procedure Example is
   -- Byte hat einen Wertebereich von 0..255
   A, B : Byte;
begin
   A := 100;
   B := A + 250; -- Ueberlauf bei B
   A := A - 250; -- Unterlauf bei A
   Ada.Text_IO.Put_Line("A: " &  A'IMG);
   Ada.Text_IO.Put_Line("B: " &  B'IMG);
end example;
```

Ausgabe des Programmes:

```
A: 106
B: 94
```

3.2 Abgeleitete Typen

Abgeleitete Typen sind Typen die von elementaren Typen abgeleitet werden. In der Regel handelt es sich dabei um Felder die aus Elementaren Typen bestehen. **Alle nicht privaten Felder, die aus elementaren Typen bestehen, werden in der ACL als n-Bit Zahlen interpretiert**, wobei das erste Element (First) als das höchstwertigste und das letzte Element (Last) als der niederwertigster Teil dieser Zahl interpretiert wird. D.h. Die Wertigkeit eines Elements sinkt mit der Höhe des Index.

Bit

```
type Bits is array (Integer range <>) of Bit;
```

Bytes

```
type Bytes is array (Integer range <>) of Byte;

subtype Byte_Word  is Bytes (0 .. 3);
subtype Byte_DWord is Bytes (0 .. 7);

subtype B_Block32  is Bytes (0 ..  3);
subtype B_Block48  is Bytes (0 ..  5);
subtype B_Block56  is Bytes (0 ..  6);
subtype B_Block64  is Bytes (0 ..  7);
subtype B_Block128 is Bytes (0 .. 15);
subtype B_Block160 is Bytes (0 .. 19);
subtype B_Block192 is Bytes (0 .. 23);
subtype B_Block256 is Bytes (0 .. 31);
```

Ein B_BlockN besteht immer aus einem N Bit-Array das in N/8 Bytes unterteilt ist. Der subtype B_Block256 ist z.B. ein 256-Bitstring der in 32 (256/8) Bytes unterteilt ist. In Ada lässt sich dies durch ein 32-elementiges Byte-Array abbilden.

Words

```
type Words   is array (Integer range <>) of Word;

subtype W_Block128  is Words(0 .. 3);
subtype W_Block160  is Words(0 .. 4);
subtype W_Block192  is Words(0 .. 5);
subtype W_Block256  is Words(0 .. 7);
subtype W_Block512  is Words(0 .. 15);
```

Ein W_BlockN besteht immer aus einem N Bit-Array das in N/32 Words unterteilt ist. Der subtype W_Block256 ist z.B. ein 256-Bitstring der in 8 (256/32) Word Elementen unterteilt ist. In Ada lässt sich dies durch ein 8-elementiges Word-Array abbilden.

DWords

```
type DWords    is array (Integer range <>) of DWord

subtype DW_Block256    is DWords(0 ..   3);
subtype DW_Block384    is DWords(0 ..   5);
subtype DW_Block512    is DWords(0 ..   7);
subtype DW_Block1024   is DWords(0 ..  15);
```

Ein DW_BlockN besteht immer aus einem N Bit-Array das in N/64 Words unterteilt ist. Der subtype DW_Block256 ist z.B. ein 256-Bitstring der in 4 (256/64) DWord Elementen unterteilt ist. In Ada lässt sich dies durch ein 4-elementiges DWord-Array abbilden.

Strings

```
subtype Hex_Byte  is String (1.. 2);
subtype Hex_Word  is String (1.. 8);
subtype Hex_DWord is String( 1..16);
```

Nachrichtenblöcke

```
subtype Message_Block_Length512  is Natural range 0 ..
64;
subtype Message_Block_Length1024 is Natural range 0 .. 128;
```

Die Message_Block_Length Typen gibt die Länge in Zeichen (Bytes) einer Nachricht innerhalb eines 512- bzw. 1024-Bit Nachrichtenblockes M an. Wenn man z.B. eine 1152-Bit lange Nachricht in 512-Bit Blöcke unterteilt, dann erhält man drei 512-Bit Blöcke. Wobei die Nachrichtenlänge im letzten Block 16 ($1280 - 2 \cdot 512 = 128/8 = 16$) ist. Die restlichen 384-Bit des letzten Nachrichtenblockes sind „leer “, d.h. sie enthalten keine Teile der ursprünglichen Nachricht. Eine Variable des Types M_Length512 würde man in diesem Fall auf 32 setzen. Diese Variablen-Typen werden also für das Auffüllen von Nachrichtenblöcken (padding) benötigt. Mehr Informationen zum Thema “Padding” gibt es im Kapitel über Hashfunktionen (**??**).

3.3 Funktionen und Prozeduren

3.3.1 Bitverschiebungs Funktionen

```
function Shift_Left   (Value  : Byte;
                       Amount : Natural)
                       return Byte;

function Shift_Right  (Value  : Byte;
                       Amount : Natural)
                       return Byte;

function Rotate_Left  (Value  : Byte;
                       Amount : Natural)
                       return Byte;

function Rotate_Right (Value  : Byte;
                       Amount : Natural)
                       return Byte;


function Shift_Left   (Value : Word;
                       Amount : Natural)
                       return Word;

function Shift_Right  (Value  : Word;
                       Amount : Natural)
                       return Word;

function Rotate_Left  (Value  : Word;
                       Amount : Natural)
                       return Word;

function Rotate_Right (Value  : Word;
                       Amount : Natural)
                       return Word;


function Shift_Left   (Value  : DWord;
                       Amount : Natural)
                       return DWord;

function Shift_Right  (Value  : DWord;
```

```
                                Amount : Natural)
                                return DWord;

function Rotate_Left  (Value  : DWord;
                       Amount : Natural)
                        return DWord;

function Rotate_Right (Value  : DWord;
                       Amount : Natural)
                       return DWord;
```

3.3.2 Basis Funktionen

“XOR”

```
function "xor" (Left, Right : Bytes)  return Bytes;
function "xor" (Left, Right : Words)  return Words;
function "xor" (Left, Right : DWords) return DWords;
```

Vorbedingung:
Left'Length = Right'Length
Exception:
Verletzung der Vorbedingung : Constraint_Byte/Word/DWord_Error

Diese Funktionen verknüpfen die beiden Eingabefelder feldweise mit der XOR-Operation. D.h. Left(Left'First) wird z.B. mit Right(Right'First) XOR-verknüpft und Left(Left'Last) wird mit Right(Right'Last) XOR-verknüpft.

“+”

```
function "+" (Left : Bytes;  Right : Byte)   return Bytes;
function "+" (Left : Byte;   Right : Bytes)  return Bytes;
function "+" (Left : Words;  Right : Word)   return Words;
function "+" (Left : Word;   Right : Words)  return Words;
function "+" (Left : Words;  Right : Byte)   return Words;
function "+" (Left : DWords; Right : DWord)  return DWords;
function "+" (Left : DWord;  Right : DWords) return DWords;
function "+" (Left : DWords; Right : Byte)   return DWords;
```

Bei diesen Funktionen werden Left und Right als Zahlen interpretiert wobei Left(Left'First) bzw. Right(Right'First) das niederwertigste Byte/Word/DWord und Left(Left'Last) bzw. Right(Right'Last) das höchstwertigste Byte /Word/DWord einer Zahl repräsentiert. Das Ergebnis dieser Funktion ist die Summe der beiden Zahlen.

Beispiel:

```
procedure plus is
  A : Byte := 200;
  B : Bytes(0..1) := (0 => 100, 1 => 16);
  begin
  B := A+B; -- B := 2#11001000#+2#10000_01100100#
  -- B(0) = 2#101100# = 44
  -- B(1) = 2#010001# = 17
end plus;
```

3.3.3 Umwandlungs-Funktionen

Im folgenden gilt:

`W : Word ≡ B0||B1||B2||B3`
`D : DWord ≡ B0||B1||B2||B3||B4||B5||B6||B7`

wobei B0 hier jeweils dem höchstwertigsten Byte und B3 bzw. B7 dem niederwertigsten Byte von W bzw. D entspricht.

ByteN

```
function Byte0 (W : Word)  return Byte;
function Byte1 (W : Word)  return Byte;
function Byte2 (W : Word)  return Byte;
function Byte3 (W : Word)  return Byte;

function Byte0 (D : DWord) return Byte;
function Byte1 (D : DWord) return Byte;
function Byte2 (D : DWord) return Byte;
function Byte3 (D : DWord) return Byte;
function Byte4 (D : DWord) return Byte;
function Byte5 (D : DWord) return Byte;
function Byte6 (D : DWord) return Byte;
```

```
function Byte7 (D : DWord) return Byte;
```

Diese Funktionen geben das entprechende N-te Byte von W bzw. D zurück. Man beachte, das hier `B0` ≡ `Byte0` usw. entspricht.

To_Bytes

```
function To_Bytes(X : Word)  return Byte_Word;
function To_Bytes(X : DWord) return Byte_DWord;

function To_Bytes(Word_Array  : Words)  return Bytes;
function To_Bytes(DWord_Array : DWords) return Bytes;

function To_Bytes(Message : String) return Bytes;
```

Die ersten beiden Funktion wandelt ein Word bzw. DWord in ein Byte_Word bzw. Byte_DWord um. Dabei wird das höchstwertigste Byte von X zum ersten Element des Rückgabewertes (First) und das niederwertigste Byte von X zum letzten Element des Rückgabewertes.

Beispiel:

```
D : DWord := 16#AA_BB_CC_DD_EE_FF_11_22#;
B : Byte_DWord := To_Bytes(D);
-- B(0) := 16#AA#; B(1) := 16#BB#; B(2) := 16#CC#;
-- B(3) := 16#DD#; B(4) := 16#EE#; B(5) := 16#FF#;
-- B(6) := 16#11#; B(7) := 16#22#;
```

Die nächsten beiden Funktion wandelt ein Word-Array, bzw. DWord-Array in ein Byte-Array um. Dabei wird das höchstwertigste Byte des ersten Arrrayelemntes zum ersten Element des Byte-Arrays und das niederwertigste Byte vom letzten Arrrayelement zum letzten Element des Byte-Arrays.

Die letzte Funktion wandelt einen (ASCII-) String in ein Byte-Array B um. Für alle $I \in$ Message'Range gilt dabei : B(I) = Character'Pos(Message(I)).

R_To_Bytes

```
function R_To_Bytes (X :  Word) return Byte_Word;
function R_To_Bytes (X : DWord) return Byte_DWord;
```

Diese Funktionen wandeln ein Word bzw. DWord X in ein Byte-Array Byte_Word bzw. Byte_DWord B um. Dabei wird das höchstwertigste Byte von X zu B(B'Last) und das niederwertigste Byte von X zu B(B'First).

To_Word

```
function To_Word (X : Byte_Word)  return Word;

function To_Word (A,B,C,D : Byte) return Word;

function To_Word (A,B,C,D : Character) return Word;
```

Die erste Funktion wandelt ein Byte_Word in ein Word um. Dabei wird X(Byte_Word'First) zum höchstwertigesten Byte des Wortes und X(Byte_Word'Last) zum niederwertigsten Byte des Wortes.

Die zweite Funktion wandelt vier Bytes (A, B, C, D) zu einem Wort um. Dabei wird das erste Byte (A) zum höchstwertigsten Byte des Wortes und das letzte Byte (D) zum niederwertigsten Byte des Wortes.

Die letzte Funktion wandelt vier Character (A, B, C, D) zu einem Wort um. Dabei wird die Position des erste Character (A'Pos) zum höchstwertigsten Byte des Wortes und die Position das letzte Characters (D'Pos) zum niederwertigsten Byte des Wortes.

R_To_Word

```
function R_To_Word (X : Byte_Word) return Word;
```

Diese Funktion wandelt ein Byte_Word in ein Word um. Dabei wird X(Byte_Word'First) zum niederwertigsten Byte und X(Byte_Word'Last) zum höchstwertigsten Byte des Resultats.

To_Words

```
function To_Words(Byte_Array : Bytes) return Words;
```

Diese Funktion wandelt ein Byte-Array B in ein Word-Array W um. Dabei wird B als Zahl interpretiert bei der B(B'First) das niederwertigste Byte und B(B'Last) das höchstwertigste Byte dieser Zahl darstellt. Diese Zahl wird in ein Word-Array konvertiert, das ebenfalls als Zahl interpretiert wird, bei der das erste Element (First) den höchstwertigsten und das letzte Element (Last) den niederwertigsten Teil der Zahl darstellt.

Beispiele:
(i)

```
B : Bytes(1..6) := (16#0A#, 16#14#, 16#1E#,
                    16#28#, 16#32#, 16#3C#);
W : Words := Byte_to_Dwords(B);
-- W(D'First) = 16#0A_14_1E_28#
-- W(D'Last)  = 16#32_3C_00_00#
```

(ii)

Eingabe: $B = \underbrace{B(1)||B(2)||B(3)||B(4)}_{W(0)}\underbrace{B(5)||B(6)||B(7)||B(8)}_{W(1)}\underbrace{B(9)||B(10)}_{W(2)}$

Ausgabe: $W = W(0)||W(1)||W(2)$

To_DWord

```
function To_DWord (X : Byte_DWord) return DWord;
```

Diese Funktion wandelt ein Byte_DWord in ein DWord um. Dabei wird X(Byte_Word'First) zum höchstwertigesten Byte des DWords und X(Byte_Word'Last) zu dessen niederwertigsten Byte.

R_To_DWord

```
function R_To_DWord (X : Byte_DWord) return DWord;
```

Diese Funktion wandelt ein Byte_DWord in ein DWord um. Dabei wird X(Byte_Word'First) zum niederwertigsten Byte und X(Byte_Word'Last) zum höchstwertigsten Byte des Resultats.

To_DWords

```
function To_DWords (Byte_Array : Bytes) return DWords;
```

Diese Funktion wandelt ein Byte_Array B in ein DWord-Array um. Dabei wird B als n-Bit Zahl interpretiert bei der B(B'First) das höchstwertigste Byte und B(B'Last) das niederwerigste Byte dieser Zahl darstellt. Diese Zahl wird in ein DWord-Array konvertiert, das ebenfalls als Zahl interpretiert wird, bei der das erste Element (First) den höchstwertigsten und das letzte Element (Last) den niederwertigsten Teil der Zahl darstellt.

Beispiel:

```
B : Bytes(1..10) : (10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
D : DWords := Byte_to_Dwords(B);
-- D(D'First) =                 16#0A_14_1E_28_32_3C_46_50#
-- D(D'Last) = D(First+1) = 16#5A_64_00_00_00_00_00_00#
```

To_String

```
function Bytes_To_String(ASCII : Bytes) return String;
```

Diese Funktion wandelt ein Byte-Array in einen String um, indem sie die einzelnen Bytes als ASCII-Code interpretiert.

To_Hex

```
function To_Hex(B : Byte)  return Hex_Byte;
function To_Hex(W : Word)  return Hex_Word;
function To_Hex(D : DWord) return Hex_DWord;
```

Diese Funktionen wandelt ein Byte, Word oder DWord in einen Hex-String der Länge 2, 8 oder 16 um. Beispielsweise entspricht `Put(To_Hex(W));` in C dem Ausdruck `printf(''%08X'', i);`.

Beispiel:

```
B : Word := 0;
W : Word := 16#AABBCCDDEEFF#;

HB: Hex_Byte:=To_Hex(W); -- Es gilt: HB="00".
HW: Hex_Word:=To_Hex(W); -- Es gilt: HW="0000AABBCCDDEEFF".
```

3.3.4 Is_Zero Funktionen

```
function Is_Zero(Byte_Array  : Bytes)  return Boolean;
function Is_Zero(Word_Array  : Words)  return Boolean;
function Is_Zero(DWord_Array : DWords) return Boolean;
```

Diese Funktionen geben "True" zurück, wenn alle Felder des übergebenen Arrays den Wert "0" enthalten. Ansonsten geben diese Funktionen "False" zurück.

3.3.5 Padding Prozeduren

```
procedure Padding(Data           : in out Bytes;
                  Message_Length : in  Word;
                  Data2          : out Bytes);

procedure Padding(Data           : in  out Words;
                  Message_Length : in  Word;
                  Data2          : out Words);

procedure Padding(Data           : in  out DWords;
                  Message_Length : in  Word;
                  Data2          : out DWords);
```

Parameter:

Data:
Data muss ein Array sein, das eine Nachricht enthält. Die Nachricht muss folgendem Bereich von Data entsprechen:
`Data(Data'First)..Data(Data'First+(Message_Length-1))`

Message_Length:
Die Länge der Nachricht die Data enthält.

Data2:
Wird für Sonderfälle benötigt. Im Normalfall gilt nach dem Prozeduraufruf Is_Zero(Data2) = true.

Vorbedingungen:

1. Data'Length = Data2'Length
2. Message_Length $\leq$ Data'Length
3. Word(Data'Length) - Message_Length > 2**16-1[1]

Nachbedingung:

1. Message_Length < Data'Length-2 $\Rightarrow$ Is_Zero(Data2) = True
2. Is_Zero(Data2) = False $\Longleftrightarrow$ Message'Length > Data'Length-2

Exceptions:
Constraint_Message_Length_Error : Message_Length > Data'Length
Constraint_Length_Error : Data'Length > (2 * Byte'Last) [1]

Diese Funktionen fügt einer Nachricht Null-Bytes/Words/Dwords an gefolgt von einem Byte/Word/DWord das der Anzahl der eingefügten Null-Bytes entspricht. Dabei können folgende Sonderfälle auftreten:

1. Message_Length+1=Data'Length
   Nachdem ein Null-Byte an die Nachricht angefügt wurde ist kein Platz mehr vorhanden um die Anzahl der eingefügten Null-Bytes/Words/DWords an die Nachricht anzuhängen. Data(Data'Last) wurde ja mit einem Null-Byte/Word/DWord aufgefüllt. Um dieses Problem zu lösen werden alle Felder von Data2 außer Data2'Length auf 0 gesetzt. Data2'Last wird auf den Wert Data2'Length gesetzt.
2. Message_Length=Data'Length
   Analog zu dem ersten Sonderfall bis auf folgende Abweichungen:

(a) Es wird kein Null-Byte/Word/DWord an die Nachricht angehängt

(b) Data2'Last := Data2'Length-1

Beispiel

```
   ...
   package BIO is new Ada.Text_Io.Modular_IO (Byte);

   X : Bytes(1..9) := (1 => 2, 2 => 4, 3 => 6,
                       4 => 8, 5 => 9, others =>1);
   Y : Bytes(1..9);
begin
   Padding(X, N, Y);
   Put("X:");
   for I in X'Range loop
      BIO.Put(X(I));
   end loop;
   New_Line;
   if Is_Zero(Y) = False  then
       Put("Y:");
       for I in Y.all'Range loop
          BIO.Put(Y.all(I));
       end loop;
    end if;
...
```

Ausgabe:

- N = 5
  X : 2 4 6 8 9 0 0 0 3
- N = 8
  X : 2 4 6 8 9 1 1 1 0
  Y : 0 0 0 0 0 0 0 0 9
- N = 9
  X : 2 4 6 8 9 1 1 1 1
  Y : 0 0 0 0 0 0 0 0 8

Kapitel 4

Crypto.Random

4.1 Einführung

Dieses Paket greift defaultmäßig auf “dev/random/” zu. Unter Linux ist “/dev/random” ein Pseudodevice das kryptographisch sichere Pseudozufallsbits liefert. Wenn sie ein Betriebsystem verwenden das kein solches Pseudodevice besitzt, dann können sie dies mit Hilfe von mit Brian Warners “Entropy Gathering Daemon”(EGD), der unter *http://www.lothar.com/tech/crypto/* zu finden ist, nachinstallieren. Der EGD ist ein Userspace-Implementierung des Linux-Kernel-Devices “dev/random/”, das kryptograpisch sichere Zuallsbits generiert.
Im privaten Teil befindet sich die Variable “Path”. Diese Variable gibt den Pfad zu einer Datei, Named Pipe oder Gerät das krytograpisch sichere Zufallsbits enthält an. Dadurch sind Sie in der Lage die ACL auch an eine kryptograpisch sichere (Pseudo-)Zufallsbitquelle eines nicht POSIX kompatiblen Betriebsystems wie Windows XP anbinden.
Weiterhin besteht die Möglichkeit unter Windows den Marsaglia- RNG zu benutzen. Er ist ein lagged- Fibonacci Sequenz Generator und erzeugt 24-Bit real numbers im Intervall [0,1].
Eigenschaften: Kombination zweier unterschiedlicher Generatoren, welche fortlaufende Bits erzeugen. Diese werden durch zwei weitere Generatoren zu einer Tabelle Kombiniert. Die Prozedur ”Start” initialisiert diese Tabelle, welche von der Funktion ”Next” genutzt wird um die nächste Zufallszahl zu generieren.

4.2 API

4.2.1 Prozeduren

```
procedure Read(B : out Byte);
procedure Read(W : out Word);
procedure Read(D : out DWord);
```

Bei diesen drei Prozeduren wird B, W oder D mit Bits aus der verwendeten Pseudozufallsbit-Quelle gefüllt.

```
procedure Read(Byte_Array  : out Bytes);
procedure Read(Word_Array  : out Words);
procedure Read(DWord_Array : out DWords);
```

Diese drei Prozeduren wirden die übergebenen Arrays mit Bits aus der verwendeten Pseudozufallsbit-Quelle gefüllt.

```
procedure Start(
New_I : Seed_Range_1 := Default_I;
New_J : Seed_Range_1 := Default_J;
New_K : Seed_Range_1 := Default_K;
New_L : Seed_Range_2 := Default_L);
```

4.2.2 Exceptions

```
Random_Source_Does_Not_Exist_Error : exception;
```

Diese Ausnahme wird aufgerufen, wenn kein (kryptograpisch sicherer) Pseudozufallsbit-Quelle gefunden wurde.

```
Random_Source_Read_Error : exception;
```

Wenn ein Lesefehler bei einem Zugriff auf die Pseudozufallsbit-Quelle auftritt, wird die folgende Aufnahme ausgeworfen.

Kapitel 5

Crypto.Symmetric

5.1 Funktion

Dies ist das Wurzelpaket für die symmetrische Kryptographie. Dieses Paket ermöglicht dem symmetrischen Teil den direkten Zugriff auf `Crypto.Types` und `Crypto.Random`. Diesen beiden Pakete stellen grundlegende Typen und Basisfunktionen für den symmetrischen Teil der ACL zur Verfügung.
Das Herzstück der symmetrischen Krypographie bilden in der ACL Blockchiffren. In dem folgenden Abschnitt wird der Aufbau einer solchen innerhalb der ACL erläutert.

5.2 Blockchiffren

Logischer Aufbau

Eine Blockchiffre ist ein drei logische Schichten unterteilt. Wobei man als Anwendungsprogrammierer nur die API der höchsten Schicht verwenden sollte. Denn nur mit dieser API ist eine sicher Nutzung einer Blockchiffre garantiert.

1. Schicht: Algorithmus

Auf dieser Ebene befindet sich der nackte Algorithmus einer symmetrischen Chiffre. Die API der einzelnen Algorithmen ist identisch und dient als Schnittstelle für die Blockchiffen-Schicht. Bei jedem Algorithmus wird die Quelle auf der er beruht genannt. Damit ist es möglich sich mit Hilfe der Quelle von der Korrektheit der Implementierung zu überzeugen. Es ist auf keinen Fall eine schlechte Idee dies auch zu tun. Mehrere Informationen zu den einzelnen Algorithmen und Chiffren erfahren Sie im nächsten Kapitel.

Bemerkung:
Verwenden sie die API dieser Schicht nur um eine Blockchiffre zu generieren.

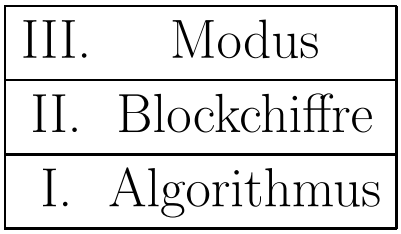


Abbildung 5.1: Die 3-Schichten-Architektur einer symmetrischen (Block-)chiffre

Verwenden Sie sie auf keinen Fall für einen anderen Zweck. Es gibt kein Szenario in dem dies einen Sinn ergibt. Diese API dient wirklich nur als Schnittstelle für die 2. Schicht.

2. Schicht: Blockchiffre

Auf dieser Ebene wird aus einem Algorithmus eine Blockchiffre im unsicheren ECB-Modus (Electronic Code Book) generiert. Verwenden sie diese API nur, wenn Sie sich auch mit symmetrischen Blockchiffren auskennen. Diese API dient eigentlich nur als Schnittstelle für die 3. Schicht. In Kapitel **??** wird beschrieben wie man aus einem Algorithmus eine Blockchiffre generiert.

3. Schicht: Modus

Diese Schicht versetzt eine Blockchiffre in einen vernünftigen Modus. Es gibt verschiedene Modi für verschiedene Szenarien die alle ihre Vor- und Nachteile haben. Alle unterstützen Modi und deren Vor- und Nachteil werden in dem Kapitel **??** ausführlich beschrieben.

Bemerkung

Mit dieser 3. Schichten Architektur ist es auch möglich Strom-Chiffren zu implementieren. Man muss dazu aus er 1. Ebene eine Prozedur zur Verfügung stellen die die nächsten n-Bit des Keystreams liefert. Damit kann man eine Oneway-Blockchiffre (Kapitel **??**) konstruieren und diese in den Oneway-OFB oder den Oneway-Counter-Modus versetzen. Im Moment beinhaltet die ACL noch keinen Strom-Chiffre. Es ist aber geplant die Strom-Chiffre Helix [**?**] zu implementieren.

Kapitel 6

Crypto.Symmetric.Algorithm

Alle Subpakete dieses Paketes sind symmetrische Chiffren oder Hashfunktionen. Hier haben Sie einen direkten Zugriff auf die einzelnen Operationen die eine Chiffre oder Hashfunktion zur Verfügung stellt. Sie sollten aber davon Abstand nehmen die vorhandene API zu benutzen. Dies ist so als ob Sie direkt Datenblöcke auf die Festplatte schreiben und von der Festplatte lesen ohne ein Dateisystem zu verwenden. Ausserdem importiert deieses Paket `Ada.IO_Exceptions` und `Gnat.Os_Lib` und stellt deren API all seinen Kindpaketen zur Verfügung.

6.1 Symmetrische Chiffren

6.1.1 Allgemein

Bei symmetrischen Chiffren werden von der ACL für den Anwendungsprogrammierer folgende Typen und Operationen zur Verfügung gestellt.

- Einen privaten Typ **Cipherkey** der deterministisch aus einem Schlüssel generiert wird. Mit Hilfe dieses **Cipherkey** kann man den die Ver- und Entschlüsselungsfunktion der Chiffre aufrufen.
- Eine Prozedur **Prepare_Key** die aus einem Schlüssel den zugehörigen Cipherkey erstellt.

  Beispiel:

  ```
  procedure Prepare_Key(Key      : in  B_Block128;
                        Cipherkey : out Cipherkey_Type);
  ```

- Eine Prozedur **Encrypt**. Diese Prozedur verschlüsselte einen Klartext (Plaintext) mit Hilfe eines Cipherkeys in einen Chiffretext (Ciphertext).

  Beispiel:

```
procedure Encrypt(Cipherkey  : in  Cipherkey_Type;
                  Plaintext  : in  B_Block128;
                  Ciphertext : out B_Block128);
```

- Eine Prozedur **Decrypt**. Dies Prozedur entschlüsselt mit Hilfe des passenden Cipherkeys einen Chiffretext (Ciphertext) in den zugehörigen Klartext (Plaintex).

  Beispiel:

```
procedure Decrypt(Cipherkey  : in  Cipherkey_Type;
                  Ciphertext : in  B_Block128;
                  Plaintext  : out B_Block128);
```

6.1.2 AES (Rijndael)

Rijndael ist der vom Nist evaluierte und 2001 verabschiedete Advanced Encryption Standard (AES) und offizieller DES Nachfolger. AES ist eine 128-Bit Blockchiffre der mit 128, 192 oder 265 Bit Schlüsseln betrieben werden kann. Wenn sie Rechtssicherheit benötigen oder sich gerne an allgemeine Standards halten, dann ist AES die Chiffre ihrer Wahl. Diese Implementation wurde erfolgreich mit den Testwerten aus der FIPS-197 [**?**] auf dessen Korrektheit getestet. Diese Implementation ist für 32-Bit Plattformen optimiert, und benutzt die Runden-Transformation für 32-Bit Prozessoren aus dem Paper „AES Proposal: Rijndael “[**?**],

API

AES mit 128-Bit Schlüssel:

```
type Cipherkey_AES128 is private;

procedure Prepare_key128(Key       : in  B_Block128;
                         Cipherkey : out Cipherkey_AES128);

procedure Encrypt128(Cipherkey  : in Cipherkey_AES128;
                     Plaintext  : in B_Block128;
                     Ciphertext : out B_Block128);

procedure Decrypt128(Cipherkey  : in Cipherkey_AES128;
                     Ciphertext : in B_Block128;
                     Plaintext  : out B_Block128);
```

AES mit 192-Bit Schlüssel:

```
type Cipherkey_AES192 is private;

procedure Prepare_key192(Key       : in  B_Block192;
                         Cipherkey : out Cipherkey_AES192);

procedure Encrypt192(Cipherkey  : in Cipherkey_AES192;
                     Plaintext  : in B_Block128;
                     Ciphertext : out B_Block128);


procedure Decrypt192(Cipherkey  : in  Cipherkey_AES192;
                     Ciphertext : in B_Block128;
                     Plaintext  : out B_Block128);
```

AES mit 256-Bit Schlüssel:

```
procedure Prepare_key256(Key       : in B_Block256;
                         Cipherkey : out Cipherkey_AES256);

procedure Encrypt256(Cipherkey  : in Cipherkey_AES256;
                     Plaintext  : in B_Block128;
                     Ciphertext : out B_Block128);

procedure Decrypt256(Cipherkey  : in Cipherkey_AES256;
                     Ciphertext : in B_Block128;
                     Plaintext  : out B_Block128);
```

6.1.3 Serpent

Serpent ist ein weitere 128-Bit Blockchiffre und einer der 5 Finalisten des AES-Wettbewerbs. Wenn Sie sehr hohen Wert auf Sicherheit und nicht so hohen Wert auf Performance legen, dann sollte Sie auf jeden Fall mal einen Blick auf diese Blockchiffre wagen. Die Implementation basiert auf der Ada-Referenzimplementation von Markus G. Kuhn. Diese kann nur mit 256-Bit Schlüsseln betrieben werden.
Eine API für 128- und 192-Bit Schlüssel ist geplant.

API

```
type Cipherkey_Serpent256 is private;

procedure Prepare_Key256(Key       : in  B_Block256;
                         Cipherkey : out Cipherkey_Serpent256);
```

```
procedure Encrypt256(Cipherkey  : in  Cipherkey_Serpent256;
                     Plaintext  : in  B_Block128;
                     Ciphertext : out B_Block128);

procedure Decrypt256(Cipherkey  : in  Cipherkey_Serpent256;
                     Ciphertext : in  B_Block128;
                     Plaintext  : out B_Block128);
```

6.1.4 Tripel DES (3DES, TDES, TDEA)

Diese 64-Bit Blockchiffre arbeitet mit einem 192-Bit Schlüssel den sie in 3*64-Bit Schlüssel zerlegt. Mit dem ersten Schlüssel wir der Klartext mittels DES (Data Encryption Standard) verschlüsselt. Die Ausgabe wird dann mit dem zweiten 64-Schlüssel mittels DES entschlüsselt und diese Ausgabe wird wiederum mit dem dritten 64-Bit Schlüssel mittels DES verschlüsselt. DES ist Software relativ langsam. Aus diesem Grund ist TDES die langsamste Blockchiffre. Man kann bei dieser Blockchiffre davon ausgehen, das diese Chiffre nicht mehr gebrochen wird, da DES über 20 Jahre von den besten Kryptoanalytikern auf Schwächen untersucht wurde. Wenn Sie keinen großen Wert auf Performance legt und sich einer effektiven Schlüssellänge von 112-Bit geben, dann sollte Sie TDES verwenden. Diese Bibliothek unterstützt zwei unterschiedliche Implementationen. Die eine stammt aus dem Buch „Angewandte Kryptographie“von Burce Scheier [**?**]. Diese ist für 32-Bit Prozessoren optimiert. Die andere Implementation ist als „obsolete “gekennzeichnet. Diese Implementation entspricht bis auf ein paar kleine Optimierungen dem Algorithmus aus der FIPS-800-67 [**?**]. Diese Implementation ist aber um einiges langsamer.

API

TDES:

```
type Cipherkey_TDES is private;

 procedure Prepare_Key(Key        : in B_Block192;
                       Cipherkey  : out Cipherkey_TDES);

 procedure Encrypt(Cipherkey  : in  Cipherkey_TDES;
                   Plaintext  : in  B_Block64;
                   Ciphertext : out B_Block64);

 procedure Decrypt(Cipherkey  : in  Cipherkey_TDES;
                   Ciphertext : in  B_Block64;
                   Plaintext  : out B_Block64);
```

Obsolete TDES:

```
type Obsolete_Cipherkey_TDES is private;

procedure Obsolete_Prepare_Key
  (Key        : in B_Block192;
   Cipherkey  : out Obsolete_Cipherkey_TDES);

procedure Obsolete_Encrypt
  (Cipherkey :in  Obsolete_Cipherkey_TDES;
   Plaintext  : in  B_Block64;
   Ciphertext : out B_Block64);

procedure Obsolete_Decrypt
  (Cipherkey : in Obsolete_Cipherkey_TDES;
   Ciphertext : in  B_Block64;
   Plaintext  : out B_Block64);
```

6.1.5 Twofish

Twofish ist eine 128-Bit Blockchiffre, und ein weiterer Finalist des AES-Wettbewerbs. Towfish hat eine klassische Feistelstruktur und schlüsselabhängige S-Boxen. Diese Implementation basiert auf der C-Referenzimplementation von Niels Ferguson [**?**] und der Twofish-Spezifikation [**?**], und kann mit 128, 192 oder 265 Bit Schlüsseln betrieben werden.

API

Twofish mit 128-Bit Schlüssel:

```
subtype Cipherkey_Twofish128 is Cipherkey_Twofish;

procedure Prepare_Key128(Key        : in B_Block128;
                         Cipherkey : out Cipherkey_Twofish128);

procedure Encrypt128(Cipherkey  : in  Cipherkey_Twofish128;
                     Plaintext  : in  B_Block128;
                     Ciphertext : out B_Block128);

procedure Decrypt128(Cipherkey  : in  Cipherkey_Twofish128;
                     Ciphertext : in  B_Block128;
                     Plaintext  : out B_Block128);
```

Twofish mit 192-Bit Schlüssel:

```
subtype Cipherkey_Twofish192 is Cipherkey_Twofish;

procedure Prepare_Key192(Key        : in B_Block192;
                         Cipherkey  : out Cipherkey_Twofish192);

procedure Encrypt192(Cipherkey  : in  Cipherkey_Twofish192;
                     Plaintext  : in  B_Block128;
                     Ciphertext : out B_Block128);


procedure Decrypt192(Cipherkey  : in  Cipherkey_Twofish192;
                     Ciphertext : in  B_Block128;
                     Plaintext  : out B_Block128);
```

Twofish mit 256-Bit Schlüssel:

```
subtype Cipherkey_Twofish256 is Cipherkey_Twofish;

procedure Prepare_Key256(Key        : in B_Block256;
                         Cipherkey  : out Cipherkey_Twofish256);

procedure Encrypt256(Cipherkey  : in  Cipherkey_Twofish256;
                     Plaintext  : in  B_Block128;
                     Ciphertext : out B_Block128);


procedure Decrypt256(Cipherkey  : in  Cipherkey_Twofish256;
                     Ciphertext : in  B_Block128;
                     Plaintext  : out B_Block128);
```

6.2 Hashfunktionen

6.2.1 Allgemein

Unter Hashfunktion sind hier kryptographische Hashfunktionen mit einem iterativen Aufbau wie beispielsweise SHA-1 [**?**] gemeint. Eine solche Hashfunktion lässt sich bei der ACL entweder durch eine Low-Level- oder einer High-Level-API ansprechen.

High-Level-API

Die High-Level-API besteht aus drei Prozeduren mit der sich Strings, Byte-Arrays und Datein hashen lassen.

Beispiel:

```
procedure Hash(Message : in String; Hash_Value : out W_Block160);
procedure Hash(Message : in Bytes;  Hash_Value : out W_Block160);

procedure F_Hash(Filename : in String; Hash_Value : out W_Block160);
```

Low-Level-API

Bevor die Low-Level-API erläutert wird folgt zunächst ein kleiner Exkurs über iterative Hashfunktionen

Exkurs: Iterative Hashfunktionen

Eine iterative Hashfunktion H zerlegt eine Nachricht M in m-Bit große Nachrichtenblöcke M_i $(1 \leq i \leq n)$. Anschließend hashed alle Nachrichenblöcke sequentiell. Damit ein Hashwert h für die komplette Nachricht M erstellt werde kann, muss sich die Hashfunktion die Hashwerte der einzelnen Nachrichtenblöcke speichern. Damit man sich nicht n Hashwerte merken muss wird der Haswert $h = h_n$ eienr Nachricht iterativ berechnet. D.h. der Hashwert des it-en Nachrichtenblocks hängt von $h_1, ..., h_{i-1}$. Im Allgemeinen gilt $h_i := H(M_i, h_{i-1})$. Für i=1 gilt damit $h_1 := H(M_1, h_0)$. Dabei ist h_0 der initale Hashwert, der zu beginn mit einem Initialen Hashwert Initialisiert wird.
Nachrichten deren Länge keinem Vielfachen von m-Bit entspricht werden durch Padding aufgebläht. I.d.R. wird der letzte Nachrichtenblock immer einer speziellen Paddingverfahren unterworfen. D.h. Nachrichten die $j \cdot m$ lang sind werden auf eine Länge von $(j + 1) \cdot m$ aufgebläht.

Mit diesem Wissen widmen wir uns nun den einzelen Funktionen und Prozeduren der Low-Level-API.

- Die Prozedure **Init** setzt den Hashwert (*Hash_Value*) auf seinen initalen Wert. Wenn eine Nachricht gehashed werde soll muss diese Prozedur aufgerufen werden.

  Beispiel:

  ```
  procedure Init(Hash_Value : out W_Block160);
  ```

- Ein Nachrichtenblock (*Message_Block*) wird mit Hilfe der Prozedure **Round** gehashed.

  Beispiel:

```
procedure Round(Message_Block : in      W_Block512;
                Hash_Value    : in out  W_Block160);
```

- Mit der Funktion **Final_Round** wird der letze Nachrichtenblock (*Last_Message_Block*) aufgebläht und der finale Hashwert zurückgeliefert. Für das Padding wird die Bytelänge muss die Bytelänge des letzten Nachrichtenblockes (*Message_Block_Length*) als Parameter mit übergeben weden.

  Beispiel:

```
function Final_Round
  (Last_Message_Block   : W_Block512;
   Last_Message_Length  : Message_Block_Length512;
   Hash_Value           : W_Block160)
  return W_Block160;
```

6.2.2 SHA-1

Bei SHA-1 [**?**] handelt es sich um kryptographsiche 160-Bit Hashfunktion. Diese Hashunktion gehört zum *Secure Hash Standard* [**?**] und ist die derzeitige Standardhashfunktion. Auf der Crypto 2005 stellten X. Wang, Y.L. Yin und H. Yu erhebliche Schwächen dieser Hashfunktion fest [**?**]. Aufgrund dieser neuen Erkenntnisse ist es nicht ratsam diese Hashfunktion zu verwenden. Der einzige Grund weshalb die ACL diese Hashfunktion unterstützt ist ihr immenser Verbeitungsrad in Standardprotokollen wie beispielsweise dem Digitalen Signatur Standard (Kapitel **??**) . [**?**] Diese Hashfunktion wird voraussichtlich nicht mehr lange von der ACL unterstützt. Aus diesme Grund sollte sie wenn möglich eine andere Hashfunktion verwenden.
Anmerkungen:
Diese Implementation von SHA-1 ist langsamer als die Implementation von SHA-256.

API

```
-- High-Level-API
procedure Hash(Message : in String; Hash_Value : out W_Block160);
procedure Hash(Message : in Bytes;  Hash_Value : out W_Block160);

procedure F_Hash(Filename : in String; Hash_Value : out W_Block160);
```

```
--Low-Level-API
 procedure Init(Hash_Value : out W_Block160);

 procedure Round(Message_Block : in      W_Block512;
                 Hash_Value    : in out  W_Block160);

 function Final_Round(Last_Message_Block   : W_Block512;
                      Last_Message_Length  : Message_Block_Length512;
                      Hash_Value           : W_Block160)
                      return W_Block160;
```

Ausnahme

```
SHA1_Constraint_Error : Exception;
```

Diese Ausnahme tritt auf, wenn eine Nachricht die Länger als $2^{64} - 1024$-Bit (ca. 2 Millionen TB) ist mittels SHA-1 gehashed werden soll.

6.2.3 SHA-256

SHA-256 [**?**] ist wie der Name schon sagt eine 256-Bit Hashfunktion und ein Mitglied der SHA-2 Familie die auf SHA-1 beruht. SHA-256 ist ebenfalls wie alle anderen SHA-2 Familien im *Secure Hash Standard* definiert. Dies Hashfamilie hat einen ausreichend langen Hashwert und macht signifikante Veränderungen gegenüber des SHA-1 Algorithmus einen sicheren Eindruck. Da diese Hashfamilie mit 32-Bit Wörtern arbeitet ist der einsatz dieser auf 32-Bit Prozessoren eine Überlegung wert. Wenn möglich sollte SHA-1 durch diese neue Generation der SHA-Hashfamilie ausgetauscht werden.

API

```
-- High-Level-API

procedure Hash(Message : in Bytes;  Hash_Value :  out W_Block256);
procedure Hash(Message : in String; Hash_Value :  out W_Block256);

procedure F_Hash(Filename : in String; Hash_Value :  out W_Block256);


-- Low-Level-API
procedure Init(Hash_Value : out W_Block256);
```

```
procedure Round(Message_Block : in W_Block512;
                Hash_Value    : in out W_Block256);

function Final_Round(Last_Message_Block   : W_Block512;
                     Last_Message_Length  : Message_Block_Length512;
                     Hash_Value           : W_Block256)
                     return W_Block256;
```

Ausnahme

```
SHA256_Constraint_Error : Exception;
```

Diese Ausnahme tritt auf, wenn eine Nachricht die Länger als $2^{64} - 1024$-Bit (ca. 2 Millionen TB) ist mittels SHA-256 gehashed werden soll.

6.2.4 SHA-512

Die 512-Bit Hashfamilie SHA-512 [**?**] ist die sicherste Mitglied der SHA-2 Familie die im *Secure Hash Standard* definiert ist. sie arbeitet mit 64-Bit Wörtern und eignet sich daher besonders für 64-Bit Architekturen. Wenn sie eine sehr sichere und standadisierte Hashfunktion suchen, dann ist diese Hashfamilie wahrscheinlich genau das was sie suchen.

API

```
-- High-Level-API
procedure Hash(Message : in Bytes;  Hash_Value : out DW_Block512);
procedure Hash(Message : in String; Hash_Value : out DW_Block512);

procedure F_Hash(Filename : in String; Hash_Value : out DW_Block512);


-- Low-Level-API
procedure Init(Hash_Value : out DW_Block512);

procedure Round(Message_Block : in DW_Block1024;
                Hash_Value    : in out DW_Block512);

function Final_Round(Last_Message_Block   : DW_Block1024;
                     Last_Message_Length  : Message_Block_Length1024;
                     Hash_Value           : DW_Block512)
                     return DW_Block512;
```

Ausnahme

```
SHA2_Constraint_Error : Exception;
```

Diese Ausnahme tritt auf, wenn eine Nachricht die Länger als 2^{128} − 2048-Bit (ca. 2^{80} TB) ist mittels SHA-512 gehashed werden soll.

6.2.5 SHA-384

Bei SHA-384 hadnelt es sich um SHA-512. Diese Hashfunktion unterschieded sich genau in zwei Punkten von SHA-512.

1. SHA-384 verwendet andere Initalwerte als SHA512.
2. Bei der finalen Rounde wird bei SHA-384 der zurückgelieferte Hashwert auf durch abschneiden auf 384-Bit gekürzt.

Da diese Hashfamilie intern einen 512-Bit Hashwert verwendet ist dies nicht zu der generischen API von *Crypto.Symmetric.Hashfunction* kompatibel. Daher sollte darauf verzichet werden diese Hashfunktion zu verwenden.

API

```
-- High-Level-API
procedure Hash(Message : in Bytes;  Hash_Value : out DW_Block384);
procedure Hash(Message : in String; Hash_Value : out DW_Block384);

procedure F_Hash(Filename : in String; Hash_Value : out DW_Block384);


-- Low-Level-API
procedure Init(Hash_Value : out DW_Block512);

procedure Round(Message_Block : in DW_Block1024;
                Hash_Value : in out DW_Block512);

function Final_Round(Last_Message_Block   : DW_Block1024;
                     Last_Message_Length  : Message_Block_Length1024;
                     Hash_Value           : DW_Block512)
                     return DW_Block384;
```

6.2.6 Whirlpool

Bei Whirlpool [**?**] handelt es sich um eine 512-Bit Hashfunktion die 64-Bit Wörter verwendet. Whirlpool wurde nicht wie die SHA-Hashfunktionen von der

NSA sondern von den beiden freien Kryptographen Paulo Barreto und Vincent Rijmen entwickelt. Diese Hashfunktion ist algorithmisch mit der Blockchiffre Rijndael aka AES verwandt. Dies liegt daran, das Vincent Rijmen einer der beiden Rijndael Designer ist. Wenn sie eine sichere Hashfunktion benötigen und keine Kompatibilität zum amerikanischen Krypto-Standard benötigen, dann sollten sie in betracht ziehen Whirlpool zu verwenden.

API

```
-- High-Level-API
procedure Hash   (Message   : in Bytes;  Hash_Value : out DW_Block512);
procedure Hash   (Message   : in String; Hash_Value : out DW_Block512);

procedure F_Hash(Filename : in String; Hash_Value : out DW_Block512);


-- Low-Level-API
procedure Init(Hash_Value : out DW_Block512);

procedure Round(Message_Block : in     DW_Block512;
                Hash_Value    : in out DW_Block512);

function Final_Round(Last_Message_Block : DW_Block512;
                     Last_Message_Length: Message_Block_Length512;
                     Hash_Value         : DW_Block512)
                     return DW_Block512;
```

Ausnahme

```
SHA2_Constraint_Error : Exception;
```

Diese Ausnahme tritt auf, wenn eine Nachricht die Länger als $2^{128} - 2048$-Bit (ca. 2^{80} TB) ist mittels SHA-512 gehashed werden soll.

6.2.7 Ausnahmen

```
File_Open_Error : exception;
```

Diese Ausnahmen wird geworfen wenn es zu einem Fehler beim Öffnen einer zu hashenden Datei kommt.

```
File_Read_Error : exception;
```

Diese Ausnahmen wird geworfen wenn es zu einem Fehler beim lesen einer zu hashenden Datei kommt.

Kapitel 7

Crypto.Symmetric.Algorithm.Oneway

Jede symmetrische Chiffre und kryptographische Hashfunktion hat ein Unterpaket „Oneway “. Bei diesem Paket handelt es sich um eine symmetrische Chiffre ohne Entschlüsslungsoperation.
Auch bei den Oneway-Algorithmen sollte sie davon Abstand nehmen die vorhanden API direkt zu verwenden.

API

Ein Oneway-Paket stellt dem Anwendungsprogrammierer folgende Typen und Operationen zur Verfügung.

- Einen privaten Typ **Cipherkey_Oneway** der deterministisch aus einem Schlüssel generiert wird. Mit dessen Hilfe kann man die Verschlüsselungsfunktion des entsprechenden Paketes aufrufen.
- Eine Prozedur **Prepare_Oneway_Key** die aus einem Schlüssel den zugehörigen Cipherkey erstellt.

  Beispiel:

  ```
  procedure Prepare_Oneway_Key(Key       : in  B_Block;
                               Cipherkey : out Cipherkey_Oneway);
  ```

- Eine Prozedur **Encrypt_Oneway**. Diese Prozedur verschlüsselte einen Klartext (Plaintext) mit Hilfe eines Cipherkeys in einen Chiffretext (Ciphertext).

  Beispiel:

  ```
  procedure Encrypt_Oneway(Cipherkey  : in  Cipherkey_Oneway;
                           Plaintext  : in  B_Block;
                           Ciphertext : out B_Block);
  ```

Kapitel 8

Crypto.Symmetric.Blockcipher

Mit Hilfe dieses generischen Paketes lässt sich aus dem Algorithmus einer symmetrischen Chiffre (siehe Kapitel **??**) eine Blockchiffre erstellen. Sie sollten davon Abstand nehmen die API dieses Paketes direkt zu verwenden, denn dieses Paket implementiert eine Blockchiffre im "unsicheren" ECB-Modus (Electronic Codebook Modus). Wenn man in diesem Modus zwei identische Klartextblöcke p1=p2 mit dem selben Schlüssel chiffriert, dann erhält man zwei identische Chiffretextblöcke c1=c2. Dadurch kann der Chiffretext noch Informationen über die Struktur des Klartextes enthalten.

8.1 API

Die API eine Blockchiffre besteht aus drei Prozeduren.

1. Die Prozedur **Prepare_Key** weist einer Blockchiffre einen Schlüssel *Key* zu.

   ```
   procedure Prepare_Key(Key : in Key_Type);
   ```

2. Die Prozedur **Encrypt** verschlüsselt (mit Hilfe eines vorher zugewiesenen Schlüssels) einen Klartextblock (Plaintext) in einen Chiffretextblock (Ciphertext)

   ```
   procedure Encrypt(Plaintext : in Block; Ciphertext : out Block);
   ```

3. Die Prozedur **Decrypt** entschlüsselt (mit Hilfe eines vorher zugewiesenen Schlüssels) einen Chiffretextblock (Ciphertext) in einen Klartextblock (Plaintext).

   ```
   procedure Decrypt(Ciphertext : in Block; Plaintext  : out Block);
   ```

8.2 Generischer Teil

```
type Block is private;
type Key_Type is private;
type Cipherkey_Type is private;

with procedure Prepare_Key(Key        : in  Key_Type;
                           Cipherkey  : out Cipherkey_Type);

with procedure Encrypt(Cipherkey  : in  Cipherkey_Type;
                       Plaintext  : in  Block;
                       Ciphertext : out Block);

with procedure Decrypt(Cipherkey  : in  Cipherkey_Type;
                       Ciphertext : in  Block;
                       Plaintext  : out Block);
```

8.3 Anwendungsbeispiel

```
with Crypto.Types;
with Crypto.Symmetric.Blockcipher;
with Crypto.Symmetric.Algorithm.TripleDES;

procedure Generic_TripleDES is
use Crypto.Types;
use Crypto.Symmetric.Algorithm.TripleDES;

   package Generic_TripleDES is
      new Crypto.Symmetric.Blockcipher
      (Block          => B_Block64,
       Key_Type       => B_Block192,
       Cipherkey_Type => Cipherkey_TDES,
       Prepare_Key    => Prepare_Key,
       Encrypt        => Encrypt,
       Decrypt        => Decrypt);
begin
   ...
end Generic_TripleDES;
```

8.4 Anmerkung

Sie müssen nicht jedes mal von neuem aus einem symmetrischen Algorithmus eine Blockchiffre generiern. Statdessen können Sie auch eine der folgenden vor-

gefertigten Blockchiffren verwenden.

- Crypto.Symmetric.Blockcipher_AES128
- Crypto.Symmetric.Blockcipher_AES192
- Crypto.Symmetric.Blockcipher_AES256
- Crypto.Symmetric.Blockcipher_Serpent256
- Crypto.Symmetric.Blockcipher_Tripledes
- Crypto.Symmetric.Blockcipher_Twofish128
- Crypto.Symmetric.Blockcipher_Twofish192
- Crypto.Symmetric.Blockcipher_Twofish256

Kapitel 9

Crypto.Symmetric.Oneway_Blockcipher

Mit Hilfe dieses generischen Paketes können Sie aus einem symmetrischen Einweg-Algorithmus (**Kapitel ??**) oder einer Hashfunktion (**Kapitel** Kapitel **??**) eine Einweg-Blockchiffre (oneway-Blockcipher) generieren. Sie sollten aber davon Abstand nehmen die API dieses Paketes direkt zu verwenden. Diese Paket implementiert nämlich eine Blockchiffre "unsicheren" im ECB-Modus (Electronic Codebook Modus). In diesem Modus werden zwei identische Klartextblöcke p1=p2, die mit dem selben Schlüssel chiffriert werden, zu zwei identischen Chiffretextblöcken c1,c2 mit c1=c2. Der Chiffretext kann also immer noch Informationen über die Struktur des Klartextes enthalten.
Bei Einweg-Blockchiffre die auf Hashfunktionen beruhen wurde die Schlüssellänge (mind. 256 Bit) so gewählt, dass es unmöglich ist den kompletten Schlüsselraum abzusuchen. Ein (irreversibler) 219-Bit Zähler der all seine Zustände durchläuft benötigt min. $10^{51} erg.$. Dies entspricht der Energie die eine typische Supernova abgibt. Damit ein 256-Bit Zähler alle Zustände durchläuft benötigt er min. die Energie von 64 Milliarden Supernoven. Der Energieaustoß unserer Sonne beträgt im Jahr gerade mal $1,21 \cdot 10^{41}$ erg. [**?**]
Damit ist eine solche Einweg-Blockchiffre so (un-)sicher wie die zugrundeliegende Hashfunktionen.

9.1 API

Die API eine Blockchiffre besteht den folgenden beiden Prozeduren.

1. Die Prozedur **Prepare_Key** weisen sie einer Blockchiffre einen Schlüssel *Key* zu.

   ```
   procedure Prepare_Key(Key  : in Key_Type);
   ```

2. Die Prozedur **Encrypt** verschlüsselt (mit Hilfe eines vorher zugewiesenen Schlüssels) einen Klartextblock (Plaintext) in einen Ciffretextblock (Ciphertext)

```
   procedure Encrypt(Plaintext : in Block; Ciphertext : out Block);
```

9.2 Generischer Teil

```
generic
   type Block is private;
   type Key_Type is private;
   type Cipherkey_Type is private;

   with procedure Prepare_Key(Key : in Key_Type;
                              Cipherkey : out Cipherkey_Type);

   with procedure Encrypt(Cipherkey  : in Cipherkey_Type;
                          Plaintext  : in Block;
                          Ciphertext : out Block);
```

9.3 Anwendungsbeispiel

```
with Crypto.Types;
with Crypto.Symmetric.Oneway_Blockcipher;
with Crypto.Symmetric.Algorithm.TripleDES.Oneway;

procedure Generic_Oneway_TripleDES is
use Crypto.Types;
use Crypto.Symmetric.Algorithm.TripleDES.Oneway;

   package Oneway_TripleDES is
      new Crypto.Symmetric.Oneway_Blockcipher
      (Block          => B_Block64,
       Key_Type       => B_Block192,
       Cipherkey_Type => Cipherkey_Oneway_TDES,
       Prepare_Key    => Prepare_Oneway_Key,
       Encrypt        => Encrypt_Oneway);

begin
   ...
end Generic_Oneway_TripleDES;
```

9.4 Anmerkung

Sie müssen nicht jedes mal auf neue aus einem Algorithmus eine Einweg-Blockchiffre generieren. Stattdessen können Sie auch eine der folgenden vorgefertigten Blockchiffren verwenden.

- Crypto.Symmetric.Oneway_Blockcipher_AES128
- Crypto.Symmetric.Oneway_Blockcipher_AES192
- Crypto.Symmetric.Oneway_Blockcipher_AES256
- Crypto.Symmetric.Oneway_Blockcipher_SHA1
- Crypto.Symmetric.Oneway_Blockcipher_SHA256
- Crypto.Symmetric.Oneway_Blockcipher_SHA512
- Crypto.Symmetric.Oneway_Blockcipher_Serpent256
- Crypto.Symmetric.Oneway_Blockcipher_Tripledes
- Crypto.Symmetric.Oneway_Blockcipher_Twofish128
- Crypto.Symmetric.Oneway_Blockcipher_Twofish192
- Crypto.Symmetric.Oneway_Blockcipher_Twofish256
- Crypto.Symmetric.Oneway_Blockcipher_Whirlpool

Kapitel 10

Crypto.Symmetric.Mode

Mit Hilfe von generischen Betriebsmodi ist es möglich eine Blockchiffre in einem sicheren Betriebsmodus zu betreiben. Ein Betriebsmodus verknüpft für gewöhnlich eine Blockchiffre mit einer Rückkopplung und einigen einfachen Operationen (+, xor), und wird mit Hilfe eines zufälligen öffentlichen Startwertes (Initial Value (IV)) initialisiert. Der Chiffretext ist damit nicht nur von dem verwendeten Chiffre, Klartext und Schlüssel abhängig, sondern auch von dem zufälligen Startwert. Wenn Sie einen Klartext mehrmals mit der gleichen Chiffre und dem gleichen Schlüssel aber unterschiedlichen IVs verschlüsselt, erhalten Sie unterschiedliche Chiffretexte. Durch die Rückkopplung werden gleiche Klartextblöcke zu unterschiedlichen Chiffretextblöcken chiffriert, d.h ein Betriebsmodus verschlüsselt zwei Klartextblöcke p1 und p2 mit p1=p2, mit überwältigender Wahrscheinlichkeit zu zwei Chiffretextblöcke c1 und c2 mit c1≠c2. Damit ist es nun möglich mehrere Nachrichten sicher mit dem selben Schlüssel zu verschlüsseln.
Vorsicht: Um einen Chiffretext zu entschlüsseln benötigt Sie den gleichen Schlüssel und Startwert wie bei der Verschlüsselung. Aus diesem Grund sollte Sie den Startwert immer mit dem zugehörigen Chiffretext aufbewahren. **Die Sicherheit eines Modus ist unabhängig vom „Bekanntheitsgrad“des Startwertes**. Daher ist es üblich, dass man den Startwert mit dem Chiffretext multipliziert ($IV * C = C' = IV||C$), d.h. man hängt den Chiffretext einfach an den Startwert an um das finale Chiffrat C' zu erhalten.
In diesem Kapitel wird auf die einzelnen Modi ihre API, Einsatzweck und Merkmale eingegangen.

10.1 API

Die API eine Blockchiffre besteht aus den folgenden 4 Prozeduren.

1. Die Prozedur **Init** initialisiert einen Blockchiffre und Modus indem sie der Blockchiffre einen Schlüssel *Key* und dem Modus einen Startwert

Initial_Value zuweist.

```
procedure Init(Key           : in Key_Type;
               Initial_Value : in Block);
```

2. Die Prozedur **Encrypt** verschlüsselt einen Klartextblock (Plaintext) in einen Chiffretextblock (Ciphertext)

   ```
   procedure Encrypt(Plaintext  : in Block;
                     Ciphertext : out Block);
   ```

3. Die Prozedur **Decrypt** entschlüsselt einen Chiffretextblock (Ciphertext) in einen Klartextblock (Plaintext).

   ```
   procedure Decrypt(Ciphertext : in Block;
                     Plaintext  : out Block);
   ```

4. Die Prozedur **Set_IV** weist dem Modus den Startwert *Initial_Value* zu. D.h. mit Hilfe dieser Prozedur lässt sich eine Modus reinitialisieren. Nach jeder ver- bzw. entschlüsselten Nachricht (Eine Nachricht besteht aus n-Klartextblöcken,dies entspricht n-Aufrufe der *Encrypt* bzw. *Decrypt*-Prozedur) muss eine Chiffre reinitialisiert werden. Außerdem muss eine Chiffre jedes mal reinitialisieren werden wenn man den Richtung (Verschlüsselung, Entschlüsselung) des Modus ändert

   ```
   procedure Set_IV(Initial_Value : in Block);
   ```

10.2 Cipher-Block-Chaining-Modus (CBC)

Paket: Crypto.Symmetric.Mode.CBC

Bei diesem Betriebsmodus fließt das Ergebnis der Verschlüsselung früherer Blöcke in die Verschlüsselung des aktuellen Blockes mit ein, indem der Klartext vor der Verschlüsselung mit dem vorherigen (zwischengespeicherten) Chiffretextblock mit der XOR-Operation verknüpft wird.

10.2.1 Verschlüsselung

Bei der Initialisierung wird der Startwert IV als C_0 zwischengespeichert. Wird nun ein Klartextblock P_i verschlüsselt, so wird dieser zuerst mit C_0 mittels der XOR-Operation verknüpft und danach mit Hilfe der Blockchiffre zu dem Chiffretextblock C_1 verschlüsselt und zwischengespeichert. Der nächste Klartextblock P_2 wird zuerst mit C_1 mittels der XOR-Operation verknüpft und danach zu C_2 verschlüsselt und zwischengespeichert usw.

Mathematische Beschreibung: $C_i = E_K(P \oplus C_{i-1})$

10.2.2 Entschlüsselung

Die Entschlüsselung verläuft analog zur Verschlüsselung. Zu Beginn wird der Betriebsmodus mit dem Startwert C_0 initialisiert bzw. mittels Set_IV reinitialisiert. Der Chiffretextblock C_1 wird nun ganz normal entschlüsselt und das Ergebnis der Entschlüsselung dann mit C_0 mittels der XOR-Operation verknüpft. Das Ergebnis dieser XOR-Operation ist der Klartextblock P_i. Als nächstes wird der Chiffretextblock C_2 entschlüsselt und das Ergebnis dieser Entschlüsselung wird dann mit C_1, mittels der XOR-Operation, zu dem Klartextblock P_2, verknüpft usw.

Mathematische Beschreibung: $P_i = C_{i-1} \oplus D_K(C_i)$

10.2.3 Verwendungszweck

- **Verschlüsselung von Dateien**
  Da mit diesem Modus keine Synchronisationsfehler behoben werden können, eignet er sich besonders für die Verschlüsselung von Dateien. Dabei kommt es praktisch nie zu synchronisations Fehlern, dafür aber gelegentlich zu Bitfehler (durch defekte Hardware o.ä.). Ein Bitfehler in einem Chiffretextblock C_i betrifft den kompletten Klartextblock P_i sowie das entsprechenden Bit im nächsten Klartextblock P_{i+1}.

- **Integritätsprüfung von Nachrichten**
  Um die Integrität einer Nachricht M zu überprüfen verschlüsseln sie diese und merken sich nur die beiden Chiffretextblöcke $C_0 = IV$ und C_n. Die restlichen Chiffretextblöcke werden nicht benötigt. Jetzt sind Sie in der Lage jederzeit festzustellen ob die Nachricht manipuliert wurde, indem Sie die Nachricht M' nochmals mit dem Startwert C_0 verschlüsseln C'_n berechnen und überprüft ob $C_n = C'_n$ gilt. Ist dies der Fall, dann wurde die M nicht manipuliert. Wenn $C_n \neq C'_n$ gilt, dann wurde entweder IV, C_n oder M manipuliert. Unter manipuliert versteht man hier das zufällige Kippen eines oder mehrer Bits.

- **Authentifizierung**
  Angenommen sie teilen mit Alice, die Sie noch nie zuvor getroffen haben, ein Geheimnis (Key). Eines Tages wollen Sie sich mit Alice treffen um mit ihr vertrauliche Daten auszutauschen. Um sicher zugehen das sich es bei der Person am vereinbarten Treffpunkt wirklich um Alice handelt nehmen Sie einfach eine Nachricht M und einen Zufälligen Startwert IV mit. Bitten Sie "Alice" M mit ihrem Geheimnis und dem Startwert IV C_n zu berechnen. Wenn dieser Wert mit dem von ihnen berechneten C_n über-

einstimmen (und weder Alice noch Sie ihr Geheimnis jemand anderem verraten hat), dann handelt es sich bei der Person mit überwältigender Wahrscheinlichkeit um Alice. Wenn die Werte nicht übereinstimmen, dann handelt es sich bei dieser Person nicht um Alice.
VORSICHT: Der CBC-MAC ist nur sicher, wenn die alle ausgetauschten Nachrichten gleich lang sind. Aus diesem Grund sollte man davon Abstand nehmen den CBC-MAC zu verwendet.

10.3 Cipher-Feedback-Modus (CFB)

Paket: Crypto.Symmetric.Mode.CFB

Der CFB-Modus wandelt eine Blockchiffre in eine selbst synchronisierte Blockchiffre um. Z.B. versetzt dies ein Terminal in die Lage, jedes eingetippte Zeichen sofort zum Host zu übertragen.
Achtung: Bei diesem Modus muss der Startwert nach jeder Nachricht geändert werden. Ansonsten ist dieses Verfahren nicht sicher.

10.3.1 Verschlüsselung

Bei der Initialisierung wird der Startwert IV als C_0 zwischengespeichert. Wenn nun n-Bytes (n < Block'Size) verschlüsselt werden soll, dann werden die n-Bytes an den Anfang des Klartextblockes P_0 kopiert. Dieser wird dann mit Nullen aufgefüllt. Bei der Verschlüsselung von P_1 wird C_0 verschlüsselt und der dadurch generierte Chiffretext C_1 wird mittels der XOR-Operation mit P_1 verknüpft. Das Ergebnis dieser Operation wird ausgegeben und als C_1 zwischengespeichert. Der nächste aufgefüllte Klartextblock P_2 wird nach der Verschlüsselung zu C_2 indem C_1 verschlüsselt und mit Hilfe der XOR-Operation mit P_2 verknüpft wird. Dies wird solange wiederholt bis die gesamte Nachricht übermittelt wurde.

Mathematische Beschreibung: $C_i = P_i \oplus E_K(C_{i-1})$

10.3.2 Entschlüsselung

Die Entschlüsselung verläuft analog zur Verschlüsselung. Zu Beginn wird der Betriebsmodus mit dem Startwert IV initialisiert bzw. mittels *Set_IV* reinitialisiert und als C_0 zwischengespeichert. Bei der Entschlüsselung eines Chiffretextblockes C_i wird zunächst C_{i-1} entschlüsselt und mit Hilfe der XOR-Operation mit C_i verknüpft. Das Ergebnis dieser Operation ist P_i. Zum Schluss wird noch C_i zwischengespeichert.

Mathematische Beschreibung: $P_i = C_i \oplus E_K(C_{i-1})$

10.3.3 Verwendungszweck

Im Gegensatz zum CBC-Modus wo die Verschlüsselung erst dann beginnen kann, wenn ein vollständiger Datenblock vorliegt, können im CFB-Modus Daten z.B. auch Byteweise (8-CFB) verschlüsselt werden. Dadurch eignet sich dieses Verfahren hervorragend für die **Verschlüsselung von Byteströme (z.B. Remoteshell)**

10.3.4 Anmerkungen

Beim n-CFB-Modus

- wirkt sich ein Fehler im Klartext auf den gesamten nachfolgenden Chiffretext aus und macht bei der Entschlüsselung selbst wieder rückgängig.
- wirkt sich ein Fehler im Chiffretext C_i auf den Klartextblock P_i und die folgenden $\frac{m}{n} - 1$ Klartextblöcke aus, wobei m die Blockgröße ist.
- Ein Angreifer kann die Nachrichtenbits im letzten Chiffretextblock verändern ohne dabei entdeckt zu werden.

10.4 Couter-Modus (CTR)

Paket: Crypto.Symmetric.Mode.CTR

Beim Counter-Modus wird die die Blockchiffre in einen Schlüsselstromgenerator umgewandelt. Die Rückkopplung hängt also nicht vom Klartext ab sondern von einem Zähler, der nach jeder Verschlüsselungsoperation um eins erhöht wird.

10.4.1 Verschlüsselung

Bei der Initialisierung wird der Zähler auf den Startwert (IV) gesetzt. Bei der Verschlüsselung des Klartextblockes P_i wird $IV + i - 1$ mit Hilfe der Blockchiffre zu dem Schlüsselstromblock K_i verschlüsselt. Danach wird mit Hilfe der XOR-Operation K_i mit P_i verknüpft. Das Ergebnis dieser Verknüpfung ist der Chiffretext C_i .

Mathematische Beschreibung: $C_i = P_i \oplus E_K(IV + i - 1)$

10.4.2 Entschlüsselung

Die Entschlüsselung verläuft analog zur Verschlüsselung. Zu Beginn wird der Zähler mit dem Startwert IV initialisiert bzw. mittels *Set_IV* reinitialisiert. Bei der Entschlüsselung eines Chiffretextblockes C_i wird zunächst $IV + i - 1$ mittels der Blockchiffre zu dem Schlüsselstromblock K_i verschlüsselt. Danach wird C_i

mit Hilfe der XOR-Operation zu mit K_i verknüpft. Das Ergebnis dieser Operation ist P_i .

Mathematische Beschreibung: $P_i = C_i \oplus E_K(IV + i - 1)$

10.4.3 Verwendungszweck

- **Ent/Verschlüsselung von Nachrichten mit wahlfreiem Zugriff** Da Sie mit dem Counter-Modus in der Lage sind gezielt einzelne Chiffretextblock zu entschlüsseln eignet sich dieses Verfahren für die Verschlüsselung von Dateien mit wahlfreiem Zugriff wie z.B. Datenbanken. Hier können sie bei einer Anfrage an eine verschlüsselte Datenbank genau die Daten, die angefragt wurden.

- **Parallele Ent/Verschlüsselung**
  Eine Parallelisierung ist möglich indem man aus dem Startwert des Zählers IV und der Länge der Nachricht L folgendes Intervall berechnet: $[IV...IV + L]$ Das Intervall lässt sich in max. L disjunkte Teilintervalle zerlegen. Die Nachrichtenblöcke der Teilintervalle können parallel ver- bzw. entschlüsselt werden.

- **Phasenweise "High Speed"-Verschlüsselung**
  Die ist eine Expertenanwendung die auf der "Low-Level-API" (**??**) des CTR-Modus beruht. Verwenden sie diese nur wenn Sie genau wissen, was Sie tun.
  Beim Counter-Modus ist es möglich beliebig viele Schlüsselstrombits, ohne das ein Nachrichtenblock benötigt wird, zu generiere. Wenn Sie mit Hilfe des Counter-Modus genügend Schlüsselstrombits generieren, dann sind Sie in der Lage Nachrichten sehr schnell zu verschlüsseln, indem Sie sie einfach mit den vorher erzeugten Schlüsselstrombits XOR-Verknüpft.

10.4.4 Anmerkungen

- Ein Bit-Fehler im Klartext wirkt sich nur auf ein Bit im Chiffretext aus und umgekehrt.

- Manipulationen am Klartext sind sehr einfach, da jede Änderung des Chiffretextes beeinflusst direkt den Klartext.

- Synchronisationsfehler (Alice und Bob haben unterschiedliche Counterstände) können nicht behoben werden.

10.4.5 Low-Level-API

Die folgende API sollten Sie nur dann verwenden, wenn Sie genau wissen was Sie tun.

```
procedure Next_Block(Keystream : out Block);
```

Diese Prozedur generiert einen Schlüssselstromblock (*Keystream*) indem sie den Wert des internen Zähler Counter zuerst zu C verschlüsselt, ihn dann um Eins erhöht und zu guter letzt C ausgibt. Bei der Initialisierung wird Counter auf IV gesetzt.
Mathematische Beschreibung: $C = E_K(Counter); \quad Counter := Counter + 1$

10.5 Output-Feedback-Modus (OFB)

Paket: Crypto.Symmetric.Mode.OFB

Der OFB-Modus transformiert wie der Counter-Modus eine Blockchiffre in eine Stromchiffre. D.h. die interne Rückkopplung ist hier unabhängig vom Klartext.

10.5.1 Verschlüsselung

Bei der Initialisierung wird der interne Schlüsselstromblock K_0 auf IV gesetzt. Bei der Verschlüsselung eines Klartextblock P_i wird K_{i-1} zu K_i verschlüsselt und mit P_i XOR verknüpft. Das Ergebnis dieser Operation, ist der Chiffretextblock C_i
Mathematische Beschreibung: $C_i = P_i \oplus K_i$

10.5.2 Entschlüsselung

Die Entschlüsselung verläuft analog zur Verschlüsselung. Zu Beginn wird der Schlüsselstromblock K_0 mit dem Startwert IV initialisiert bzw. mittels *Set_IV* reinitialisiert. Bei der Entschlüsselung eines Chiffretextblock C_i wird K_{i-1} zu K_i verschlüsselt und mit C_i XOR verknüpft. Das Ergebnis dieser Operation, ist der Klartextblock P_i. Die Chiffretextblöcke müssen in der gleichen Reihenfolge in der sie geniertet wurden entschlüsselt werden.
Mathematische Beschreibung: $P_i = C_i \oplus K_i$

10.5.3 Verwendung

Diesen Modus macht eigentlich nur mit der "Low-Level-API" für Experten (**??**) Sinn. Mit Hilfe dieser können Sie einen Schlüsselstrom ohne Klartextblöcke generieren. Dadurch sind Sie in der Lage Klartextblöcke sehr schnell zu verschlüsseln. Zum Beispiel wäre es denkbar nachts Schlüsselstromblöcke zu generieren und mit Hilfe dieser dann Klartextblöcke am Tag zu verschlüsselt. Dieser Modus eignet sich daher besonders gut, wenn **phasenweise sehr schnell Klartextblöcke verschlüsseln** werden müssen.

10.5.4 Anmerkungen

- Der Schlüsselstrom wiederholt sich irgendwann. D.h. $\exists\ L : K_0 = K_L$ Wenn m die Blockgröße in Bits ist, beträgt die durchschnittliche Länge eines Zyklus $2^m - 1$ Bits
- Ein Bit-Fehler im Klartext wirkt sich nur auf ein Bit im Chiffretext aus und umgekehrt.
- Manipulationen am Klartext sind sehr einfach, da jede Änderung des Chiffretextes beeinflusst direkt den Klartext.
- Synchronisationsfehler (Alice und Bob haben unterschiedliche Counterstände) können nicht behoben werden.

10.5.5 Low-Level-API

Die folgende API sollten Sie nur dann verwenden, wenn Sie genau wissen was Sie tun.

```
procedure Next_Block(Keystream : out Block);
```

Bei der Initialisierung mittels der *Init*-Prozedur wird der Startwert (IV) als Schlüsselstromblock K_0 zwischengespeichert. Jedes mal wenn nun die *Next_Block*-Prozedur aufgerufen wird, passiert folgendes: Der Schlüsselstromblock K_i wird zu K_{i+1} verschlüsselt, zwischengespeichert und als Keystream ausgegeben.

Mathematische Beschreibung: $K_i = E_K(K_{i-1})$

10.6 Generischer Teil

```
generic
   with package C is
      new Crypto.Symmetric.Blockcipher(<>);

   with function "xor" (Left, Right  : in C.Block)
                       return C.Block is <>;

   -- Diese Funktion wird nur beim Counter-Mode benoetigt
   with function "+"   (Left : C.Block; Right : Byte)
                       return C.Block is <>;
```

10.7 Anwendungsbeispiel

```
with Crypto.Types;
with Ada.Text_IO;
with Crypto.Symmetric.Blockcipher_Tripledes;
with Crypto.Symmetric.Mode.CBC;


procedure Bsp_Modus_CBC is
   use Ada.Text_IO;
   use Crypto.Types;

  package TDES renames Crypto.Symmetric.Blockcipher_Tripledes;

  --Benutze die TDES im sicheren CBC-Modus
   package TDES_CBC is new Crypto.Symmetric.Mode.CBC(TDES);

  use TDES_CBC;

  -- Schluessel
  Key : B_Block192 :=
              (16#00#, 16#00#, 16#00#, 16#00#, 16#00#, 16#00#,
               16#00#, 16#00#, 16#00#, 16#00#, 16#00#, 16#00#,
               16#00#, 16#00#, 16#00#, 16#00#, 16#01#, 16#23#,
               16#45#, 16#67#, 16#89#, 16#ab#, 16#cd#, 16#ef#);

   --Startwert
  IV : B_Block64 := (16#12#, 16#34#, 16#56#, 16#78#,
                     16#90#, 16#ab#, 16#cd#, 16#ef#);

   -- Klartext
   P_String : String :="Now is the time for all .";

   --Klartext wird in drei 64-Bit Bloecke unterteilt
   P : array (1..3) of B_Block64 :=
     ((To_Bytes(P_String(1..8))),
      (To_Bytes(P_String(9..16))),
      (To_Bytes(P_String(17..24))));

   -- Chiffretrext
   C : array (0..3) of B_Block64;
  begin
    --1. Initialisierung
    Init(Key, IV);
```

```
      -- 1a) Chiffreblock = Startwert.
      C(0) := IV;

    -- 2. Verschluesselung
    for I in P'Range loop
       Encrypt(P(I), C(I));
    end loop;

    -- Fuer die Entschluesselung wird die Chiffre mit dem
    -- gleichen Startwert wie bei der Entschluesselung reinitalisiert
      Set_IV(C(0));

    -- 3. Entschluesselung
    for I in P'Range loop
       Decrypt(C(I), P(I));
       Put(To_String(P(I)));
    end loop;
end Bsp_Modus_CBC;
```

Kapitel 11

Crypto.Symmmetric.Mode.Oneway

Dieses generische Paket betreibt eine Einwegblockchiffre in einen bestimmten Einweg-(Betriebs-)modus. Dieser verknüpft gewöhnlich eine Einweg-Blockchiffre mit einer Rückkopplung und einigen einfachen Operationen (+, xor). Ein Einweg-Modus wird mit Hilfe eines zufälligen öffentlichen Startwert (Initial Value (IV)) initialisiert. Der Chiffretext ist damit nicht nur von der verwendeten Chiffre, Klartext und Schlüssel abhängig, sondern auch von dem zufälligen Startwert. Wenn Sie nun einen Klartext mehrmals mit der gleichen Chiffre und dem gleichen Schlüssel aber unterschiedlichen IVs verschlüsselt, erhalten Sie unterschiedliche Chiffretext und durch die Rückkopplung in einem Einweg-Modus werden gleiche Klartextblöcke zu unterschiedlichen Chiffretextblöcken chiffriert, d.h ein Einweg-Betriebsmodus verschlüsselt zwei Klartextblöcke p1 und p2 mit p1=p2, mit überwältigender Wahrscheinlichkeit, zu zwei Chiffretextblöcke c1 und c2 mit c1≠c2. Damit ist es nun möglich mehrere Nachrichten mit dem selben Schlüssel zu verschlüsseln.
Vorsicht: Um einen Chiffretext zu entschlüsseln benötigen Sie den gleichen Schlüssel und Startwert wie bei der Verschlüsselung. Aus diesem Grund sollte der Startwert immer mit dem zugehörigen Chiffretext aufbewahrt werden. **Die Sicherheit eines Modus ist unabhängig vom „Bekanntheitsgrad“des Startwertes**. Daher Multipliziert man den Startwert meist mit dem Chiffretext zu dem endgültigen Chiffrat indem man den Startwert vor dem Chiffretext hängt ($IV * C = C' = IV||C$).

11.1 Anmerkungen

- Für die Einweg-Modus gilt das gleiche wie für einen normalen Modus. Falls ein normaler Modus auch als Einweg-Modus zur Verfügung steht, dann sollte Sie diesen dem normalen Modus vorziehen, da dieser etwas schlanker ist.

- Die **API** ist identisch zu den normalen Modi [**??**]
- Unterstütze Modi
  - Cipher-Feedback-Modus (CFB) [**??**]
  - Counter-Modus (CTR) [**??**]
  - Output-Feedback-Modus (OFB) [**??**]

11.2 Anwendungsbeispiel

```
with Crypto.Types, Ada.Text_IO, Crypto.Symmetric.Mode.Oneway_CTR;
with Crypto.Symmetric.Oneway_Blockcipher_Twofish128;

procedure Bsp_Oneway_Modus_CTR
   package TF128 renames Crypto.Symmetric.Oneway_Blockcipher_Twofish128;

   -- Benutze die Blockchiffre in einem sicheren Modus
   package Twofish128 is new Crypto.Symmetric.Mode.Oneway_CTR(TF128);

   use Ada.Text_IO, Crypto.Types, Twofish128;

   --Schluessel
   Key : B_Block128:=(16#2b#, 16#7e#, 16#15#, 16#16#, 16#28#, 16#Ae#,
                      16#D2#, 16#A6#, 16#Ab#, 16#F7#, 16#15#, 16#88#,
                      16#09#, 16#Cf#, 16#4f#, 16#3c#);

   --Startwert
   IV : B_Block128 := (15 => 1, others => 0);

   --Nachricht
   Message : String :="All your Base are belong to us! ";

   -- Nachrichtenbloecke
   P : array(1..2) of B_Block128 :=
     (To_Bytes(Message(1..16)), To_Bytes(Message(17..32)));

   --Chiffretextbloecke
   C : array(0..2) of B_Block128;

begin
   --Initialisierung
   Init(Key, IV);
```

```
   -- 1. Chiffreblock = Startwert.
   C(0) := IV;

    -- Verschluesselung
   for I in P'Range loop
      Encrypt(P(I), C(I));
   end loop;

   -- Fuer die Entschluesselung wird die Chiffre mit dem
   -- gleichen Startwert wie bei der Entschluesselung reinitalisiert
   Set_IV(C(0));

   -- Entschluesselung
   for I in P'Range loop
      Decrypt(C(I), P(I));
      Put(To_String(P(I)));
   end loop;

end Bsp_Oneway_Modus_CTR;
```

Kapitel 12

Crypto.Hashfunction

Mit Hilfe dieses generischen Paketes lässt sich aus dem Algorithmus einer kryptographsichen Hashfunktion (siehe Kapitel **??**) eine krypto. Hashfunktion erstellen die sich hervorragend für folgende Zwecke einsetzen:

- Integritätsüberprüfung von Nachrichten
- Generierung und Verifizierung von digitalen Signaturen.
- Generierung von Zufallszahlen bzw. Zufallsbits

Der Zweck dieses Paketes ist es die API für Hashfunktionen zu vereintheilichen und zu vereinfachen. Aus Sicht der Sicherheit spricht hier nichst dagegen direkt die native API aus `Crypto.Symmetric.Algorithm` zu verwenden.

12.1 Generischer Teil

```
generic
   type Hash_Type                  is private;
   type Message_Type               is private;
   type Message_Block_Length_Type  is private;

   with procedure Init(Hash_Value : out Hash_Type) is <>;

   with procedure Round(Message_Block : in     Message_Type;
                        Hash_Value    : in out Hash_Type) is <>;

   with function Final_Round(Last_Message_Block   : Message_Type;
                             Last_Message_Length  :
                             Message_Block_Length_Type;
                             Hash_Value           : Hash_Type)
```

```
                                return Hash_Type is <>;

   with procedure Hash(Message     : in Bytes;
                       Hash_Value  : out  Hash_Type) is <>;

   with procedure Hash(Message     : in String;
                       Hash_Value  : out  Hash_Type) is <>;

   with procedure F_Hash(Filename  : in String;
                         Hash_Value : out  Hash_Type) is <>;
```

12.2 API

Die API einer generischen Hashfunktion besteht aus einer High- und einer Low-Level-API. Die Low-Level-API sollen sie nur verwenden, wenn sie sich mit krptographischen Hashfunktionen auskennen. Wenn dies nicht der Fall ist, dann verwenden sie bitte die folgende High-Level-API.

12.2.1 High-Level-API

```
   function Hash   (Message  : Bytes)  return Hash_Type;
   function Hash   (Message  : String) return Hash_Type;

   function F_Hash(Filename : String) return Hash_Type;
```

Die Funktion **Hash** liefert den Hashwert einer Nachricht (*Message*). Bei der Nachricht kann es sich dabei entweder um ein Byte-Array oder einen String handeln. Die Funktion **F_Hash** gibt den Hashwert der Datei *Filename* zurück. Beispielsweise liefert die Codezeile

```
H := F_Hash("/bin/ls")
```

den Hashwert von `/bin/ls`.

12.2.2 Low-Level-API

Die Low-Level-API besteht aus einer Funktion und zwei Prozeduren.

- Die Prozedur **Init** initialisiert bzw. reinitailisert die Hashfunktion. Jedesmal wenn eine Nachricht gehashed werde soll muss zunächst diese Prozedure aufgerufen werden.

  ```
  procedure Init;
  ```

- Mit der Prozedure **Round**, können iterativ Nachrichtenblöcke gehashed werden.

```
procedure Round(Message_Block : in Message_Type);
```

- Die Funktion **Final_Round** padded und hashed anschließend einen Nachrichtenblock *Last_Message_Block*. Auf Grund des Paddings muss die Bytelänge des Nachrichtenmaterials *Last_Message_Length* angegeben werden. Denn eine Nachricht ist i.d.R. kürzer als eine Nachrichtenblock vom Typ *Message_Type*. Der Rückgabewert dieser Funktion entspricht dem finalen Hashwert einer Nachricht.

```
function Final_Round(Last_Message_Block   : Message_Type;
                     Last_Message_Length  : Message_Block_Length_Type)
                     return Hash_Type;
```

12.3 Anwendungsbeispiel

12.3.1 High-Level-API

Das folgende Beispiel gibt den SHA-256 Hashwert von **/bin/ls** aus.

```
with Ada.Text_IO;
with Crypto.Types,
with Crypto.Hashfunction_SHA256;


use Crypto.Types, Ada.Text_IO;

procedure example is
   package SHA256 renames Crypto.Hashfunction_SHA256;

   Hash : W_Block256 := SHA256.F_Hash("/bin/ls");
begin
   for I in Hash'Range loop
      Put(To_Hex(Hash(I)));
   end loop;
   Put_Line("  /bin/ls");
end example;
```

12.3.2 Low-Level-API

```
with Ada.Text_IO;
with Crypto.Types;
with Crypto.Hashfunction;
with Crypto.Symmetric.Algorithm.SHA256;

use Ada.Text_IO;
use Crypto.Types;
use Crypto.Symmetric.Algorithm.SHA256;

pragma Elaborate_All (Crypto.Hashfunction);

procedure Example_Hashing is
   package WIO is new  Ada.Text_IO.Modular_IO(Word);

    package SHA256 is
      new Crypto.Hashfunction(Hash_Type      => W_Block256,
                              Message_Type => W_Block512,
                              Message_Block_Length_Type =>
                              Crypto.Types.Message_Block_Length512);

   Message : String :=  "All your base are belong to us!";

   W : Words := To_Words(To_Bytes(Message));

   M : W_Block512 := (others => 0);
   H : W_Block256;
begin
   M(W'Range) := W;

   SHA256.Init;

   -- Berechne den Hashwert von Message
   H := SHA256.Final_Round(M, Message'Last);

   -- Gib den Hashwert von Message aus
   for I in W_Block256'Range loop
      WIO.Put(H(I), Base => 16);
      New_line;
   end loop;
   New_Line;
end Example_Hashing;
```

12.4 Anmerkung

Sie müssen nicht jedes mal von neuem aus einem dafür vorgesehenen symmetrischen Algorithmus eine Hashfunktion generieren. Stattdessen können Sie auch eine der vorgefertigten Hashfunktionen verwenden.

- Crypto.Hashfunktion_SHA1
- Crypto.Hashfunktion_SHA256
- Crypto.Hashfunktion_SHA512
- Crypto.Hashfunktion_Whirlpool

Kapitel 13

Crypto.Symmetric.MAC

Ein Message Authentication Code (MAC) dient zur Sicherung der Integrität und Authentizität einer Nachricht. Er gewährt keine Verbindlichkeit, da er nur einen symmetrischen Schlüssel verwendet. Bei den asymmertrischen Verfahren spricht man von digitalen Signaturen. Bei MACs von „authentication Tags “. Der Autor dieser Bibliothek bezeichnet diese meist als (digitiale) Stempel.

13.1 Randomized MAC (RMAC)

Der RMAC wurde von Eliane Jaulmes, Antoine Joux and Frederic Valette entwickelt. Er basiert auf dem CBC-MAC und basiert daher auf einer Einweg-Blockchiffre (Kapitel **??**) . Er ist beweisbar sicher gegenüber „Geburtstagsparadoxon-Angriffe “. Er benötigt zwei Schlüssel. Der digitale Stempel ist doppelt so lange wie die verwendete Blockchiffre.

Mathematische Beschreibung:

$\mathrm{RMAC}_{K_1,K_2}(E,M) = E_{K2\oplus R}(C_n)$ mit $C_i = E_{K_1}(M_i \oplus C_{i-1}), \quad R \in_R \{0,1\}^{|K_1|}$ und $C_0 = 0$

13.1.1 Generischer Teil

```
generic
   with package C is
      new Crypto.Symmetric.Oneway_Blockcipher(<>);

   with procedure Read (Random : out C.Key_Type) is <>;
   with function "xor" (Left, Right : C.Block)
                       return C.Block is <>;
   with function "xor" (Left, Right : C.Key_type)
                       return C.Key_Type is <>;
```

13.1.2 API

High-Level-API

```
type Blocks  is array (Integer range <>) of Block;

procedure Sign(Message : in Blocks;
               Key1, Key2 : in Key_Type;
               R    : out Key_Type;
               Tag : out Block);


function Verify(Message : in Blocks;
                Key1, Key2 : in Key_Type;
                R    : in Key_Type;
                Tag : in Block) return Boolean;
```

Die Funktion **Sign** signiert eine Nachricht **Message** unter den beiden Schlüssel **Key1** und **Key2**. Sie liefert einen digitiale Stempel $S = Tag||R$. Wobei es sich bei **R** um eine zufällig generierte Zahl handelt.

Die Funktion **Verify** liefert *true* zurück, wenn $S = Tag||R$ ein gültiger Stempel ist ansonsten *false* zurück. D.h. wenn einer oder Mehrere Parameter nicht mit denen aus der Prozedur **Sign** übereinstimmt, dann liefert die Funktion mit überwältigender Wahrscheinlichkeit *false* zurück.

Low-Level-API

```
procedure Init(Key1, Key2 : in Key_Type);

procedure Sign(Message_Block : in Block);
procedure Final_Sign(Final_Message_Block : in Block;
                     R    : out Key_Type;
                     Tag : out Block);

procedure Verify(Message_Block : in Block);
function Final_Verify(Final_Message_Block  : in Block;
                      R    : in Key_Type;
                      Tag : in Block)
                     return Boolean;
```

- Die Prozedur **Init** initalisiert einen RMAC mit den beiden Schlüsseln **Key1** und **Key2** und einem internen Anfangszustand.
- Die Prozedur **Sign** „hashed “einen Nachrichtenblock (**Message_Block**).
- Die Prozedur **Final_Sign** hashed den finalen Nachrichtenblock **Final_Message_Block**, generiert eine Zufallszahl **R** berechnet **Tag** und setzt den RMAC auf seinen Anfangszustand zurück.
- Die Prozedur **Verify** „hashed “einen Nachrichtenblock (**Message_Block**).
- Die Funktion **Final_Verify** hashed den finalen Nachrichtenblock **Final_Message_Block** und verifiziert mit Hilfe eines internen Zustandes und **R**, ob es sich bei **Tag** bzw. **R**||**Tag** um einen gültigen Stempel handelt. Falls ja gibt sie *true*, ansonsten *false* zurück.

13.1.3 Anwendungsbeispiel

```
with Crypto.Types; use Crypto.Types;
with Ada.Text_IO;  use Ada.Text_IO;
with Crypto.Symmetric.MAC.RMAC;
with Crypto.Symmetric.Oneway_Blockcipher_AES128;

pragma Elaborate_All (Crypto.Symmetric.MAC.RMAC);

procedure Example is
  package AES128 renames Crypto.Symmetric.Oneway_Blockcipher_AES128;
  package RMAC is new Crypto.Symmetric.MAC.RMAC(AES128);
  use RMAC;

  Key1 : B_Block128 :=
         (
          16#00#, 16#01#, 16#02#, 16#03#, 16#04#, 16#05#,
          16#06#, 16#07#, 16#08#, 16#09#, 16#0a#, 16#0b#,
          16#0c#, 16#0d#, 16#0e#, 16#0f#
         );

  Key2 : B_Block128 :=
         (
          16#00#, 16#11#, 16#22#, 16#33#, 16#44#, 16#55#, 16#66#,
          16#77#, 16#88#, 16#99#, 16#aa#, 16#bb#, 16#cc#, 16#dd#,
          16#ee#, 16#ff#
         );
```

```
      R      : B_Block128;
      Tag    : B_Block128;

      M : B_Block256 := To_Bytes("ALL_YOUR_BASE_ARE_BELONG_TO_US!_");
      Message : RMAC.Blocks(0..1) := (0 => M(0..15), 1 => M(16..31));
begin
  Init(K1,K2);

  Sign(Message(0));
  Final_Sign(Message(1), R, Tag) ;

  Verify(Message(0));
  Put_Line(Final_Verify(Message(1), R, Tag));
end Example;
```

13.2 Hashfunction MAC (HMAC)

Bei Mihir Bellare, Ran Canettiy und Hugo Krawczykz handelt es sich um die HMAC Designer. Der HMAC basiert auf einer Hashfunktion.

Mathematische Beschreibung:

$\text{HMAC}_K(H, M) = H(K \oplus opad, H(K \oplus ipad || M))$ mit $opad = \{0x5C\}^n$ und $ipad = \{0x36\}^n$

13.2.1 Generischer Teil

```
with package H is new Crypto.Hashfunction(<>);

 with function "xor"
   (Left, Right : H.Message_Type)  return H.Message_Type is <>;
 with procedure Fill36 (Ipad : out  H.Message_Type) is <>;
 with procedure Fill5C (Opad : out  H.Message_Type) is <>;

 with procedure Copy
   (Source : in H.Hash_Type; Dest : out H.Message_Type) is <>;

 with function M_B_Length
    (Z : H.Hash_Type) return H.Message_Block_Length_Type is <>;
```

- Die beiden Prozeduren **Fill36** und **Fill5C** füllen einen *Message_Type* **Ipad** bzw. **Opad** mit dem Wert `0x36` bzw. `0x5C` auf.

- Die Prozedur **Copy** kopiert den Inhalt von **Soucre** nach **Destination**. Ist der *Message_Type* „kleiner “als der *Hash_Type* der Hashfunktion, dann wird der Rest mit Nullen aufgefüllt.
- Die Funktion **M_B_Length** gibt die Länge in Bytes von von **Z** bzw. dem *Hash_Typ* zurück.

All diese Hilfsfmethoden sind in *Crypto.Symmetric.MAC* definiert.

13.2.2 API

```
procedure Init(Key : in Message_Type);

procedure Sign(Message_Block : in Message_Type);

procedure Final_Sign
  (Final_Message_Block         : in Message_Type;
   Final_Message_Block_Length  : in Message_Block_Length_Type;
   Tag                         : out Hash_Type);


procedure Verify(Message_Block : in Message_Type);

function Final_Verify
  (Final_Message_Block         : Message_Type;
   Final_Message_Block_Length  : Message_Block_Length_Type;
   Tag                         : Hash_Type) return Boolean;
```

- Die Prozedur **Init** initialisiert den HMAC mit dem Schlüssel **Key** und einem internen Anfangszustand.
- Die Prozedur **Sign** hashed einen Nachrichtenblock.
-
- Die Prozedur **Final_Sign** hashed einen Nachrichtenblock der Bytelänge **Final_Message_Block_Length**, gibt den digitalen Stempel **Tag** zurück und setzt den HMAC auf seine Anfangszustand zurück.
- Die Prozedur **Verify** hashed einen Nachrichtenblock.

- Die Funktion **Final_Verify** hashed einen Nachrichtenblock der Bytelänge **Final_Message_Block_Length** und verifiziert mit Hilfe eines internen Zustandes ob es sich bei **Tag** um einen gültigen Stempel handelt. Falls ja gibt sie *true*, ansonsten *false* zurück.

13.2.3 Anwendungsbeispiel

```
with Ada.Text_IO;
with Crypt.Types;
with Crypto.Symmetric.MAC;
with Crypto.Symmetric.MAC.HMAC;
with Crypto.Hashfunction_SHA256;

use  Ada.Text_IO;
use  Crypt.Types;

pragma Elaborate_All (Crypto.Symmetric.MAC.HMAC);


procedure Example is

   package RMAC is new Crypto.Symmetric.MAC.HMAC
             (H            => Crypto.Hashfunction_SHA256,
              Copy         => Crypto.Symmetric.Mac.Copy,
              Fill36       => Crypto.Symmetric.Mac.Fill36,
              Fill5C       => Crypto.Symmetric.Mac.Fill5C,
              M_B_Length => Crypto.Symmetric.Mac.M_B_Length);

   use RMAC;

   -- 160-Bit Key
   Key1 : W_Block512 := (0 => 16#0b_0b_0b_0b#, 1 => 16#0b_0b_0b_0b#,
                         2 => 16#0b_0b_0b_0b#, 3 => 16#0b_0b_0b_0b#,
                         4 => 16#0b_0b_0b_0b#, others  => 0);

    -- "Hi There" Len = 8 Byte
    Message : W_Block512 := (0 => 16#48_69_20_54#, 1 => 16#68_65_72_65#,
                             others => 0);

    Tag : W_Block256;
```

```
begin
   Init(Key1);
   Final_Sign(Message, 8, Tag);

   Put_Line(Final_Verify(Message1, 8, Tag));
end Example;
```

13.2.4 Anmerkungen

Sie müssen nicht jedes mal von neuem aus einer Hashfunktion einen HMAC generieren. Stattdessen können Sie auch eine der vorgefertigten HMACs verwenden.

- Crypto.Symmetric.MAC.HMAC_SHA1
- Crypto.Symmetric.MAC.HMAC_SHA256
- Crypto.Symmetric.MAC.HMAC_SHA512
- Crypto.Symmetric.MAC.HMAC_Whirlpool

Kapitel 14

Crypto.Types.Big_Numbers

Dieses Paket stellt den generischen Typen *Big_Unsigned*, mit einem ganzen Satz von Prozeduren und Funktionen, zur Verfügung. Dieser Typ verwendet intern ein Array das aus k CPU-Wörtern besteht. Dieses Array wird als ein modularer Typ interpretiert. Aus diesem Grund es ist nur möglich Big_Unsigneds deren Bitlänge einem Vielfachem der CPU-Wortlänge entspricht zu generieren. Das dieses Paket ohne Zeiger und Inline-Assembler arbeitet, ist es um ein vielfaches langsamer als z.B. die auf Effizienz optimierte MPI des GnuPG[**?**].
Dieses Paket basiert auf Jerome Delcourts Big_Number-Bibliothek[**?**]. Ursprünglich sollte diese Bibliothek an dieser Stelle verwendet werden. Es stellte sich aber heraus, das diese doch nicht nicht den gewünschten Anforderungen entsprach. Aus diesem Grund wurde dieses Paket noch mal komplett neu geschrieben. Einen wesentlichen Beitrag zu dem derzeitigen Code trugen die Analyse des Quellcodes von java.math.BigInteger[**?**] und Bob Debliers beecyrpt[**?**] sowie folgende Quellen [**?**, **?**, **?**, **?**, **?**] bei.

Notation

- $|X|\hat{=}$: Bitlänge von X.
- $|CPU|$: Länge eines CPU-Wortes.
  (Bei einem n-Bit Prozessor gilt i.d.R. $|CPU| = n$)

14.1 API

14.1.1 Generischer Teil

```
generic
   Size : Positive;
```

Vorbedingung: $Size = k \cdot |CPU| \quad k \in N$

Exception: $Size \neq k \cdot |CPU| \quad k \in N$: Constraint_Size_Error.

Size gibt die Bitlänge des Types *Big_Unsigned* an.

14.1.2 Typen

```
type Big_Unsigned is private;
```

Dies ist dar Basis-Typ des Paketes. Big_Unsigned repräsentiert eine modulare n-Bit Zahl. ($n = k \cdot m$ wobei m die Länge eines CPU-Wortes ist). Eine Variable von diesem Typ wir immer mit der Konstanten Big_Unsigned_Zero (**??**) initialisiert.

```
subtype Number_Base is Integer range 2 .. 16;
```

Dieser Typ wird später bei der Konvertierung einer Big_Unsigned in einen String benötigt. Ein Big_Unsigned Variable kann nur als eine Zahl zu einer Basis dieses Types dargestellt werden.

```
type Mod_Type is mod 2**System.Word_Size;
 for Mod_Type'Size use System.Word_Size;
```

Mod_Type ist ein Modulare Typ der die Länge eines CPU-Wortes hat. Bei einem 24-Bit Prozessor ist System.Word_Size = 24 bei einem 32-Bit Prozessor ist System.Word_Size = 32 usw.

14.1.3 Konstanten

```
Big_Unsigned_Zero     : constant Big_Unsigned; -- = 0
Big_Unsigned_One      : constant Big_Unsigned; -- = 1
Big_Unsigned_Two      : constant Big_Unsigned; -- = 2
Big_Unsigned_Three    : constant Big_Unsigned; -- = 3
Big_Unsigned_Four     : constant Big_Unsigned; -- = 4
Big_Unsigned_Sixteen  : constant Big_Unsigned; -- = 16
Big_Unsigned_First    : constant Big_Unsigned; -- = 0
Big_Unsigned_Last     : constant Big_Unsigned; -- = "Big_Unsigned'Last"
```

14.2 Vergleichsoperationen

```
-- Vergleiche Big_Unsigned mit Big_Unsigned

function "="(Left, Right : Big_Unsigned) return Boolean;
function "<"(Left, Right : Big_Unsigned) return Boolean;
function ">"(Left, Right : Big_Unsigned) return Boolean;

function "<="(Left, Right : Big_Unsigned) return Boolean;
function ">="(Left, Right : Big_Unsigned) return Boolean;

function Min(X, Y : in Big_Unsigned) return Big_Unsigned;
function Max(X, Y : in Big_Unsigned) return Big_Unsigned;


-- Vergleiche Big_Unsigned mit Mod_Type

function "=" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return  Boolean;

function "=" (Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Boolean;

function "<" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return Boolean;

function "<" (Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Boolean;

function ">" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return  Boolean;

function ">" (Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Boolean;
```

```
function "<=" (Left  : Big_Unsigned;
               Right : Mod_Type)
               return  Boolean;

function "<="(Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Boolean;

function ">=" (Left  : Big_Unsigned;
               Right : Mod_Type)
               return  Boolean;

function ">=" (Left  : Mod_Type;
               Right : Big_Unsigned)
               return  Boolean;
```

14.2.1 Elementare Operationen

```
function "+" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "+" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return  Big_Unsigned;

function "+" (Left :  Mod_Type;
              Right : Big_Unsigned)
              return  Big_Unsigned;


function "-" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "-" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return  Big_Unsigned;

function "-" (Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Big_Unsigned;
```

```
function "*" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "*" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return  Big_Unsigned;

function "*" (Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Big_Unsigned;


function "/" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "/" (Left  : Big_Unsigned;
              Right : Mod_Type)
              return  Big_Unsigned;

function "/" (Left  : Mod_Type;
              Right : Big_Unsigned)
              return  Big_Unsigned;


function "xor" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "and" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "or"  (Left, Right : Big_Unsigned) return Big_Unsigned;

function "**"  (Left, Right : Big_Unsigned) return Big_Unsigned;

function "mod" (Left, Right : Big_Unsigned) return Big_Unsigned;

function "mod" (Left  : Big_Unsigned;
                Right : Mod_Type)
                return  Big_Unsigned;
```

Multiplikation

Zusätzlich zu der elementaren Multiplikation wurden noch weitere Algorithmen implementiert, die ihr Potential bei größeren Zahlen entfalten. Leider lassen die theoretischen Laufzeiten nicht in der ACL erreichen. Je komplexer die Algorithmen werden, desto mehr Rechenoperationen und Variablen werden benötigt. An

dieser Stelle lässt sich der Flaschenhals erkennen, welcher in der Initialisierung von *Big_Unsigned* steckt.

Weitere Multiplikationsalgorihmen sind:

```
function Russ      (Left, Right : Big_Unsigned)
                   return Big_Unsigned;
```

Die Russische Bauernmultiplikation $O(N^2)$ basierend auf Bitoperationen.

```
function Karatsuba (Left, Right : Big_Unsigned)
                   return Big_Unsigned;
```

Der Karatsuba $\left(O\left(N^{log_2 3}\right) = O\left(N^{1.585}\right)\right)$ Algorithmus teilt die Faktoren intern in Polynome 1. Grades und brechnet die Teilprodukte mit der Schulmethode.

```
function Karatsuba_P (Left, Right : Big_Unsigned)
                     return Big_Unsigned;
```

Die Berechnung der Teilprodukte wird parallel in Tasksausgeführt.

```
function Toom_Cook (Left, Right : Big_Unsigned)
                   return Big_Unsigned;
```

Die hier Implementierte variante ist der Toom-Cook-3-Way $\left(O\left(N^{1,465}\right)\right)$ Algorithmus nach D. Knuth. Er teilt die Faktoren intern in Polynome 2. Grades und brechnet die Teilprodukte mit der Schulmethode.

```
function Toom_Cook_P (Left, Right : Big_Unsigned)
                     return Big_Unsigned;
```

Die Berechnung der Teilprodukte wird parallel in Tasksausgeführt.

Alle Algorithmen lassen sich explizit Aufrufen und zur Multiplikation zweier *Big_Unsigned* benutzen.

Eine Performance-Steigerung lässt sich dennoch durch die Verzahnung der Algorithmen erreichen, welche versucht die Stärken der verschiedenen Algorithmen zu kombinieren. Intern ruft dazu der überladene Operator " $*$ " für kleine Faktoren die Schulmethode, für größere (ca. 3100 Bit) den parallelen Karatsuba und ab ca. 3900 Bit Länge den parallelen Toom-Cook-Algorithmus auf.

14.3 Utils

In dem separatem Body Crypto.Types.Big_Numbers.Utils verbergen sich sehr viele nützliche Funktionen und Prozeduren. Der Zugriff erfolgt über das Präfix **Utils.**

```
procedure Swap(X, Y : in out Big_Unsigned);
```

Diese Prozedur vertauscht *X* mit *Y*.

```
procedure Set_Least_Significant_Bit(X : in out Big_Unsigned);
```

Diese Prozedur setzt das niederwertigste Bit von *X* auf 1.
Dadurch ist *X* nach diesem Prozeduraufruf immer ungerade.

```
procedure Set_Most_Significant_Bit(X : in out Big_Unsigned);
```

Diese Prozedur setzt das höchstwertigste Bit von *X* auf 1.
Damit ist *X* nach dem Prozeduraufruf eine Size-Bit Zahl.

```
function Is_Odd(X : Big_Unsigned) return Boolean;
```

Diese Funktion liefert *True* zurück, wenn *X* ungerade ist, ansonsten *False*.

```
function Is_Even(X : Big_Unsigned) return Boolean;
```

Diese Funktion liefert *True* zurück, wenn X gerade ist, ansonsten *False*.

```
procedure Inc(X : in out Big_Unsigned);
```

Diese Prozedur erhöht X um 1.

```
procedure Dec(X : in out Big_Unsigned);
```

Diese Prozedur vermindert X um 1.

```
function Shift_Left (Value  : Big_Unsigned;
                     Amount : Natural)
                     return   Big_Unsigned;
```

Diese Funktion berechnet $Value * 2^{Amount}$.

```
function Shift_Right (Value  : Big_Unsigned;
                      Amount : Natural)
                      return    Big_Unsigned;
```

Diese Funktion berechnet $\lfloor Value/2^{Amount} \rfloor$.

```
function Rotate_Left(Value  : Big_Unsigned;
                     Amount : Natural)
                     return   Big_Unsigned;
```

Diese Funktion berechnet $((Value * 2^{Amount}) \oplus (\lfloor Value/2^{|Value|-Amount} \rfloor))$ mod $2^{|Value|+1}$.

```
function Rotate_Right(Value  : Big_Unsigned;
                      Amount : Natural)
                      return   Big_Unsigned;
```

Diese Funktion berechnet $((Value * 2^{|Value|-Amount}) \oplus (\lfloor Value/2^{Amount} \rfloor))$ mod $2^{|Value|+1}$.

```
function Get_Random return Big_Unsigned;
```

Diese Funktion generierte eine Zufallszahl aus dem Intervall {0...,Big_Unsigned_Last}.

```
function Bit_Length(X : Big_Unsigned) return Natural;
```

Diese Funktion berechnet die Bitlänge von X.

```
function Lowest_Set_Bit(X : Big_Unsigned) return Natural;
```

Diese Funktion berechnet die Position des niederwertigsten Bits von X, das den Wert "1" hat.

Exception:
X = Big_Unsigned_Zero : Is_Zero_Error

```
function Gcd(Left, Right : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet den größten gemeinsamen Teiler von *Left* und *Right*.

```
function Length_In_Bytes(X : Big_Unsigned) return Natural;
```

Diese Funktion berechnet die Anzahl der Bytes die man benötigt um X als Byte-Array auszugeben.

```
function To_Big_Unsigned(X : Bytes) return Big_Unsigned;
```

Diese Funktion wandelt ein Byte-Array X in eine Big_Unsigned B um. Dabei wird X(X'First) zum höchstwertigstem Byte von B und X(X'Last) zum niederwertigstem Byte von B.

Exception:
$X'Length * Byte'Size > Size$: Constraint_Error

```
function To_Bytes(X : Big_Unsigned) return Bytes;
```

Diese Funktion wandelt X in ein Byte-Array B um. Dabei wird das höchstwertigste Byte von X zu B(B'First), und das niederwertigste Byte von X wird zu B(B'Last).

```
function To_String (Item : Big_Unsigned;
                    Base : Number_Base := 10)
                    return String;
```

Diese Funktion wandelt *Item* in einen String um. Die Umwandlung erfolgt dabei zur Basis *Base* ($Base \in \{2, \ldots, 16\}$) Der String hat folgenden Aufbau:

Base=10: Einen Zahl zur Basis 10 (Bsp. "1325553")

Base/=10: *Base#* eine Zahl zur Basis B# (Bsp. "12#AB45623A3402#")

```
function To_Big_Unsigned(S : String) return Big_Unsigned;
```

Dies Funktion wandelt eine Zeichenkette S in eine Big_Unsigned um. Die übergebene Zeichenkette wird als Zahl in dezimaler Notation oder als eine Zahl zur Basis B ($B \in \{2, ..., 16\}$) betrachtet und muss folgenden Aufbau haben:

- B#eine Zahl zur Basis B# (Bsp. S = "16#FF340A12B1#")
- eine Zahl zur Basis 10 (Bsp. S = "333665")

Exception:

S ist eine leere Zeichenkette	: Conversion_Error
S hat eine ungültige Basis	: Conversion_Error
S hat ungültige Ziffern	: Conversion_Error

```
procedure Put (Item : in Big_Unsigned;
               Base : in Number_Base := 10);
```

Diese Prozedur gibt die Big_Unsigned *Item* auf der Standardausgabe zur Basis *Base* ($Base \in \{2, ..., 16\}$) aus.

```
procedure Put_Line(Item : in Big_Unsigned;
                   Base : in Number_Base := 10);
```

Diese Prozedur gibt die Big_Unsigned *Item* auf der Standardausgabe, inklusive Zeilenumbruch, zur Basis *Base* ($Base \in \{2, ..., 16\}$) aus.

```
procedure Big_Div (Dividend, Divisor : in Big_Unsigned;
                   Quotient, Remainder : out Big_Unsigned);
```

Diese Prozedur berechnet den Quotienten *Quotient* und den Rest *Remainder* einer ganzzahligen Division. Es gilt:

- $Quotient := \lfloor \frac{Dividend}{Divisor} \rfloor$
- $Remainder := Dividend \mod Divisor$

Exception:
X = Big_Unsigned_Zero : Is_Zero_Error

```
procedure Short_Div (Dividend  : in  Big_Unsigned;
                     Divisor   : in  Mod_Type;
                     Quotient  : out Big_Unsigned;
                     Remainder : out Mod_Type);
```

Diese Prozedur berechnet den Quotienten *Quotient* und den Rest *Remainder* einer ganzzahligen Division. Es gilt:

- $Quotient := \lfloor \frac{Dividend}{Divisor} \rfloor$
- $Remainder := Dividend \mod Divisor$

Exception:
X = Big_Unsigned_Zero : Is_Zero_Error

14.4 Mod_Utils

In dem separatem Body Crypto.Types.Big_Numbers.Mod_Utils befinden sich Funktionen und Prozeduren die man häufig in der Public-Key-Kryptographie benötigt. Der Zugriff erfolgt über das Präfix **Mod_Utils.**

```
function Add (Left, Right, N : Big_Unsigned)
                  return Big_Unsigned;
```

Diese Funktion berechnet $Left + Right \pmod N$.

```
function Sub (Left, Right, N : Big_Unsigned)
                  return Big_Unsigned;
```

Diese Funktion berechnet $Left - Right \pmod N$.

```
function Div (Left, Right, N : Big_Unsigned)
                  return Big_Unsigned;
```

Diese Funktion berechnet $Left/Right \pmod N$.

Exception:
$Right$ = Big_Unsigned_Zero : Constraint_Error.

```
function Mult (Left, Right, N : Big_Unsigned)
                   return Big_Unsigned;
```

Diese Funktion berechnet $Left \cdot Right \pmod N$.

```
function Pow (Base, Exponent, N : Big_Unsigned)
                  return Big_Unsigned;
```

Diese Funktion berechnet $Base^{Exponent} \pmod N$.

```
function Get_Random (N : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet eine zufällige Big_Unsigned$B$ mit $B < N$.

```
function Inverse (X, N : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet das Inverse (bezüglich der Multiplikation) von X mod N. Falls kein Inverses von X mod N existiert gibt diese Funktion Big_Unsigned_Zero zurück.

```
function Get_Prime(N : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet mit einer überwältigenden Wahrscheinlichkeit eine Primzahl P mit $P < N$. Sie benutzt dazu die Funktion *Is_Prime* (**??**).

Exception:
$N <=$ Big_Unsigned_Two : Constraint_Error

```
function Get_N_Bit_Prime(N : Positive) return Big_Unsigned;
```

Diese Funktion berechnet mit einer überwältigenden Wahrscheinlichkeit eine N-Bit-Primzahl. Sie benutzt dazu die Funktion *Is_Prime* (**??**).

Exception:
$(N = 2) \vee (N > Size)$: Constraint_Error

```
Is_Prime(X : Big_Unsigned) return Boolean;
```

Dies Funktion gibt mit überwältigender Wahrscheinlichkeit *False* zurück, wenn es sich bei X nicht um eine Primzahl handelt.
Funktionsweise:

1. Es wird getestet ob X durch eine einstellige Primzahl (2,3,5,7) teilbar ist.
2. Es wird getestet ob X durch eine zweistellige Primzahl teilbar ist.
3. Es wird getestet ob es sich bei X um eine Lucas-Lemehr-Primzahl handelt.
4. Es wird getestet ob X durch eine dreistellige Primzahl teilbar ist.
5. Es werden 2-50 Miller-Rabin-Test durchgeführt. Wobei die Anzahl der Tests von X abhängig ist. Je größer $|X|$, desto weniger Test werden durchgeführt.

```
function Looks_Like_A_Prime(X : Big_Unsigned) return Boolean;
```

Dies Funktion gibt mit hoher Wahrscheinlichkeit *False* zurück, wenn es sich bei X nicht um eine Primzahl handelt.

Funktionsweise:
Wie *Is_Prime* (**??**) mit dem Unterschied, daß anstelle des Miller-Rabin-Tests ein einfacherer aber unzuverlässiger Primzahlentest, der 2-50 Zufallszahlen zieht und testet ob es sich dabei um einen Miller-Rabin-Zeugen oder einen echten Teiler von X handelt. Wenn keiner dieser Zufallszahlen ein Miller-Rabin-Zeuge oder ein echter Teiler von X ist, geht die Funktion davon aus, das es sich bei X um eine Primzahl handelt.

```
function Passed_Miller_Rabin_Test (X : Big_Unsigned;
                                   S : Positive)
                                   return Boolean;
```

Dies Funktion gibt *True* zurück, wenn X eine gewisse Anzahl von Miller-Rabin-Test (S-Stück) überlebt. Die Wahrscheinlichkeit das eine Pseudoprimzahl diesen Test besteht ist kleiner als $\frac{1}{2^{2S}}$.

```
function Jacobi(X, N : Big_Unsigned) return Integer;
```

Diese Funktion berechnet das Jacobisymbol für eine Element X aus $\mathbf{Z}_N$.

Vorbedingung:
N muss ungerade sein.

Rückgabewerte:
Jacobi(N,C) = 0 : X mod N = 0
Jacobi(N,C) = 1 : X ist ein quadratischer Rest mod N
Jacobi(N,C) = -1 : X ist ein quadratischer Rest mod N

Ausnahme:
N ungerade : Constraint_Error

14.5 Binäre Körper GF(2^m)

14.5.1 Erläuterungen

In dem separatem Body Crypto.Types.Big_Numbers.Mod_Binfield_Utils befinden sich Funktionen und Prozeduren bei denen eine `Big_Unsigned` Variable als Element aus dem Körper GF(2^m) interpretiert wird.

Beispiel:

```
-- Es wird das 1., 2. und 6. Bit von A gesetzt.
-- Danach enspricht A dem Element/Polynom z^5 + z + 1
A := Big_Unsigned_One;
A := A xor Shift_left(Big_Unsigned_One,1);
A := A xor Shift_left(Big_Unsigned_One,5);
```

Die `Big_Unsigned` Variable F ist im folgenden immer ein irreduzible Polynom $f(z)$ vom Grad m. Der Zugriff erfolgt über das Präfix **Binfield_Utils.**

14.5.2 API

```
function B_Add(Left,Right : Big_Unsigned) return Big_Unsigned;
function B_Sub(Left,Right : Big_Unsigned) return Big_Unsigned;
```

Diese beiden Funktion berechnen Left xor Right. Dies entspricht einer Addtion bzw. Subtraktion in GF(2^m).

```
function B_Mult(Left, Right, F : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet Left · Right mod F.

```
function B_Square(A, F : Big_Unsigned)
return Big_Unsigned;
```

Diese Funktion berechnet A^2 mod F.

```
function B_Div (Left, Right, F : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet Left/Right mod F.

```
function B_Mod(Left, Right  : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet Left mod Right.

```
function B_Inverse(X, F : Big_Unsigned) return Big_Unsigned;
```

Diese Funktion berechnet $X^{-}1 \mod F$.

14.6 Exceptions

```
Constraint_Size_Error : exception;
Conversion_Error      : exception;
Division_By_Zero      : exception;
Is_Zero_Error         : exception;
```

Kapitel 15

Acl.Crypto.Asymmetric

Beschreibung

Dies ist das Wurzelpaket für die asymmetrische Kryptographie.

Funktion

Dieses Paket ermöglicht dem symmetrischen Teil den direkten Zugriff auf Crypto.Types, welches grundlegende Typen und deren Basisfunktionen zur Verfügung stellt.
Des weiteren ermöglicht diese Paket dem asymmetrischen Teil den Zugriff auf Crypto.Types.Big_Numbers.

15.1 Exceptions

```
Invalid_Private_Key_Error : exception;
Invalid_Public_Key_Error  : exception;
Plaintext_Too_Long_Error  : exception;
Decrypt_Error             : exception;
```

Kapitel 16

Acl.Crypto.Asymmetric.DSA

Bei diesem generischen Paket handelt es sich um eine Implementierung des DSA-Signaturalgorithmus. Der DSA-Algorithmus ist Teil des digitalen Signatur Standards (DSS) [**?**] der von der NSA entworfen und 1994 vom NIST verabschiedet wurde. Die Bezeichner wurden aus dem DSS übernommen.
Mit Hilfe von DSA sind Sie in der Lage digitale Unterschriften zu erstellen und deren Authentizität zu verifizieren. Um eine Nachricht digital zu unterschreiben (signieren) benötigt Sie einen öffentlichen und einen privaten Schlüssel. Den privaten Schlüssel benötigt Sie für den Signierungsprozess und den öffentlichen Schlüssel für den Verifikationsprozess. Bei diesen Prozessen wird nicht die Nachricht selbst sondern deren SHA-1 Hash-Wert **??** unterschrieben. Jemand der den privaten Schlüssel nicht kennt, ist nicht in der Lage eine gültige (verifizierbare) Signatur zu erstellen. Aber jeder der den öffentlichen Schlüssel kennt, kann sich von der Korrektheit einer Signatur überzeugen indem er sie dem Verifizierungsprozess unterwirft.

16.1 Generischer Teil

```
generic
   Size : Positive;
```

Exception $Size \neq 512 + i64 \quad (i \in \{0, \ldots, 8\})$: Constraint_Error;

16.2 API

16.2.1 Typen

```
subtype DSA_Number is Bytes(0..Size/8-1);
type Public_Key_DSA  is private;
type Private_Key_DSA is private;
```

```
type Signature_DSA    is private;
```

Bei DSA_Number handeltes sich um ein Byte-Array das als Zahl interpretiert wird. Das erste Element des Arrays (First) enstpricht dabei dem dem höchstwertigsten Byte und das letzte Element des Arrays (Last) dem niederwertigsten Byte dieser Zahl.

16.2.2 Prozeduren und Funktionen

```
procedure Gen_Key(Public_Key  : out Public_Key_DSA;
                  Private_Key : out Private_Key_DSA);
```

Diese Prozedur erzeugt ein Schlüsselpaar, das aus einem öffentlichen *Public_Key* und einem privaten Schlüssel *Private_Key* besteht.

```
procedure Sign(Private_Key  : in  Private_Key_DSA;
               SHA1_Hash    : in  W_Block160;
               Signature    : out Signature_DSA);
```

Diese Prozedur signiert den SHA-1 Hashwert (*SHA1_Hash*) einer Nachricht mit einem privaten Schlüssel (*Private_Key*).

```
function Verify(Public_Key  : Public_Key_DSA;
                SHA1_Hash   : W_Block160;
                Signature   : Signature_DSA) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe des öffentlichen Schlüssels *Public_Key* verifiziert werden kann, ob es sich bei *Signature* um eine gültige Unterschrift des SHA-1 Hashwertes *SHA1_Hash* handelt. Ist dies nicht der Fall gibt sie "False" zurück.
Mit einem öffentlichen Schlüssel, können nur Signaturen des zugehörigen privaten Schlüssels verifiziert werden. Passen die beiden Schlüssel nicht zusammen, gibt diese Funktion "False" zurück.
Wenn Bob anstatt seines den Ausweis den von Alice mit sich führt, dann kann man anhand dieses Ausweises auch nicht die Identität von Bob überprüfen, obwohl Alice Ausweis gültig ist.

```
procedure Sign_File(Filename    : in  String;
                    Private_Key : in  Private_Key_DSA;
                    Signature   : out Signature_DSA);
```

Mit dieser Prozedur kann man mit Hilfe eines privaten Schlüssels *Private_Key* eine Datei (*Filename*) signieren.

Exception:

Die Datei *Filename* ist größer als 2^{26} TB	: SHA1_Constraint_Error
Private_Key wurde nicht initialisiert	: Invalid_Private_Key
Lesefehler	: File_Open_Error

```
function Verify_File(Filename   : String;
                     Public_Key : Public_Key_DSA;
                     Signature  : Signature_DSA) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe des öffentlichen Schlüssels *Public_Key* verifiziert werden kann, ob es sich bei *Signature* um eine gültige Unterschrift der Datei (*Filename*) handelt. Ist dies nicht der Fall gibt sie "False" zurück.
Mit einem öffentlichen Schlüssel, können nur Signaturen des dazugehörigen privaten Schlüssels verifiziert werden. Passen die beiden Schlüssel nicht zusammen, gibt diese Funktion "False" zurück.

Exception:

Die Datei *Filename* ist größer als 2^{26} TB	: SHA1_Constraint_Error
Public_Key wurde nicht initialisiert	: Invalid_Public_Key
Lesefehler	: File_Open_Error

```
function Verify_Key_Pair(Private_Key : Private_Key_DSA;
                         Public_Key
: Public_Key_DSA) return Boolean;
```

Diese Funktion gibt "True" zurück, wenn der privaten *Private_Key* und der öffentliche Schlüssel *Public_Key* zusammengehören, d.h. ein Paar bilden, ansonsten gibt sie "False" zurück.

Exception:
Public_Key wurde nicht initialisiert : Invalid_Public_Key_Error
Private_Key wurde nicht initialisiert : Invalid_Private_Key_Error

---

```
procedure Get_Public_Key(Public_Key : in Public_Key_DSA;
                         P : out DSA_Number;
                         Q : out DSA_Number;
                         G : out DSA_Number;
                         Y : out DSA_Number);
```

Diese Prozedur zerlegt einen öffentlichen Schlüssel *Public_Key* in folgende Komponenten:

- Eine Size-Bit-Primzahl P.
- Eine 160-Bit-Primzahl Q.
- Einen Generator G der eine Untergruppe von P mit der Ordnung Q erzeugt.
- Der eigentliche öffentliche Schlüssel Y.

Mit Hilfe dieser Werte lässt sich der öffentliche Schlüssel zu einem späteren Zeitpunkt wieder rekonstruieren.

```
procedure Get_Private_Key(Private_Key : in Private_Key_DSA;
                          P : out DSA_Number;
                          Q : out DSA_Number;
                          G : out DSA_Number;
                          X : out DSA_Number);
```

Diese Prozedur zerlegt einen privaten Schlüssel *Private_Key* in folgende Komponenten:

- Eine Size-Bit-Primzahl P.
- Eine 160-Bit-Primzahl Q.
- Einen Generator G der eine Untergruppe von P mit der Ordnung Q erzeugt.
- Der eigentliche geheime Schlüssel X.

Mit Hilfe dieser Werte lässt sich der private Schlüssel zu einem späteren Zeitpunkt wieder rekonstruieren.

```
procedure Set_Public_Key(P : in DSA_Number;
                         Q : in DSA_Number;
                         G : in DSA_Number;
                         Y : in DSA_Number;
                         Public_Key : out Public_Key_DSA);
```

Mit Hilfe dieser Prozedur ist es möglich einen öffentlichen Schlüssel *Public_Key* zu (re-)konstruieren. Man benötigt dazu folgende Werte:

- Eine Size-Bit-Primzahl P.
- Eine 160-Bit-Primzahl Q.
- Einen Generator G der eine Untergruppe von Z_P^* mit der Ordnung Q erzeugt.
- Den eigentliche öffentlichen Schlüssel Y.

Exception:
P, Q, G oder Y unzulässig : Invalid_Public_Key_Error.

```
procedure Set_Private_Key(P : in DSA_Number;
                          Q : in DSA_Number;
                          G : in DSA_Number;
                          X : in DSA_Number;
                          Private_Key : out Private_Key_DSA);
```

Mit Hilfe dieser Prozedur ist es möglich einen privaten Schlüssel (*Private_key*) zu (re-)konstruieren. Man benötigt dazu folgende Werte:

- Eine Size-Bit-Primzahl P.
- Eine 160-Bit-Primzahl Q.
- Einen Generator G der eine Untergruppe von Z_P^* mit der Ordnung Q erzeugt.
- Den eigentliche geheime Schlüssel X.

Exception:
P, Q, G oder Y unzulässig : Invalid_Private_Key_Error.

16.3 Anwendungsbeispiel

```
with Crypto.Types.Big_Numbers;
with Crypto.Asymmetric.Dsa;
with Ada.Text_IO; use Ada.Text_IO;

procedure Example_DSA is
   package DSA is new Crypto.Asymmetric.DSA(512);
   use DSA;

   Public_Key  : Public_Key_DSA;
   Private_Key : Private_Key_DSA;
   Signature   : Signature_DSA;

begin
   --Schluesselgenerierung
   Gen_Key(Public_Key, Private_Key);

   --Signierung
   Sign_File("example_dsa.adb", Private_Key, Signature);
```

```
   -- Verifikation
   if Verify_File("example_dsa.adb", Public_Key, Signature) then
      Put_Line("OK");
   else Put_Line("Implementation error.");
   end if;

end Example_DSA;
```

Kapitel 17

Acl.Crypto.Asymmetric.RSA

Bei diesem generischen Paket handelt es sich um eine RSA (Rivest Shamir Adelman) Implementierung. Klartextblöcke (Plaintext) bzw. Chiffretextblöcke (Ciphertext) können mit Hilfe von RSAES-OAEP [**?**] ver- bzw. entschlüsselt werden. RSAES-PKCS1-v1_5 wurde nicht implementiert, da im PKCS #1 v2.1 (Public-Key Cryptography Standards) empfohlen wird für neue Anwendungen RSAES-OEAP zu verwenden.

OEAP-Details

- Diese Implementation verwendet SHA1 innerhalb der MGF1 (Mask Generation Function 1)
- Diese Implementation unterstüzt nicht das optionale Label L, d.h. L ist immer ein leerer String

17.1 Generischer Teil

```
generic
   Size : Positive;
```

Vorbedingung:
Size $\geq$512

Exception:
Size $<$ 512 : Constraint_Size_Error;

17.2 API

17.2.1 Typen

```
subtype RSA_Number is Bytes(0..Size/8-1);
type Public_Key_RSA  is private;
type Private_Key_RSA is private;
```

Bei RSA_Number handeltes sich um ein Byte-Array das als Zahl interpretiert wird. Das erste Element des Arrays (First) enstpricht dabei dem dem höchstwertigsten Byte und das letzte Element des Arrays (Last) dem niederwertigsten Byte dieser Zahl.

17.2.2 Prozeduren und Funktionen

```
procedure Gen_Key(Public_Key  : out Public_Key_RSA;
                  Private_Key : out Private_Key_RSA);
```

Dies Prozedure erzeugt ein Schlüsselpaar, das aus einem öffentlichen *Public_Key* und einem privaten Schlüssel *Private_Key* besteht.

```
function Verify_Key_Pair(Private_Key : Private_Key_RSA;
                         Public_Key
: Public_Key_RSA) return Boolean;
```

Diese Funkton gibt "True" zurück, wenn der privaten *Private_Key* und der öffentliche Schlüssel *Public_Key* zusammengehören, d.h. ein Paar bilden, ansonsten gibt sie "False" zurück.

Exception:
Public_Key wurde nicht initalisiert : Invalid_Public_Key_RSA
Private_Key wurde nicht initalisiert : Invalid_Private_Key_RSA

```
function OAEP_Encrypt(Public_Key : in Public_Key_RSA;
                      Plaintext  : in
Bytes) return RSA_Number;
```

Diese Funktion verschlüsselt einen Klartext(-block) (*Plaintext*) mit dem OEAP-Verfahren [**?**] und gibt den Chiffretext zurück.

Vorbedingung:
Plaintext'Length $\leq Size - 336 (= 2 \cdot 160 + 2 \cdot 8)$
Public_Key is ein zulässiger Schlüssel.

Exceptions:

Plaintext'Length > Size - 336	:	Plaintext_Too_Long_Error
Public_Key wurde nicht initalisiert	:	Invalid_Public_Key_Error
Unzulässiger Public_Key	:	Invalid_Public_Key_Error

```
function OAEP_Decrypt(Private_Key : in Private_Key_RSA;
                      Ciphertext  : in
RSA_Number) return Bytes;
```

Diese Funktion entschlüsselt einen Chiffretext(-block) (*Ciphertext*), mit Hilfe eines privaten Schlüssels (*Private_Key*). Der entschlüsselte Text entspricht nur dem "orginal" Klartext, wenn der dabei benutze öffentliche Schlüssel und *Private_Key* ein Schlüsselpaar bilden (**??**).

Vorbedingungen:

- *Private_Key* ist ein zulässiger Schlüssel.
- *Ciphertext* wurde mit dem RSAES-OEAP-Verfahren verschlüsselt.
- Der öffenltiche Schlüssel mit dem *Ciphertext* verschlüsselt wurde und *Private_Key* bilden ein Schlüsselpaar.

Exception:
Verletzung einer Vorbedingung : Decrypt_Error

```
procedure Get_Public_Key(Public_Key : in Public_Key_RSA;
                         N : out RSA_Number;
                         E : out RSA_Number);
```

Diese Prozedur zerlegt einen öffentlichen Schlüssel *Public_Key* in folgende Komponenten:

- Einen Size-Bit-RSA-Modulus N. (N = PQ P,Q Prime)
- Einen öffentlichen RSA-Exponenten E.

Mit Hilfe dieser Werte lässt sich der öffentliche Schlüssel zu einem späteren Zeitpunkt wieder rekonstruieren.

```
procedure Get_Private_Key(Private_Key : in Private_Key_RSA;
                          N    : out RSA_Number;
                          D    : out RSA_Number;
                          Phi  : out RSA_Number);
```

Diese Prozedur zerlegt einen privaten Schlüssel *Private_Key* in folgende Komponenten:

- Einen Size-Bit-RSA-Modulus N. (N = PQ P,Q Prim)
- Einen privaten RSA-Exponenten D.
- Phi = (P-1)(Q-1).

Mit Hilfe dieser Werte lässt sich der private Schlüssel zu einem späteren Zeitpunkt wieder rekonstruieren.

```
procedure Set_Public_Key(N : in RSA_Number;
                         E : in RSA_Number;
                         Public_Key : out Public_Key_RSA);
```

Mit Hilfe dieser Prozedur ist es möglich einen öffentlichen Schlüssel *Public_Key* zu (re-)konstruieren. Man benötigt dazu folgende Werte:

- Einen Size-Bit-RSA-Modulus *N*.
- Einen öffentlichen RSA-Exponenten *E*.

Exception:
N oder E unzulässig : Invalid_Public_Key_Error.

```
procedure Set_Private_Key(N     : in RSA_Number;
                          D     : in RSA_Number;
                          Phi   : in RSA_Number;
                          Private_Key : out Private_Key_RSA);
```

Mit Hilfe dieser Prozedur ist es möglich einen privaten Schlüssel *Private_Key* zu (re-)konstruieren. Man benötigt dazu folgende Werte:

- Einen Size-Bit-RSA-Modulus *N*. (N = PQ P,Q Prim)
- Einen privaten RSA-Exponenten *D*.
- Phi = (P-1)(Q-1).

Exception:
N, D oder S unzulässig : Invalid_Public_Key_Error.

17.2.3 Low-Level-API

Diese API sollten sie nur verwenden, wenn Sie wissen was Sie tun. Bei einem naiven Einsatz können kritische Sicherheitsprobleme auftreten, da identische Klartexte zu identischen Chiffretexten verschlüsselt werden.

```
procedure Encrypt(Public_Key : in  Public_Key_RSA;
                  Plaintext  : in  RSA_Number;
                  Ciphertext : out RSA_Number);
```

Diese Prozedur verschlüsselt einen Klartext (*Plaintext*) mit Hilfe eines öffentlichen Schlüssels (*Public_Key*) zu einem Chiffretext (*Ciphertext*). Sie verwendet dabei das "naive" RSA-Verfahren ($c = p^d \pmod{n}$).

```
procedure Decrypt(Private_Key : in  Private_Key_RSA;
                  Ciphertext  : in  RSA_Number;
                  Plaintext   : out RSA_Number);
```

Diese Prozedur entschlüsselt einen Chiffretext (*Ciphertext*) mit Hilfe eines privaten Schlüssels (*Private_Key*) zu einem Klartext (*Plaintext*). Sie verwendet dabei das "naive" RSA-Verfahren ($p = c^e \pmod{n}$).

17.2.4 Anwendungsbeispiel

```
with Crypto.Types;
with Crypto.Asymmetric.RSA;
with Ada.Text_IO;

procedure Example_RSA is
   package RSA is new Crypto.Asymmetric.RSA(512);
   use Crypto.Types;
   use Ada.Text_IO;
   use RSA;

   Message : Bytes := To_Bytes("All your base...");
   Public_Key  : Public_Key_RSA;
   Private_Key : Private_Key_RSA;

begin
   --Generierung des Schluesselpaares
   Gen_Key(Public_Key, Private_Key);

   declare
      --Verschluesselung
      Ciphertext : RSA_Number := OAEP_Encrypt(Public_Key, Message);
```

```
      -- Entschluesselung
      Plaintext : Bytes := OAEP_Decrypt(Private_Key, Ciphertext);

   begin
      -- Ausgabe des Chiffretextes
      Put(To_String(Ciphertext)); New_Line;

      -- Ausgabe des entschluesselten Chiffretextes
      Put(To_String(Plaintext));
   end;
end Example_RSA;
```

Kapitel 18

Ellyptische Kurven

Die ACL unterstützt die beiden folgenden Klasen von ellyptischen Kurven

- $y^2 = x^3 + ax + b \mod p$ mit $p \in \mathbf{P} \setminus \{2, 3\}$
  Hier handelt es sich um ellyptische Kurven über dem endlichen Körper Z_p.
- $y^2 + xy = x^3 + a(z)x^2 + b(z) \mod f(z)$
  Hier handelt es sich über nicht supersingulare ellyptische Kurven über den endlichen Körper $GF(2^{\deg f(z)})$.
  Vorraussetzung: $f(z)$ ein irreduzibles Polynom.

18.1 Wurzelpaket: Types.Elliptic_Curves

Dieses generische Paket ist das Wurzelpaket für ellyptische Kurven. In ihm ist der Basistype `EC_Point` für ellyptische Kurven definiert.

18.1.1 API

Generischer Teil

```
generic
with package Big is new Crypto.Types.Big_Numbers(<>);
```

Typen

Dieses Paket stellt den Typ *EC_Point* zur Verfügung.

```
-- (x,y)
type EC_Point is record
   X : Big.Big_Unsigned;
   Y : Big.Big_Unsigned;
end record;
```

Die Konstante `EC_Point_Infinity` entspricht dem Punkt ∞. Dieser Punkt befindet sich auf jeder ellyptischen Kurve und entspricht dem neutrale Element der Addition.

Prozeduren

```
procedure Put(Item : in EC_Point; Base : in Big.Number_Base := 10;

procedure Put_Line(Item : in EC_Point; Base : in Big.Number_Base := 10);
```

Die beiden Prozeduren geben einen `EC_Point` auf der Standardausgabe in Form eines Tupels aus.

Beispiel:

```
procedure Example is
  P : EC_Point(X => Big_unsigned_Three, Y => Big_Unsigned_one)
begin
  Put(P);                 -- Ausgabe: "(3,1)"
  Put(P, Base => 2);      -- Ausgabe: "(2#11, 2#1#)"
end Example;
```

18.2 Kinderpakete

Die beiden Kinderpakete

- Types.Elliptic_Curves.Zp
- Types.Elliptic_Curves.NSS_BF

verfügen über die folgende Typen

Typen

Das Paket *Types.Elliptic_Curves.Zp* stellt den Typ *Elliptic_Curve_Zp* zur Verfügung.

```
-- (A,B,P)
type Elliptic_Curve_Zp is record
  A : Big_Unsigned;
  B : Big_Unsigned;
  P : Big_Unsigned;
end record;
```

API

```
procedure Init(A, B, P : in Big_Unsigned);
procedure Init(ECZ : in Elliptic_Curve_Zp);

procedure Init(A, B, F : in Big_Unsigned);
```

Diese Prozeduren initialiseren die ellyptische Kurven. Die ersten beiden bilden die ellyptische Kurve $y^2 = x^3 + Ax + B \mod P$, die dritte die Kurve $y^2 + xy = x^3 + Ax^2 + B \mod F$.
Diese Funktion testet nicht, ob es sich bei `P` bzw. `F` wirklich um eine Primzahl bzw. irreduzibles Polynom handelt oder die Parameter tatsächlich eine ellyptische Kurve bilden.

```
function Is_Elliptic_Curve return Boolean;
```

Diese Funktion testet, ob durch die Initalisierung mittels `Init` ein ellyptische Kurve konstruiert worden ist. Wenn dies nicht der Fall ist, dann liefern alle folgenden Funktionen und Prozeduren wahrscheinlich nicht das gewünschte Ergebnis.

```
function On_Elliptic_Curve(X : EC_Point) return Boolean;
```

Diese Funktion testet, ob `X` ein Punkt auf der eyllptischen Kurve ist.

```
function Negative(X : EC_Point) return EC_Point;
```

Dies Funktion berechnet `-X`.

```
function "+"(Left, Right : EC_Point) return EC_Point;
```

Dies Funktion berechnet `Left + Right`.

```
function Double(X : EC_Point) return EC_Point;
```

Dies Funktion berechnet `2X`.

Kapitel 19

Ellyptische Kurven Datenbank ZP

Die ACL bietet für die ellyptischen Kurven der Form

- $y^2 = x^3 + ax + b \mod p$ mit $p \in \mathbf{P} \setminus \{2, 3\}$
  Hier handelt es sich um ellyptische Kurven über dem endlichen Körper Z_p.

eine Datenbank, welche die durch NIST (National Institute of Standards and Technology) herausgegebenen elliptischen Kurven enthält.

Es sind die folgenden Kurven enthalten.

- Curve P-192
- Curve P-224
- Curve P-256
- Curve P-384
- Curve P-521
- Test Curve (5 bit)

19.1 API

19.1.1 Generischer Teil

```
generic
with package Big is new Crypto.Types.Big_Numbers(<>);
```

19.1.2 Typen

Dieses Paket stellt den Typ *Bit_Length* und den Typ *Precomputed_Elliptic_Curve* zur Verfügung.

```
type Bit_Length is new natural;

      type Precomputed_Elliptic_Curve is record
        -- prime modulus
            p      : String(1..192) := (others=>' ');
            -- order
            r      : String(1..192) := (others=>' ');
            -- 160-bit input seed to SHA-1
            s      : String(1..192) := (others=>' ');
            -- output of SHA-1
            c      : String(1..192) := (others=>' ');
            -- coefficient b (satisfying b*b*c = -27 (mod p))
            b      : String(1..192) := (others=>' ');
            -- base point x coordinate
            Gx     : String(1..192) := (others=>' ');
            -- base point x coordinate
            Gy     : String(1..192) := (others=>' ');
            -- Bit lenght
            length : Bit_Length;
      end record;
```

19.1.3 Prozeduren

```
   procedure Get_Elliptic_Curve(ECZ
: out Elliptic_Curve_Zp;
                                ECP    : out EC_Point;
                                order  : out Big_Unsigned;
                                length : in  Bit_Length);
```

Diese Prozedur holt alle Variablen aus der Datenbank, die zum Rechnen mit Elliptische Kurven benötigt werden. Die Kurve hat mindestens die kryptografische Sicherheit von *length*.

Exception:
BitLength is not supported. (Max BitLength = 521) : LEN_EX

```
procedure Set_Elliptic_Curve_Map;
```

Diese Prozedur initiert die Datenbank. Sie ist nur für den internen Gebrauch notwendig.

19.2 Anwendungsbeispiel

```
with Crypto.Types; use Crypto.Types;
with Crypto.Types.Big_Numbers;
with Crypto.Types.Elliptic_Curves.Zp;
with Crypto.Types.Elliptic_Curves.Zp.Database;

procedure Example_EC_DB_ZP is
  package Big is new Crypto.Types.Big_Numbers(Size);
  use Big;
  package EC  is new Crypto.Types.Elliptic_Curves(Big);
  use EC;
  package Zp is new EC.Zp;
  use Zp;
        package DB is new Zp.Database;
        use DB;

  EC_ZP : Public_Key_ECNR;
  EC_P  : Private_Key_ECNR;
  order : Signature_ECNR;

begin
        Get_Elliptic_Curve(EC_ZP, EC_P, order, 168);
end Example_EC_DB_ZP;
```

Kapitel 20

Acl.Crypto.Asymmetric.ECDSA

Bei diesem generischen Paket handelt es sich um eine Implementierung des ECDSA-Signaturalgorithmus. Der ECDSA-Algorithmus ist die Erweiterung des DSA-Algorithmus auf den Bereich der Elliptischen Kurven.
Mit Hilfe von ECDSA sind Sie in der Lage digitale Unterschriften zu erstellen und deren Authentizität zu verifizieren. Um eine Nachricht digital zu unterschreiben (signieren) benötigt Sie einen öffentlichen und einen privaten Schlüssel. Den privaten Schlüssel benötigt Sie für den Signierungsprozess und den öffentlichen Schlüssel für den Verifikationsprozess. Bei diesen Prozessen wird nicht die Nachricht selbst sondern deren SHA-1 Hash-Wert **??** unterschrieben. Jemand der den privaten Schlüssel nicht kennt, ist nicht in der Lage eine gültige (verifizierbare) Signatur zu erstellen. Aber jeder der den öffentlichen Schlüssel kennt, kann sich von der Korrektheit einer Signatur überzeugen indem er sie dem Verifizierungsprozess unterwirft.

20.1 Generischer Teil

```
generic
   Size : Positive;
```

Exception $Size \neq 512 + i64 \quad (i \in \{0, \ldots, 8\})$: Constraint_Error;

20.2 API

20.2.1 Typen

```
type Public_Key_ECDSA  is private;
type Private_Key_ECDSA is private;
type Signature_ECDSA   is private;
```

Der *Public_Key_ECDSA* ist ein beinhaltet folgende Variablen:

```
E : Elliptic_Curve_Zp;
P : EC_Point;
n : Big_Unsigned;
Q : EC_Point;
```

E ist eine elliptische Kurve üpber ZP. P ist ein öffentlicher Punkt auf dieser Kurve, dessen Ordnung n bekannt ist. Q ist der öffenliche Teil des Schlüssels.

Der *Private_Key_ECDSA* beinhaltet folgende Variablen:

```
Q : EC_Point;
n : Big_Unsigned;
```

Q ist ein Punkt auf der Elliptischen Kurve und der öffenliche Teil des Schlüssels. n ist eine Zahl und der geheime Teil des Schlüssels.

Die *Signatur_ECDSA* enthält zwei Zahlen.

```
R : Big_Unsigned;
S : Big_Unsigned;
```

20.2.2 Prozeduren und Funktionen

```
procedure Gen_Public_Key(Public_Key : out Public_Key_ECDSA;
                         length     : in DB.Bit_Length);
```

Diese Prozedur holt eine Elliptische Kurve aus der Datenbank. Die Kurve hat mindestens die kryptografische Sicherheit von *length*.

Exception:
BitLength is not supported. (Max BitLength = 521) : LEN_EX

```
procedure Gen_Private_Key(Public_Key  : in out Public_Key_ECDSA;
                          Private_Key : out Private_Key_ECDSA);
```

Diese Prozedur erzeugt ein Schlüsselpaar, den *Private_Key*. Hierzu wird eine zufällige Zahl generiert, diese ist der geheime Teil des Schlüssels. Des weiteren wird ein Punkt als öffentlicher Teil des Schlüssels, als Produkt aus dem eben generierten Zahl und dem öffentlichen Punkt des *Public_Key* berechnet. Dieser Punkt wird sowohl im öffentlichen als auch im privaten Schlüssel gespeichert.

```
procedure Sign(  Public_Key   : in Public_Key_ECDSA;
                                                        Private
Private_Key_ECDSA;
                 SHA1_Hash    : in  W_Block160;
                 Signature    : out Signature_ECDSA);
```

Diese Prozedur signiert den SHA-1 Hashwert (*SHA1_Hash*) einer Nachricht mit einem privaten Schlüssel (*Private_Key*).

```
function Verify(Public_Key   : Public_Key_ECDSA;
                SHA1_Hash    : W_Block160;
                Signature    : Signature_ECDSA) return Boolean;
```

Diese Funktion gibt den Wert “True” zurück, wenn mit Hilfe des öffentlichen Schlüssels *Public_Key* verifiziert werden kann, ob es sich bei *Signature* um eine gültige Unterschrift des SHA-1 Hashwertes *SHA1_Hash* handelt. Ist dies nicht der Fall gibt sie “False” zurück.
Mit einem öffentlichen Schlüssel, können nur Signaturen des zugehörigen privaten Schlüssels verifiziert werden. Passen die beiden Schlüssel nicht zusammen, gibt diese Funktion “False” zurück.
Wenn Bob anstatt seines den Ausweis den von Alice mit sich führt, dann kann man anhand dieses Ausweises auch nicht die Identität von Bob überprüfen, obwohl Alice Ausweis gültig ist.

```
procedure Sign_File(Filename   : in  String;
                    Public_Key : in  Public_Key_ECDSA;
                    Private_Key : in
Private_Key_ECDSA;
                    Signature  : out Signature_ECDSA);
```

Mit dieser Prozedur kann man mit Hilfe eines privaten Schlüssels *Private_Key* eine Datei (*Filename*) signieren.

Exception:

Die Datei *Filename* ist größer als 2^{26} TB	: SHA1_Constraint_Error
Private_Key wurde nicht initialisiert	: Invalid_Private_Key
Lesefehler	: File_Open_Error

```
function Verify_File(Filename   : String;
                     Public_Key : Public_Key_ECDSA;
                     Signature  : Signature_ECDSA) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe des öffentlichen Schlüssels *Public_Key* verifiziert werden kann, ob es sich bei *Signature* um eine gültige Unterschrift der Datei (*Filename*) handelt. Ist dies nicht der Fall gibt sie "False" zurück.
Mit einem öffentlichen Schlüssel, können nur Signaturen des dazugehörigen privaten Schlüssels verifiziert werden. Passen die beiden Schlüssel nicht zusammen, gibt diese Funktion "False" zurück.

Exception:

Die Datei *Filename* ist größer als 2^{26} TB	: SHA1_Constraint_Error
Public_Key wurde nicht initialisiert	: Invalid_Public_Key
Lesefehler	: File_Open_Error

```
function Verify_Key_Pair(Private_Key : Private_Key_ECDSA;
                         Public_Key
: Public_Key_ECDSA) return Boolean;
```

Diese Funktion gibt "True" zurück, wenn der privaten *Private_Key* und der öffentliche Schlüssel *Public_Key* zusammengehören, d.h. ein Paar bilden, ansonsten gibt sie "False" zurück.

Exception:
Public_Key wurde nicht initialisiert : Invalid_Public_Key_Error
Private_Key wurde nicht initialisiert : Invalid_Private_Key_Error

---

20.3 Anwendungsbeispiel

```
with Crypto.Types.Big_Numbers;
with Crypto.Asymmetric.ECDSA;
with Ada.Text_IO; use Ada.Text_IO;

procedure Example_ECDSA is
   package ECDSA is new Crypto.Asymmetric.ECDSA(512);
   use ECDSA;

   Public_Key  : Public_Key_ECDSA;
   Private_Key : Private_Key_ECDSA;
   Signature   : Signature_ECDSA;

begin
   --Schluesselgenerierung
   Gen_Public_Key(Public_Key, 178);
   Gen_Private_Key(Public_Key, Private_Key);

   --Signierung
   Sign_File("example_ecdsa.adb",Public_Key, Private_Key, Signature);

   --Verifikation
   if Verify_File("example_ecdsa.adb", Public_Key, Signature) then
      Put_Line("OK");
   else Put_Line("Implementation error.");
   end if;
```

```
end Example_ECDSA;
```

Kapitel 21

Acl.Crypto.Asymmetric.ECNR

Bei diesem generischen Paket handelt es sich um eine Implementierung des ECNR-Signaturalgorithmus. Der ECNR-Algorithmus ist die Erweiterung des NR-Algorithmus auf den Bereich der Elliptischen Kurven.
Mit Hilfe von ECNR sind Sie in der Lage digitale Unterschriften zu erstellen und deren Authentizität zu verifizieren. Um eine Nachricht digital zu unterschreiben (signieren) benötigt Sie einen öffentlichen und einen privaten Schlüssel. Den privaten Schlüssel benötigt Sie für den Signierungsprozess und den öffentlichen Schlüssel für den Verifikationsprozess. Bei diesen Prozessen wird nicht die Nachricht selbst sondern deren SHA-1 Hash-Wert **??** unterschrieben. Jemand der den privaten Schlüssel nicht kennt, ist nicht in der Lage eine gültige (verifizierbare) Signatur zu erstellen. Aber jeder der den öffentlichen Schlüssel kennt, kann sich von der Korrektheit einer Signatur überzeugen indem er sie dem Verifizierungsprozess unterwirft.
Dieser Algorithmus ist patentrechtlich geschützt zur Verwendung muss die Genehmigung Eigentümers eingeholt werden.

21.1 Mathematische Beschreibung

21.1.1 Voraussetzung

Eine Elliptische Kurve E der Form $y^2 = x^3 + a * x + bmodp$.
Ein Punkt P der auf der Kurve E liegt und von der Ordnung n ist.

Ein Schlüsselpaar besteht aus einem öffentlichen Punkt Q und einer geheimen Zahl d welche die Gleichung $Q = d * P$ erfüllen.

Es werden also folgende Parameter benötigt: a, b, p, P und n. Sowie ein Schlüsselpaar (d, Q) und f, dies repräsentiert die Nachricht.

Die zu berechnende Signatur besteht aus zwei Zahlen (R, S).

21.1.2 Signieren

1. Generiere ein Schlüsselpaar V(öffentlicher Punkt mit (x_V, y_V)) und u(geheime Zahl). V darf nicht der Punkt im Unendlichen sein.
2. Wandle x_V in eine Zahl i um.
3. Berechne $R = i + fmodn$. Ist $R = 0$ dann beginne wieder mit Schritt 1.
4. Berechne $S = u - dR$.
5. Gib (R, S) als Schlüsselpaar aus.

21.1.3 Verifizieren

1. Kontrolliere ob R im Bereich $[1, n-1]$ und S im Bereich $[0, n-1]$ liegen.
2. Berechne einen Punkt G mit $G = S * P + R * Q$. G darf nicht der Punkt im Unendlichen sein.
3. Wandle x_G in eine Zahl i um.
4. Berechne $f' = c - imodn$.
5. Ist $f' = f$ dann gebe valide aus.

21.2 Generischer Teil

```
generic
  Size : Positive;
```

Exception $Size \neq 512 + i64 \quad (i \in \{0, \ldots, 8\})$: Constraint_Error;

21.3 API

21.3.1 Typen

```
type Public_Key_ECNR  is private;
type Private_Key_ECNR is private;
type Signature_ECNR   is private;
```

Der *Public_Key_ECNR* ist ein beinhaltet folgende Variablen:

```
E : Elliptic_Curve_Zp;
P : EC_Point;
n : Big_Unsigned;
Q : EC_Point;
```

E ist eine elliptische Kurve üpber ZP. P ist ein öffentlicher Punkt auf dieser Kurve, dessen Ordnung n bekannt ist. Q ist der öffenliche Teil des Schlüssels.

Der *Private_Key_ECNR* beinhaltet folgende Variablen:

```
Q : EC_Point;
d : Big_Unsigned;
```

Q ist ein Punkt auf der Elliptischen Kurve und der öffenliche Teil des Schlüssels. n ist eine Zahl und der geheime Teil des Schlüssels.

Die *Signatur_ECNR* enthält zwei Zahlen.

```
R : Big_Unsigned;
S : Big_Unsigned;
```

21.3.2 Prozeduren und Funktionen

```
procedure Gen_Public_Key(Public_Key : out Public_Key_ECNR;
                         length     : in DB.Bit_Length);
```

Diese Prozedur holt eine Elliptische Kurve aus der Datenbank. Die Kurve hat mindestens die kryptografische Sicherheit von *length*.

Exception:
BitLength is not supported. (Max BitLength = 521) : LEN_EX

```
procedure Gen_Private_Key(Public_Key
: in out Public_Key_ECNR;
                          Private_Key : out
Private_Key_ECNR);
```

Diese Prozedur erzeugt ein Schlüsselpaar, den *Private_Key*. Hierzu wird eine zufällige Zahl generiert, diese ist der geheime Teil des Schlüssels. Des weiteren wird ein Punkt als öffentlicher Teil des Schlüssels, als Produkt aus dem eben generierten Zahl und dem öffentlichen Punkt des *Public_Key* berechnet. Dieser Punkt wird sowohl im öffentlichen als auch im privaten Schlüssel gespeichert.

```
procedure Sign(Public_Key  : in  Public_Key_ECNR;
               Private_Key : in  Private_Key_ECNR;
               SHA1_Hash   : in  W_Block160;
               Signature   : out Signature_ECNR);
```

Diese Prozedur signiert den SHA-1 Hashwert (*SHA1_Hash*) einer Nachricht mit einem privaten Schlüssel (*Private_Key*).

```
function Verify(Public_Key : Public_Key_ECNR;
                SHA1_Hash  : W_Block160;
                Signature  : Signature_ECNR) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe des öffentlichen Schlüssels *Public_Key* verifiziert werden kann, ob es sich bei *Signature* um eine gültige Unterschrift des SHA-1 Hashwertes *SHA1_Hash* handelt. Ist dies nicht der Fall gibt sie "False" zurück.
Mit einem öffentlichen Schlüssel, können nur Signaturen des zugehörigen privaten Schlüssels verifiziert werden. Passen die beiden Schlüssel nicht zusammen, gibt diese Funktion "False" zurück.
Wenn Bob anstatt seines den Ausweis den von Alice mit sich führt, dann kann man anhand dieses Ausweises auch nicht die Identität von Bob überprüfen, obwohl Alice Ausweis gültig ist.

```
procedure Sign_File(Filename    : in  String;
                    Public_Key  : in  Public_Key_ECNR;
                    Private_Key : in  Private_Key_ECNR;
                    Signature   : out Signature_ECNR);
```

Mit dieser Prozedur kann man mit Hilfe eines privaten Schlüssels *Private_Key* eine Datei (*Filename*) signieren.

Exception:

Die Datei *Filename* ist größer als 2^{26} TB	: SHA1_Constraint_Error
Private_Key wurde nicht initialisiert	: Invalid_Private_Key
Lesefehler	: File_Open_Error

```
function Verify_File(Filename   : String;
                     Public_Key : Public_Key_ECNR;
                     Signature  : Signature_ECNR) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe des öffentlichen Schlüssels *Public_Key* verifiziert werden kann, ob es sich bei *Signature* um eine gültige Unterschrift der Datei (*Filename*) handelt. Ist dies nicht der Fall gibt sie "False" zurück.
Mit einem öffentlichen Schlüssel, können nur Signaturen des dazugehörigen privaten Schlüssels verifiziert werden. Passen die beiden Schlüssel nicht zusammen, gibt diese Funktion "False" zurück.

Exception:

Die Datei *Filename* ist größer als 2^{26} TB	: SHA1_Constraint_Error
Public_Key wurde nicht initialisiert	: Invalid_Public_Key
Lesefehler	: File_Open_Error

```
function Verify_Key_Pair(Private_Key : Private_Key_ECNR;
                         Public_Key
: Public_Key_ECNR) return Boolean;
```

Diese Funktion gibt "True" zurück, wenn der privaten *Private_Key* und der öffentliche Schlüssel *Public_Key* zusammengehören, d.h. ein Paar bilden, ansonsten gibt sie "False" zurück.

Exception:

Public_Key wurde nicht initialisiert	: Invalid_Public_Key_Error
Private_Key wurde nicht initialisiert	: Invalid_Private_Key_Error

---

21.4 Anwendungsbeispiel

```
with Crypto.Types.Big_Numbers;
with Crypto.Asymmetric.ECNR;
with Ada.Text_IO; use Ada.Text_IO;

procedure Example_ECNR is
   package ECNR is new Crypto.Asymmetric.ECNR(512);
   use ECNR;
```

```
   Public_Key   : Public_Key_ECNR;
   Private_Key  : Private_Key_ECNR;
   Signature    : Signature_ECNR;

begin
   --Schluesselgenerierung
   Gen_Public_Key(Public_Key, 178);
   Gen_Private_Key(Public_Key, Private_Key);

   --Signierung
   Sign_File("example_ecnr.adb",Public_Key, Private_Key, Signature);

   --Verifikation
   if Verify_File("example_ecnr.adb", Public_Key, Signature) then
        Put_Line("OK");
   else Put_Line("Implementation error.");
   end if;

end Example_ECNR;
```

Kapitel 22

Acl.Crypto.Symmetric.Algorithm.ECDH

Bei diesem generischen Paket handelt es sich um eine Implementierung des ECDH-Schlüsselgenerierungsalgorithmus. Der ECDH-Algorithmus ist die Erweiterung des DH-Algorithmus auf den Bereich der Elliptischen Kurven.
Mit Hilfe von ECDH sind zwei Parteien in der Lage ein gemeinsamen geheimen Schlüssel aus den eigenen Schlüsselpaar und aus dem öffentlichen Schlüssel der anderen Partei zu erzeugen.

22.1 Mathematische Beschreibung

22.1.1 Voraussetzung

Eine Elliptische Kurve E der Form $y^2 = x^3 + a * x + bmodp$.
Ein Punkt P der auf der Kurve E liegt und von der Ordnung n ist.

Ein Schlüsselpaar besteht aus einem öffentlichen Punkt Q und einer geheimen Zahl d welche die Gleichung $Q = d * P$ erfüllen.

Es werden also folgende Parameter benötigt: a, b, p, P und n. Sowie ein Schlüsselpaar (d, Q).

Die öffentlichen Schlüssel der anderen Partei werden mit $'$ gekennzeichnet.

22.1.2 Schlüsselgenerierung

1. Berechne einen Punkt G mit $G = d * Q'$. G darf nicht der Punkt im Unendlichen sein.
2. Wandle x_G in eine Zahl z um.
3. Gib z als gemeinsamen Schlüssel aus.

22.2 Generischer Teil

```
generic
  Size : Positive;
```

Exception $Size \neq 512 + i64 \quad (i \in \{0, \ldots, 8\})$: Constraint_Error;

22.3 API

22.3.1 Typen

```
type Public_Key_ECDH;
type Private_Key_ECDH is private;
type Shared_Key_ECDH;
```

Der *Public_Key_ECDH* ist ein beinhaltet folgende Variablen:

```
E : Elliptic_Curve_Zp;
P : EC_Point;
n : Big_Unsigned;
Q : EC_Point;
```

E ist eine elliptische Kurve üpber ZP. P ist ein öffentlicher Punkt auf dieser Kurve, dessen Ordnung n bekannt ist. Q ist der öffenliche Teil des Schlüssels.

Der *Private_Key_ECDH* beinhaltet folgende Variablen:

```
Q : EC_Point;
d : Big_Unsigned;
```

Q ist ein Punkt auf der Elliptischen Kurve und der öffenliche Teil des Schlüssels. n ist eine Zahl und der geheime Teil des Schlüssels.

Die *Shared_Key_ECDH* enthält einen Punkt.

```
W : EC_Point;
```

22.3.2 Prozeduren und Funktionen

```
procedure Gen_Public_Key(Public_Key_A : out Public_Key_ECDH;
                                 length
: in DB.Bit_Length);
```

Diese Prozedur holt eine Elliptische Kurve aus der Datenbank. Die Kurve hat mindestens die kryptografische Sicherheit von *length*.

Exception:
BitLength is not supported. (Max BitLength = 521) : LEN_EX

```
procedure Gen_Single_Private_Key(Public_Key_A
: in out Public_Key_ECDH;
                                 Private_Key_A : out
Private_Key_ECDH);
```

Diese Prozedur erzeugt ein Schlüsselpaar, den *Private_Key_A*. Hierzu wird eine zufällige Zahl generiert, diese ist der geheime Teil des Schlüssels. Des weiteren wird ein Punkt als öffentlicher Teil des Schlüssels, als Produkt aus dem eben generierten Zahl und dem öffentlichen Punkt des *Public_Key_A* berechnet. Dieser Punkt wird sowohl im öffentlichen als auch im privaten Schlüssel gespeichert.

```
procedure Gen_Shared_Private_Key(Public_Key_B : in
Public_Key_ECDH;
                                 Private_Key_A : in
Private_Key_ECDH;
                                 Shared_Key_A
: out Shared_Key_ECDH);
```

Diese Prozedur erzeugt einen öffentlichen Schlüssel, den *Shared_Key_A*. Hierzu wird der öffentliche Schlüssel von Public_Key_B mit dem geheimen Schlüssel vom Private_Key _A multiplitiert.

```
function Verify(Public_Key_A   : Public_Key_ECDH;
                Public_Key_B   : Public_Key_ECDH;
                Private_Key_A  : Private_Key_ECDH;
                Private_Key_B  : Private_Key_ECDH;
                Shared_Key_A   : Shared_Key_ECDH;
                Shared_Key_B   : Shared_Key_ECDH) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe sich auf Basis aller Informationen der der gleich geheime Schlüssel für beide Parteien berechnet werden konnte.

22.4 Anwendungsbeispiel

```
with Crypto.Types.Big_Numbers;
with Crypto.Asymmetric.ECDH;
with Ada.Text_IO; use Ada.Text_IO;

procedure Example_ECDH is
   package ECDH is new Crypto.Asymmetric.ECDH(512);
   use ECDH;

   Public_Key_A   : Public_Key_ECDH;
   Public_Key_B   : Public_Key_ECDH;
   Private_Key_A  : Private_Key_ECDH;
   Private_Key_B  : Private_Key_ECDH;
   Shared_Key_A   : Signature_ECDH;
   Shared_Key_B   : Signature_ECDH;

begin
   --Schluesselgenerierung
   Gen_Public_Key(Public_Key_A, 178);
   Gen_Single_Private_Key(Public_Key_A, Private_Key_A);

   Gen_Public_Key(Public_Key_B, 178);
   Gen_Single_Private_Key(Public_Key_B, Private_Key_B);

   --ECDH-Schluesselgenerierung
   Gen_Shared_Private_Key(Public_Key_B, Private_Key_A, Shared_Key_A);
   Gen_Shared_Private_Key(Public_Key_A, Private_Key_B, Shared_Key_B);
```

```
   -- Verifikation
   if Verify(Public_Key_A, Public_Key_B, Private_Key_A, Private_Key_B, Shar
         Put_Line("OK");
   else Put_Line("Implementation error.");
   end if;

end Example_ECDH;
```

Kapitel 23

Acl.Crypto.Symmetric.Algorithm.ECMQ

Bei diesem generischen Paket handelt es sich um eine Implementierung des ECMQV-Schlüsselgenerierungsalgorithmus. Der ECMQV-Algorithmus ist die Erweiterung des MQV-Algorithmus auf den Bereich der Elliptischen Kurven.
Mit Hilfe von ECMQV sind zwei Parteien in der Lage ein gemeinsamen geheimen Schlüssel aus den eigenen Schlüsselpaar und aus dem öffentlichen Schlüssel der anderen Partei zu erzeugen.
Dieser Algorithmus ist patentrechtlich geschützt zur Verwendung muss die Genehmigung Eigentümers eingeholt werden.

23.1 Mathematische Beschreibung

23.1.1 Voraussetzung

Eine Elliptische Kurve E der Form $y^2 = x^3 + a * x + bmodp$.
Ein Punkt P der auf der Kurve E liegt und von der Ordnung n ist.

Ein Schlüsselpaar besteht aus einem öffentlichen Punkt Q und einer geheimen Zahl d welche die Gleichung $Q = d * P$ erfüllen.

Es werden also folgende Parameter benötigt: a, b, p, P und n. Sowie zwei Schlüsselpaare (d_1, Q_1) und (d_2, Q_2).

Die öffentlichen Schlüssel der anderen Partei werden mit $'$ gekennzeichnet.

23.1.2 Schlüsselgenerierung

1. Berechne $h = 2^{(log_2 r)/2}$
2. Wandle x_{Q2} in eine Zahl i um.
3. Berechne $t = imodh$.

4. Berechne $t = t + h$.
5. Wandle x'_{Q2} in eine Zahl i' um.
6. Berechne $t' = i' mod h$.
7. Berechne $t' = t' + h$.
8. Berechne $e = t * d_1 + d_2 mod n$.
9. Berechne einen Punkt G mit $G = e(Q'_2 + t' * Q'_1)$. G darf nicht der Punkt im Unendlichen sein.
10. Wandle x_G in eine Zahl z um.
11. Gib z als gemeinsamen Schlüssel aus.

23.2 Generischer Teil

```
generic
   Size : Positive;
```

Exception $Size \neq 512 + i64 \quad (i \in \{0, \ldots, 8\})$: Constraint_Error;

23.3 API

23.3.1 Typen

```
type Public_Key_ECMQV;
type Private_Key_ECMQV is private;
type Shared_Key_ECMQV;
```

Der *Public_Key_ECMQV* ist ein beinhaltet folgende Variablen:

```
E : Elliptic_Curve_Zp;
P : EC_Point;
n : Big_Unsigned;
Q : EC_Point;
```

E ist eine elliptische Kurve üpber ZP. P ist ein öffentlicher Punkt auf dieser Kurve, dessen Ordnung n bekannt ist. Q ist der öffenliche Teil des Schlüssels.

Der *Private_Key_ECMQV* beinhaltet folgende Variablen:

```
Q : EC_Point;
d : Big_Unsigned;
```

Q ist ein Punkt auf der Elliptischen Kurve und der öffenliche Teil des Schlüssels.
n ist eine Zahl und der geheime Teil des Schlüssels.

Die *Shared_Key_ECMQV* enthält einen Punkt.

```
z : Big_Unsigned;
```

23.3.2 Prozeduren und Funktionen

```
procedure Gen_Single_Public_Key(Public_Key_A : out Public_Key_ECMQV;
                                length
: in DB.Bit_Length);
```

Diese Prozedur holt eine Elliptische Kurve aus der Datenbank. Die Kurve hat mindestens die kryptografische Sicherheit von *length*.

Exception:
BitLength is not supported. (Max BitLength = 521) : LEN_EX

```
procedure Gen_Public_Keys(Public_Key_A_1 : out Public_Key_ECMQV;
                          Public_Key_A_2 : out Public_Key_ECMQV;
                          length
: in DB.Bit_Length);
```

Diese Prozedur holt zwei Elliptische Kurve aus der Datenbank. Eine Kurve hat mindestens die kryptografische Sicherheit von *length*. Hierzu wird die Funktion *Gen_Single_Public_Key* genutzt.

Exception:
BitLength is not supported. (Max BitLength = 521) : LEN_EX

```
procedure Gen_Single_Private_Key(Public_Key_A
: in out Public_Key_ECMQV;
                                 Private_Key_A : out
Private_Key_ECMQV);
```

Diese Prozedur erzeugt ein Schlüsselpaar, den *Private_Key_A*. Hierzu wird eine zufällige Zahl generiert, diese ist der geheime Teil des Schlüssels. Des weiteren wird ein Punkt als öffentlicher Teil des Schlüssels, als Produkt aus dem eben generierten Zahl und dem öffentlichen Punkt des *Public_Key_A* berechnet. Dieser Punkt wird sowohl im öffentlichen als auch im privaten Schlüssel gespeichert.

```
procedure Gen_Private_Keys(Public_Key_A_1
: in out Public_Key_ECMQV;
                           Public_Key_A_2
: in out Public_Key_ECMQV
                           Private_Key_A_1 : out
Private_Key_ECMQV;
                           Private_Key_A_2 : out
Private_Key_ECMQV);
```

Diese Prozedur erzeugt zwei Schlüsselpaare *Private_Key_A_1* und *Private_Key_A_2*. Hierzu wird die Funktion *Gen_Single_Private_Key* genutzt.

```
procedure Gen_Shared_Private_Key(Public_Key_B_1 : in
Public_Key_ECMQV;
                                 Public_Key_B_2 : in
Public_Key_ECMQV;
                                 Private_Key_A_1 : in
Private_Key_ECMQV;
                                 Private_Key_A_2 : in
Private_Key_ECMQV;
                                 Shared_Key_A
: out Shared_Key_ECMQV);
```

Diese Prozedur erzeugt einen öffentlichen Schlüssel, den *Shared_Key_A* aus den Parametern.

```
function Verify(Public_Key_A_1  : Public_Key_ECMQV;
                Public_Key_A_2  : Public_Key_ECMQV;
                Public_Key_B_1  : Public_Key_ECMQV;
                Public_Key_B_2  : Public_Key_ECMQV;
                Private_Key_A_1 : Private_Key_ECMQV;
                Private_Key_A_2 : Private_Key_ECMQV;
                Private_Key_B_1 : Private_Key_ECMQV;
                Private_Key_B_2 : Private_Key_ECMQV;
                Shared_Key_A    : Shared_Key_ECMQV;
                Shared_Key_B    : Shared_Key_ECMQV) return Boolean;
```

Diese Funktion gibt den Wert "True" zurück, wenn mit Hilfe sich auf Basis aller Informationen der der gleich geheime Schlüssel für beide Parteien berechnet werden konnte.

23.4 Anwendungsbeispiel

```
with Crypto.Types.Big_Numbers;
with Crypto.Asymmetric.ECMQV;
with Ada.Text_IO; use Ada.Text_IO;

procedure Example_ECMQV is
   package ECMQV is new Crypto.Asymmetric.ECMQV(512);
   use ECMQV;

   Public_Key_A_1  : Public_Key_ECMQV;
   Public_Key_A_2  : Public_Key_ECMQV;
   Public_Key_B_1  : Public_Key_ECMQV;
   Public_Key_B_2  : Public_Key_ECMQV;
   Private_Key_A_1 : Private_Key_ECMQV;
   Private_Key_A_2 : Private_Key_ECMQV;
   Private_Key_B_1 : Private_Key_ECMQV;
   Private_Key_B_2 : Private_Key_ECMQV;
   Shared_Key_A    : Signature_ECMQV;
   Shared_Key_B    : Signature_ECMQV;

begin
   --Schluesselgenerierung
   Gen_Public_Keys(Public_Key_A_1, Public_Key_A_2, 178);
   Gen_Public_Keys(Public_Key_B_1, Public_Key_B_2, 178);
```

```
   Gen_Private_Keys(Public_Key_A_1, Public_Key_A_2,
                    Private_Key_A_1, Private_Key_A_2);
   Gen_Private_Keys(Public_Key_B_1, Public_Key_B_2,
                    Private_Key_B_1, Private_Key_B_2);

   --ECMQV-Schluesselgenerierung
   Gen_Shared_Private_Key(Public_Key_B_1, Public_Key_B_2,
                          Private_Key_A_1, Private_Key_A_2,
                          Shared_Key_A);
   Gen_Shared_Private_Key(Public_Key_A_1, Public_Key_A_2,
                          Private_Key_B_1, Private_Key_B_2,
                          Shared_Key_B);

   --Verifikation
   if Verify(Public_Key_A_1, Public_Key_A_2,
             Public_Key_B_1, Public_Key_B_2,
             Private_Key_A_1, Private_Key_A_2,
             Private_Key_B_1, Private_Key_B_2,
             Shared_Key_A,Shared_Key_B) then
      Put_Line("OK");
   else Put_Line("Implementation error.");
   end if;

end Example_ECMQV;
```

Kapitel 24

Acl.Crypto.Symmetric.Algorithm.ECIES

Bei diesem generischen Paket handelt es sich um eine Implementierung des ECIES-Verschlüsslungsalgorithmus. Der ECIES-Algorithmus ist die Erweiterung des IES-Algorithmus auf den Bereich der Elliptischen Kurven.
Mit Hilfe von ECIES sind zwei Parteien in der Lage Daten mittels einem gemeinsamen geheimen Schlüssel zu ver- und entschlüsseln. Um einen gemeinsamen Schlüssel zu erzeugen wird oft der ECDH-Algorithmus verwendet.

24.1 Beschreibung

24.1.1 Voraussetzung

Ein gemeinsamer Schlüssel der mittels ECDH berechnet wurde. In dieser Variante werden folgende Algorithmen/Procedere genutzt um ECIES zu realisieren.

- Punktkompression: **keine**
- Hashalgorithmus: **Sha-256**
- Verschlüsslungsalgorithmus: **AES-256**
- Mac-Algorithmus: **HMAC-Sha-256**

24.1.2 Verschlüsselung

1. Generiere aus dem gemeinsamen ECDH-Schlüssel einen Schlüssel für den Verschlüsslungsalgorithmus und einen Schlüssel für den Mac-Algorithmus. Unter Verwendung, der vorher festgelegten Hashfunktion.
2. Verschlüssel die Nachricht, unter Verwendung des erzeugten Verschlüsslungsschlüssels.
3. Berechne den Mac für die verschlüsselte Nachricht unter Verwendung des erzeugten Mac-Schlüssels.
4. Gib die verschlüsselte Nachricht und den zugehörigen Mac aus.

24.1.3 Entschlüsselung

1. Generiere aus dem gemeinsamen ECDH-Schlüssel einen Schlüssel für den Verschlüsslungsalgorithmus und einen Schlüssel für den Mac-Algorithmus. Unter Verwendung, der vorher festgelegten Hashfunktion.
2. Entschlüssel die Nachricht, unter Verwendung des erzeugten Verschlüsslungsschlüssels.
3. Berechne den Mac für die verschlüsselte Nachricht unter Verwendung des erzeugten Mac-Schlüssels.
4. Vergleiche den gegeben und den erzeugten Mac.
5. Sind die beiden Macs gleich, so gib die unverschlüsselte Nachricht aus. Ansonsten gib einen Fehler aus.

24.2 Generischer Teil

```
generic
   Size : Positive;
```

Exception $Size \neq 512 + i64 \quad (i \in \{0, \ldots, 8\})$: Constraint_Error;

24.3 API

24.3.1 Typen

```
type Cipher_ECIES is private;
type Plain_ECIES is private;
```

Der *Cipher_ECIES* ist ein beinhaltet folgende Variablen:

```
Public_Point          : EC_Point;
Message_Block_Count   : natural := 0;
Mac                   : W_Block256;
Cipher                : Unbounded_String;
Cipher_Map            : Container_Cipher_Map.Map;
```

Public_Point ist ein öffentlicher Schlüssel. *Message_Block_Count* ist der Zähler für die Anzahl der Nachrichtenblöcke. *Mac* ist der Mac über die verschlüsselten Nachrichtblöcke. *Cipher* ist die verschlüsselte Nachricht. *Cipher_Map* enthält die verschlüsselte Nachricht in Blöcke aufgeteilt. Als Schlüssel für die Map fungiert die Nummer des Nachrichtenblockes.

Der *Plain_ECIES* beinhaltet folgende Variablen:

```
Message_Block_Count : natural := 1;
AES_Key             : B_Block256;
Mac_Key             : W_Block512;
Message             : Unbounded_String;
Message_Map         : Container_Message_Map.Map;
```

Message_Block_Count ist der Zähler für die Anzahl der Nachrichtenblöcke. *AES_Key* ist der Schlüssel für den AES-256. *Mac_Key* ist der Schlüssel für den HMac-SHA-256. *Message* ist die unschlüsselte Nachricht. *Message_Map* enthält die unschlüsselte Nachricht in Blöcke aufgeteilt. Als Schlüssel für die Map fungiert die Nummer des Nachrichtenblockes.

24.3.2 Prozeduren und Funktionen

```
--internal purpose
procedure Message_Prepare(Plain   : out Plain_ECIES;
                          Message : in  String);
```

Diese Prozedur wandelt einen String in eine Map mit entsprechend vielen *B_Block128* um. Sie ist nur für interne Zwecke bestimmt.

```
--internal purpose
procedure Key_Prepare(AES_Key    : out B_Block256;
                      Mac_Key    : out W_Block512;
                      Shared_Key : in
Shared_Key_ECDH);
```

Diese Prozedur berechnet aus dem, vorher mittels ECDH erzeugten, *Shared_Key* sowohl den Schlüssel für AES-256 (*AES_Key*) als auch den für den HMAC-SHA-256 (*Mac_Key*). Sie ist nur für interne Zwecke bestimmt.

```
--internal purpose
procedure Mac_Compute(Mac_Key : in  W_Block512;
                      Cipher  : in  Cipher_ECIES;
                      Mac     : out W_Block256);
```

Diese Prozedur berechnet aus dem, vorher erzeugten (*Mac_Key*) einen HMAC-SHA-256 über die verschlüsselten Nachrichtenblöcke. Sie ist nur für interne Zwecke bestimmt.

```
procedure Encrypt(Public_Key_A : in  Public_Key_ECDH;
                  Shared_Key   : in  Shared_Key_ECDH;
                  Plaintext    : in  String;
                  Cipher       : out Cipher_ECIES);
```

Diese Prozedur verschlüsselt den *Plaintext* nach dem oben beschriebenen Prinzip von ECIES. Dies geschieht unter Verwendung des *Shared_Key*. Diese Prozedur kann verwendet werden, falls man bereits einen *Shared_Key* berechnet hat.

```
procedure Encrypt(Public_Key_A  : in  Public_Key_ECDH;
                  Public_Key_B  : in  Public_Key_ECDH;
                  Private_Key_A : in
Private_Key_ECDH;
                  Plaintext     : in  String;
                  Cipher        : out Cipher_ECIES);
```

Diese Prozedur verschlüsselt den *Plaintext* nach dem oben beschriebenen Prinzip von ECIES. Diese Prozedur berechnet zuerst den den gemeinsamen ECDH-Schlüssel aus dem *Public_Key_B* und dem *Private_Key_A*. Im Anschluss verwendet sie die andere *Encrypt* um zu entschlüsseln. Außerdem testet Sie die Parameter der beiden *Public_Key_ECDH* auf Gleichheit.

Exception:
Found different curve or domainparameter. : CURVE_EX

```
procedure Decrypt(Shared_Key : in  Shared_Key_ECDH;
                  Cipher     : in  Cipher_ECIES;
                  Plaintext  : out Unbounded_String);
```

Diese Prozedur entschlüsselt den *Cipher* nach dem oben beschriebenen Prinzip von ECIES. Dies geschieht unter Verwendung des *Shared_Key*. Diese Prozedur kann verwendet werden, falls man bereits einen *Shared_Key* berechnet hat.

Exception:
Found different MACs. : MAC_EX

```
   procedure Decrypt(Public_Key_B  : in  Public_Key_ECDH;
                     Private_Key_A : in
Private_Key_ECDH;
                     Cipher        : in  Cipher_ECIES;
                     Plaintext     : out Unbounded_String);
```

Diese Prozedur entschlüsselt den *Cipher* nach dem oben beschriebenen Prinzip von ECIES. Diese Prozedur berechnet zuerst den den gemeinsamen ECDH-Schlüssel aus dem *Public_Key_B* und dem *Private_Key_A*. Im Anschluss verwendet sie die andere *Decrypt* um zu verschlüsseln.

Exception:
Found different MACs. : MAC_EX

24.4 Anwendungsbeispiel

```
with Crypto.Symmetric.Algorithm.ECIES;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

procedure Example_ECIES is
   package ECIES is new  Crypto.Symmetric.Algorithm.ECIES(512);
   use ECIES;

   Public_Key_A  : ECDH.Public_Key_ECDH;
   Public_Key_B  : ECDH.Public_Key_ECDH;
   Private_Key_A : ECDH.Private_Key_ECDH;
   Private_Key_B : ECDH.Private_Key_ECDH;
   Cipher              : Cipher_ECIES;
      Input                  : String := "Hi there";
      Temp                   : Unbounded_String;
      Output                 : Unbounded_String;
   Result         : Boolean := False;

begin
   --ECDH
   Gen_Public_Key(Public_Key_A, 178);
   Gen_Single_Private_Key(Public_Key_A, Private_Key_A);

   Gen_Public_Key(Public_Key_B, 178);
```

```
      Gen_Single_Private_Key(Public_Key_B, Private_Key_B);

      --ECIES
            Encrypt(Public_Key_A, Public_Key_B, Private_Key_A, Input, Cipher);

      Decrypt(Public_Key_B, Private_Key_A, Cipher, Output);

            Append(Temp, Input);
            for I in 1 .. Input'Length loop
              if to_string(Temp)(I) = to_string(Output)(I) then
                Result := True;
                    else
                Result := False;
                    end if;
            end loop;

            if Result then Put_Line("ECIES - OK");
            else Put_Line("ECIES - Implementation error.")
            end if;

end Example_ECIES;
```

Kapitel 25

Crypto.Certificate

Mit Hilfe eines asymmetrischen Kryptosystems können Nachrichten in einem Netzwerk digital signiert und verschlüsselt werden. Für jede verschlüsselte Übermittlung benötigt der Sender allerdings den öffentlichen Schlüssel (Public-Key) des Empfängers. Um zu überprüfen, ob es sich tatsächlich um den Schlüssel des Empfängers handelt und nicht um eine Fälschung, benutzt man digitale Zertifikate, welche die Authentizität eines öffentlichen Schlüssels und seinen zulässigen Anwendungs- und Geltungsbereich bestätigen. X.509 ist derzeit der wichtigste Standard für digitale Zertifikate. Die aktuelle Version ist X.509v3.

25.1 X.509

Durch den Einsatz des Crypto.Certificate werden Zertifikate verifiziert und im Falle einer erfolgeichen Prüfung können sie für ihre Zwecke weiter benutzt werden. Andernfalls gilt das Zertifikat als nicht vertrauenswürdig und wird nicht weiter benutzt.
Unter solchen umständen sollten in Zukunft die Kinderzertifikate, also die Zertifikate die von einem hierarchisch höheren Signiert wurden, auch nicht mehr vertraut werden.

25.2 API

```
procedure Get_Cert(Filename: String);
```

Die Prozedur dient zum einlesen eines Zertifikats. Akzeptiert werden *.txt und *.cer Files. Allerdings können nur die CER-Files auf Echtheit überprüft werden. Beim Importieren wird die Struktur und die Gültigkeitsdauer des Zertifikats überprüft. Ist das Zertifikat noch aktuell, wird die Signatur mit dem öffentlichen Schlüssel des Austellers entschlüsselt und mit dem Zertifikat-Hash verglichen. Bei einem Wurzelzertifikat befindet sich der Public Key in dem selbigen. So wird die Korrektheit der Werte sichergestellt.

```
procedure CertOut;
```

Die Prozedur *CertOut* gibt das Zertifikat in Textform aus.

```
function AnalyseSigned return Unbounded_String;
```

Die Funktion *AnalyseSigned* entschlüsselt die Signatur des in *Get_Cert* Importierten Zertifikats und gibt einen Unbounded_String zurück. Man erhält bei erfolgreicher Dekodierung einen Hash-Wert (die Länge variiert je nach verwendeten Hash-Typ MD5 oder Sha-1). Ansonsten erhält man einen Null_Unbounded_String.

25.3 Exceptions

```
Cert_Typ_Error   :          exception;
```

Diese Ausnahme wird geworfen wenn es sich beim Importieren um ein falsches Format handelt.

```
Cert_Structure_Error : exception;
```

Diese Ausnahme wird geworfen wenn die Struktur des Zertifikats nicht stimmt.

```
ASN1_Structure_Error : exception;
```

Diese Ausnahme wird im Kinderpaket crypto.certificate.asn1 geworfen, wenn die Struktur der Binärdaten nicht nach dem asn1 Standart vorliegen.

```
Conversion_Error : exception;
```

Diese Ausnahme wird im Kinderpaket crypto.certificate.asn1 geworfen, wenn keine Binärdaten vorliegen.

25.4 Kinderpaket

Das Paket Crypto.Certificate.ASN1 dient lediglich zur Konvertierung eines Zertifikats im Format CER.
Es verfügt über folgende High-Level-API

```
procedure ASN1_Out;
```

Mit Hilfe dieser Prozedur wird das Zertifikat in Binär Darstellung ausgegeben. Es muss vorher ein Zertifikat mittels *Get_Cert(Filename: String)* eingelesen sein.

Die Low-Level API

```
procedure Parse(ASN1Data : Unbounded_String);;
```

Diese Prozedur wird direkt beim Einlesen eines Zertifikats im CER-Format benutzt und dient zum Entschlüsseln der Werte eines nach dem Standard ASN1 (mit Distinct Encoding Rules (DER)) binär codierten Zertifikat.

25.5 Anwendungsbeispiel

```
with Ada.Text_IO; use Ada.Text_IO;
with Crypto.Certificate;
use Crypto.Certificate;

procedure Example_Cert is

begin
   --Zertifikat importiern
   Get_Cert("1.cer");

   --Zertifikat ausgeben
   CertOut;

end Example_Cert;
```