crypto

crypto 3.6
June 23, 2015

June 23, 2015

1 Crypto 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) 1998-2011 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
```

```
 * 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 an Erlang API to cryptographic functions, see *crypto(3)*. Note that the API is on a fairly low level and there are some corresponding API functions available in *public_key(3)*, on a higher abstraction level, that uses the crypto application in its implementation.

DEPENDENCIES

The current crypto implementation uses nifs to interface OpenSSLs crypto library and requires *OpenSSL* package version 0.9.8 or higher.

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

SEE ALSO

application(3)

crypto

Erlang module

This module provides a set of cryptographic functions.

- Hash functions - **Secure Hash Standard**, **The MD5 Message Digest Algorithm (RFC 1321)** and **The MD4 Message Digest Algorithm (RFC 1320)**
- Hmac functions - **Keyed-Hashing for Message Authentication (RFC 2104)**
- Block ciphers - DES and AES in Block Cipher Modes - **ECB, CBC, CFB, OFB, CTR and GCM**
- **RSA encryption RFC 1321**
- Digital signatures **Digital Signature Standard (DSS)** and **Elliptic Curve Digital Signature Algorithm (ECDSA)**
- **Secure Remote Password Protocol (SRP - RFC 2945)**
- gcm: Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", National Institute of Standards and Technology SP 800- 38D, November 2007.

DATA TYPES

```
key_value()  = integer() | binary()
```

Always `binary()` when used as return value

```
rsa_public()  = [key_value()] = [E, N]
```

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

```
rsa_private() = [key_value()] = [E, N, D] | [E, N, D, P1, P2, E1, E2, C]
```

Where E is the public exponent, N is public modulus and D is the private exponent.The longer key format contains redundant information that will make the calculation faster. P1,P2 are first and second prime factors. E1,E2 are first and second exponents. C is the CRT coefficient. Terminology is taken from **RFC 3447**.

```
dss_public() = [key_value()] = [P, Q, G, Y]
```

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

```
dss_private() =  [key_value()] = [P, Q, G, X]
```

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

```
srp_public() = key_value()
```

Where is `A` or `B` from **SRP design**

```
srp_private() = key_value()
```

Where is a or b from **SRP design**

Where Verifier is v, Generator is g and Prime is N, DerivedKey is X, and Scrambler is u (optional will be generated if not provided) from **SRP design** Version = '3' | '6' | '6a'

```
dh_public() = key_value()
```

```
dh_private() = key_value()
```

```
dh_params() = [key_value()] = [P, G]
```

```
ecdh_public() = key_value()
```

```
ecdh_private() = key_value()
```

```
ecdh_params() =  ec_named_curve() | ec_explicit_curve()
```

```
ec_explicit_curve() =
    {ec_field(), Prime :: key_value(), Point :: key_value(), Order :: integer(), CoFactor :: none | integer()}
```

```
ec_field() = {prime_field, Prime :: integer()} |
    {characteristic_two_field, M :: integer(), Basis :: ec_basis()}
```

```
ec_basis() =  {tpbasis, K :: non_neg_integer()} |
    {ppbasis, K1 :: non_neg_integer(), K2 :: non_neg_integer(), K3 :: non_neg_integer()} |
    onbasis
```

```
ec_named_curve() ->
      sect571r1| sect571k1| sect409r1| sect409k1| secp521r1| secp384r1| secp224r1| secp224k1|
      secp192k1| secp160r2| secp128r2| secp128r1| sect233r1| sect233k1| sect193r2| sect193r1|
      sect131r2| sect131r1| sect283r1| sect283k1| sect163r2| secp256k1| secp160k1| secp160r1|
      secp112r2| secp112r1| sect113r2| sect113r1| sect239k1| sect163r1| sect163k1| secp256r1|
      secp192r1|
      brainpoolP160r1| brainpoolP160t1| brainpoolP192r1| brainpoolP192t1| brainpoolP224r1|
      brainpoolP224t1| brainpoolP256r1| brainpoolP256t1| brainpoolP320r1| brainpoolP320t1|
      brainpoolP384r1| brainpoolP384t1| brainpoolP512r1| brainpoolP512t1
```

Note that the *sect* curves are GF2m (characteristic two) curves and are only supported if the underlying OpenSSL has support for them. See also *crypto:supports/0*

```
stream_cipher() = rc4 | aes_ctr
```

```
block_cipher() =  aes_cbc128 | aes_cfb8 | aes_cfb128 | aes_ige256 | blowfish_cbc |
     blowfish_cfb64 | des_cbc | des_cfb | des3_cbc | des3_cbf
     | des_ede3 | rc2_cbc
```

```
aead_cipher() =  aes_gcm | chacha20_poly1305
```

```
stream_key() =  aes_key() | rc4_key()
```

```
block_key() =  aes_key() |  blowfish_key() | des_key()| des3_key()
```

```
aes_key() = iodata()
```

Key length is 128, 192 or 256 bits

```
rc4_key() = iodata()
```

Variable key length from 8 bits up to 2048 bits (usually between 40 and 256)

```
blowfish_key() = iodata()
```

Variable key length from 32 bits up to 448 bits

```
des_key() = iodata()
```

Key length is 64 bits (in CBC mode only 8 bits are used)

```
des3_key() = [binary(), binary(), binary()]
```

Each key part is 64 bits (in CBC mode only 8 bits are used)

```
digest_type() =  md5 | sha | sha224 | sha256 | sha384 | sha512
```

```
 hash_algorithms() =  md5 | ripemd160 | sha | sha224 | sha256 | sha384 | sha512
```

md4 is also supported for hash_init/1 and hash/2. Note that both md4 and md5 are recommended only for compatibility with existing applications.

```
 cipher_algorithms() = des_cbc | des_cfb |  des3_cbc | des3_cbf | des_ede3 |
     blowfish_cbc | blowfish_cfb64 | aes_cbc128 | aes_cfb8 | aes_cfb128| aes_cbc256 | aes_ige256 | aes_gcm |
```

```
 public_key_algorithms() =   rsa |dss | ecdsa | dh | ecdh | ec_gf2m
```

Note that ec_gf2m is not strictly a public key algorithm, but a restriction on what curves are supported with ecdsa and ecdh.

Exports

`block_encrypt(Type, Key, PlainText) -> CipherText`

Types:

Type = des_ecb | blowfish_ecb | aes_ecb
Key = block_key()
PlainText = iodata()

Encrypt `PlainText` according to `Type` block cipher.

May throw exception `notsup` in case the chosen `Type` is not supported by the underlying OpenSSL implementation.

`block_decrypt(Type, Key, CipherText) -> PlainText`

Types:

Type = des_ecb | blowfish_ecb | aes_ecb
Key = block_key()
PlainText = iodata()

Decrypt `CipherText` according to `Type` block cipher.

May throw exception `notsup` in case the chosen `Type` is not supported by the underlying OpenSSL implementation.

`block_encrypt(Type, Key, Ivec, PlainText) -> CipherText`
`block_encrypt(AeadType, Key, Ivec, {AAD, PlainText}) -> {CipherText, CipherTag}`

Types:

Type = block_cipher()
AeadType = aead_cipher()
Key = block_key()
PlainText = iodata()
AAD = IVec = CipherText = CipherTag = binary()

Encrypt `PlainText` according to `Type` block cipher. `IVec` is an arbitrary initializing vector.

In AEAD (Authenticated Encryption with Associated Data) mode, encrypt `PlainText`according to `Type` block cipher and calculate `CipherTag` that also authenticates the `AAD` (Associated Authenticated Data).

May throw exception `notsup` in case the chosen `Type` is not supported by the underlying OpenSSL implementation.

`block_decrypt(Type, Key, Ivec, CipherText) -> PlainText`
`block_decrypt(AeadType, Key, Ivec, {AAD, CipherText, CipherTag}) -> PlainText | error`

Types:

Type = block_cipher()
AeadType = aead_cipher()
Key = block_key()
PlainText = iodata()
AAD = IVec = CipherText = CipherTag = binary()

Decrypt `CipherText` according to `Type` block cipher. `IVec` is an arbitrary initializing vector.

In AEAD (Authenticated Encryption with Associated Data) mode, decrypt `CipherText`according to `Type` block cipher and check the authenticity the `PlainText` and `AAD` (Associated Authenticated Data) using the `CipherTag`. May return `error` if the decryption or validation fail's

May throw exception `notsup` in case the chosen `Type` is not supported by the underlying OpenSSL implementation.

`bytes_to_integer(Bin) -> Integer`

Types:

`Bin = binary() - as returned by crypto functions`

`Integer = integer()`

Convert binary representation, of an integer, to an Erlang integer.

`compute_key(Type, OthersPublicKey, MyKey, Params) -> SharedSecret`

Types:

`Type = dh | ecdh | srp`

`OthersPublicKey = dh_public() | ecdh_public() | srp_public()`

`MyKey = dh_private() | ecdh_private() | {srp_public(),srp_private()}`

`Params = dh_params() | ecdh_params() | SrpUserParams | SrpHostParams`

`SrpUserParams = {user, [DerivedKey::binary(), Prime::binary(), Generator::binary(), Version::atom() | [Scrambler:binary()]]}`

`SrpHostParams = {host, [Verifier::binary(), Prime::binary(), Version::atom() | [Scrambler::binary]]}`

`SharedSecret = binary()`

Computes the shared secret from the private key and the other party's public key. See also *public_key:compute_key/2*

`exor(Data1, Data2) -> Result`

Types:

`Data1, Data2 = iodata()`

`Result = binary()`

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

`generate_key(Type, Params) -> {PublicKey, PrivKeyOut}`
`generate_key(Type, Params, PrivKeyIn) -> {PublicKey, PrivKeyOut}`

Types:

`Type = dh | ecdh | srp`

`Params = dh_params() | ecdh_params() | SrpUserParams | SrpHostParams`

`SrpUserParams = {user, [Generator::binary(), Prime::binary(), Version::atom()]}`

`SrpHostParams = {host, [Verifier::binary(), Generator::binary(), Prime::binary(), Version::atom()]}`

`PublicKey = dh_public() | ecdh_public() | srp_public()`

`PrivKeyIn = undefined | dh_private() | ecdh_private() | srp_private()`

`PrivKeyOut = dh_private() | ecdh_private() | srp_private()`

Generates public keys of type `Type`. See also *public_key:generate_key/1*

`hash(Type, Data) -> Digest`

Types:

`Type = md4 | hash_algorithms()`

`Data = iodata()`

`Digest = binary()`

Computes a message digest of type `Type` from `Data`.

May throw exception `notsup` in case the chosen `Type` is not supported by the underlying OpenSSL implementation.

`hash_init(Type) -> Context`

Types:

`Type = md4 | hash_algorithms()`

Initializes the context for streaming hash operations. `Type` determines which digest to use. The returned context should be used as argument to *hash_update*.

May throw exception `notsup` in case the chosen `Type` is not supported by the underlying OpenSSL implementation.

`hash_update(Context, Data) -> NewContext`

Types:

`Data = iodata()`

Updates the digest represented by `Context` using the given `Data`. `Context` must have been generated using *hash_init* or a previous call to this function. `Data` can be any length. `NewContext` must be passed into the next call to `hash_update` or *hash_final*.

`hash_final(Context) -> Digest`

Types:

`Digest = binary()`

Finalizes the hash operation referenced by `Context` returned from a previous call to *hash_update*. The size of `Digest` is determined by the type of hash function used to generate it.

`hmac(Type, Key, Data) -> Mac`

`hmac(Type, Key, Data, MacLength) -> Mac`

Types:

`Type = hash_algorithms() - except ripemd160`

`Key = iodata()`

`Data = iodata()`

`MacLength = integer()`

`Mac = binary()`

Computes a HMAC of type `Type` from `Data` using `Key` as the authentication key.

`MacLength` will limit the size of the resultant `Mac`.

`hmac_init(Type, Key) -> Context`

Types:

`Type = hash_algorithms() - except ripemd160`

`Key = iodata()`

`Context = binary()`

Initializes the context for streaming HMAC operations. `Type` determines which hash function to use in the HMAC operation. `Key` is the authentication key. The key can be any length.

```
hmac_update(Context, Data) -> NewContext
```

Types:

Context = NewContext = binary()

Data = iodata()

Updates the HMAC represented by `Context` using the given `Data`. `Context` must have been generated using an HMAC init function (such as *hmac_init*). `Data` can be any length. `NewContext` must be passed into the next call to `hmac_update` or to one of the functions *hmac_final* and *hmac_final_n*

Warning:

Do not use a `Context` as argument in more than one call to hmac_update or hmac_final. The semantics of reusing old contexts in any way is undefined and could even crash the VM in earlier releases. The reason for this limitation is a lack of support in the underlying OpenSSL API.

```
hmac_final(Context) -> Mac
```

Types:

Context = Mac = binary()

Finalizes the HMAC operation referenced by `Context`. The size of the resultant MAC is determined by the type of hash function used to generate it.

```
hmac_final_n(Context, HashLen) -> Mac
```

Types:

Context = Mac = binary()

HashLen = non_neg_integer()

Finalizes the HMAC operation referenced by `Context`. `HashLen` must be greater than zero. `Mac` will be a binary with at most `HashLen` bytes. Note that if HashLen is greater than the actual number of bytes returned from the underlying hash, the returned hash will have fewer than `HashLen` bytes.

```
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">>}]
```

Note:

From OTP R16 the *numeric version* represents the version of the OpenSSL *header files* (`openssl/opensslv.h`) used when crypto was compiled. The text variant represents the OpenSSL library used at runtime. In earlier OTP versions both numeric and text was taken from the library.

`mod_pow(N, P, M) -> Result`

Types:

`N, P, M = binary() | integer()`

`Result = binary() | error`

Computes the function `N^P mod M`.

`next_iv(Type, Data) -> NextIVec`

`next_iv(Type, Data, IVec) -> NextIVec`

Types:

`Type = des_cbc | des3_cbc | aes_cbc | des_cfb`

`Data = iodata()`

`IVec = NextIVec = binary()`

Returns the initialization vector to be used in the next iteration of encrypt/decrypt of type `Type`. `Data` is the encrypted data from the previous iteration step. The `IVec` argument is only needed for `des_cfb` as the vector used in the previous iteration step.

`private_decrypt(Type, CipherText, PrivateKey, Padding) -> PlainText`

Types:

`Type = rsa`

`CipherText = binary()`

`PrivateKey = rsa_private()`

`Padding = rsa_pkcs1_padding | rsa_pkcs1_oaep_padding | rsa_no_padding`

`PlainText = binary()`

Decrypts the `CipherText`, encrypted with *public_encrypt/4* (or equivalent function) using the `PrivateKey`, and returns the plaintext (message digest). This is a low level signature verification operation used for instance by older versions of the SSL protocol. See also *public_key:decrypt_private/[2,3]*

`private_encrypt(Type, PlainText, PrivateKey, Padding) -> CipherText`

Types:

`Type = rsa`

`PlainText = binary()`

The size of the `PlainText` 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 N is public modulus of the RSA key.

`PrivateKey = rsa_private()`

`Padding = rsa_pkcs1_padding | rsa_no_padding`

`CipherText = binary()`

Encrypts the `PlainText` using the `PrivateKey` and returns the ciphertext. This is a low level signature operation used for instance by older versions of the SSL protocol. See also *public_key:encrypt_private/[2,3]*

`public_decrypt(Type, CipherText, PublicKey, Padding) -> PlainText`

Types:

`Type = rsa`

`CipherText = binary()`

`PublicKey = rsa_public()`

`Padding = rsa_pkcs1_padding | rsa_no_padding`

`PlainText = binary()`

Decrypts the `CipherText`, encrypted with *private_encrypt/4*(or equivalent function) using the `PrivateKey`, and returns the plaintext (message digest). This is a low level signature verification operation used for instance by older versions of the SSL protocol. See also *public_key:decrypt_public/[2,3]*

`public_encrypt(Type, PlainText, PublicKey, Padding) -> CipherText`

Types:

`Type = rsa`

`PlainText = binary()`

The size of the `PlainText` 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 N is public modulus of the RSA key.

`PublicKey = rsa_public()`

`Padding = rsa_pkcs1_padding | rsa_pkcs1_oaep_padding | rsa_no_padding`

`CipherText = binary()`

Encrypts the `PlainText` (message digest) using the `PublicKey` and returns the `CipherText`. This is a low level signature operation used for instance by older versions of the SSL protocol. See also *public_key:encrypt_public/[2,3]*

`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_seed(Seed) -> ok`

Types:

`Seed = binary()`

Set the seed for PRNG to the given binary. This calls the RAND_seed function from openssl. Only use this if the system you are running on does not have enough "randomness" built in. Normally this is when *stong_rand_bytes/1* returns `low_entropy`

`rand_uniform(Lo, Hi) -> N`

Types:

`Lo, Hi, N = integer()`

Generate a random number `N, Lo =< N < Hi.` Uses the `crypto` library pseudo-random number generator. `Hi` must be larger than `Lo`.

`sign(Algorithm, DigestType, Msg, Key) -> binary()`

Types:

Algorithm = rsa | dss | ecdsa

Msg = binary() | {digest,binary()}

The msg is either the binary "cleartext" data to be signed or it is the hashed value of "cleartext" i.e. the digest (plaintext).

DigestType = digest_type()

Key = rsa_private() | dss_private() | [ecdh_private(),ecdh_params()]

Creates a digital signature.

Algorithm `dss` can only be used together with digest type `sha`.

See also *public_key:sign/3*

`start() -> ok`

Equivalent to application:start(crypto).

`stop() -> ok`

Equivalent to application:stop(crypto).

`strong_rand_bytes(N) -> binary()`

Types:

N = integer()

Generates N bytes randomly uniform 0..255, and returns the result in a binary. Uses a cryptographically secure prng seeded and periodically mixed with operating system provided entropy. By default this is the `RAND_bytes` method from OpenSSL.

May throw exception `low_entropy` in case the random generator failed due to lack of secure "randomness".

`stream_init(Type, Key) -> State`

Types:

Type = rc4

State = opaque()

Key = iodata()

Initializes the state for use in RC4 stream encryption *stream_encrypt* and *stream_decrypt*

`stream_init(Type, Key, IVec) -> State`

Types:

Type = aes_ctr

State = opaque()

Key = iodata()

IVec = binary()

Initializes the state for use in streaming AES encryption using Counter mode (CTR). `Key` is the AES key and must be either 128, 192, or 256 bts long. `IVec` is an arbitrary initializing vector of 128 bits (16 bytes). This state is for use with *stream_encrypt* and *stream_decrypt*.

`stream_encrypt(State, PlainText) -> { NewState, CipherText}`

Types:

Text = iodata()

CipherText = binary()

Encrypts `PlainText` according to the stream cipher `Type` specified in stream_init/3. `Text` can be any number of bytes. The initial `State` is created using *stream_init*. `NewState` must be passed into the next call to `stream_encrypt`.

`stream_decrypt(State, CipherText) -> { NewState, PlainText }`

Types:

CipherText = iodata()

PlainText = binary()

Decrypts `CipherText` according to the stream cipher `Type` specified in stream_init/3. `PlainText` can be any number of bytes. The initial `State` is created using *stream_init*. `NewState` must be passed into the next call to `stream_decrypt`.

`supports() -> AlgorithmList`

Types:

AlgorithmList = [{hashs, [hash_algorithms()]}, {ciphers, [cipher_algorithms()]}, {public_keys, [public_key_algorithms()]}

Can be used to determine which crypto algorithms that are supported by the underlying OpenSSL library

`ec_curves() -> EllipticCurveList`

Types:

EllipticCurveList = [ec_named_curve()]

Can be used to determine which named elliptic curves are supported.

`ec_curve(NamedCurve) -> EllipticCurve`

Types:

NamedCurve = ec_named_curve()

EllipticCurve = ec_explicit_curve()

Return the defining parameters of a elliptic curve.

`verify(Algorithm, DigestType, Msg, Signature, Key) -> boolean()`

Types:

Algorithm = rsa | dss | ecdsa

Msg = binary() | {digest,binary()}

The msg is either the binary "cleartext" data or it is the hashed value of "cleartext" i.e. the digest (plaintext).

DigestType = digest_type()

Signature = binary()

Key = rsa_public() | dss_public() | [ecdh_public(),ecdh_params()]

Verifies a digital signature

Algorithm `dss` can only be used together with digest type `sha`.

See also *public_key:verify/4*