crypto

crypto 2.0.1
September 13 2010

September 13 2010

1 User's Guide

The *Crypto* application provides functions for computation of message digests, and functions for encryption and decryption.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

For full OpenSSL and SSLeay license texts, see *Licenses*.

1.1 Licenses

This chapter contains in extenso versions of the OpenSSL and SSLeay licenses.

1.1.1 OpenSSL License

```
/* ====================================================================
 * Copyright (c) 1998-2002 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
```

```
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 * ====================================================================
 *
 * This product includes cryptographic software written by Eric Young
 * (eay@cryptsoft.com).  This product includes software written by Tim
 * Hudson (tjh@cryptsoft.com).
 *
 */
```

1.1.2 SSLeay License

```
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

1.1 Licenses

```
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

2 Reference Manual

The Crypto Application provides functions for computation of message digests, and encryption and decryption functions.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

For full OpenSSL and SSLeay license texts, see *Licenses*.

crypto

Application

The purpose of the Crypto application is to provide message digest and DES encryption for SMNPv3. It provides computation of message digests MD5 and SHA, and CBC-DES encryption and decryption.

Configuration

The following environment configuration parameters are defined for the Crypto application. Refer to application(3) for more information about configuration parameters.

`debug = true | false <optional>`

Causes debug information to be written to standard error or standard output. Default is `false`.

OpenSSL libraries

The current implementation of the Erlang Crypto application is based on the *OpenSSL* package version 0.9.7 or higher. There are source and binary releases on the web.

Source releases of OpenSSL can be downloaded from the **OpenSSL** project home page, or mirror sites listed there.

The same URL also contains links to some compiled binaries and libraries of OpenSSL (see the `Related/Binaries` menu) of which the **Shining Light Productions Win32 and OpenSSL** pages are of interest for the Win32 user.

For some Unix flavours there are binary packages available on the net.

If you cannot find a suitable binary OpenSSL package, you have to fetch an OpenSSL source release and compile it.

You then have to compile and install the library `libcrypto.so` (Unix), or the library `libeay32.dll` (Win32).

For Unix The `crypto_drv` dynamic driver is delivered linked to OpenSSL libraries in `/usr/local/lib`, but the default dynamic linking will also accept libraries in `/lib` and `/usr/lib`.

If that is not applicable to the particular Unix operating system used, the example `Makefile` in the Crypto `priv/obj` directory, should be used as a basis for relinking the final version of the port program.

For `Win32` it is only required that the library can be found from the `PATH` environment variable, or that they reside in the appropriate `SYSTEM32` directory; hence no particular relinking is need. Hence no example `Makefile` for Win32 is provided.

SEE ALSO

application(3)

crypto

Erlang module

This module provides a set of cryptographic functions.

References:

- md4: The MD4 Message Digest Algorithm (RFC 1320)
- md5: The MD5 Message Digest Algorithm (RFC 1321)
- sha: Secure Hash Standard (FIPS 180-2)
- hmac: Keyed-Hashing for Message Authentication (RFC 2104)
- des: Data Encryption Standard (FIPS 46-3)
- aes: Advanced Encryption Standard (AES) (FIPS 197)
- ecb, cbc, cfb, ofb: Recommendation for Block Cipher Modes of Operation (NIST SP 800-38A).
- rsa: Recommendation for Block Cipher Modes of Operation (NIST 800-38A)
- dss: Digital Signature Standard (FIPS 186-2)

The above publications can be found at **NIST publications**, at **IETF**.

Types

```
byte() = 0 ... 255
ioelem() = byte() | binary() | iolist()
iolist() = [ioelem()]
Mpint() = <<ByteLen:32/integer-big, Bytes:ByteLen/binary>>
```

Exports

start() -> ok

Starts the crypto server.

stop() -> ok

Stops the crypto server.

info() -> [atom()]

Provides the available crypto functions in terms of a list of atoms.

info_lib() -> [{Name,VerNum,VerStr}]

Types:

Name = binary()

VerNum = integer()

VerStr = binary()

Provides the name and version of the libraries used by crypto.

Name is the name of the library. VerNum is the numeric version according to the library's own versioning scheme. VerStr contains a text variant of the version.

```
> info_lib().
[{<<"OpenSSL">>,9469983,<<"OpenSSL 0.9.8a 11 Oct 2005">>}]
```

md4(Data) -> Digest

Types:

Data = iolist() | binary()

Digest = binary()

Computes an MD4 message digest from Data, where the length of the digest is 128 bits (16 bytes).

md4_init() -> Context

Types:

Context = binary()

Creates an MD4 context, to be used in subsequent calls to md4_update/2.

md4_update(Context, Data) -> NewContext

Types:

Data = iolist() | binary()

Context = NewContext = binary()

Updates an MD4 Context with Data, and returns a NewContext.

md4_final(Context) -> Digest

Types:

Context = Digest = binary()

Finishes the update of an MD4 Context and returns the computed MD4 message digest.

md5(Data) -> Digest

Types:

Data = iolist() | binary()

Digest = binary()

Computes an MD5 message digest from Data, where the length of the digest is 128 bits (16 bytes).

md5_init() -> Context

Types:

Context = binary()

Creates an MD5 context, to be used in subsequent calls to md5_update/2.

md5_update(Context, Data) -> NewContext

Types:

Data = iolist() | binary()

Context = NewContext = binary()

Updates an MD5 `Context` with `Data`, and returns a `NewContext`.

`md5_final(Context) -> Digest`

Types:

Context = Digest = binary()

Finishes the update of an MD5 `Context` and returns the computed `MD5` message digest.

`sha(Data) -> Digest`

Types:

Data = iolist() | binary()

Digest = binary()

Computes an `SHA` message digest from `Data`, where the length of the digest is 160 bits (20 bytes).

`sha_init() -> Context`

Types:

Context = binary()

Creates an SHA context, to be used in subsequent calls to `sha_update/2`.

`sha_update(Context, Data) -> NewContext`

Types:

Data = iolist() | binary()

Context = NewContext = binary()

Updates an SHA `Context` with `Data`, and returns a `NewContext`.

`sha_final(Context) -> Digest`

Types:

Context = Digest = binary()

Finishes the update of an SHA `Context` and returns the computed `SHA` message digest.

`md5_mac(Key, Data) -> Mac`

Types:

Key = Data = iolist() | binary()

Mac = binary()

Computes an `MD5 MAC` message authentification code from `Key` and `Data`, where the the length of the Mac is 128 bits (16 bytes).

`md5_mac_96(Key, Data) -> Mac`

Types:

Key = Data = iolist() | binary()

Mac = binary()

Computes an `MD5 MAC` message authentification code from `Key` and `Data`, where the length of the Mac is 96 bits (12 bytes).

`sha_mac(Key, Data) -> Mac`

Types:

Key = Data = iolist() | binary()

Mac = binary()

Computes an `SHA MAC` message authentification code from `Key` and `Data`, where the length of the Mac is 160 bits (20 bytes).

`sha_mac_96(Key, Data) -> Mac`

Types:

Key = Data = iolist() | binary()

Mac = binary()

Computes an `SHA MAC` message authentification code from `Key` and `Data`, where the length of the Mac is 96 bits (12 bytes).

`des_cbc_encrypt(Key, IVec, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts `Text` according to DES in CBC mode. `Text` must be a multiple of 64 bits (8 bytes). `Key` is the DES key, and `IVec` is an arbitrary initializing vector. The lengths of `Key` and `IVec` must be 64 bits (8 bytes).

`des_cbc_decrypt(Key, IVec, Cipher) -> Text`

Types:

Key = Cipher = iolist() | binary()

IVec = Text = binary()

Decrypts `Cipher` according to DES in CBC mode. `Key` is the DES key, and `IVec` is an arbitrary initializing vector. `Key` and `IVec` must have the same values as those used when encrypting. `Cipher` must be a multiple of 64 bits (8 bytes). The lengths of `Key` and `IVec` must be 64 bits (8 bytes).

`des_cbc_ivec(Data) -> IVec`

Types:

Data = iolist() | binary()

IVec = binary()

Returns the `IVec` to be used in a next iteration of `des_cbc_[encrypt|decrypt]`. `Data` is the encrypted data from the previous iteration step.

`des3_cbc_encrypt(Key1, Key2, Key3, IVec, Text) -> Cipher`

Types:

Key1 =Key2 = Key3 Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts `Text` according to DES3 in CBC mode. `Text` must be a multiple of 64 bits (8 bytes). `Key1`, `Key2`, `Key3`, are the DES keys, and `IVec` is an arbitrary initializing vector. The lengths of each of `Key1`, `Key2`, `Key3` and `IVec` must be 64 bits (8 bytes).

`des3_cbc_decrypt(Key1, Key2, Key3, IVec, Cipher) -> Text`

Types:

Key1 = Key2 = Key3 = Cipher = iolist() | binary()

IVec = Text = binary()

Decrypts `Cipher` according to DES3 in CBC mode. `Key1`, `Key2`, `Key3` are the DES key, and `IVec` is an arbitrary initializing vector. `Key1`, `Key2`, `Key3` and `IVec` must and `IVec` must have the same values as those used when encrypting. `Cipher` must be a multiple of 64 bits (8 bytes). The lengths of `Key1`, `Key2`, `Key3`, and `IVec` must be 64 bits (8 bytes).

`des_ecb_encrypt(Key, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

Cipher = binary()

Encrypts `Text` according to DES in ECB mode. `Key` is the DES key. The lengths of `Key` and `Text` must be 64 bits (8 bytes).

`des_ecb_decrypt(Key, Cipher) -> Text`

Types:

Key = Cipher = iolist() | binary()

Text = binary()

Decrypts `Cipher` according to DES in ECB mode. `Key` is the DES key. The lengths of `Key` and `Cipher` must be 64 bits (8 bytes).

`blowfish_ecb_encrypt(Key, Text) -> Cipher`
`blowfish_ecb_decrypt(Key, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts the first 64 bits of `Text` using Blowfish in ECB mode. `Key` is the Blowfish key. The length of `Text` must be at least 64 bits (8 bytes).

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Decrypts the first 64 bits of `Text` using Blowfish in ECB mode. `Key` is the Blowfish key. The length of `Text` must be at least 64 bits (8 bytes).

`blowfish_cbc_encrypt(Key, Text) -> Cipher`
`blowfish_cbc_decrypt(Key, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts `Text` using Blowfish in CBC mode. `Key` is the Blowfish key, and `IVec` is an arbitrary initializing vector. The length of `IVec` must be 64 bits (8 bytes). The length of `Text` must be a multiple of 64 bits (8 bytes).

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Decrypts `Text` using Blowfish in CBC mode. `Key` is the Blowfish key, and `IVec` is an arbitrary initializing vector. The length of `IVec` must be 64 bits (8 bytes). The length of `Text` must be a multiple 64 bits (8 bytes).

`blowfish_cfb64_encrypt(Key, IVec, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts `Text` using Blowfish in CFB mode with 64 bit feedback. `Key` is the Blowfish key, and `IVec` is an arbitrary initializing vector. The length of `IVec` must be 64 bits (8 bytes).

`blowfish_cfb64_decrypt(Key, IVec, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Decrypts `Text` using Blowfish in CFB mode with 64 bit feedback. `Key` is the Blowfish key, and `IVec` is an arbitrary initializing vector. The length of `IVec` must be 64 bits (8 bytes).

`blowfish_ofb64_encrypt(Key, IVec, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts `Text` using Blowfish in OFB mode with 64 bit feedback. `Key` is the Blowfish key, and `IVec` is an arbitrary initializing vector. The length of `IVec` must be 64 bits (8 bytes).

`aes_cfb_128_encrypt(Key, IVec, Text) -> Cipher`
`aes_cbc_128_encrypt(Key, IVec, Text) -> Cipher`

Types:

Key = Text = iolist() | binary()

IVec = Cipher = binary()

Encrypts `Text` according to AES in Cipher Feedback mode (CFB) or Cipher Block Chaining mode (CBC). `Text` must be a multiple of 128 bits (16 bytes). `Key` is the AES key, and `IVec` is an arbitrary initializing vector. The lengths of `Key` and `IVec` must be 128 bits (16 bytes).

`aes_cfb_128_decrypt(Key, IVec, Cipher) -> Text`
`aes_cbc_128_decrypt(Key, IVec, Cipher) -> Text`

Types:

Key = Cipher = iolist() | binary()

IVec = Text = binary()

Decrypts `Cipher` according to Cipher Feedback Mode (CFB) or Cipher Block Chaining mode (CBC). `Key` is the AES key, and `IVec` is an arbitrary initializing vector. `Key` and `IVec` must have the same values as those used when encrypting. `Cipher` must be a multiple of 128 bits (16 bytes). The lengths of `Key` and `IVec` must be 128 bits (16 bytes).

aes_cbc_ivec(Data) -> IVec

Types:

Data = iolist() | binary()

IVec = binary()

Returns the `IVec` to be used in a next iteration of `aes_cbc_*_[encrypt|decrypt]`. `Data` is the encrypted data from the previous iteration step.

erlint(Mpint) -> N
mpint(N) -> Mpint

Types:

Mpint = binary()

N = integer()

Convert a binary multi-precision integer `Mpint` to and from an erlang big integer. A multi-precision integer is a binary with the following form: `<<ByteLen:32/integer, Bytes:ByteLen/binary>>` where both `ByteLen` and `Bytes` are big-endian. Mpints are used in some of the functions in `crypto` and are not translated in the API for performance reasons.

rand_bytes(N) -> binary()

Types:

N = integer()

Generates N bytes randomly uniform 0..255, and returns the result in a binary. Uses the `crypto` library pseudo-random number generator.

rand_uniform(Lo, Hi) -> N

Types:

Lo, Hi, N = Mpint | integer()

Mpint = binary()

Generate a random number `N, Lo =< N < Hi.` Uses the `crypto` library pseudo-random number generator. The arguments (and result) can be either erlang integers or binary multi-precision integers.

mod_exp(N, P, M) -> Result

Types:

N, P, M, Result = Mpint

Mpint = binary()

This function performs the exponentiation `N ^ P mod M`, using the `crypto` library.

rsa_sign(Data, Key) -> Signature
rsa_sign(DigestType, Data, Key) -> Signature

Types:

Data = Mpint

Key = [E, N, D]

E, N, D = Mpint

Where `E` is the public exponent, `N` is public modulus and `D` is the private exponent.

DigestType = md5 | sha

The default `DigestType` is sha.

Mpint = binary()

Signature = binary()

Calculates a `DigestType` digest of the `Data` and creates a RSA signature with the private key `Key` of the digest.

rsa_verify(Data, Signature, Key) -> Verified
rsa_verify(DigestType, Data, Signature, Key) -> Verified

Types:

Verified = boolean()

Data, Signature = Mpint

Key = [E, N]

E, N = Mpint

Where `E` is the public exponent and `N` is public modulus.

DigestType = md5 | sha

The default `DigestType` is sha.

Mpint = binary()

Calculates a `DigestType` digest of the `Data` and verifies that the digest matches the RSA signature using the signer's public key `Key`.

rsa_public_encrypt(PlainText, PublicKey, Padding) -> ChipherText

Types:

PlainText = binary()

PublicKey = [E, N]

E, N = Mpint

Where `E` is the public exponent and `N` is public modulus.

Padding = rsa_pkcs1_padding | rsa_pkcs1_oaep_padding | rsa_no_padding

ChipherText = binary()

Encrypts the `PlainText` (usually a session key) using the `PublicKey` and returns the cipher. The `Padding` decides what padding mode is used, `rsa_pkcs1_padding` is PKCS #1 v1.5 currently the most used mode and `rsa_pkcs1_oaep_padding` is EME-OAEP as defined in PKCS #1 v2.0 with SHA-1, MGF1 and an empty encoding parameter. This mode is recommended for all new applications. The size of the `Msg` must be less than `byte_size(N)-11` if `rsa_pkcs1_padding` is used, `byte_size(N)-41` if `rsa_pkcs1_oaep_padding` is used and `byte_size(N)` if `rsa_no_padding` is used. Where byte_size(N) is the size part of an `Mpint-1`.

rsa_private_decrypt(ChipherText, PrivateKey, Padding) -> PlainText

Types:

ChipherText = binary()

PrivateKey = [E, N, D]

E, N, D = Mpint

Where `E` is the public exponent, `N` is public modulus and `D` is the private exponent.

Padding = rsa_pkcs1_padding | rsa_pkcs1_oaep_padding | rsa_no_padding

PlainText = binary()

Decrypts the `ChipherText` (usually a session key encrypted with *rsa_public_encrypt/3*) using the `PrivateKey` and returns the message. The `Padding` is the padding mode that was used to encrypt the data, see *rsa_public_encrypt/3*.

`rsa_private_encrypt(PlainText, PrivateKey, Padding) -> ChipherText`

Types:

PlainText = binary()

PrivateKey = [E, N, D]

E, N, D = Mpint

Where `E` is the public exponent, `N` is public modulus and `D` is the private exponent.

Padding = rsa_pkcs1_padding | rsa_no_padding

ChipherText = binary()

Encrypts the `PlainText` using the `PrivateKey` and returns the cipher. The `Padding` decides what padding mode is used, `rsa_pkcs1_padding` is PKCS #1 v1.5 currently the most used mode. The size of the `Msg` must be less than `byte_size(N)-11` if `rsa_pkcs1_padding` is used, and `byte_size(N)` if `rsa_no_padding` is used. Where byte_size(N) is the size part of an `Mpint-1`.

`rsa_public_decrypt(ChipherText, PublicKey, Padding) -> PlainText`

Types:

ChipherText = binary()

PublicKey = [E, N]

E, N = Mpint

Where `E` is the public exponent and `N` is public modulus

Padding = rsa_pkcs1_padding | rsa_no_padding

PlainText = binary()

Decrypts the `ChipherText` (encrypted with *rsa_private_encrypt/3*) using the `PrivateKey` and returns the message. The `Padding` is the padding mode that was used to encrypt the data, see *rsa_private_encrypt/3*.

`dss_sign(Data, Key) -> Signature`
`dss_sign(DigestType, Data, Key) -> Signature`

Types:

DigestType = sha | none (default is sha)

Data = Mpint | ShaDigest

Key = [P, Q, G, X]

P, Q, G, X = Mpint

Where `P`, `Q` and `G` are the dss parameters and `X` is the private key.

ShaDigest = binary() with length 20 bytes

Signature = binary()

Creates a DSS signature with the private key `Key` of a digest. If `DigestType` is 'sha', the digest is calculated as SHA1 of `Data`. If `DigestType` is 'none', `Data` is the precalculated SHA1 digest.

`dss_verify(Data, Signature, Key) -> Verified`
`dss_verify(DigestType, Data, Signature, Key) -> Verified`

Types:

Verified = boolean()

DigestType = sha | none

Data = Mpint | ShaDigest

Signature = Mpint

Key = [P, Q, G, Y]

P, Q, G, Y = Mpint

Where `P`, `Q` and `G` are the dss parameters and `Y` is the public key.

ShaDigest = binary() with length 20 bytes

Verifies that a digest matches the DSS signature using the public key `Key`. If `DigestType` is 'sha', the digest is calculated as SHA1 of `Data`. If `DigestType` is 'none', `Data` is the precalculated SHA1 digest.

`rc4_encrypt(Key, Data) -> Result`

Types:

Key, Data = iolist() | binary()

Result = binary()

Encrypts the data with RC4 symmetric stream encryption. Since it is symmetric, the same function is used for decryption.

`dh_generate_key(DHParams) -> {PublicKey,PrivateKey}`
`dh_generate_key(PrivateKey, DHParams) -> {PublicKey,PrivateKey}`

Types:

DHParameters = [P, G]

P, G = Mpint

Where `P` is the shared prime number and `G` is the shared generator.

PublicKey, PrivateKey = Mpint()

Generates a Diffie-Hellman `PublicKey` and `PrivateKey` (if not given).

`dh_compute_key(OthersPublicKey, MyPrivateKey, DHParams) -> SharedSecret`

Types:

DHParameters = [P, G]

P, G = Mpint

Where `P` is the shared prime number and `G` is the shared generator.

OthersPublicKey, MyPrivateKey = Mpint()

SharedSecret = binary()

Computes the shared secret from the private key and the other party's public key.

`exor(Data1, Data2) -> Result`

Types:

Data1, Data2 = iolist() | binary()

Result = binary()

Performs bit-wise XOR (exclusive or) on the data supplied.

DES in CBC mode

The Data Encryption Standard (DES) defines an algorithm for encrypting and decrypting an 8 byte quantity using an 8 byte key (actually only 56 bits of the key is used).

When it comes to encrypting and decrypting blocks that are multiples of 8 bytes various modes are defined (NIST SP 800-38A). One of those modes is the Cipher Block Chaining (CBC) mode, where the encryption of an 8 byte segment depend not only of the contents of the segment itself, but also on the result of encrypting the previous segment: the encryption of the previous segment becomes the initializing vector of the encryption of the current segment.

Thus the encryption of every segment depends on the encryption key (which is secret) and the encryption of the previous segment, except the first segment which has to be provided with an initial initializing vector. That vector could be chosen at random, or be a counter of some kind. It does not have to be secret.

The following example is drawn from the old FIPS 81 standard (replaced by NIST SP 800-38A), where both the plain text and the resulting cipher text is settled. The following code fragment returns `true'.

```
Key = <<16#01,16#23,16#45,16#67,16#89,16#ab,16#cd,16#ef>>,
IVec = <<16#12,16#34,16#56,16#78,16#90,16#ab,16#cd,16#ef>>,
P = "Now is the time for all ",
C = crypto:des_cbc_encrypt(Key, IVec, P),
   % Which is the same as
P1 = "Now is t", P2 = "he time ", P3 = "for all ",
C1 = crypto:des_cbc_encrypt(Key, IVec, P1),
C2 = crypto:des_cbc_encrypt(Key, C1, P2),
C3 = crypto:des_cbc_encrypt(Key, C2, P3),

C = <<C1/binary, C2/binary, C3/binary>>,
C = <<16#e5,16#c7,16#cd,16#de,16#87,16#2b,16#f2,16#7c,
       16#43,16#e9,16#34,16#00,16#8c,16#38,16#9c,16#0f,
       16#68,16#37,16#88,16#49,16#9a,16#7c,16#05,16#f6>>,
<<"Now is the time for all ">> ==
                  crypto:des_cbc_decrypt(Key, IVec, C).
```

The following is true for the DES CBC mode. For all decompositions `P1 ++ P2 = P` of a plain text message `P` (where the length of all quantities are multiples of 8 bytes), the encryption `C` of `P` is equal to `C1 ++ C2`, where `C1` is obtained by encrypting `P1` with `Key` and the initializing vector `IVec`, and where `C2` is obtained by encrypting `P2` with `Key` and the initializing vector `last8(C1)`, where `last(Binary)` denotes the last 8 bytes of the binary `Binary`.

Similarly, for all decompositions `C1 ++ C2 = C` of a cipher text message `C` (where the length of all quantities are multiples of 8 bytes), the decryption `P` of `C` is equal to `P1 ++ P2`, where `P1` is obtained by decrypting `C1` with `Key` and the initializing vector `IVec`, and where `P2` is obtained by decrypting `C2` with `Key` and the initializing vector `last8(C1)`, where `last8(Binary)` is as above.

For DES3 (which uses three 64 bit keys) the situation is the same.