Ada Crypto Library (ACL) - Documentation

Christian Forler

November 29, 2005

Contents

Chapter 1

Introduction

The Ada Crypto Library (ACL) is a free cryptographic library for Ada. One of the two main design objectives of this library was to create an **intuitive and clean API**. The other goal was to use a clean programming style in order to actively support and simplify formal code verification. Due to this, the ACL has been coded without the following 'features':

- access types (pointers)
- inline assembler
- tagged types (objects)
- goto statements

Goto statements and access types have been avoided as they make the source more complex and might cause problems during formal verification of the code. No tagged types have been used as procedures and methods may be overwritten and the actual method that is being utilised is determined during runtime (dynamic dispatching). This results in massive problems during formal code verification. Inline assembler has been avoided cause it involves a very unclean programming style and can be used for omitting the strict typing of ADA. Due to this restrictions, the actual code quality increases with respect to verification and security, however they also lead to decreased performance. If you favor performance over clean design, you should possibly look for another cryptographic library. Unfortunately, there are currently no other free cryptographic libraries for ADA I can refer to.
The ACL has the status of a draft. Neither a formal code review nor a code audit have been performed. At the time of writing, the author of the ACL is the only one who has dealt with the implementation's details. For instance, there might be the possibility that sensible material, such as the key, might still be resident within the RAM after a program using the ACL has been terminated. A storage pool dealing with this restriction is currently being developed. Most

of the other cryptographic libraries that share this limitation do not inform about the possible weakness. Those libraries should not be used as they can not be considered reliable.
The following paragraph subsumes the disadvantages of the ACL I know:

- missing cleanup of stack and heap
- no big endian support
- bad performance

This documentation subsumes the installation and the topological structure. Afterwards, it solely describes the API. Each package and it's API are introduced within a separate chapter. Each chapter concludes with an example.
If you should have any questions about the ACL, encounter a bug or want to contribute one or several packages, feel free to contact me via email at: crypto@shortie.inka.de.

1.0.1 TODO List

- optimisation
- formal proofs of correctness
- own storage pool
- translation of several packages to SPARK

1.1 Installation

1.1.1 Linux libacl-dev

On a Linux system, the following packages are required for installing the ACL:

- tar
- gnat
- make
- bunzip2
- binutils

Compilation

The following commands can be used for decompressing and compiling the ACL as well as the regression test included.

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

Testing

Before you actually install the ACL you should run the regression test in order to make sure the ACL properly works on your computer system. The regression test approximately takes 30 seconds on a PII 450. The following command sequence can be used for running the regression test.

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

Installation

If no errors were encountered during the regression test, the ACL can be installed by issuing the following command:

```
su -c ‘‘make install’’
```

De-installation

The following command can be used for removing the ACL:

```
su -c ‘‘make uninstall’’
```

Recompilation

The following commands can be used for recompiling the ACL:

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

Shared Library

The ACL may be installed as a shared library (libacl.so) with the following commands:

- `make shared`
- `make shared_install`

Shared Library de-installation

The shared library can be de-installed with the following command:

```
su -c ‘‘make shared_uninstall’’
```

Adaptations

The Makefile can be found in the base directory. It can be used for adapting the following variables:

- LIBDIR : installation path for the shared library.
- ACLDIR : installation path for the ACL.
- ACLDIR : installation path for the ACL documentation.

1.2 Layout of Directories and Packages

1.2.1 Directories

- doc : documentation
- ref : references and specifications
- src : source code
- test : regression test

1.2.2 Package Layout

The Ada Crypto Library (ACL) subsumes the following packages (components).

Crypto:
This is the root package of the ACL. All other ACL packages start with the *Crypto* prefix.

Crypto.Types:
The fundamental types of the ACL (such as Byte) and their corresponding basic functionality are located within this package. Utilisation of the ACL is very limited without including this package.

Crypto.Random:
This package implements an interface between an external random bit generator an the ACL.

Crypto.Symmetric:
This is the root package for the symmetric branch.

Crypto.Symmetric.Algorithm:
This branch of the ACL tree contains the symmetric algorithms such as DES.

Crypto.Symmetric.Algorithm.Oneway:
Each algorithm has an oneway algorithm. Symmetric oneway algorithms are symmetric algorithms that only work in one direction, and can either be used for en- or decryption of data.

Crypto.Symmetric.Blockcipher:
This generic package can be used for generating a block cipher out of a symmetric algorithm.

Crypto.Symmetric.Oneway_Blockcipher:
This package can be used for generating a oneway block cipher out of a symmetric oneway algorithm.

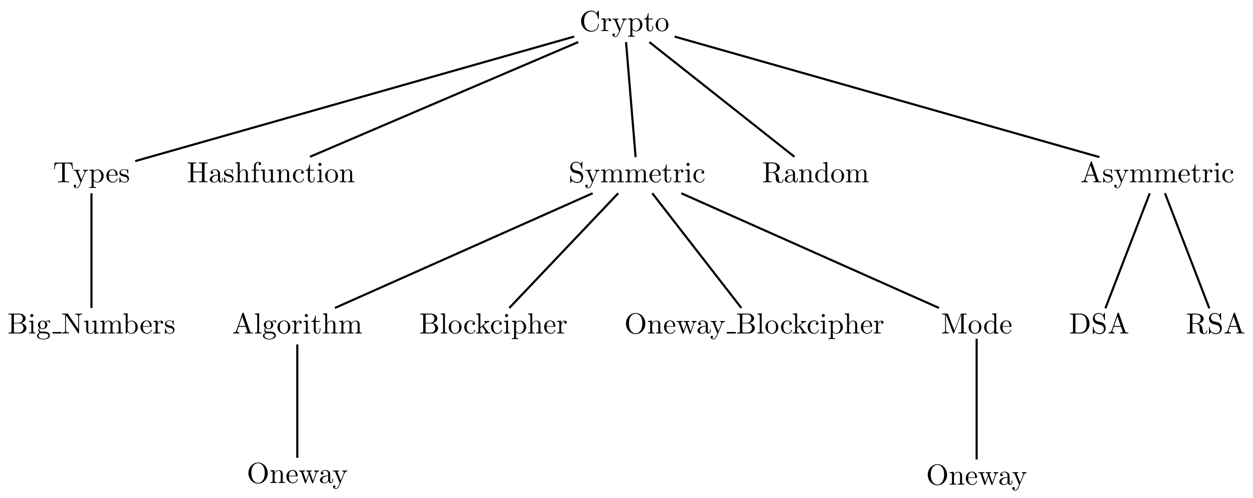
Crypto.Symmetric.Mode:
This branch subsumes multiple modes of operation for block ciphers.

Crypto.Symmetric.Mode.Oneway:
This branch subsumes multiple modes of operation for oneway block ciphers.

Crypto.Asymmetric:
This is the root package of the asymmetric branch.

Crypto.Asymmetric.DSA:
This package can be used for generating and verifying digital signatures.

Crypto.Asymmetric.RSA:
This package can be used for asymmetric en- and decryption of data.


Crypto
Types
Hashfunction
Symmetric
Random
Asymmetric
Big_Numbers
Algorithm
Blockcipher
Oneway_Blockcipher
Mode
DSA
RSA
Oneway
Oneway

Chapter 2

Crypto

Crypto is an empty root package. This package does not contain any code. It's sole purpose is to provide an unique namespace for all ACL packages. As all packages are descendants of this package, their names start with the prefix Crypto (e.g. Crypto.Foo.Bar).

Chapter 3

Crypto.Types

This package provides the fundamental types of the ACL and their base functions.

IMPORTANT:
When using the ACL you should in any case import this package via

```
with Crypto.Types;
```

3.1 Types

The following types are defined within this package.

3.1.1 Fundamental Types

The fundamental respective primary types are exclusively modular types, this means that an overflow or underrun of a variable does not lead to an exception. If the result of an operation is not within the scope of a modular type, $n := 2 ** Type'Size = Type'Last + 1$ will be added to or subtracted for it until the result is within the scope of the modular type.

```
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;
```

3.1.2 Example

```
with Ada.Text_IO;
with Crypto.Types;
procedure Example is
   use Crypto.Types;
   -- Byte has a scope of 0..255
   A, B : Byte;
begin
   A := 100;
   B := A + 250; -- Overflow at B
   A := A - 250; -- Underrun at A
   Ada.Text_IO.Put_Line("A:␣" & A'IMG);
   Ada.Text_IO.Put_Line("B:␣" & B'IMG);
end example;
```

Program output:

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

3.1.3 Derived Types

Derived types are types that are derived from elementary types. Regularly, these are arrays, consisting of elementary types. **All non-private arrays consisting of elementary types are interpreted as n-bit numbers within the ACL**, where the first element (First) is considered to be the most-significant and the last element (Last) the least-significant element of the number. This is a fundamental characteristic of the ACL.

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);
```

A B_BlockN always consists of a N bit array separated into N/8 Bytes. The subtype B_Block256 for example is a 256-bitstring, separated into 32 (258/8) byte blocks. In Ada, it can be represented by a byte array consisting of 32 elements.

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);
```

A W_BlockN always consists of a N bit array separated into N/32 Words. The subtype W_Block256 for example is a 256-bitstring, separated into 8 (256/32) Word elements. In Ada, it can be represented by a Word array consisting of 8 elements.

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);
```

A DW_BlockN always consists of a N bit array separated into N/64 Words. The subtype DW_Block256 for example is a 256-bitstring separated into 4 (256/64) DWord elements. In Ada, it can be represented by a DWord array with 4 elements.

Message blocks

```
subtype Message_Block_Length512  is Natural range 0 ..
63;
subtype Message_Block_Length1024 is Natural range 0 .. 127;
```

The Message_Block_Length types indicate the length of the actual message stored within a 512 or 1024 bit message block M in characters (Bytes). For example, splitting a 1152 bit message into 512 bit blocks results in three 512 bit blocks. The actual message length of the last block is 16 ($1280 - 2 \cdot 512 = 128/8 = 16$). The remaining 384 bits of the last message block are "empty", this means that they do not contain any part of the original message. A variable of type M_Length512 would be set to 32 in this case. As seen above, these variable types are used for message block padding. More information about padding can be found in the chapter about hash functions (**??**).

3.1.4 Bit-shifting Functions

```
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;
```

3.1.5 Basic Functions

XOR

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

Prerequisite:
Left'Length = Right'Length
Exception:
Prerequisite violation : Constraint_Byte/Word/DWord_Error

This functions perform a field-wise concatenation of the first two input fields by using the XOR operation. Left(Left'First) is XOR concatenated with Right(Right'First) and Left(Left'Last) with Right(Right'Last).

+

```
  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;
```

This function interprets Left and Right as numbers where Left(Left'First) and Right(Right'First) denotes the least significant Byte/Word/DWord and Left(Left'Last) and Right(Right'Last) denotes the most significant Byte/-Word/DWord of a number. The result of this function is the sum of the two numbers.

Example:

```
procedure plus is
  A : Byte := 200 -- = 2#11001000#
  B : Bytes(0..1) := (0 => 100, 1 => 16); -- = 2#10000_01100100#
  begin
  B := A+B; -- = 2#11001000#+2#10000_01100100# = 2#10001_00101100#
  -- B(0) = 2#101100# = 44
  -- B(1) = 2#010001# = 17
end plus;
```

3.1.6 Transformation Functions

Byte0

```
function Byte0 (W : Word) return Byte;
```

This function returns the most significant byte of W.

Byte1

```
function Byte1 (W : Word) return Byte;
```

Let $W = B0||B1||B2||B3$ and B0 be the most significant byte of W and B3 be the least significant byte, then this function returns B1.

Byte2

```
function Byte2 (W : Word) return Byte;
```

Let $W = B0||B1||B2||B3$ and B0 be the most significant byte of W and B3 be the least significant byte, then this function returns B2.

Byte3

```
function Byte3 (W : Word) return Byte;
```

This function returns the least significant byte of W.

Bytes_To_DWord

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

This function transfers a Byte_DWord to a DWord. X(Byte_Word'First) becomes the most significant byte of the DWord and X(Byte_Word'Last) its least significant byte.

Bytes_To_DWords

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

This function transforms a Byte_Array B into a DWord-Array. B is interpreted as a n-bit number, with B(B'First) as the most significant byte and B(B'Last) as the least significant byte. This number is converted into a DWord-Array, that is also being interpreted as a number with the first element (First) as most significant and the last element as least significant part of the number.

Example:

```
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#
```

Bytes_To_Word

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

This function transforms a Byte_Word into a Word. X(Byte_Word'First) becomes the most significant byte of the word and X(Byte_Word'Last) the least significant one.

Bytes_To_Word

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

This function transforms four Bytes (A, B, C, D) into a word. The first Byte (A) becomes the most significant Byte of the word, the last Byte (D) the least significant one.

Bytes_To_Words

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

This function transforms a Byte array B into a Word array W. B is interpreted as a number consisting of a least significant byte B(B'First) and most significant byte B(B'Last). This number is converted into a Word array, that is also being interpreted as a number with the first Element (First) as the most significant and the last element (Last) as the least significant part of the number.

Examples:
(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)

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

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

Bytes_To_String

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

This function transforms a Byte array into a String by interpreting the single contained bytes as ASCII-Code.

Characters_To_Word

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

This function transforms four Characters (A, B, C, D) into a Word. The first Character (A'Pos) becomes the most significant byte of the resulting Word, the last character (D'Pos) the least significant one.

DWord_To_Bytes

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

This function transforms a DWord X into a Byte_DWord B. The most significant Byte of X becomes B(B'First), the least significant one becomes B(B'Last).

Example:

```
D : DWord := 16#AA_BB_CC_DD_EE_FF#;
B : Byte_DWord := 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#00#; B(7) := 16#00#;
```

R_Bytes_To_DWord

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

This function transforms a Byte_DWord into a DWord. X(Byte_Word'First) becomes the least significant and X(Byte_Word'Last) the most significant byte of the result.

R_Bytes_To_Word

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

This function transforms a Byte_Word into a Word. X(Byte_Word'First) becomes the least significant and X(Byte_Word'Last) the most significant byte of the result.

R_DWord_To_Bytes

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

This function transforms a DWord X into a Byte_DWord B. The most significant byte of X becomes B(B'Last), the least significant byte of X becomes B(B'First).

R_Word_To_Bytes

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

This function transforms a Word X into a Byte_Word B. The most significant byte of X becomes B(B'Last), the least significant byte of X becomes B(B'First).

String_To_Bytes

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

This function transforms a (ASCII) String into a Byte array B. For all $I \in$ Message'Range: B(I) = Character'Pos(Message(I)).

Word_To_Bytes

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

This function transforms a Word X into a Byte_Word B. The most significant Byte of X becomes B(B'First) the least significant one becomes B(B'Last).

Words_To_Bytes

```
function Words_To_Bytes(Word_Array : Words) return Bytes;
```

This function transforms a Word array W into a Byte array B. The most significant byte of W(W'First) becomes B(B'First), the least significant byte W(W'Last) becomes B(B'Last).

3.1.7 Functions

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

These functions return “True” if all fields of the array parameter contain the value “0”. Otherwise, “False” is returned.

3.1.8 Padding Procedures

```
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);
```

Parameters:

Data: Data has to be an array containing a message. The message has to correspond to the following part of Data:
Data(Data’First)..Data(Data’First+(Message_Length-1))

Message_Length: The length of the message stored in Data.

Data2: Is used for some special cases. Regularly Is_Zero(Data2) = true after calling this function.

Prerequisites:

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

Postconditions:

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]

This functions add a number of null-Bytes/Words/Dwords to a message, followed by a Byte/Word/DWord representing the actual number of null-Bytes inserted. This might result in the following special cases:

1. Message_Length+1=Data'Length
   After adding a null Byte to the message, no additional space is available for appending the actual number of Null-Bytes/Words/DWords inserted to the message as Data(Data'Last) has been filled up by the null-Byte/Word/DWord. In order to solve this problem, all fields of Data2 except for Data2'Length are set to 0. Data2'Last is set to Data2'Length.

2. Message_Length=Data'Length
   This case is treated analogous to the first special case with the following deviations:

   (a) No null-Byte/Word/DWord is appended to the message

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

Example

```
...
  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;
...
```

Output:

- 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

Chapter 4

Crypto.Random

This package uses "dev/urandom/" as default. On Linux systems, "/dev/urandom" is a pseudo-device capable of producing cryptographically secure pseudo-random bits. If you should be using an operating system without such a pseudo-device, you can use Brian Warner's "Entropy Gathering Daemon"(EGD) that can be found at *http://www.lothar.com/tech/crypto/*. The EGD is a userspace implementation of the Linux kernel driver "dev/random/", that can be used for generating cryptographically secure pseudo-random bits.
The private part includes the "Path" variable. These variable contains the path to a file, a named pipe or a device storing cryptographically secure random bits. As a result, you can even link the ACL to a cryptographically secure pseudo-random bit source on non-POSIX compliant operating systems such as Windows XP.

4.1 API

```
procedure Read(B : out Byte);
```

This procedure fills a Byte with a random value.

```
procedure read(Byte_Array : out Bytes);
```

This procedure fills a Byte array with random values.

```
procedure Read(W : out Word);
```

This procedure fills a Word with a random value.

```
procedure Read(Word_Array : out Words);
```

This procedure fills a Word array with random values.

4.1.1 Exceptions

```
Random_Source_Does_Not_Exist_Error : exception;
```

This exception is raised if no (cryptographically secure) pseudo-random bit source could be found.

```
Random_Source_Read_Error : exception;
```

This exception is raised if a read error occurs while accessing the pseudo-random bit source.

Chapter 5

Crypto.Symmetric

Description

This is the root package for symmetric cryptography.

Function

This package provides the symmetric part of the ACL with direct access to Crypto.Types, which contains the fundamental types and corresponding base functions.

5.1 Logical Setup

A block cipher is separated into three logical layers. An application programmer should solely use the API of the upmost layer. Only with this API, a secure application of a block cipher can be guaranteed.

Layer 1: Algorithm

This layer provides the pure algorithm of a symmetric cipher. The API of the different algorithms is identical and provides as an interface for the block cipher layer. With each algorithm, a reference to the corresponding source (e.g. a paper) is provided. As a result, it is possible to verify the implementation by consulting this reference. Generally, this is always a good thing to do. Additional information regarding the particular algorithms and ciphers can be found within the next chapter.

Comment:
The API of this layer should only be used for generating a block cipher. It should by no means be used for any other purpose. There is no scenario where this would make any sense. This API only serves as an interface for layer 2.

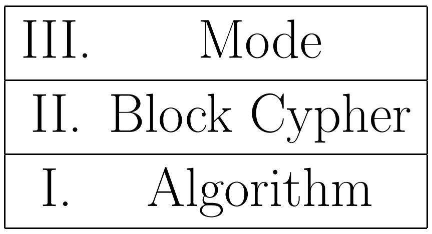


Figure 5.1: The three-layered architecture of a symmetric (block) cipher

Layer 2: Block Cypher

At this layer, a block cipher is generated from the algorithm in the insecure ECB-mode (Electronic Code Book). You should only use this API when you are familiar with symmetric block ciphers. Actually, this API serves as an interface for layer 3. Chapter **??** explains how a block cipher is generated from an algorithm.

Layer 3: Mode

This layer transforms a block cipher into a reasonable mode. There are different modes for different use cases, each of them having there particular assets and drawbacks. Chapter **??** provides a detailed description.

Comment

By using this three-layered architecture, it is also possible to implement stream ciphers. In order to do this, you have to provide the first layer with a procedure for receiving the next n bits of the key stream. Afterwards, you can construct an oneway block cipher (Chapter **??**) and and transform it to the oneway-OFB or oneway-counter mode. The ACL currently does not provide any stream cipher. However, there are plans for implementing the stream cipher Helix [1].

Bibliography

[1] Niels Ferguson, Doug Whiting, Bruce Schneier, Stefan Lucks John Kelsey, and Tadayoshi Kohno. *Helix - Fast Encryption and Authentication in a Single Cryptographic Primitive.*