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
 * 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).
 */
1.2 FIPS mode

This chapter describes FIPS mode support in the crypto application.
1.2 FIPS mode

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 FIPS mode

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",
    ["/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.
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.

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

List all engines currently loaded

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. Thoose 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
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

Block Ciphers

To be used in `block_encrypt/3`, `block_encrypt/4`, `block_decrypt/3` and `block_decrypt/4`.

Available in all OpenSSL compatible with Erlang CRYPTO if not disabled by configuration.
To dynamically check availability, check that the name in the *Cipher and Mode* column is present in the list with the *cipher* tag in the return value of *crypto:supports()*.

<table>
<thead>
<tr>
<th>Cipher and Mode</th>
<th>Key length [bytes]</th>
<th>IV length [bytes]</th>
<th>Block size [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_cbc</td>
<td>16, 24, 32</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>aes_cbc128</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>aes_cbc256</td>
<td>32</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>aes_cfb8</td>
<td>16, 24, 32</td>
<td>16</td>
<td>any</td>
</tr>
<tr>
<td>aes_ecb</td>
<td>16, 24, 32</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>aes_ige256</td>
<td>16</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>blowfish_cbc</td>
<td>4-56</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>blowfish_cfb64</td>
<td>#1</td>
<td>8</td>
<td>any</td>
</tr>
<tr>
<td>blowfish_ecb</td>
<td>#1</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>blowfish_ofb64</td>
<td>#1</td>
<td>8</td>
<td>any</td>
</tr>
<tr>
<td>des3_cbc (=DES EDE3 CBC)</td>
<td>[8,8,8]</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>des3_cfb (=DES EDE3 CFB)</td>
<td>[8,8,8]</td>
<td>8</td>
<td>any</td>
</tr>
<tr>
<td>des_cbc</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>des_cfb</td>
<td>8</td>
<td>8</td>
<td>any</td>
</tr>
<tr>
<td>des_ecb</td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>des_ede3 (=DES EDE3 CBC)</td>
<td>[8,8,8]</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>rc2_cbc</td>
<td>#1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.1: Block cipher key lengths

**AEAD Ciphers**

To be used in *block_encrypt/4* and *block_decrypt/4*.

To dynamically check availability, check that the name in the *Cipher and Mode* column is present in the list with the *cipher* tag in the return value of *crypto:supports()*.
1.5 Algorithm Details

### Cipher and Mode Details

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_ccm</td>
<td>16, 24, 32</td>
<td>7-13</td>
<td>any</td>
<td>even 4-16 default: 12</td>
<td>any</td>
<td>#1.1.0</td>
</tr>
<tr>
<td>aes_gcm</td>
<td>16, 24, 32</td>
<td>#1</td>
<td>any</td>
<td>1-16 default: 16</td>
<td>any</td>
<td>#1.1.0</td>
</tr>
<tr>
<td>chacha20_poly1305</td>
<td>1-16</td>
<td>any</td>
<td>16</td>
<td>any</td>
<td>#1.1.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: AEAD cipher key lengths

### Stream Ciphers

To be used in `stream_init/2` and `stream_init/3`.

To dynamically check availability, check that the name in the `Cipher and Mode` column is present in the list with the `cipher` tag in the return value of `crypto:supports()`.

<table>
<thead>
<tr>
<th>Cipher and Mode</th>
<th>Key length [bytes]</th>
<th>IV length [bytes]</th>
<th>Supported with OpenSSL versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_ctr</td>
<td>16, 24, 32</td>
<td>16</td>
<td>#1.0.1</td>
</tr>
<tr>
<td>rc4</td>
<td>#1</td>
<td></td>
<td>all</td>
</tr>
</tbody>
</table>

Table 5.3: Stream cipher key lengths

### Message Authentication Codes (MACs)

**CMAC**

To be used in `cmac/3` and `cmac/4`.

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 with the `macs` tag in the return value of `crypto:supports()`. Also check that the name in the `Cipher and Mode` column is present in the list with the `cipher` tag in the return value.

<table>
<thead>
<tr>
<th>Cipher and Mode</th>
<th>Key length [bytes]</th>
<th>Max Mac Length [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_cbc</td>
<td>16, 24, 32</td>
<td>16</td>
</tr>
<tr>
<td>aes_cbc128</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>aes_cbc256</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>
1.5 Algorithm Details

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key_Length</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_cfb8</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>blowfish_cbc</td>
<td>4-56</td>
<td>8</td>
</tr>
<tr>
<td>blowfish_cfb64</td>
<td>#1</td>
<td>1</td>
</tr>
<tr>
<td>blowfish_ecb</td>
<td>#1</td>
<td>8</td>
</tr>
<tr>
<td>blowfish_ofb64</td>
<td>#1</td>
<td>1</td>
</tr>
<tr>
<td>des3_cbc (=DES EDE3 CBC)</td>
<td>[8,8,8]</td>
<td>8</td>
</tr>
<tr>
<td>des3_cfb (=DES EDE3 CFB)</td>
<td>[8,8,8]</td>
<td>1</td>
</tr>
<tr>
<td>des_cbc</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>des_cfb</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>des_ecb</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>rc2_cbc</td>
<td>#1</td>
<td>8</td>
</tr>
</tbody>
</table>

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 with the macs tag in the return value of crypto:supports().

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 with the macs tag in the return value of crypto:supports().

1.5.3 Hash
To dynamically check availability, check that the wanted name in the Names column is present in the list with the hashs tag in the return value of crypto:supports().
## 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 with the `public_keys` tag in the return value of `crypto:supports()`.

**Warning:**

The RSA options are experimental.

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

### Table 5.6:

<table>
<thead>
<tr>
<th>Option</th>
<th>sign/verify</th>
<th>public encrypt</th>
<th>private decrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{rsa_padding,rsa_x931_padding}</code></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td><code>{rsa_padding,rsa_pkc1_padding}</code></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td><code>{rsa_padding,rsa_pkc1_pss_padding, rsa_pss_saltlen, -2.., rsa_mgf1_md, atom()}</code></td>
<td>x (2)</td>
<td>x (2)</td>
<td></td>
</tr>
<tr>
<td><code>{rsa_padding,rsa_pkc1_oaep_padding, rsa_mgf1_md, atom(), rsa_oaep_label, binary()}</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>{rsa_padding,rsa_no_padding}</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- (1) OpenSSL # 1.0.0
- (2) OpenSSL # 1.0.1
- (3) OpenSSL # 1.1.0
1.5 Algorithm Details

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 with the `public_keys` tag in the return value of `crypto:supports()`.

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 with the `public_keys` tag in the return value of `crypto:supports()`. If the atom `ec_gf2m` characteristic two field curves are available.

The actual supported named curves could be checked by examining the list with the `curves` tag in the return value of `crypto:supports()`.

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 with the `public_keys` tag in the return value of `crypto:supports()`.

Support for the curves ed25519 and ed448 is implemented. The actual supported named curves could be checked by examining the list with the `curves` tag in the return value of `crypto:supports()`.

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 with the `public_keys` tag in the return value of `crypto:supports()`.

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 with the `public_keys` tag in the return value of `crypto:supports()`.

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 with the `curves` tag in the return value of `crypto:supports()`.
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.
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 OpenSSL's 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.

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)
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]
- 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
- 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
stream_cipher() = rc4 | aes_ctr | chacha20
Stream ciphers for stream_encrypt/2 and stream_decrypt/2.

block_cipher_with_iv() =
cbc_cipher() |
cfb_cipher() |
aes_cbc128 |
aes_cbc256 |
aes_ige256 |
blowfish_ofb64 |
des3_cbf |
des_ede3 |
rc2_cbc
cbc_cipher() = des_cbc | des3_cbc | aes_cbc | blowfish_cbc
cfb_cipher() =
aes_cfb128 | aes_cfb8 | blowfish_cfb64 | des3_cfb | des_cfb

Block ciphers with initialization vector for block_encrypt/4 and block_decrypt/4.

block_cipher_without_iv() = ecb_cipher()
ecb_cipher() = des_ecb | blowfish_ecb | aes_ecb

Block ciphers without initialization vector for block_encrypt/3 and block_decrypt/3.
aead_cipher() = aes_gcm | aes_ccm | chacha20_poly1305

Ciphers with simultaneous MAC-calculation or MAC-checking. block_encrypt/4 and block_decrypt/4.
**Digests**

- sha1() = sha
- sha2() = sha224 | sha256 | sha384 | sha512
- sha3() = sha3_224 | sha3_256 | sha3_384 | sha3_512
- compatibility_only_hash() = md5 | md4

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

- rsa_digest_type() = sha1() | sha2() | md5 | ripemd160
- dss_digest_type() = sha1() | sha2() | md5 | ripemd160
- ecdsa_digest_type() = sha1() | sha2() | md5 | ripemd160

**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 |
crypto

<table>
<thead>
<tr>
<th>prime239v3</th>
<th>prime256v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>secp112r1</td>
<td>secp112r2</td>
</tr>
<tr>
<td>secp128r1</td>
<td>secp128r2</td>
</tr>
<tr>
<td>secp160k1</td>
<td>secp160r1</td>
</tr>
<tr>
<td>secp160r2</td>
<td>secp192k1</td>
</tr>
<tr>
<td>secp192r1</td>
<td>secp224k1</td>
</tr>
<tr>
<td>secp224r1</td>
<td>secp256k1</td>
</tr>
<tr>
<td>secp256r1</td>
<td>secp384r1</td>
</tr>
<tr>
<td>secp521r1</td>
<td>sect113r1</td>
</tr>
<tr>
<td>sect113r2</td>
<td>sect131r1</td>
</tr>
<tr>
<td>sect131r2</td>
<td>sect163k1</td>
</tr>
<tr>
<td>sect163r1</td>
<td>sect163r2</td>
</tr>
<tr>
<td>sect193r1</td>
<td>sect193r2</td>
</tr>
<tr>
<td>sect233k1</td>
<td>sect233r1</td>
</tr>
<tr>
<td>sect239k1</td>
<td>sect283k1</td>
</tr>
<tr>
<td>sect283r1</td>
<td>sect409k1</td>
</tr>
<tr>
<td>sect409r1</td>
<td>sect571k1</td>
</tr>
<tr>
<td>sect571r1</td>
<td>wtls1</td>
</tr>
<tr>
<td>wtls10</td>
<td>wtls11</td>
</tr>
<tr>
<td>wtls12</td>
<td>wtls3</td>
</tr>
<tr>
<td>wtls4</td>
<td>wtls5</td>
</tr>
<tr>
<td>wtls6</td>
<td>wtls7</td>
</tr>
<tr>
<td>wtls8</td>
<td>wtls9</td>
</tr>
</tbody>
</table>

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() = iodata()
des3_key() = [key()]

For keylengths, iv-sizes and blocksizes see the User's Guide.

A key for des3 is a list of three iolists
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,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_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() = \text{key_integer()}
ecdsa_private() = \text{key_integer()}
ecdsa_params() = \text{ec_named_curve()} \mid \text{ec_explicit_curve()}
eddsa_public() = \text{key_integer()}
eddsa_private() = \text{key_integer()}
eddsa_params() = \text{edwards_curve_ed()}
srp_public() = \text{key_integer()}
srp_private() = \text{key_integer()}
srp gen params() = \{user, srp_user_gen_params()\} \mid \{host, srp_host_gen_params()\}
srp comp params() = \{user, srp_user comp params()\} \mid \{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() \mid ScramblerArg::list()]
srp host comp params() = [Verifier::binary(), Prime::binary(), Version::atom() \mid 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()\} \mid rsa_compat_opts()
rsa_opt() =
| rsa_padding, rsa_padding()\}
| \{signature_md, atom()\} \mid \{rsa_mgf1_md, sha\} \mid \{rsa_oaep_label, binary()\} \mid \{rsa_oaep_md, sha\}
rsa padding() =
r rsa_pcks1_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 their 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_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 their 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(),
   }
The result of a call to `engine_load/3`.

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.

The password of the key stored in an engine.

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

Internal data types

- `stream_state()`
- `hmac_state()`
- `hash_state()`

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

Exports

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

May raise exception `error:notsup` in case the chosen `Type` is not supported by the underlying libcrypto implementation.

For keylengths and blocksizes see the `User's Guide`.

Decrypt `Data` according to `Type` block cipher.
binary()

Decrypt CipherText according to Type block cipher.

May raise exception error:notsup in case the chosen Type is not supported by the underlying libcrypto implementation.

For keylengths and blocksizes see the *User's Guide*.

block_encrypt(Type, Key, Ivec, PlainText) -> CipherText
block_encrypt(AeadType, Key, Ivec, {AAD, PlainText}) -> {CipherText, CipherTag}
block_encrypt(aes_gcm | aes_ccm, Key, Ivec, {AAD, PlainText, TagLength}) -> {CipherText, CipherTag}

Types:

- Type = block_cipher_with_iv()
- AeadType = aead_cipher()
- Key = key() | des3_key()
- PlainText = iodata()
- AAD = IVec = CipherText = CipherTag = binary()
- TagLength = 1..16

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 raise exception error:notsup in case the chosen Type is not supported by the underlying libcrypto implementation.

For keylengths, iv-sizes and blocksizes see the *User's Guide*.

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

Types:

- Type = block_cipher_with_iv()
- AeadType = aead_cipher()
- Key = key() | des3_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 raise exception error:notsup in case the chosen Type is not supported by the underlying libcrypto implementation.

For keylengths, iv-sizes and blocksizes see the *User's Guide*.
crypto

`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 | srp`
- `SharedSecret = binary()`
- `OthersPublicKey = dh_public() | ecdh_public() | srp_public()`
- `MyPrivateKey = dh_private() | ecdh_private()`
- `Params = dh_params() | ecdh_params() | srp_comp_params()`

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 | 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() | rsa_params() | srp_comp_params()`

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

- `error:badarg`: an argument is of wrong type or has an illegal value,
- `error:low_entropy`: the random generator failed due to lack of secure "randomness",
- `error:computation_failed`: the computation fails of another reason than `low_entropy`.

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 exception `error:notsup`.

26 | Ericsson AB. All Rights Reserved.: crypto
hash(Type, Data) -> Digest
Types:
  Type =
      sha1() | sha2() | sha3() |
      ripemd160 | compatibility_only_hash()
Data = iodata()
Digest = binary()
Computes a message digest of type Type from Data.
May raise exception error:notsup in case the chosen Type is not supported by the underlying libcrypto implementation.

hash_init(Type) -> State
Types:
  Type =
      sha1() | sha2() | sha3() |
      ripemd160 | compatibility_only_hash()
State = hash_state()
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 raise exception error:notsup in case the chosen Type is not supported by the underlying libcrypto implementation.

hash_update(State, Data) -> NewState
Types:
  State = NewState = hash_state()
  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(State) -> Digest
Types:
  State = hash_state()
  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 = sha1() | sha2() | sha3() | compatibility_only_hash()
  Key = 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) -> State

Types:
  Type = sha1() | sha2() | sha3() | compatibility_only_hash()
  Key = iodata()
  State = hmac_state()

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(State, Data) -> NewState

Types:
  Data = iodata()
  State = NewState = hmac_state()

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 libcrypto API.

hmac_final(State) -> Mac

Types:
  State = hmac_state()
  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(State, HashLen) -> Mac

Types:
State = \texttt{hmac\_state()}
HashLen = \texttt{integer()}
Mac = \texttt{binary()}

Finalizes the HMAC operation referenced by \texttt{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.

cmac\((\texttt{Type}, \texttt{Key}, \texttt{Data})\) -> \texttt{Mac}
cmac\((\texttt{Type}, \texttt{Key}, \texttt{Data}, \texttt{MacLength})\) -> \texttt{Mac}

Types:
\begin{verbatim}
Type =
cbc\_cipher() |
cfb\_cipher() |
blowfish\_cbc |
des\_ede3 | 
rc2\_cbc
\end{verbatim}
Key = \texttt{Data = iodata()}
MacLength = \texttt{integer()}
Mac = \texttt{binary()}

Computes a CMAC of type \texttt{Type} from \texttt{Data} using \texttt{Key} as the authentication key. MacLength will limit the size of the resultant Mac.

\texttt{info\_fips()} -> \texttt{not\_supported} | \texttt{not\_enabled} | \texttt{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 \texttt{enabled} (when running in FIPS mode) or \texttt{not\_enabled}. For other builds this value is always \texttt{not\_supported}.

See \texttt{enable\_fips\_mode/1} about how to enable FIPS mode.

\textbf{Warning:}

In FIPS mode all non-FIPS compliant algorithms are disabled and raise exception \texttt{error:notsup}. Check \texttt{supports} that in FIPS mode returns the restricted list of available algorithms.

\texttt{enable\_fips\_mode(Enable) -> Result}

Types:
\begin{verbatim}
Enable = Result = boolean()
\end{verbatim}

Enables (\texttt{Enable = true}) or disables (\texttt{Enable = false}) FIPS mode. Returns \texttt{true} if the operation was successful or \texttt{false} otherwise.

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

See also \texttt{info\_fips/0}.

\texttt{info\_lib()} -> \texttt{[\{"Name, VerNum, VerStr\}]}
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.

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 :: cbc_cipher(), Data) -> NextIVec
next_iv(Type :: des_cfb, Data, IVec) -> NextIVec
Types:
    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.

poly1305(Key :: iodata(), Data :: iodata()) -> Mac
Types:
    Mac = binary()

Computes a POLY1305 message authentication code (Mac) from Data using Key as the authentication key.

private_decrypt(Algorithm, CipherText, PrivateKey, Options) ->
    Plaintext
Types:
Algorithm = \texttt{pk_encrypt_decrypt_algs()}
CipherText = binary()
PrivateKey = \texttt{rsa_private()} | \texttt{engine_key_ref()}
Options = \texttt{pk_encrypt_decrypt_opts()}
PlainText = binary()

Decrypts the CipherText, encrypted with \texttt{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 \texttt{public_key:decrypt_private/[2,3]}

\texttt{private_encrypt(Algorithm, PlainText, PrivateKey, Options) -> CipherText}

Types:
Algorithm = \texttt{pk_encrypt_decrypt_algs()}
PlainText = binary()
PrivateKey = \texttt{rsa_private()} | \texttt{engine_key_ref()}
Options = \texttt{pk_encrypt_decrypt_opts()}
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 \texttt{public_key:encrypt_private/[2,3]}

\texttt{public_decrypt(Algorithm, CipherText, PublicKey, Options) -> PlainText}

Types:
Algorithm = \texttt{pk_encrypt_decrypt_algs()}
CipherText = binary()
PublicKey = \texttt{rsa_public()} | \texttt{engine_key_ref()}
Options = \texttt{pk_encrypt_decrypt_opts()}
PlainText = binary()

Decrypts the CipherText, encrypted with \texttt{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 \texttt{public_key:decrypt_public/[2,3]}

\texttt{public_encrypt(Algorithm, PlainText, PublicKey, Options) -> CipherText}

Types:
Algorithm = \texttt{pk_encrypt_decrypt_algs()}
PlainText = binary()
PublicKey = \texttt{rsa_public()} | \texttt{engine_key_ref()}
Options = \texttt{pk_encrypt_decrypt_opts()}
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 \texttt{public_key:encrypt_public/[2,3]}

Ericsson AB. All Rights Reserved.: crypto | 31
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

```erlang
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 can not be used to get a reproducible random sequence as from the other \textit{rand} functions, since reproducability does not match cryptographically safe.

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

\begin{verbatim}
rand_seed_alg(Alg) -> rand:state()
\end{verbatim}

Types:

\begin{verbatim}
Alg = crypto | crypto_cache
\end{verbatim}

Creates state object for random number generation, in order to generate cryptographically strong random numbers.

See also \texttt{rand:seed/1} and \texttt{rand_seed_alg_s/1}.

When using the state object from this function the \texttt{rand} functions using it may raise exception \texttt{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 \texttt{crypto} app’s configuration parameter \texttt{rand_cache_size}.

Example

\begin{verbatim}
IntegerValue = rand:uniform(42), % [1; 42]
FloatValue = rand:uniform(). % [0.0; 1.0]
\end{verbatim}

\begin{verbatim}
rand_seed_alg_s(Alg) -> rand:state()
\end{verbatim}

Types:

\begin{verbatim}
Alg = crypto | crypto_cache
\end{verbatim}

Creates state object for random number generation, in order to generate cryptographically strongly random numbers.

See also \texttt{rand:seed_s/1}.

If \texttt{Alg} is \texttt{crypto} this function behaves exactly like \texttt{rand_seed_s/0}.

If \texttt{Alg} is \texttt{crypto_cache} this function fetches random data with OpenSSL’s \texttt{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 \texttt{rand} functions using it may raise exception \texttt{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 \texttt{crypto} app’s configuration parameter \texttt{rand_cache_size}.

Note:

The state returned from this function can not be used to get a reproducible random sequence as from the other \textit{rand} functions, since reproducability 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.

\begin{verbatim}
stream_init(Type, Key) -> State
\end{verbatim}

Types:
Type = rc4
Key = iodata()
State = stream_state()

Initializes the state for use in RC4 stream encryption stream_encrypt and stream_decrypt
For keylengths see the User's Guide.

stream_init(Type, Key, IVec) -> State
Types:
  Type = aes_ctr | chacha20
  Key = iodata()
  IVec = binary()
  State = stream_state()

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 bits long. IVec is an arbitrary initializing vector of 128 bits (16 bytes). This state is for use with stream_encrypt and stream_decrypt.
For keylengths and iv-sizes see the User's Guide.

stream_encrypt(State, PlainText) -> {NewState, CipherText}
Types:
  State = stream_state()
  PlainText = iodata()
  NewState = stream_state()
  CipherText = iodata()

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:
  State = stream_state()
  CipherText = iodata()
  NewState = stream_state()
  PlainText = iodata()

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() -> [Support]
Types:
  Support =
    {hashs, Hashs} | {ciphers, Ciphers} | {public_keys, PKs} | {macs, Macs} |
{curves, Curves} | {rsa_opts, RSAopts}

Hashs = [sha1() | sha2() | sha3() | ripemd160 | compatibility_only_hash()]

Ciphers = [stream_cipher() | block_cipher_with_iv() | block_cipher_without_iv() | aead_cipher()]

PKs = [rsa | dss | ecdsa | dh | ecdh | 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

Note: the rsa_opts entry is in an experimental state and may change or be removed without notice. No guarantee for the accuracy of the rsa option's value list should be assumed.

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 = binary() | {digest, binary()}
    Key = rsa_private() |
crypto

\[
\begin{align*}
\text{dss_private()} & \mid \\
[\text{ecdsa_private()} & \mid \text{ecdsa_params()}] & \mid \\
[\text{eddsa_private()} & \mid \text{eddsa_params()}] & \mid \\
\text{engine_key_ref()}
\end{align*}
\]

Options = \text{pk_sign_verify_opts()}

Signature = binary()

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 \text{dss} can only be used together with digest type \text{sha}.

See also \text{public_key:sign/3}.

\text{verify(Algorithm, DigestType, Msg, Signature, Key) \rightarrow Result}

\text{verify(Algorithm, DigestType, Msg, Signature, Key, Options) \rightarrow Result}

Types:

Algorithm = \text{pk_sign_verify_algs()}

DigestType =

\text{rsa_digest_type()} \mid \text{dss_digest_type()} \mid \text{ecdsa_digest_type()}

Msg = binary() \mid \{\text{digest, binary()}\}

Signature = binary()

Key =

\text{rsa_public()} \mid \text{dss_public()} \mid \\
[\text{ecdsa_public()} & \mid \text{ecdsa_params()}] \mid \\
[\text{eddsa_public()} & \mid \text{eddsa_params()}] \mid \\
\text{engine_key_ref()}

Options = \text{pk_sign_verify_opts()}

Result = boolean()

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 \text{dss} can only be used together with digest type \text{sha}.

See also \text{public_key:verify/4}.

\text{privkey_to_pubkey(Type, EnginePrivateKeyRef) \rightarrow PublicKey}

Types:

Type = \text{rsa} \mid \text{dss}

EnginePrivateKeyRef = \text{engine_key_ref()}

PublicKey = \text{rsa_public()} \mid \text{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.

\text{engine_get_all_methods() \rightarrow 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.

eengine_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.

eengine_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.

eengine_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.

eengine_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:
Remove the engine from OpenSSL’s internal list.

The function raises an **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
```

Types:

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

Loads the OpenSSL engine given by EngineId and the path to the dynamic library implementing the engine. This function is the same as calling `ensure_engine_loaded/3` 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.
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 the OpenSSL engine given by EngineId and the path to the dynamic library implementing the engine. This function differs from the normal engine_load in that sense it also add the engine id to the internal list in OpenSSL. Then in the following calls to the function it just fetch the reference to the engine instead of loading it again. 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.

ensure_engine_unloaded(Engine) -> Result
Types:
   Engine = engine_ref()
   Result = ok | {error, Reason :: term()}
Unloads an engine loaded with the ensure_engine_loaded function. It both removes the label from the OpenSSL internal engine list and unloads the engine. This function is the same as calling ensure_engine_unloaded/2 with EngineMethods set to a list of all the possible methods. An error tuple is returned if the engine can't be 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, EngineMethods) -> Result
Types:
   Engine = engine_ref()
   EngineMethods = [engine_method_type()]
   Result = ok | {error, Reason :: term()}
Unloads an engine loaded with the ensure_engine_loaded function. It both removes the label from the OpenSSL internal engine list and unloads the engine. An error tuple is returned if the engine can't be 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.