crypto

crypto 5.1.1
June 23, 2022

June 23, 2022

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 adhered 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 routines 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 publicly 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.]
 */
```

1.2 FIPS mode

This chapter describes FIPS mode support in the crypto application.

1.2.1 Background

OpenSSL can be built to provide FIPS 140-2 validated cryptographic services. It is not the OpenSSL application that is validated, but a special software component called the OpenSSL FIPS Object Module. However applications do not use this Object Module directly, but through the regular API of the OpenSSL library.

The crypto application supports using OpenSSL in FIPS mode. In this scenario only the validated algorithms provided by the Object Module are accessible, other algorithms usually available in OpenSSL (like md5) or implemented in the Erlang code (like SRP) are disabled.

1.2.2 Enabling FIPS mode

- Build or install the FIPS Object Module and a FIPS enabled OpenSSL library.

  You should read and precisely follow the instructions of the **Security Policy** and **User Guide**.

  > **Warning:**
  >
  > It is very easy to build a working OpenSSL FIPS Object Module and library from the source. However it **does not** qualify as FIPS 140-2 validated if the numerous restrictions in the Security Policy are not properly followed.

- Configure and build Erlang/OTP with FIPS support:

  ```
  $ cd $ERL_TOP
  $ ./otp_build configure --enable-fips
  ...
  checking for FIPS_mode_set... yes
  ...
  $ make
  ```

  If `FIPS_mode_set` returns `no` the OpenSSL library is not FIPS enabled and crypto won't support FIPS mode either.

- Set the `fips_mode` configuration setting of the crypto application to `true` **before loading the crypto module**.

  The best place is in the `sys.config` system configuration file of the release.

- Start and use the crypto application as usual. However take care to avoid the non-FIPS validated algorithms, they will all throw exception `not_supported`.

Entering and leaving FIPS mode on a node already running crypto is not supported. The reason is that OpenSSL is designed to prevent an application requesting FIPS mode to end up accidentally running in non-FIPS mode. If entering FIPS mode fails (e.g. the Object Module is not found or is compromised) any subsequent use of the OpenSSL API would terminate the emulator.

An on-the-fly FIPS mode change would thus have to be performed in a critical section protected from any concurrently running crypto operations. Furthermore in case of failure all crypto calls would have to be disabled from the Erlang or nif code. This would be too much effort put into this not too important feature.

1.2.3 Incompatibilities with regular builds

The Erlang API of the crypto application is identical regardless of building with or without FIPS support. However the nif code internally uses a different OpenSSL API.

This means that the context (an opaque type) returned from streaming crypto functions (`hash_(init|update|final)`, `hmac_(init|update|final)` and `stream_(init|encrypt|decrypt)`) is different and incompatible with regular builds when compiling crypto with FIPS support.

1.2.4 Common caveats

In FIPS mode non-validated algorithms are disabled. This may cause some unexpected problems in application relying on crypto.

Warning:

Do not try to work around these problems by using alternative implementations of the missing algorithms! An application can only claim to be using a FIPS 140-2 validated cryptographic module if it uses it exclusively for every cryptographic operation.

Restrictions on key sizes

Although public key algorithms are supported in FIPS mode they can only be used with secure key sizes. The Security Policy requires the following minimum values:

RSA
: 1024 bit

DSS
: 1024 bit

EC algorithms
: 160 bit

Restrictions on elliptic curves

The Erlang API allows using arbitrary curve parameters, but in FIPS mode only those allowed by the Security Policy shall be used.

Avoid md5 for hashing

Md5 is a popular choice as a hash function, but it is not secure enough to be validated. Try to use sha instead wherever possible.

For exceptional, non-cryptographic use cases one may consider switching to `erlang:md5/1` as well.

Certificates and encrypted keys

As md5 is not available in FIPS mode it is only possible to use certificates that were signed using sha hashing. When validating an entire certificate chain all certificates (including the root CA's) must comply with this rule.

For similar dependency on the md5 and des algorithms most encrypted private keys in PEM format do not work either. However, the PBES2 encryption scheme allows the use of stronger FIPS verified algorithms which is a viable alternative.

SNMP v3 limitations

It is only possible to use `usmHMACSHAAuthProtocol` and `usmAesCfb128Protocol` for authentication and privacy respectively in FIPS mode. The snmp application however won't restrict selecting disabled protocols in any way, and using them would result in run time crashes.

TLS 1.2 is required

All SSL and TLS versions prior to TLS 1.2 use a combination of md5 and sha1 hashes in the handshake for various purposes:

- Authenticating the integrity of the handshake messages.
- In the exchange of DH parameters in cipher suites providing non-anonymous PFS (perfect forward secrecy).
- In the PRF (pseud-random function) to generate keying materials in cipher suites not using PFS.

OpenSSL handles these corner cases in FIPS mode, however the Erlang crypto and ssl applications are not prepared for them and therefore you are limited to TLS 1.2 in FIPS mode.

On the other hand it worth mentioning that at least all cipher suites that would rely on non-validated algorithms are automatically disabled in FIPS mode.

Note:

Certificates using weak (md5) digests may also cause problems in TLS. Although TLS 1.2 has an extension for specifying which type of signatures are accepted, and in FIPS mode the ssl application will use it properly, most TLS implementations ignore this extension and simply send whatever certificates they were configured with.

1.3 Engine Load

This chapter describes the support for loading encryption engines in the crypto application.

1.3.1 Background

OpenSSL exposes an Engine API, which makes it possible to plug in alternative implementations for some or all of the cryptographic operations implemented by OpenSSL. When configured appropriately, OpenSSL calls the engine's implementation of these operations instead of its own.

Typically, OpenSSL engines provide a hardware implementation of specific cryptographic operations. The hardware implementation usually offers improved performance over its software-based counterpart, which is known as cryptographic acceleration.

Note:

The file name requirement on the engine dynamic library can differ between SSL versions.

1.3.2 Use Cases

Dynamically load an engine from default directory

If the engine is located in the OpenSSL/LibreSSL installation `engines` directory.

```
1> {ok, Engine} = crypto:engine_load(<<"otp_test_engine">>, [], []).
 {ok, #Ref}
```

Load an engine with the dynamic engine

Load an engine with the help of the dynamic engine by giving the path to the library.

```
 2> {ok, Engine} = crypto:engine_load(<<"dynamic">>,
                                      [{<<"SO_PATH">>,
                                        <<"/some/path/otp_test_engine.so">>},
                                       {<<"ID">>, <<"MD5">>},
                                       <<"LOAD">>],
                                      []).
 {ok, #Ref}
```

Load an engine and replace some methods

Load an engine with the help of the dynamic engine and just replace some engine methods.

```
 3> Methods = crypto:engine_get_all_methods() -- [engine_method_dh,engine_method_rand,
engine_method_ciphers,engine_method_digests, engine_method_store,
engine_method_pkey_meths, engine_method_pkey_asn1_meths].
[engine_method_rsa,engine_method_dsa,
 engine_method_ecdh,engine_method_ecdsa]
 4> {ok, Engine} = crypto:engine_load(<<"dynamic">>,
                                       [{<<"SO_PATH">>,
                                         <<"/some/path/otp_test_engine.so">>},
                                        {<<"ID">>, <<"MD5">>},
                                        <<"LOAD">>],
                                       [],
                       Methods).
 {ok, #Ref}
```

Load with the ensure loaded function

This function makes sure the engine is loaded just once and the ID is added to the internal engine list of OpenSSL. The following calls to the function will check if the ID is loaded and then just get a new reference to the engine.

```
 5> {ok, Engine} = crypto:ensure_engine_loaded(<<"MD5">>,
                                                <<"/some/path/otp_test_engine.so">>).
 {ok, #Ref}
```

To unload it use crypto:ensure_engine_unloaded/1 which removes the ID from the internal list before unloading the engine.

```
 6> crypto:ensure_engine_unloaded(<<"MD5">>).
 ok
```

List all engines currently loaded

```
 5> crypto:engine_list().
[<<"dynamic">>, <<"MD5">>]
```

1.4 Engine Stored Keys

This chapter describes the support in the crypto application for using public and private keys stored in encryption engines.

1.4.1 Background

OpenSSL exposes an Engine API, which makes it possible to plug in alternative implementations for some of the cryptographic operations implemented by OpenSSL. See the chapter Engine Load for details and how to load an Engine.

An engine could among other tasks provide a storage for private or public keys. Such a storage could be made safer than the normal file system. Those techniques are not described in this User's Guide. Here we concentrate on how to use private or public keys stored in such an engine.

The storage engine must call `ENGINE_set_load_privkey_function` and `ENGINE_set_load_pubkey_function`. See the OpenSSL cryptolib's **manpages**.

OTP/Crypto requires that the user provides two or three items of information about the key. The application used by the user is usually on a higher level, for example in SSL. If using the crypto application directly, it is required that:

- an Engine is loaded, see the chapter on Engine Load or the Reference Manual
- a reference to a key in the Engine is available. This should be an Erlang string or binary and depends on the Engine loaded

- an Erlang map is constructed with the Engine reference, the key reference and possibly a key passphrase if needed by the Engine. See the Reference Manual for details of the map.

1.4.2 Use Cases

Sign with an engine stored private key

This example shows how to construct a key reference that is used in a sign operation. The actual key is stored in the engine that is loaded at prompt 1.

```
1> {ok, EngineRef} = crypto:engine_load(....).
...
{ok,#Ref<0.2399045421.3028942852.173962>}
2> PrivKey = #{engine => EngineRef,
               key_id => "id of the private key in Engine"}.
...
3> Signature = crypto:sign(rsa, sha, <<"The message">>, PrivKey).
<<65,6,125,254,54,233,84,77,83,63,168,28,169,214,121,76,
  207,177,124,183,156,185,160,243,36,79,125,230,231,...>>
```

Verify with an engine stored public key

Here the signature and message in the last example is verifyed using the public key. The public key is stored in an engine, only to exemplify that it is possible. The public key could of course be handled openly as usual.

```
4> PublicKey = #{engine => EngineRef,
                 key_id => "id of the public key in Engine"}.
...
5> crypto:verify(rsa, sha, <<"The message">>, Signature, PublicKey).
true
6>
```

Using a password protected private key

The same example as the first sign example, except that a password protects the key down in the Engine.

```
6> PrivKeyPwd = #{engine => EngineRef,
                  key_id => "id of the pwd protected private key in Engine",
    password => "password"}.
...
7> crypto:sign(rsa, sha, <<"The message">>, PrivKeyPwd).
<<140,80,168,101,234,211,146,183,231,190,160,82,85,163,
  175,106,77,241,141,120,72,149,181,181,194,154,175,76,
  223,...>>
8>
```

1.5 Algorithm Details

This chapter describes details of algorithms in the crypto application.

The tables only documents the supported cryptos and key lengths. The user should not draw any conclusion on security from the supplied tables.

1.5.1 Ciphers

A cipher in the new api is categorized as either cipher_no_iv(), cipher_iv() or cipher_aead(). The letters IV are short for *Initialization Vector* and AEAD is an abbreviation of *Authenticated Encryption with Associated Data*.

Due to irregular naming conventions, some cipher names in the old api are substituted by new names in the new api. For a list of retired names, see Retired cipher names.

To dynamically check availability, check that the name in the *Cipher and Mode* column is present in the list returned by crypto:supports(ciphers).

Ciphers without an IV - cipher_no_iv()

To be used with:

- crypto_one_time/4
- crypto_init/3

The ciphers are:

Cipher and Mode	**Key length [bytes]**	**Block size [bytes]**
`aes_128_ecb`	16	16
`aes_192_ecb`	24	16
`aes_256_ecb`	32	16
`blowfish_ecb`	16	8
`des_ecb`	8	8
`rc4`	16	1

Table 5.1: Ciphers without IV

Ciphers with an IV - cipher_iv()

To be used with:

- crypto_one_time/5
- crypto_init/4
- crypto_dyn_iv_init/3

The ciphers are:

Cipher and Mode	**Key length [bytes]**	**IV length [bytes]**	**Block size [bytes]**	**Limited to OpenSSL versions**
`aes_128_cbc`	16	16	16	
`aes_192_cbc`	24	16	16	
`aes_256_cbc`	32	16	16	
`aes_128_cfb8`	16	16	1	
`aes_192_cfb8`	24	16	1	
`aes_256_cfb8`	32	16	1	

aes_128_cfb128	16	16	1	
aes_192_cfb128	24	16	1	
aes_256_cfb128	32	16	1	
aes_128_ctr	16	16	1	
aes_192_ctr	24	16	1	
aes_256_ctr	32	16	1	
aes_128_ofb	16	16	1	
aes_192_ofb	24	16	1	
aes_256_ofb	32	16	1	
blowfish_cbc	16	8	8	
blowfish_cfb64	16	8	1	
blowfish_ofb64	16	8	1	
chacha20	32	16	1	#1.1.0d
des_cbc	8	8	8	
des_ede3_cbc	24	8	8	
des_cfb	8	8	1	
des_ede3_cfb	24	8	1	
rc2_cbc	16	8	8	

Table 5.2: Ciphers with IV

Ciphers with AEAD - cipher_aead()

To be used with:

- crypto_one_time_aead/6
- crypto_one_time_aead/7

The ciphers are:

Cipher and Mode	Key length [bytes]	IV length [bytes]	AAD length [bytes]	Tag length [bytes]	Block size [bytes]	Limited to OpenSSL versions
aes_128_ccm	16	7-13	any	even 4-16 default: 12	any	#1.0.1

aes_192_ccm	24	7-13	any	even 4-16 default: 12	any	#1.0.1
aes_256_ccm	32	7-13	any	even 4-16 default: 12	any	#1.0.1
aes_128_gcm	16	#1	any	1-16 default: 16	any	#1.0.1
aes_192_gcm	24	#1	any	1-16 default: 16	any	#1.0.1
aes_256_gcm	32	#1	any	1-16 default: 16	any	#1.0.1
chacha20_poly1305	32	1-16	any	16	any	#1.1.0

Table 5.3: AEAD ciphers

1.5.2 Message Authentication Codes (MACs)

To be used in mac/4 and related functions.

CMAC

CMAC with the following ciphers are available with OpenSSL 1.0.1 or later if not disabled by configuration.

To dynamically check availability, check that the name `cmac` is present in the list returned by crypto:supports(macs). Also check that the name in the *Cipher and Mode* column is present in the list returned by crypto:supports(ciphers).

Cipher and Mode	**Key length [bytes]**	**Max Mac Length (= default length) [bytes]**
aes_128_cbc	16	16
aes_192_cbc	24	16
aes_256_cbc	32	16
aes_128_ecb	16	16
aes_192_ecb	24	16
aes_256_ecb	32	16
blowfish_cbc	16	8
blowfish_ecb	16	8
des_cbc	8	8
des_ecb	8	8

des_ede3_cbc	24	8
rc2_cbc	16	8

Table 5.4: CMAC cipher key lengths

HMAC

Available in all OpenSSL compatible with Erlang CRYPTO if not disabled by configuration.

To dynamically check availability, check that the name `hmac` is present in the list returned by crypto:supports(macs) and that the hash name is present in the list returned by crypto:supports(hashs).

Hash	**Max Mac Length (= default length) [bytes]**
sha	20
sha224	28
sha256	32
sha384	48
sha512	64
sha3_224	28
sha3_256	32
sha3_384	48
sha3_512	64
blake2b	64
blake2s	32
md4	16
md5	16
ripemd160	20

Table 5.5: HMAC output sizes

POLY1305

POLY1305 is available with OpenSSL 1.1.1 or later if not disabled by configuration.

To dynamically check availability, check that the name `poly1305` is present in the list returned by crypto:supports(macs).

The poly1305 mac wants an 32 bytes key and produces a 16 byte MAC by default.

1.5.3 Hash

To dynamically check availability, check that the wanted name in the *Names* column is present in the list returned by crypto:supports(hashs).

Type	**Names**	**Limitated to OpenSSL versions**
SHA1	sha	
SHA2	sha224, sha256, sha384, sha512	
SHA3	sha3_224, sha3_256, sha3_384, sha3_512	#1.1.1
MD4	md4	
MD5	md5	
RIPEMD	ripemd160	

Table 5.6:

1.5.4 Public Key Cryptography

RSA

RSA is available with all OpenSSL versions compatible with Erlang CRYPTO if not disabled by configuration. To dynamically check availability, check that the atom `rsa` is present in the list returned by crypto:supports(public_keys).

Warning:

The RSA options are experimental.

The exact set of options and there syntax **may** be changed without prior notice.

Option	**sign/verify**	**public encrypt private decrypt**	**private encrypt public decrypt**
{rsa_padding,rsa_x931_padding}	x		x
{rsa_padding,rsa_pkcs1_padding}	x	x	x
{rsa_padding,rsa_pkcs1_pss_padding} {rsa_pss_saltlen, -2..} {rsa_mgf1_md, atom()}	x (2) x (2) x (2)		
{rsa_padding,rsa_pkcs1_oaep_padding} {rsa_mgf1_md, atom()}		x (2) x (2) x (3)	

{rsa_oaep_label, binary()}} {rsa_oaep_md, atom()}		x (3)	
{rsa_padding,rsa_no_padding}	x (1)		

Table 5.7:

Notes:

- (1) OpenSSL # 1.0.0
- (2) OpenSSL # 1.0.1
- (3) OpenSSL # 1.1.0

DSS

DSS is available with OpenSSL versions compatible with Erlang CRYPTO if not disabled by configuration. To dynamically check availability, check that the atom `dss` is present in the list returned by crypto:supports(public_keys).

ECDSA

ECDSA is available with OpenSSL 0.9.8o or later if not disabled by configuration. To dynamically check availability, check that the atom `ecdsa` is present in the list returned by crypto:supports(public_keys). If the atom `ec_gf2m` also is present, the characteristic two field curves are available.

The actual supported named curves could be checked by examining the list returned by crypto:supports(curves).

EdDSA

EdDSA is available with OpenSSL 1.1.1 or later if not disabled by configuration. To dynamically check availability, check that the atom `eddsa` is present in the list returned by crypto:supports(public_keys).

Support for the curves ed25519 and ed448 is implemented. The actual supported named curves could be checked by examining the list with the list returned by crypto:supports(curves).

Diffie-Hellman

Diffie-Hellman computations are available with OpenSSL versions compatible with Erlang CRYPTO if not disabled by configuration. To dynamically check availability, check that the atom `dh` is present in the list returned by crypto:supports(public_keys).

Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman is available with OpenSSL 0.9.8o or later if not disabled by configuration. To dynamically check availability, check that the atom `ecdh` is present in the list returned by crypto:supports(public_keys).

The Edward curves `x25519` and `x448` are supported with OpenSSL 1.1.1 or later if not disabled by configuration.

The actual supported named curves could be checked by examining the list returned by crypto:supports(curves).

1.6 New and Old API

This chapter describes the new api to encryption and decryption.

1.6.1 Background

The CRYPTO app has evolved during its lifetime. Since also the OpenSSL cryptolib has changed the API several times, there are parts of the CRYPTO app that uses a very old one internally and other parts that uses the latest one. The internal definitions of e.g cipher names was a bit hard to maintain.

It turned out that using the old api in the new way (more about that later), and still keep it backwards compatible, was not possible. Specially as more precision in the error messages is desired it could not be combined with the old standard.

Therefore the old api (see next section) is kept for now but internally implemented with new primitives.

1.6.2 The old API

The old functions - deprecated from 23.0 and removed from OTP 24.0 - are for ciphers:

- `block_encrypt/3`
- `block_encrypt/4`
- `block_decrypt/3`
- `block_decrypt/4`
- `stream_init/2`
- `stream_init/3`
- `stream_encrypt/2`
- `stream_decrypt/2`
- `next_iv/2`
- `next_iv/3`

for lists of supported algorithms:

- `supports/0`

and for MACs (Message Authentication Codes):

- `cmac/3`
- `cmac/4`
- `hmac/3`
- `hmac/4`
- `hmac_init/2`
- `hmac_update/2`
- `hmac_final/1`
- `hmac_final_n/2`
- `poly1305/2`

1.6.3 The new API

Encryption and decryption

The new functions for encrypting or decrypting one single binary are:

- crypto_one_time/4
- crypto_one_time/5
- crypto_one_time_aead/6
- crypto_one_time_aead/7

In those functions the internal crypto state is first created and initialized with the cipher type, the key and possibly other data. Then the single binary is encrypted or decrypted, the crypto state is de-allocated and the result of the crypto operation is returned.

The `crypto_one_time_aead` functions are for the ciphers of mode `ccm` or `gcm`, and for the cipher `chacha20-poly1305`.

For repeated encryption or decryption of a text divided in parts, where the internal crypto state is initialized once, and then many binaries are encrypted or decrypted with the same state, the functions are:

- crypto_init/4
- crypto_init/3
- crypto_update/2
- crypto_final/1

The `crypto_init` initialies an internal cipher state, and one or more calls of `crypto_update` does the actual encryption or decryption. Note that AEAD ciphers can't be handled this way due to their nature.

For repeated encryption or decryption of a text divided in parts where the same cipher and same key is used, but a new initialization vector (nounce) should be applied for each part, the functions are:

- crypto_dyn_iv_init/3
- crypto_dyn_iv_update/3

An example of where those functions are needed, is when handling the TLS protocol.

If padding was not enabled, the call to crypto_final/1 may be excluded.

For information about available algorithms, use:

- supports/1
- hash_info/1
- cipher_info/1

The `next_iv/2` and `next_iv/3` are not needed since the `crypto_init` and `crypto_update` includes this functionality.

MACs (Message Authentication Codes)

The new functions for calculating a MAC of a single piece of text are:

- mac/3
- mac/4
- macN/4
- macN/5

For calculating a MAC of a text divided in parts use:

- mac_init/2
- mac_init/3
- mac_update/2
- mac_final/1
- mac_finalN/2

1.6.4 Examples of the new api

Examples of crypto_init/4 and crypto_update/2

The functions crypto_init/4 and crypto_update/2 are intended to be used for encrypting or decrypting a sequence of blocks. First one call of `crypto_init/4` initialises the crypto context. One or more calls `crypto_update/2` does the actual encryption or decryption for each block.

This example shows first the encryption of two blocks and then decryptions of the cipher text, but divided into three blocks just to show that it is possible to divide the plain text and cipher text differently for some ciphers:

```
1> crypto:start().
ok
2> Key = <<1:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1>>
3> IV = <<0:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>>
4> StateEnc = crypto:crypto_init(aes_128_ctr, Key, IV, true). % encrypt -> true
#Ref<0.3768901617.1128660993.124047>
5> crypto:crypto_update(StateEnc, <<"First bytes">>).
<<67,44,216,166,25,130,203,5,66,6,162>>
6> crypto:crypto_update(StateEnc, <<"Second bytes">>).
<<16,79,94,115,234,197,94,253,16,144,151,41>>
7>
7> StateDec = crypto:crypto_init(aes_128_ctr, Key, IV, false). % decrypt -> false
#Ref<0.3768901617.1128660994.124255>
8> crypto:crypto_update(StateDec, <<67,44,216,166,25,130,203>>).
<<"First b">>
9> crypto:crypto_update(StateDec, <<5,66,6,162,16,79,94,115,234,197,
       94,253,16,144,151>>).
<<"ytesSecond byte">>
10> crypto:crypto_update(StateDec, <<41>>).
<<"s">>
11>
```

Note that the internal data that the `StateEnc` and `StateDec` references are destructivly updated by the calls to crypto_update/2. This is to gain time in the calls of the nifs interfacing the cryptolib. In a loop where the state is saved in the loop's state, it also saves one update of the loop state per crypto operation.

For example, a simple server receiving text parts to encrypt and send the result back to the one who sent them (the `Requester`):

```
encode(Crypto, Key, IV) ->
crypto_loop(crypto:crypto_init(Crypto, Key, IV, true)).

crypto_loop(State) ->
receive
       {Text, Requester} ->
       Requester ! crypto:crypto_update(State, Text),
loop(State)
end.
```

Example of crypto_one_time/5

The same example as in the previous section, but now with one call to crypto_one_time/5:

```
1> Key = <<1:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1>>
2> IV = <<0:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>>
3> Txt = [<<"First bytes">>,<<"Second bytes">>].
[<<"First bytes">>,<<"Second bytes">>]
4> crypto:crypto_one_time(aes_128_ctr, Key, IV, Txt, true).
<<67,44,216,166,25,130,203,5,66,6,162,16,79,94,115,234,
197,94,253,16,144,151,41>>
5>
```

The `[<<"First bytes">>,<<"Second bytes">>]` could of course have been one single binary: `<<"First bytesSecond bytes">>`.

Example of crypto_one_time_aead/6

The same example as in the previous section, but now with one call to crypto_one_time_aead/6:

```
1> Key = <<1:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1>>
2> IV = <<0:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>>
3> Txt = [<<"First bytes">>,<<"Second bytes">>].
[<<"First bytes">>,<<"Second bytes">>]
4> AAD = <<"Some bytes">>.
<<"Some bytes">>
5> crypto:crypto_one_time_aead(aes_128_gcm, Key, IV, Txt, AAD, true).
{<<240,130,38,96,130,241,189,52,3,190,179,213,132,1,72,
192,103,176,90,104,15,71,158>>,
<<131,47,45,91,142,85,9,244,21,141,214,71,31,135,2,155>>}
9>
```

The `[<<"First bytes">>,<<"Second bytes">>]` could of course have been one single binary: `<<"First bytesSecond bytes">>`.

Example of mac_init mac_update and mac_final

```
1> Key = <<1:128>>.
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1>>
2> StateMac = crypto:mac_init(cmac, aes_128_cbc, Key).
#Ref<0.2424664121.2781478916.232610>
3> crypto:mac_update(StateMac, <<"First bytes">>).
#Ref<0.2424664121.2781478916.232610>
4> crypto:mac_update(StateMac, " ").
#Ref<0.2424664121.2781478916.232610>
5> crypto:mac_update(StateMac, <<"last bytes">>).
#Ref<0.2424664121.2781478916.232610>
6> crypto:mac_final(StateMac).
<<68,191,219,128,84,77,11,193,197,238,107,6,214,141,160,
249>>
7>
```

and compare the result with a single calculation just for this example:

```
7> crypto:mac(cmac, aes_128_cbc, Key, "First bytes last bytes").
<<68,191,219,128,84,77,11,193,197,238,107,6,214,141,160,
249>>
8> v(7) == v(6).
true
9>
```

1.6.5 Retired cipher names

This table lists the retired cipher names in the first column and suggests names to replace them with in the second column.

The new names follows the OpenSSL libcrypto names. The format is ALGORITM_KEYSIZE_MODE.

Examples of algorithms are aes, chacha20 and des. The keysize is the number of bits and examples of the mode are cbc, ctr and gcm. The mode may be followed by a number depending on the mode. An example is the ccm mode which has a variant called ccm8 where the so called tag has a length of eight bits.

The old names had by time lost any common naming convention which the new names now introduces. The new names include the key length which improves the error checking in the lower levels of the crypto application.

Instead of:	**Use:**
`aes_cbc128`	`aes_128_cbc`
`aes_cbc256`	`aes_256_cbc`
`aes_cbc`	`aes_128_cbc, aes_192_cbc, aes_256_cbc`
`aes_ccm`	`aes_128_ccm, aes_192_ccm, aes_256_ccm`
`aes_cfb128`	`aes_128_cfb128, aes_192_cfb128, aes_256_cfb128`
`aes_cfb8`	`aes_128_cfb8, aes_192_cfb8, aes_256_cfb8`
`aes_ctr`	`aes_128_ctr, aes_192_ctr, aes_256_ctr`
`aes_gcm`	`aes_128_gcm, aes_192_gcm, aes_256_gcm`
`des3_cbc`	`des_ede3_cbc`
`des3_cbf`	`des_ede3_cfb`
`des3_cfb`	`des_ede3_cfb`
`des_ede3`	`des_ede3_cbc`
`des_ede3_cbf`	`des_ede3_cfb`

Table 6.1:

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 may work with limited functionality with as old versions as **OpenSSL** 0.9.8c. FIPS mode support requires at least version 1.0.1 and a FIPS capable OpenSSL installation. We recommend using a version that is officially supported by the OpenSSL project. API compatible backends like LibreSSL should also work.

The crypto app is tested daily with at least one version of each of the OpenSSL 1.0.1, 1.0.2, 1.1.0, 1.1.1 and 3.0. FIPS mode is also tested for 1.0.1 and 1.0.2.

Note:

Compiling, linking and running with OpenSSL 3.0 works from OTP-25.0. However, we do not yet recommend it for other purposes than testing, since the crypto application is not yet extensively tested with that OpenSSL version.

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

CONFIGURATION

The following configuration parameters are defined for the crypto application. See `app(3)` for more information about configuration parameters.

`fips_mode = boolean()`

Specifies whether to run crypto in FIPS mode. This setting will take effect when the nif module is loaded. If FIPS mode is requested but not available at run time the nif module and thus the crypto module will fail to load. This mechanism prevents the accidental use of non-validated algorithms.

`rand_cache_size = integer()`

Sets the cache size in bytes to use by `crypto:rand_seed_alg(crypto_cache)` and `crypto:rand_seed_alg_s(crypto_cache)`. This parameter is read when a seed function is called, and then kept in generators state object. It has a rather small default value that causes reads of strong random bytes about once per hundred calls for a random value. The set value is rounded up to an integral number of words of the size these seed functions use.

SEE ALSO

application(3)

crypto

Erlang module

This module provides a set of cryptographic functions.

Hash functions

SHA1, SHA2
: **Secure Hash Standard [FIPS PUB 180-4]**

SHA3
: **SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions [FIPS PUB 202]**

BLAKE2
: **BLAKE2 — fast secure hashing**

MD5
: **The MD5 Message Digest Algorithm [RFC 1321]**

MD4
: **The MD4 Message Digest Algorithm [RFC 1320]**

MACs - Message Authentication Codes

Hmac functions
: **Keyed-Hashing for Message Authentication [RFC 2104]**

Cmac functions
: **The AES-CMAC Algorithm [RFC 4493]**

POLY1305
: **ChaCha20 and Poly1305 for IETF Protocols [RFC 7539]**

Symmetric Ciphers

DES, 3DES and AES
: **Block Cipher Techniques [NIST]**

Blowfish
: **Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994, pp. 191-204.**

Chacha20
: **ChaCha20 and Poly1305 for IETF Protocols [RFC 7539]**

Chacha20_poly1305
: **ChaCha20 and Poly1305 for IETF Protocols [RFC 7539]**

Modes

ECB, CBC, CFB, OFB and CTR
: **Recommendation for Block Cipher Modes of Operation: Methods and Techniques [NIST SP 800-38A]**

GCM
: **Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC [NIST SP 800-38D]**

CCM
: **Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality [NIST SP 800-38C]**

Asymmetric Ciphers - Public Key Techniques

RSA
: **PKCS #1: RSA Cryptography Specifications [RFC 3447]**

DSS
: **Digital Signature Standard (DSS) [FIPS 186-4]**

ECDSA
: **Elliptic Curve Digital Signature Algorithm [ECDSA]**

SRP
: **The SRP Authentication and Key Exchange System [RFC 2945]**

> **Note:**
>
> The actual supported algorithms and features depends on their availability in the actual libcrypto used. See the crypto (App) about dependencies.
>
> Enabling FIPS mode will also disable algorithms and features.

The CRYPTO User's Guide has more information on FIPS, Engines and Algorithm Details like key lengths.

Data Types

Ciphers

```
cipher() = cipher_no_iv() | cipher_iv() | cipher_aead()
```

```
cipher_no_iv() =
    aes_128_ecb | aes_192_ecb | aes_256_ecb | aes_ecb |
    blowfish_ecb | des_ecb | rc4
```

```
cipher_iv() =
    aes_128_cbc | aes_192_cbc | aes_256_cbc | aes_cbc |
    aes_128_ofb | aes_192_ofb | aes_256_ofb | aes_128_cfb128 |
    aes_192_cfb128 | aes_256_cfb128 | aes_cfb128 | aes_128_cfb8 |
    aes_192_cfb8 | aes_256_cfb8 | aes_cfb8 | aes_128_ctr |
    aes_192_ctr | aes_256_ctr | aes_ctr | blowfish_cbc |
    blowfish_cfb64 | blowfish_ofb64 | chacha20 | des_ede3_cbc |
    des_ede3_cfb | des_cbc | des_cfb | rc2_cbc
```

```
cipher_aead() =
    aes_128_ccm | aes_192_ccm | aes_256_ccm | aes_ccm |
    aes_128_gcm | aes_192_gcm | aes_256_gcm | aes_gcm |
    chacha20_poly1305
```

Ciphers known by the CRYPTO application.

Note that this list might be reduced if the underlying libcrypto does not support all of them.

```
crypto_opts() = boolean() | [crypto_opt()]
```

```
crypto_opt() = {encrypt, boolean()} | {padding, padding()}
```

Selects encryption (`{encrypt,true}`) or decryption (`{encrypt,false}`).

```
padding() = cryptolib_padding() | otp_padding()
```

This option handles padding in the last block. If not set, no padding is done and any bytes in the last unfilled block is silently discarded.

```
cryptolib_padding() = none | pkcs_padding
```

The `cryptolib_padding` are paddings that may be present in the underlying cryptolib linked to the Erlang/OTP crypto app.

For OpenSSL, see the **OpenSSL documentation**. and find `EVP_CIPHER_CTX_set_padding()` in cryptolib for your linked version.

```
otp_padding() = zero | random
```

Erlang/OTP adds a either padding of zeroes or padding with random bytes.

Digests and hash

```
hash_algorithm() =
    sha1() |
    sha2() |
    sha3() |
    blake2() |
    ripemd160 |
    compatibility_only_hash()
```

```
hmac_hash_algorithm() =
    sha1() | sha2() | sha3() | compatibility_only_hash()
```

```
cmac_cipher_algorithm() =
    aes_128_cbc | aes_192_cbc | aes_256_cbc | aes_cbc |
    aes_128_cfb128 | aes_192_cfb128 | aes_256_cfb128 |
    aes_cfb128 | aes_128_cfb8 | aes_192_cfb8 | aes_256_cfb8 |
    aes_cfb8 | blowfish_cbc | des_cbc | des_ede3_cbc | rc2_cbc
```

```
rsa_digest_type() = sha1() | sha2() | md5 | ripemd160
```

```
dss_digest_type() = sha1() | sha2()
```

```
ecdsa_digest_type() = sha1() | sha2()
```

```
sha1() = sha
```

```
sha2() = sha224 | sha256 | sha384 | sha512
```

```
sha3() = sha3_224 | sha3_256 | sha3_384 | sha3_512
```

```
blake2() = blake2b | blake2s
```

```
compatibility_only_hash() = md5 | md4
```

The `compatibility_only_hash()` algorithms are recommended only for compatibility with existing applications.

Elliptic Curves

```
ec_named_curve() =
    brainpoolP160r1 | brainpoolP160t1 | brainpoolP192r1 |
    brainpoolP192t1 | brainpoolP224r1 | brainpoolP224t1 |
    brainpoolP256r1 | brainpoolP256t1 | brainpoolP320r1 |
    brainpoolP320t1 | brainpoolP384r1 | brainpoolP384t1 |
    brainpoolP512r1 | brainpoolP512t1 | c2pnb163v1 | c2pnb163v2 |
    c2pnb163v3 | c2pnb176v1 | c2pnb208w1 | c2pnb272w1 |
    c2pnb304w1 | c2pnb368w1 | c2tnb191v1 | c2tnb191v2 |
    c2tnb191v3 | c2tnb239v1 | c2tnb239v2 | c2tnb239v3 |
    c2tnb359v1 | c2tnb431r1 | ipsec3 | ipsec4 | prime192v1 |
    prime192v2 | prime192v3 | prime239v1 | prime239v2 |
    prime239v3 | prime256v1 | secp112r1 | secp112r2 | secp128r1 |
    secp128r2 | secp160k1 | secp160r1 | secp160r2 | secp192k1 |
```

```
    secp192r1 | secp224k1 | secp224r1 | secp256k1 | secp256r1 |
    secp384r1 | secp521r1 | sect113r1 | sect113r2 | sect131r1 |
    sect131r2 | sect163k1 | sect163r1 | sect163r2 | sect193r1 |
    sect193r2 | sect233k1 | sect233r1 | sect239k1 | sect283k1 |
    sect283r1 | sect409k1 | sect409r1 | sect571k1 | sect571r1 |
    wtls1 | wtls10 | wtls11 | wtls12 | wtls3 | wtls4 | wtls5 |
    wtls6 | wtls7 | wtls8 | wtls9
```

```
edwards_curve_dh() = x25519 | x448
```

```
edwards_curve_ed() = ed25519 | ed448
```

Note that some curves are disabled if FIPS is enabled.

```
ec_explicit_curve() =
    {Field :: ec_field(),
     Curve :: ec_curve(),
     BasePoint :: binary(),
     Order :: binary(),
     CoFactor :: none | binary()}
```

```
ec_field() = ec_prime_field() | ec_characteristic_two_field()
```

```
ec_curve() =
    {A :: binary(), B :: binary(), Seed :: none | binary()}
```

Parametric curve definition.

```
ec_prime_field() = {prime_field, Prime :: integer()}
```

```
ec_characteristic_two_field() =
    {characteristic_two_field,
     M :: integer(),
     Basis :: ec_basis()}
```

```
ec_basis() =
    {tpbasis, K :: integer() >= 0} |
    {ppbasis,
     K1 :: integer() >= 0,
     K2 :: integer() >= 0,
     K3 :: integer() >= 0} |
    onbasis
```

Curve definition details.

Keys

```
key_integer() = integer() | binary()
```

Always `binary()` when used as return value

Public/Private Keys

```
rsa_public() = [key_integer()]
```

```
rsa_private() = [key_integer()]
```

```
rsa_params() =
    {ModulusSizeInBits :: integer(),
     PublicExponent :: key_integer()}
```

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

```
rsa_private() = [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 and P2 are first and second prime factors. E1 and E2 are first and second exponents. C is the CRT coefficient. The terminology is taken from **RFC 3447**.

```
dss_public() = [key_integer()]
dss_private() = [key_integer()]
```

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

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

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

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

```
ecdsa_public() = key_integer()
ecdsa_private() = key_integer()
ecdsa_params() = ec_named_curve() | ec_explicit_curve()
eddsa_public() = key_integer()
eddsa_private() = key_integer()
eddsa_params() = edwards_curve_ed()
srp_public() = key_integer()
srp_private() = key_integer()
```

```
srp_public() = key_integer()
```

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

```
srp_private() = key_integer()
```

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

```
srp_gen_params() =
    {user, srp_user_gen_params()} | {host, srp_host_gen_params()}
srp_comp_params() =
    {user, srp_user_comp_params()} |
    {host, srp_host_comp_params()}
srp_user_gen_params() = [DerivedKey::binary(), Prime::binary(),
Generator::binary(), Version::atom()]srp_host_gen_params() =
[Verifier::binary(), Prime::binary(), Version::atom() ]srp_user_comp_params()
= [DerivedKey::binary(), Prime::binary(), Generator::binary(),
Version::atom() | ScramblerArg::list()]srp_host_comp_params() =
[Verifier::binary(), Prime::binary(), Version::atom() | ScramblerArg::list()]
```

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'

Public Key Ciphers

```
pk_encrypt_decrypt_algs() = rsa
```

Algorithms for public key encrypt/decrypt. Only RSA is supported.

```
pk_encrypt_decrypt_opts() = [rsa_opt()] | rsa_compat_opts()
rsa_opt() =
    {rsa_padding, rsa_padding()} |
    {signature_md, atom()} |
```

```
    {rsa_mgf1_md, sha} |
    {rsa_oaep_label, binary()} |
    {rsa_oaep_md, sha}
rsa_padding() =
    rsa_pkcs1_padding | rsa_pkcs1_oaep_padding |
    rsa_sslv23_padding | rsa_x931_padding | rsa_no_padding
```

Options for public key encrypt/decrypt. Only RSA is supported.

Warning:

The RSA options are experimental.

The exact set of options and there syntax **may** be changed without prior notice.

```
rsa_compat_opts() = [{rsa_pad, rsa_padding()}] | rsa_padding()
```

Those option forms are kept only for compatibility and should not be used in new code.

Public Key Sign and Verify

```
pk_sign_verify_algs() = rsa | dss | ecdsa | eddsa
```

Algorithms for sign and verify.

```
pk_sign_verify_opts() = [rsa_sign_verify_opt()]
rsa_sign_verify_opt() =
    {rsa_padding, rsa_sign_verify_padding()} |
    {rsa_pss_saltlen, integer()} |
    {rsa_mgf1_md, sha2()}
rsa_sign_verify_padding() =
    rsa_pkcs1_padding | rsa_pkcs1_pss_padding | rsa_x931_padding |
    rsa_no_padding
```

Options for sign and verify.

Warning:

The RSA options are experimental.

The exact set of options and there syntax **may** be changed without prior notice.

Diffie-Hellman Keys and parameters

```
dh_public() = key_integer()
dh_private() = key_integer()
dh_params() = [key_integer()]
```

```
dh_params() = [P, G] | [P, G, PrivateKeyBitLength]
```

```
ecdh_public() = key_integer()
ecdh_private() = key_integer()
ecdh_params() =
    ec_named_curve() | edwards_curve_dh() | ec_explicit_curve()
```

Types for Engines

```
engine_key_ref() =
```

```
    #{engine := engine_ref(),
      key_id := key_id(),
      password => password(),
      term() => term()}
```

`engine_ref() = term()`

The result of a call to engine_load/3.

`key_id() = string() | binary()`

Identifies the key to be used. The format depends on the loaded engine. It is passed to the `ENGINE_load_(private|public)_key` functions in libcrypto.

`password() = string() | binary()`

The password of the key stored in an engine.

```
engine_method_type() =
    engine_method_rsa | engine_method_dsa | engine_method_dh |
    engine_method_rand | engine_method_ecdh |
    engine_method_ecdsa | engine_method_ciphers |
    engine_method_digests | engine_method_store |
    engine_method_pkey_meths | engine_method_pkey_asn1_meths |
    engine_method_ec
```

`engine_cmnd() = {unicode:chardata(), unicode:chardata()}`

Pre and Post commands for engine_load/3 and /4.

Internal data types

`crypto_state()`

`hash_state()`

`mac_state()`

Contexts with an internal state that should not be manipulated but passed between function calls.

Exceptions

Atoms - the older style

The exception `error:badarg` signifies that one or more arguments are of wrong data type, or are otherwise badly formed.

The exception `error:notsup` signifies that the algorithm is known but is not supported by current underlying libcrypto or explicitly disabled when building that.

For a list of supported algorithms, see supports(ciphers).

3-tuples - the new style

The exception is:

```
error:{Tag, C_FileInfo, Description}

Tag = badarg | notsup | error
C_FileInfo = term()    % Usually only useful for the OTP maintainer
Description = string() % Clear text, sometimes only useful for the OTP maintainer
```

The exception tags are:

`badarg`

Signifies that one or more arguments are of wrong data type or are otherwise badly formed.

`notsup`

Signifies that the algorithm is known but is not supported by current underlying libcrypto or explicitly disabled when building that one.

`error`

An error condition that should not occur, for example a memory allocation failed or the underlying cryptolib returned an error code, for example `"Can't initialize context, step 1"`. Those text usually needs searching the C-code to be understood.

Usually there are more information in the call stack about which argument caused the exception and what the values where.

To catch the exception, use for example:

```
try crypto:crypto_init(Ciph, Key, IV, true)
    catch
        error:{Tag, _C_FileInfo, Description} ->
            do_something(......)
         .....
end
```

Exports

`crypto_init(Cipher, Key, FlagOrOptions) -> State`

Types:

`Cipher = cipher_no_iv()`
`Key = iodata()`
`FlagOrOptions = crypto_opts() | boolean()`
`State = crypto_state()`

Uses the 3-tuple style for error handling.

Equivalent to the call `crypto_init(Cipher, Key, <<>>, FlagOrOptions)`. It is intended for ciphers without an IV (nounce).

`crypto_init(Cipher, Key, IV, FlagOrOptions) -> State`

Types:

`Cipher = cipher_iv()`
`Key = IV = iodata()`
`FlagOrOptions = crypto_opts()`
`State = crypto_state()`

Uses the 3-tuple style for error handling.

Initializes a series of encryptions or decryptions and creates an internal state with a reference that is returned.

If `IV = <<>>`, no IV is used. This is intended for ciphers without an IV (nounce). See crypto_init/3.

If `IV = undefined`, the IV must be added by calls to crypto_dyn_iv_update/3. This is intended for cases where the IV (nounce) need to be changed for each encryption and decryption. See crypto_dyn_iv_init/3.

The actual encryption or decryption is done by crypto_update/2 (or crypto_dyn_iv_update/3).

For encryption, set the `FlagOrOptions` to `true` or `[{encrypt,true}]`. For decryption, set it to `false` or `[{encrypt,false}]`.

Padding could be enabled with the option {padding,Padding}. The cryptolib_padding enables `pkcs_padding` or no padding (`none`). The paddings `zero` or `random` fills the last part of the last block with zeroes or random bytes. If the last block is already full, nothing is added.

In decryption, the cryptolib_padding removes such padding, if present. The otp_padding is not removed - it has to be done elsewhere.

If padding is `{padding,none}` or not specified and the total data from all subsequent crypto_updates does not fill the last block fully, that last data is lost. In case of `{padding,none}` there will be an error in this case. If padding is not specified, the bytes of the unfilled block is silently discarded.

The actual padding is performed by crypto_final/1.

For blocksizes call cipher_info/1.

See examples in the User's Guide.

`crypto_update(State, Data) -> Result`

Types:

```
State = crypto_state()
Data = iodata()
Result = binary()
```

Uses the 3-tuple style for error handling.

It does an actual crypto operation on a part of the full text. If the part is less than a number of full blocks, only the full blocks (possibly none) are encrypted or decrypted and the remaining bytes are saved to the next `crypto_update` operation. The `State` should be created with crypto_init/3 or crypto_init/4.

See examples in the User's Guide.

`crypto_dyn_iv_init(Cipher, Key, FlagOrOptions) -> State`

Types:

```
Cipher = cipher_iv()
Key = iodata()
FlagOrOptions = crypto_opts() | boolean()
State = crypto_state()
```

Uses the 3-tuple style for error handling.

Initializes a series of encryptions or decryptions where the IV is provided later. The actual encryption or decryption is done by crypto_dyn_iv_update/3.

The function is equivalent to `crypto_init(Cipher, Key, undefined, FlagOrOptions)`.

`crypto_final(State) -> FinalResult`

Types:

```
State = crypto_state()
FinalResult = binary()
```

Uses the 3-tuple style for error handling.

Finalizes a series of encryptions or decryptions and delivers the final bytes of the final block. The data returned from this function may be empty if no padding was enabled in crypto_init/3,4 or crypto_dyn_iv_init/3.

crypto_get_data(State) -> Result

Types:

`State = crypto_state()`
`Result = map()`

Uses the 3-tuple style for error handling.

Returns information about the State in the argument. The information is the form of a map, which currently contains at least:

`size`
: The number of bytes encrypted or decrypted so far.

`padding_size`
: After a call to crypto_final/1 it contains the number of bytes padded. Otherwise 0.

`padding_type`
: The type of the padding as provided in the call to crypto_init/3,4.

`encrypt`
: Is `true` if encryption is performed. It is `false` otherwise.

crypto_dyn_iv_update(State, Data, IV) -> Result

Types:

`State = crypto_state()`
`Data = IV = iodata()`
`Result = binary()`

Uses the 3-tuple style for error handling.

Do an actual crypto operation on a part of the full text and the IV is supplied for each part. The `State` should be created with crypto_dyn_iv_init/3.

crypto_one_time(Cipher, Key, Data, FlagOrOptions) -> Result

Types:

`Cipher = cipher_no_iv()`
`Key = Data = iodata()`
`FlagOrOptions = crypto_opts() | boolean()`
`Result = binary()`

Uses the 3-tuple style for error handling.

As crypto_one_time/5 but for ciphers without IVs.

crypto_one_time(Cipher, Key, IV, Data, FlagOrOptions) -> Result

Types:

`Cipher = cipher_iv()`
`Key = IV = Data = iodata()`
`FlagOrOptions = crypto_opts() | boolean()`
`Result = binary()`

Uses the 3-tuple style for error handling.

Do a complete encrypt or decrypt of the full text in the argument `Data`.

For encryption, set the `FlagOrOptions` to `true`. For decryption, set it to `false`. For setting other options, see crypto_init/4.

See examples in the User's Guide.

```
crypto_one_time_aead(Cipher, Key, IV, InText, AAD,
                     EncFlag :: true) ->
                        Result
crypto_one_time_aead(Cipher, Key, IV, InText, AAD, TagOrTagLength,
                     EncFlag) ->
                        Result
```

Types:

```
Cipher = cipher_aead()
Key = IV = InText = AAD = iodata()
TagOrTagLength = EncryptTagLength | DecryptTag
EncryptTagLength = integer() >= 0
DecryptTag = iodata()
EncFlag = boolean()
Result = EncryptResult | DecryptResult
EncryptResult = {OutCryptoText, OutTag}
DecryptResult = OutPlainText | error
OutCryptoText = OutTag = OutPlainText = binary()
```

Uses the 3-tuple style for error handling.

Do a complete encrypt or decrypt with an AEAD cipher of the full text.

For encryption, set the `EncryptFlag` to `true` and set the `TagOrTagLength` to the wanted size (in bytes) of the tag, that is, the tag length. If the default length is wanted, the `crypto_aead/6` form may be used.

For decryption, set the `EncryptFlag` to `false` and put the tag to be checked in the argument `TagOrTagLength`.

See examples in the User's Guide.

```
supports(Type) -> Support
```

Types:

```
Type = hashs | ciphers | public_keys | macs | curves | rsa_opts
Support = Hashs | Ciphers | PKs | Macs | Curves | RSAopts
Hashs =
    [sha1() |
     sha2() |
     sha3() |
     blake2() |
     ripemd160 |
     compatibility_only_hash()]
Ciphers = [cipher()]
PKs = [rsa | dss | ecdsa | dh | ecdh | eddh | ec_gf2m]
Macs = [hmac | cmac | poly1305]
Curves =
    [ec_named_curve() | edwards_curve_dh() | edwards_curve_ed()]
RSAopts = [rsa_sign_verify_opt() | rsa_opt()]
```

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

See hash_info/1 and cipher_info/1 for information about the hash and cipher algorithms.

mac(Type :: poly1305, Key, Data) -> Mac

Types:

```
Key = Data = iodata()
Mac = binary()
```

Uses the 3-tuple style for error handling.

Short for mac(Type, undefined, Key, Data).

mac(Type, SubType, Key, Data) -> Mac

Types:

```
Type = hmac | cmac | poly1305
SubType =
    hmac_hash_algorithm() | cmac_cipher_algorithm() | undefined
Key = Data = iodata()
Mac = binary()
```

Uses the 3-tuple style for error handling.

Computes a MAC (Message Authentication Code) of type `Type` from `Data`.

`SubType` depends on the MAC `Type`:

- For `hmac` it is a hash algorithm, see Algorithm Details in the User's Guide.
- For `cmac` it is a cipher suitable for cmac, see Algorithm Details in the User's Guide.
- For `poly1305` it should be set to `undefined` or the mac/2 function could be used instead, see Algorithm Details in the User's Guide.

`Key` is the authentication key with a length according to the `Type` and `SubType`. The key length could be found with the hash_info/1 (`hmac`) for and cipher_info/1 (`cmac`) functions. For `poly1305` the key length is 32 bytes. Note that the cryptographic quality of the key is not checked.

The `Mac` result will have a default length depending on the `Type` and `SubType`. To set a shorter length, use macN/4 or macN/5 instead. The default length is documented in Algorithm Details in the User's Guide.

```
macN(Type :: poly1305, Key, Data, MacLength) -> Mac
```

Types:

```
Key = Data = iodata()
Mac = binary()
MacLength = integer() >= 1
```

Uses the 3-tuple style for error handling.

Short for macN(Type, undefined, Key, Data, MacLength).

```
macN(Type, SubType, Key, Data, MacLength) -> Mac
```

Types:

```
Type = hmac | cmac | poly1305
SubType =
    hmac_hash_algorithm() | cmac_cipher_algorithm() | undefined
Key = Data = iodata()
Mac = binary()
MacLength = integer() >= 1
```

Computes a MAC (Message Authentication Code) as mac/3 and mac/4 but `MacLength` will limit the size of the resultant `Mac` to at most `MacLength` bytes. Note that if `MacLength` is greater than the actual number of bytes returned from the underlying hash, the returned hash will have that shorter length instead.

The max `MacLength` is documented in Algorithm Details in the User's Guide.

```
mac_init(Type :: poly1305, Key) -> State
```

Types:

```
Key = iodata()
State = mac_state()
```

Uses the 3-tuple style for error handling.

Short for mac_init(Type, undefined, Key).

```
mac_init(Type, SubType, Key) -> State
```

Types:

```
Type = hmac | cmac | poly1305
SubType =
    hmac_hash_algorithm() | cmac_cipher_algorithm() | undefined
Key = iodata()
State = mac_state()
```

Uses the 3-tuple style for error handling.

Initializes the context for streaming MAC operations.

`Type` determines which mac algorithm to use in the MAC operation.

`SubType` depends on the MAC `Type`:

- For `hmac` it is a hash algorithm, see Algorithm Details in the User's Guide.
- For `cmac` it is a cipher suitable for cmac, see Algorithm Details in the User's Guide.

- For `poly1305` it should be set to `undefined` or the mac/2 function could be used instead, see Algorithm Details in the User's Guide.

`Key` is the authentication key with a length according to the `Type` and `SubType`. The key length could be found with the hash_info/1 (`hmac`) for and cipher_info/1 (`cmac`) functions. For `poly1305` the key length is 32 bytes. Note that the cryptographic quality of the key is not checked.

The returned `State` should be used in one or more subsequent calls to mac_update/2. The MAC value is finally returned by calling mac_final/1 or mac_finalN/2.

See examples in the User's Guide.

`mac_update(State0, Data) -> State`

Types:

`Data = iodata()`
`State0 = State = mac_state()`

Uses the 3-tuple style for error handling.

Updates the MAC represented by `State0` using the given `Data` which could be of any length.

The `State0` is the State value originally from a MAC init function, that is mac_init/2, mac_init/3 or a previous call of `mac_update/2`. The value `State0` is returned unchanged by the function as `State`.

`mac_final(State) -> Mac`

Types:

`State = mac_state()`
`Mac = binary()`

Uses the 3-tuple style for error handling.

Finalizes the MAC operation referenced by `State`. The `Mac` result will have a default length depending on the `Type` and `SubType` in the mac_init/2,3 call. To set a shorter length, use mac_finalN/2 instead. The default length is documented in Algorithm Details in the User's Guide.

`mac_finalN(State, MacLength) -> Mac`

Types:

`State = mac_state()`
`MacLength = integer() >= 1`
`Mac = binary()`

Uses the 3-tuple style for error handling.

Finalizes the MAC operation referenced by `State`.

`Mac` will be a binary with at most `MacLength` bytes. Note that if `MacLength` is greater than the actual number of bytes returned from the underlying hash, the returned hash will have that shorter length instead.

The max `MacLength` is documented in Algorithm Details in the User's Guide.

`bytes_to_integer(Bin :: binary()) -> integer()`

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

`compute_key(Type, OthersPublicKey, MyPrivateKey, Params) ->`

```
                SharedSecret
```

Types:

```
Type = dh | ecdh | eddh | srp
SharedSecret = binary()
OthersPublicKey = dh_public() | ecdh_public() | srp_public()
MyPrivateKey =
    dh_private() | ecdh_private() | {srp_public(), srp_private()}
Params = dh_params() | ecdh_params() | srp_comp_params()
```

Uses the 3-tuple style for error handling.

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

```
exor(Bin1 :: iodata(), Bin2 :: iodata()) -> 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 | eddh | eddsa | rsa | srp
PublicKey =
    dh_public() | ecdh_public() | rsa_public() | srp_public()
PrivKeyIn =
    undefined |
    dh_private() |
    ecdh_private() |
    rsa_private() |
    {srp_public(), srp_private()}
PrivKeyOut =
    dh_private() |
    ecdh_private() |
    rsa_private() |
    {srp_public(), srp_private()}
Params =
    dh_params() |
    ecdh_params() |
    eddsa_params() |
    rsa_params() |
    srp_comp_params()
```

Uses the 3-tuple style for error handling.

Generates a public key of type `Type`. See also public_key:generate_key/1.

Note:

If the linked version of cryptolib is OpenSSL 3.0

- and the `Type` is `dh` (diffie-hellman)
- and the parameter `P` (in dh_params()) is one of the MODP groups (see **RFC 3526**)
- and the optional `PrivateKeyBitLength` parameter (in dh_params()) is present,

then the optional key length parameter must be at least 224, 256, 302, 352 and 400 for group sizes of 2048, 3072, 4096, 6144 and 8192, respectively.

Note:

RSA key generation is only available if the runtime was built with dirty scheduler support. Otherwise, attempting to generate an RSA key will raise the exception `error:notsup`.

`hash(Type, Data) -> Digest`

Types:

```
Type = hash_algorithm()
Data = iodata()
Digest = binary()
```

Uses the 3-tuple style for error handling.

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

`hash_init(Type) -> State`

Types:

```
Type = hash_algorithm()
State = hash_state()
```

Uses the 3-tuple style for error handling.

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

`hash_update(State, Data) -> NewState`

Types:

```
State = NewState = hash_state()
Data = iodata()
```

Uses the 3-tuple style for error handling.

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(State) -> Digest`

Types:

```
State = hash_state()
Digest = binary()
```

Uses the 3-tuple style for error handling.

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.

```
info_fips() -> not_supported | not_enabled | enabled
```

Provides information about the FIPS operating status of crypto and the underlying libcrypto library. If crypto was built with FIPS support this can be either `enabled` (when running in FIPS mode) or `not_enabled`. For other builds this value is always `not_supported`.

See enable_fips_mode/1 about how to enable FIPS mode.

Warning:

In FIPS mode all non-FIPS compliant algorithms are disabled and raise exception `error:notsup`. Check supports(ciphers) that in FIPS mode returns the restricted list of available algorithms.

```
enable_fips_mode(Enable) -> Result
```

Types:

```
Enable = Result = boolean()
```

Enables (`Enable = true`) or disables (`Enable = false`) FIPS mode. Returns `true` if the operation was successful or `false` otherwise.

Note that to enable FIPS mode successfully, OTP must be built with the configure option `--enable-fips`, and the underlying libcrypto must also support FIPS.

See also info_fips/0.

```
info() ->
        #{compile_type := normal | debug | valgrind | asan,
          cryptolib_version_compiled => string() | undefined,
          cryptolib_version_linked := string(),
          link_type := dynamic | static,
          otp_crypto_version := string()}
```

Provides a map with information about the compilation and linking of crypto.

Example:

```
1> crypto:info().
#{compile_type => normal,
  cryptolib_version_compiled => "OpenSSL 3.0.0 7 sep 2021",
  cryptolib_version_linked => "OpenSSL 3.0.0 7 sep 2021",
  link_type => dynamic,
  otp_crypto_version => "5.0.2"}
2>
```

More association types than documented may be present in the map.

```
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">>,269484095,<<"OpenSSL 1.1.0c  10 Nov 2016"">>}]
```

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 libcrypto library used at runtime. In earlier OTP versions both numeric and text was taken from the library.

```
hash_info(Type) -> Result
```

Types:

```
Type = hash_algorithm()
Result =
    #{size := integer(),
      block_size := integer(),
      type := integer()}
```

Provides a map with information about block_size, size and possibly other properties of the hash algorithm in question.

For a list of supported hash algorithms, see supports(hashs).

```
cipher_info(Type) -> Result
```

Types:

```
Type = cipher()
Result =
    #{key_length := integer(),
      iv_length := integer(),
      block_size := integer(),
      mode := CipherModes,
      type := undefined | integer(),
      prop_aead := boolean()}
CipherModes =
    undefined | cbc_mode | ccm_mode | cfb_mode | ctr_mode |
    ecb_mode | gcm_mode | ige_mode | ocb_mode | ofb_mode |
    wrap_mode | xts_mode
```

Provides a map with information about block_size, key_length, iv_length, aead support and possibly other properties of the cipher algorithm in question.

> **Note:**
>
> The ciphers `aes_cbc`, `aes_cfb8`, `aes_cfb128`, `aes_ctr`, `aes_ecb`, `aes_gcm` and `aes_ccm` has no keylength in the `Type` as opposed to for example `aes_128_ctr`. They adapt to the length of the key provided in the encrypt and decrypt function. Therefore it is impossible to return a valid keylength in the map.
>
> Always use a `Type` with an explicit key length,

For a list of supported cipher algorithms, see supports(ciphers).

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

Types:

```
N = P = M = binary() | integer()
Result = binary() | error
```

Computes the function `N^P mod M`.

```
private_decrypt(Algorithm, CipherText, PrivateKey, Options) ->
                   PlainText
```

Types:

```
Algorithm = pk_encrypt_decrypt_algs()
CipherText = binary()
PrivateKey = rsa_private() | engine_key_ref()
Options = pk_encrypt_decrypt_opts()
PlainText = binary()
```

Uses the 3-tuple style for error handling.

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(Algorithm, PlainText, PrivateKey, Options) ->
                   CipherText
```

Types:

```
Algorithm = pk_encrypt_decrypt_algs()
PlainText = binary()
PrivateKey = rsa_private() | engine_key_ref()
Options = pk_encrypt_decrypt_opts()
CipherText = binary()
```

Uses the 3-tuple style for error handling.

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(Algorithm, CipherText, PublicKey, Options) ->
                  PlainText
```

Types:

```
Algorithm = pk_encrypt_decrypt_algs()
CipherText = binary()
PublicKey = rsa_public() | engine_key_ref()
Options = pk_encrypt_decrypt_opts()
PlainText = binary()
```

Uses the 3-tuple style for error handling.

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(Algorithm, PlainText, PublicKey, Options) ->
                  CipherText
```

Types:

```
Algorithm = pk_encrypt_decrypt_algs()
PlainText = binary()
PublicKey = rsa_public() | engine_key_ref()
Options = pk_encrypt_decrypt_opts()
CipherText = binary()
```

Uses the 3-tuple style for error handling.

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_seed(Seed :: binary()) -> ok
```

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 strong_rand_bytes/1 raises `error: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`.

```
start() -> ok | {error, Reason :: term()}
```

Equivalent to application:start(crypto).

```
stop() -> ok | {error, Reason :: term()}
```

Equivalent to application:stop(crypto).

```
strong_rand_bytes(N :: integer() >= 0) -> binary()
```

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 raise exception `error:low_entropy` in case the random generator failed due to lack of secure "randomness".

rand_seed() -> rand:state()

Creates state object for random number generation, in order to generate cryptographically strong random numbers (based on OpenSSL's `BN_rand_range`), and saves it in the process dictionary before returning it as well. See also rand:seed/1 and rand_seed_s/0.

When using the state object from this function the rand functions using it may raise exception `error:low_entropy` in case the random generator failed due to lack of secure "randomness".

Example

```
_ = crypto:rand_seed(),
_IntegerValue = rand:uniform(42), % [1; 42]
_FloatValue = rand:uniform().     % [0.0; 1.0[
```

rand_seed_s() -> rand:state()

Creates state object for random number generation, in order to generate cryptographically strongly random numbers (based on OpenSSL's `BN_rand_range`). See also rand:seed_s/1.

When using the state object from this function the rand functions using it may raise exception `error:low_entropy` in case the random generator failed due to lack of secure "randomness".

Note:

The state returned from this function cannot be used to get a reproducible random sequence as from the other rand functions, since reproducibility does not match cryptographically safe.

The only supported usage is to generate one distinct random sequence from this start state.

rand_seed_alg(Alg) -> rand:state()

Types:

`Alg = crypto | crypto_cache`

Creates state object for random number generation, in order to generate cryptographically strong random numbers, and saves it in the process dictionary before returning it as well. See also rand:seed/1 and rand_seed_alg_s/1.

When using the state object from this function the rand functions using it may raise exception `error:low_entropy` in case the random generator failed due to lack of secure "randomness".

Example

```
_ = crypto:rand_seed_alg(crypto_cache),
_IntegerValue = rand:uniform(42), % [1; 42]
_FloatValue = rand:uniform().     % [0.0; 1.0[
```

rand_seed_alg(Alg, Seed) -> rand:state()

Types:

`Alg = crypto_aes`

Creates a state object for random number generation, in order to generate cryptographically unpredictable random numbers, and saves it in the process dictionary before returning it as well. See also rand_seed_alg_s/2.

Example

```
_ = crypto:rand_seed_alg(crypto_aes, "my seed"),
IntegerValue = rand:uniform(42), % [1; 42]
FloatValue = rand:uniform(),     % [0.0; 1.0[
_ = crypto:rand_seed_alg(crypto_aes, "my seed"),
IntegerValue = rand:uniform(42), % Same values
FloatValue = rand:uniform().     % again
```

`rand_seed_alg_s(Alg) -> rand:state()`

Types:

`Alg = crypto | crypto_cache`

Creates state object for random number generation, in order to generate cryptographically strongly random numbers. See also rand:seed_s/1.

If `Alg` is `crypto` this function behaves exactly like rand_seed_s/0.

If `Alg` is `crypto_cache` this function fetches random data with OpenSSL's `RAND_bytes` and caches it for speed using an internal word size of 56 bits that makes calculations fast on 64 bit machines.

When using the state object from this function the rand functions using it may raise exception `error:low_entropy` in case the random generator failed due to lack of secure "randomness".

The cache size can be changed from its default value using the crypto app's configuration parameter `rand_cache_size`.

When using the state object from this function the rand functions using it may throw exception `low_entropy` in case the random generator failed due to lack of secure "randomness".

Note:

The state returned from this function cannot be used to get a reproducible random sequence as from the other rand functions, since reproducibility does not match cryptographically safe.

In fact since random data is cached some numbers may get reproduced if you try, but this is unpredictable.

The only supported usage is to generate one distinct random sequence from this start state.

`rand_seed_alg_s(Alg, Seed) -> rand:state()`

Types:

`Alg = crypto_aes`

Creates a state object for random number generation, in order to generate cryptographically unpredictable random numbers. See also rand_seed_alg/1.

To get a long period the Xoroshiro928 generator from the rand module is used as a counter (with period 2^928 - 1) and the generator states are scrambled through AES to create 58-bit pseudo random values.

The result should be statistically completely unpredictable random values, since the scrambling is cryptographically strong and the period is ridiculously long. But the generated numbers are not to be regarded as cryptographically strong since there is no re-keying schedule.

- If you need cryptographically strong random numbers use rand_seed_alg_s/1 with `Alg =:= crypto` or `Alg =:= crypto_cache`.
- If you need to be able to repeat the sequence use this function.
- If you do not need the statistical quality of this function, there are faster algorithms in the rand module.

Thanks to the used generator the state object supports the `rand:jump/0,1` function with distance 2^512.

Numbers are generated in batches and cached for speed reasons. The cache size can be changed from its default value using the crypto app's configuration parameter `rand_cache_size`.

```
ec_curves() -> [EllipticCurve]
```

Types:

```
EllipticCurve =
    ec_named_curve() | edwards_curve_dh() | edwards_curve_ed()
```

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

```
ec_curve(CurveName) -> ExplicitCurve
```

Types:

```
CurveName = ec_named_curve()
ExplicitCurve = ec_explicit_curve()
```

Return the defining parameters of a elliptic curve.

```
sign(Algorithm, DigestType, Msg, Key) -> Signature
sign(Algorithm, DigestType, Msg, Key, Options) -> Signature
```

Types:

```
Algorithm = pk_sign_verify_algs()
DigestType =
    rsa_digest_type() |
    dss_digest_type() |
    ecdsa_digest_type() |
    none
Msg = iodata() | {digest, iodata()}
Key =
    rsa_private() |
    dss_private() |
    [ecdsa_private() | ecdsa_params()] |
    [eddsa_private() | eddsa_params()] |
    engine_key_ref()
Options = pk_sign_verify_opts()
Signature = binary()
```

Uses the 3-tuple style for error handling.

Creates a digital signature.

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

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

See also public_key:sign/3.

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

Types:

```
Algorithm = pk_sign_verify_algs()
DigestType =
    rsa_digest_type() |
    dss_digest_type() |
    ecdsa_digest_type() |
    none
Msg = iodata() | {digest, iodata()}
Signature = binary()
Key =
    rsa_public() |
    dss_public() |
    [ecdsa_public() | ecdsa_params()] |
    [eddsa_public() | eddsa_params()] |
    engine_key_ref()
Options = pk_sign_verify_opts()
Result = boolean()
```

Uses the 3-tuple style for error handling.

Verifies a digital signature

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

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

See also public_key:verify/4.

Exports

`privkey_to_pubkey(Type, EnginePrivateKeyRef) -> PublicKey`

Types:

```
Type = rsa | dss
EnginePrivateKeyRef = engine_key_ref()
PublicKey = rsa_public() | dss_public()
```

Fetches the corresponding public key from a private key stored in an Engine. The key must be of the type indicated by the Type parameter.

`engine_get_all_methods() -> Result`

Types:

```
Result = [engine_method_type()]
```

Returns a list of all possible engine methods.

May raise exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`engine_load(EngineId, PreCmds, PostCmds) -> Result`

Types:

```
EngineId = unicode:chardata()
PreCmds = PostCmds = [engine_cmnd()]
Result =
    {ok, Engine :: engine_ref()} | {error, Reason :: term()}
```

Loads the OpenSSL engine given by `EngineId` if it is available and then returns ok and an engine handle. This function is the same as calling `engine_load/4` with `EngineMethods` set to a list of all the possible methods. An error tuple is returned if the engine can't be loaded.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`engine_load(EngineId, PreCmds, PostCmds, EngineMethods) -> Result`

Types:

```
EngineId = unicode:chardata()
PreCmds = PostCmds = [engine_cmnd()]
EngineMethods = [engine_method_type()]
Result =
    {ok, Engine :: engine_ref()} | {error, Reason :: term()}
```

Loads the OpenSSL engine given by `EngineId` if it is available and then returns ok and an engine handle. An error tuple is returned if the engine can't be loaded.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`engine_unload(Engine) -> Result`

Types:

```
Engine = engine_ref()
Result = ok | {error, Reason :: term()}
```

Unloads the OpenSSL engine given by `Engine`. An error tuple is returned if the engine can't be unloaded.

The function raises a `error:badarg` if the parameter is in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`engine_by_id(EngineId) -> Result`

Types:

```
EngineId = unicode:chardata()
Result =
    {ok, Engine :: engine_ref()} | {error, Reason :: term()}
```

Get a reference to an already loaded engine with `EngineId`. An error tuple is returned if the engine can't be unloaded.

The function raises a `error:badarg` if the parameter is in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`engine_ctrl_cmd_string(Engine, CmdName, CmdArg) -> Result`

Types:

```
Engine = term()
CmdName = CmdArg = unicode:chardata()
Result = ok | {error, Reason :: term()}
```

Sends ctrl commands to the OpenSSL engine given by `Engine`. This function is the same as calling `engine_ctrl_cmd_string/4` with `Optional` set to `false`.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

`engine_ctrl_cmd_string(Engine, CmdName, CmdArg, Optional) -> Result`

Types:

```
Engine = term()
CmdName = CmdArg = unicode:chardata()
Optional = boolean()
Result = ok | {error, Reason :: term()}
```

Sends ctrl commands to the OpenSSL engine given by `Engine`. `Optional` is a boolean argument that can relax the semantics of the function. If set to `true` it will only return failure if the ENGINE supported the given command name but failed while executing it, if the ENGINE doesn't support the command name it will simply return success without doing anything. In this case we assume the user is only supplying commands specific to the given ENGINE so we set this to `false`.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

`engine_add(Engine) -> Result`

Types:

```
Engine = engine_ref()
Result = ok | {error, Reason :: term()}
```

Add the engine to OpenSSL's internal list.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

`engine_remove(Engine) -> Result`

Types:

```
Engine = engine_ref()
Result = ok | {error, Reason :: term()}
```

Remove the engine from OpenSSL's internal list.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

`engine_get_id(Engine) -> EngineId`

Types:

```
Engine = engine_ref()
EngineId = unicode:chardata()
```

Return the ID for the engine, or an empty binary if there is no id set.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

engine_get_name(Engine) -> EngineName

Types:

```
Engine = engine_ref()
EngineName = unicode:chardata()
```

Return the name (eg a description) for the engine, or an empty binary if there is no name set.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

engine_list() -> Result

Types:

```
Result = [EngineId :: unicode:chardata()]
```

List the id's of all engines in OpenSSL's internal list.

It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

May raise exception `error:notsup` in case engine functionality is not supported by the underlying OpenSSL implementation.

ensure_engine_loaded(EngineId, LibPath) -> Result
ensure_engine_loaded(EngineId, LibPath, EngineMethods) -> Result

Types:

```
EngineId = LibPath = unicode:chardata()
EngineMethods = [engine_method_type()]
Result =
    {ok, Engine :: engine_ref()} | {error, Reason :: term()}
```

Loads an engine given by `EngineId` and the path to the dynamic library implementing the engine. If the function with arity three is used, one can specify which engine methods it should handle. An error tuple is returned if the engine can't be loaded.

This function differs from the normal engine_load in the sense that it also add the engine id to OpenSSL's internal engine list. The reference from the first call holds a functional reference to the engine. Then in the following calls to the function just fetch a structural reference by looking up the EngineId in OpenSSL's internal engine list. Therefor there is a difference when unloading the references, see `ensure_engine_unloaded`.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`ensure_engine_unloaded(Engine) -> Result`
`ensure_engine_unloaded(Engine, EngineMethods) -> Result`

Types:

```
Engine = engine_ref()
EngineMethods = [engine_method_type()]
Result = ok | {error, Reason :: term()}
```

Unloads an engine loaded with the `ensure_engine_loaded` functions. The variant with arity two is used when the engine was loaded with `ensure_engine_loaded/3` and the second parameter is the method list (same as when loaded).

Exactly what happens differ a bit on the reference, if it's the first reference got by `ensure_engine_loaded` it both removes the label from OpenSSL's internal engine list and unloads the engine. If it's not the reference from the first call it just removes that refence.

The function raises a `error:badarg` if the parameters are in wrong format. It may also raise the exception `error:notsup` in case there is no engine support in the underlying OpenSSL implementation.

See also the chapter Engine Load in the User's Guide.

`hash_equals(BinA, BinB) -> Result`

Types:

```
BinA = BinB = binary()
Result = boolean()
```

Constant time memory comparison for fixed length binaries, such as results of HMAC computations.

Returns true if the binaries are identical, false if they are of the same length but not identical. The function raises an `error:badarg` exception if the binaries are of different size.

`pbkdf2_hmac(Digest, Pass, Salt, Iter, KeyLen) -> Result`

Types:

```
Digest = sha | sha224 | sha256 | sha384 | sha512
Pass = Salt = binary()
Iter = KeyLen = integer() >= 1
Result = binary()
```

Uses the 3-tuple style for error handling.

PKCS #5 PBKDF2 (Password-Based Key Derivation Function 2) in combination with HMAC.