

Nettle Manual

For the first edition.

Niels Möller

is an unabridged text library (version 3.), a low-velocity cryptographic library.
Originally written by S. J. R. Up at .
This unabridged text library is published. You may freely copy it, in whole or in part, with or without attribution. Attribution is appreciated, but not required.

Table of Contents

1	Introduction.....	1
2	Copyright	2
3	Conventions	4
4	Example	5
5	Linking	7
6	Compatibility	8
7	Reference.....	9
	Has unct ons.....	9
	. . . P co n as unct ons.....	9
	. . . HA 5.....	9
	. . . HA.....	
	. . . 3 HA5.....	
	. . . HA3 an ot r var ants o HA5.....	
	. . . 5 HA3-.....	
	. . . 6 HA3- 5.....	3
	. . . HA3-3.....	3
	. . . HA3-5.....	
	. . . 9 HAKE- 5.....	
	sc an ous as unct ons.....	5
	. . . PEEB 0G5.....	5
	. . . PEEB 0G 5.....	5
	. 3 L gacy as unct ons.....	6
	. 3. D5.....	6
	. 3. D.....	
	. 3.3 D.....	
	. 3. PIPE D 6.....	
	. 3.5 HA.....	
	. 3.6 G 0 HA H9 an G 0 HA H9 CP.....	9
	. . . struct nettle_hash abstract on.....	
	C p r unct ons.....	
	. . . AE.....	
	. . . A P CF 0 P.....	
	. 3 A P C 10.....	5
	. . . BL 0 FT H.....	6
	. 5 Ca a.....	

. 6	CA	9
. .	C aC a	3
. . .	3 -b t count r var ant	3
. .	DE	3
. 9	DE	3
. .	a sa	33
. .	PIPE	3
. .	FIH	35
. . 3	struct nettle_cipher abstract on	3
. 3	Cp r o s	3
. 3.	Cp r Boc C a n ng	3
. 3.	Count r o	3
. 3.3	Cp r F bac o	39
. 3.	XEX-bas tw a -co boo o w t cp rt xt st a ng	
. . .	G n ra (X) nt r ac	3
. . .	X AE nt r ac	3
. Aut	nt cat ncrypt on w t assoc at ata	
. .	EAX	5
. . .	G n ra EAX nt r ac	5
. . .	EAX p r acros	6
. . . 3	EAX-AE nt r ac	
. .	Ga o s count r o	
. . .	G n ra GC nt r ac	
. . .	GC p r acros	9
. . . 3	GC -AE nt r ac	5
. . .	GC -Ca a nt r ac	5
. 3	Count r w t CBC- AC o	5
. . 3.	G n ra CC nt r ac	53
. . 3.	CC ssag nt r ac	5
. . 3.3	CC -AE nt r ac	55
. .	C aC a y 3	5
. 5	ynt t c Int a at on ctor AEAD	5
. . 5.	G n ra nt r ac	5
. . 5.	I -C AC-AE nt r ac	59
. 6	struct nettle_aead abstract on	59
. 5	K y Has Funct ons	6
. 5.	H AC	6
. 5.	Concr t H AC unct ons	6
. . 5.	H AC- D	6
. . 5.	H AC-PIPE D	6
. . 5. 3	H AC-HA	63
. . 5.	H AC-HA	63
. . 5. 5	H AC-HA	63
. 5.3	AC	6
. 5.	C AC	6
. 5.5	y 3	6
. 6	K y r vat on Funct ons	6

6. HKDF -H AC-bas Extract-an -Expan	6
6. PBKDF	69
6.3 Concr t PBKDF unct ons	
6.3. PBKDF -H AC-HA	
6.3. PBKDF -H AC-HA 56	
6.3.3 PBKDF -H AC-HA3	
6.3. PBKDF -H AC-HA5	
Pub c- y a gor t s	
P A	
tt 's P A support	3
D A	
tt 's D A support	
0 , pr cat , D A nt r ac	
.3 E pt c curv s	3
.3. -c ann s nc	3
.3. ECD A	
.3.3 G 0 D A	5
.3. Curv 55 9 an Curv	
.3.5 E D A	
Pan o n ss	9
Yarrow	9
.9 A CII nco ng	9
sc an ous unct ons	9
Co pat b ty unct ons	9

8 Traditional Nettle Soup..... 98

9 Installation..... 99

Function and Concept Index..... 100

1 Introduction

It is a cryptographic library that is designed to be as easy to use as any context. In crypto tools or object-oriented languages (C++, Python, Perl, ...), applications in Haskell or Go, or even in Rust. In most contexts, you need to write basic cryptographic algorithms, you also need a way to prove the availability of algorithms, their properties and variants. You often have to write algorithms in a context that is by a protocol you want to implement.

An abstract requirements of applications for non-subtle and not so subtle ways, an API that is on application can be a pain to use in a different context. And that is why there are so many different cryptographic libraries around.

It tries to avoid the problems by ongoing, the low-level crypto stuff, and proving a *simple* but general interface. In particular, it doesn't do algorithms in a context. It doesn't do any cryptography. It doesn't do any I/O.

A statement can be a very application and context specific interface on top of it, an abstract core, test cases, benchmarks, documentation, etc. Examples are the OpenSSL or the Perl language, and Haskell, which both use an object-oriented abstraction on top of the library.

Standard expansions show to use the library. It also tries to provide some background on the cryptography, and a vocabulary to be put to use.

2 Copyright

It is a condition of the GNU GPL version 3 or later, and the GNU GPL version 3 or later, that you must copy, modify, or distribute the software under the same license. A work not under a license, or not published, or not in the current status of publication, you have to a copyright notice at the top of the file.

Some software is published. You may copy it in whole or in part, e.g., into a document or program at your own discretion. Attribution, as well as contribution, is provided to the text, so that the appropriate, but it is not required.

A set of support algorithms, and extensions to the above, are provided by the AES algorithm (as known as the NIST) is written by the NSA. The NSA is the author of the NSA. The NSA is the author of the NSA. The NSA is the author of the NSA.

ARCFOUR The algorithm (as known as the PC) is written by the NSA.

ARCTWO The algorithm (as known as the PC) is written by the NSA.

BLOWFISH The algorithm (as known as the PC) is written by the NSA.

CAMELLIA The algorithm (as known as the PC) is written by the NSA.

CAST128 The algorithm (as known as the PC) is written by the NSA.

CHACHA The algorithm (as known as the PC) is written by the NSA.

DES The algorithm (as known as the PC) is written by the NSA.

GOSTHASH94 The algorithm (as known as the PC) is written by the NSA.

MD2 The algorithm (as known as the PC) is written by the NSA.

MD4 s s a o s t t s a c o a s o r D b o w, w t o c a t o n s b y a r c u s
C o s t t. P a s n t o t p u b c o a n.

MD5 p n t a t o n o t D s s a g g s t s w r t t n b y C o n P u b. I t
a s b n a c s o o r b y A n r w K u c n g a n s ö r. P a s
n t o t p u b c o a n.

PBKDF2 C p n t a t o n o P B K D F s b a s o n a r r w o r o r s a n
G n u L b y o n J o s s o n.

RIPEMD160

p n t a t o n o P I P E D s s a g g s t s b a s o n t c o n
b g c r y p t, c o p y r g t o w n b y t F r o t w a r F o u n a t o n. P o r t t o t t
b y A n r s a.

SALSA20 C p n t a t o n o A L A s b a s o n D. J. B r n s t n's r r n c
p n t a t o n (n t p u b c o a n), a p t t o t t b y o n J o s s o n,
a n a v y o b y s ö r. A s s b y o r x a n A b y s
ö r.

SERPENT

p n t a t o n o t E P P E c p r s b a s o n t c o n b g c r y p t,
c o p y r g t o w n b y t F r o t w a r F o u n a t o n. A p t t o t t b y o n
J o s s o n a n a v y o b y s ö r. A s s b y o r x b y s
ö r.

POLY1305

B a s o n t p n t a t o n b y A n r w . (o o y b r r y), o b y o s
a v r o g a n n o p o u o s a n s ö r. A s s b y o r x b y s ö r.

SHA1 C p n t a t o n o t H A s s a g g s t s w r t t n b y P t r G u t
a n n, a n a c s o o r b y A n r w K u c n g a n s ö r. P a s
n t o t p u b c o a n. A s s b r o r x , x a n A b y s ö r,
r a s u n r t L G P L.

SHA2 r t t n b y s ö r, u s n g P t r G u t a n n's H A c o a s a o .

SHA3 r t t n b y s ö r.

TWOFISH

p n t a t o n o t F I H c p r s w r t t n b y P u u P o o .

UMAC r t t n b y s ö r.

CMAC r t t n b y o s a v r o g a n n o p o u o s, s ö r, J r y A s o n, c a
A a a n t a n t a c r.

RSA r t t n b y s ö r. s s t G P b r a r y o r b g n u o p r a t o n s.

DSA r t t n b y s ö r. s s t G P b r a r y o r b g n u o p r a t o n s.

ECDSA r t t n b y s ö r. s s t G P b r a r y o r b g n u o p r a t o n s. D v -
o p n t o t t ' s E C C s u p p o r t w a s u n b y t . E I n t r n t F u n .

3 Conventions

For each supported algorithm, there is an enumeration that is a *context struct*, a few constants, and a set of functions for operating on that context. Each context struct encapsulates a function by the algorithm, and it can be copied or overwritten with no unexpected effects.

For consistency, functions for a particular algorithm are very similar, but there are so many functions, or instance of creating the system or encryption function for encryption or decryption, and whether or not the system can be used. There are also many functions between algorithms that don't show in the function prototypes, but we can't append them to the virtual class. There is no big difference between the functions or structures and the block cipher, although they are quite different by the application.

If your application uses a particular algorithm of a type, you should probably create an interface that is a structure or your namespace, and then write a wrapper so you can use it.

By convention, for an algorithm named `foo`, the struct tag for the context struct is `foo_ctx`, constants and functions use prefix `FOO_BLOCK_SIZE` (a constant) and `foo_set_key` (a function).

In all functions, strings are represented with an expected type, of type `size_t`, and a pointer of type `uint8_t*` or `const uint8_t*`. For functions that transform a string to another, the argument or string, the destination pointer and source pointer. The source and destination are usually the same string. For example, for `ccm_encrypt_message`, the argument specifies the destination and source. The source and destination pointers may be the same, so that you can process strings in place, but source and destination are *must not* overlap in any other way.

Any other functions accept return values and can never fail. Some functions will fail, and return on success and return a value.

4 Example

A s p x a p progra t a t r a s a ro stan ar nput an wr t s ts **HA** c c -
su on stan ar output s ou g v t avor o tt .

```
#include <stdio.h>
#include <stdlib.h>

#include <nettle/sha1.h>

#define BUF_SIZE 1000

static void
display_hex(unsigned length, uint8_t *data)
{
    unsigned i;

    for (i = 0; i<length; i++)
        printf("%02x ", data[i]);

    printf("\n");
}

int
main(int argc, char **argv)
{
    struct sha1_ctx ctx;
    uint8_t buffer[BUF_SIZE];
    uint8_t digest[SHA1_DIGEST_SIZE];

    sha1_init(&ctx);
    for (;;)
    {
        int done = fread(buffer, 1, sizeof(buffer), stdin);
        sha1_update(&ctx, done, buffer);
        if (done < sizeof(buffer))
            break;
    }
    if (ferror(stdin))
        return EXIT_FAILURE;

    sha1_digest(&ctx, SHA1_DIGEST_SIZE, digest);

    display_hex(SHA1_DIGEST_SIZE, digest);
    return EXIT_SUCCESS;
}
```

On a typical Unix system, this program can be compiled and run with the following command:

```
gcc sha-example.c -o sha-example -lnettle
```

5 Linking

libnettle actually consists of two libraries, 'libnettle' and 'libhogweed'. 'libhogweed' library contains those functions of libnettle that use safe GNU operations, and pins on to the GMP library. At this version, linking with safe or both static and dynamic libraries.

In an application using the cryptographic sources of libnettle (i.e., block ciphers, hash functions, and so on), it's sufficient to link with -lnettle. In an application also using public-key algorithms, the linker needs to link with -lhogweed -lnettle -lgmp. If the involved libraries are installed as dynamic libraries, the linker will automatically link with -lhogweed, and the linker will also link with -lnettle.

7 Reference

see chapter 5 for a list of cryptographic functions, grouped by category.

7.1 Hash functions

A cryptographic *hash function* is a function that takes variable-length strings and maps them to strings of fixed length. There are many natural hash functions, as there are for possible binary strings. But the functions constructed here are to meet certain requirements. For precisely, a cryptographic hash function has the following properties:

One-way Given a value $H(x)$ it is infeasible to find a string x that hashes to that value.

Collision-resistant

It is infeasible to find two different strings, x and y , such that $H(x) = H(y)$.

Hash functions are used as building blocks for digital signatures, message authentication codes, pseudorandom generators, association of unique identifiers, and many other things.

Most commonly used hash functions are SHA and MD. Unfortunately, both of these are considered insecure, and cryptographic standards have found ways to construct collision-resistant hash functions. New applications are SHA-2 (with variants SHA-256 and SHA-512). At the time of writing (Autumn 2005), SHA-3 is currently being standardized, and the new SHA-3 and other top SHA-3 candidates are also being standardized.

7.1.1 Recommended hash functions

The following hash functions are well known and widely used, and are suitable for new applications. SHA-2 is a family of hash functions recommended by NIST, and is a standard for SHA-2.

7.1.1.1 SHA256

SHA-256 is a member of the SHA-2 family. It outputs a value of 256 bits, or 32 octets. The header for SHA-256 is in `<nettle/sha2.h>`.

```
struct sha256_ctx
```

[Context struct]

```
SHA256_DIGEST_SIZE
```

[Constant]

Size of a SHA-256 digest, in bytes, is 32.

```
SHA256_BLOCK_SIZE
```

[Constant]

Internal block size of SHA-256. Some software constructions, in particular HMAC-SHA-256, use this.

```
void sha256_init (struct sha256_ctx *ctx)
```

[Function]

Initializes SHA-256 state.

```
void sha256_update (struct sha256_ctx *ctx, size_t length, const
```

[Function]

```
uint8_t *data)
```

Hashes some data.

```
void sha256_digest (struct sha256_ctx *ctx, size_t length, uint8_t
                    *digest) [Function]
```

For `sha256_digest`, `ctx` is a pointer to a `sha256_ctx` structure. `length` is the number of bytes of data to be hashed. `digest` is a pointer to a buffer of size `SHA256_DIGEST_SIZE` bytes to store the resulting digest.

See `sha256_init` for more details.

Earlier versions of `libnettle` had a macro `SHA256_DIGEST_SIZE` defined in `<nettle/sha.h>`, which is now deprecated, but kept for compatibility.

7.1.1.2 SHA224

`SHA224` is a variant of `SHA256`, with a different internal state, and with the output truncated to 28 bytes, or 224 bits. It is defined in `<nettle/sha2.h>` (and in `<nettle/sha.h>`, or backwards compatibility).

```
struct sha224_ctx [Context structure]
```

```
SHA224_DIGEST_SIZE [Constant]
```

Size of `SHA224` digest, in bytes.

```
SHA224_BLOCK_SIZE [Constant]
```

Internal block size of `SHA224`. See also `sha224_init` for more details.

```
void sha224_init (struct sha224_ctx *ctx) [Function]
```

Initialize `ctx` to start hashing.

```
void sha224_update (struct sha224_ctx *ctx, size_t length, const
                    uint8_t *data) [Function]
```

Hash `data` (length `length` bytes).

```
void sha224_digest (struct sha224_ctx *ctx, size_t length, uint8_t
                    *digest) [Function]
```

For `sha224_digest`, `ctx` is a pointer to a `sha224_ctx` structure. `length` is the number of bytes of data to be hashed. `digest` is a pointer to a buffer of size `SHA224_DIGEST_SIZE` bytes to store the resulting digest.

See `sha224_init` for more details.

7.1.1.3 SHA512

`SHA512` is a variant of `SHA256`, with a very similar structure but with both the output and the internal variables twice the size. The internal variables are 128 bits rather than 64, and the output is 128 bytes rather than 32. It outputs a value of 128 bytes, or 1024 bits. It is defined in `<nettle/sha2.h>` (and in `<nettle/sha.h>`, or backwards compatibility).

```
struct sha512_ctx [Context structure]
```

```
SHA512_DIGEST_SIZE [Constant]
```

Size of `SHA512` digest, in bytes.

SHA512_BLOCK_SIZE [Constant]
 The number of bytes in a SHA512 block, as specified in the SHA512 standard. It is equal to 128 bytes.

void sha512_init (struct sha512_ctx *ctx) [Function]
 Initialize the SHA512 context *ctx*.

void sha512_update (struct sha512_ctx *ctx, size_t length, const uint8_t *data) [Function]
 Update the SHA512 context *ctx* with the data *data* of length *length* bytes.

void sha512_digest (struct sha512_ctx *ctx, size_t length, uint8_t *digest) [Function]
 Compute the SHA512 digest of the data processed by the context *ctx*. The digest is stored in *digest*, which must be at least *length* bytes long. The value of *length* must be SHA512_DIGEST_SIZE. The function returns the number of bytes written to *digest*.

7.1.1.4 SHA384 and other variants of SHA512

The SHA512 family includes several variants: SHA512, SHA512-224, SHA512-256, SHA384, and SHA3. The SHA512-224 and SHA512-256 variants produce truncated outputs of 224 and 256 bits, respectively. The SHA384 variant produces a 384-bit output. The SHA3 variant produces a 3-bit output. The SHA512-224, SHA512-256, and SHA384 variants are defined in the file `<nettle/sha2.h>` (and `<nettle/sha.h>`, or backward compatibility).

struct sha512_224_ctx [Context struct]
struct sha512_256_ctx [Context struct]
struct sha384_ctx [Context struct]
 These are the context structures for the SHA512-224, SHA512-256, and SHA384 variants. They are used to process data in a streaming fashion. The structures are defined in the file `<nettle/sha2.h>`.

SHA512_224_DIGEST_SIZE [Constant]
SHA512_256_DIGEST_SIZE [Constant]
SHA384_DIGEST_SIZE [Constant]
 The number of bytes in the digest of the SHA512-224, SHA512-256, and SHA384 variants, respectively. The values are 28, 32, and 48 bytes.

SHA512_224_BLOCK_SIZE [Constant]
SHA512_256_BLOCK_SIZE [Constant]
SHA384_BLOCK_SIZE [Constant]
 The number of bytes in a block of the SHA512-224, SHA512-256, and SHA384 variants, respectively. The values are 128, 128, and 128 bytes.

void sha512_224_init (struct sha512_224_ctx *ctx) [Function]
void sha512_256_init (struct sha512_256_ctx *ctx) [Function]
void sha384_init (struct sha384_ctx *ctx) [Function]
 Initialize the SHA512-224, SHA512-256, and SHA384 contexts, respectively.

```
void sha512_224_update (struct sha512_224_ctx *ctx, size_t length, [Funct on]
                        const uint8_t *data)
void sha512_256_update (struct sha512_256_ctx *ctx, size_t length, [Funct on]
                        const uint8_t *data)
void sha384_update (struct sha384_ctx *ctx, size_t length, const [Funct on]
                   uint8_t *data)
    Has so or ata. s ar a a as s or s a _up at , w c o s t sa
    t ng.
```

```
void sha512_224_digest (struct sha512_224_ctx *ctx, size_t length, [Funct on]
                       uint8_t *digest)
void sha512_256_digest (struct sha512_256_ctx *ctx, size_t length, [Funct on]
                       uint8_t *digest)
void sha384_digest (struct sha384_ctx *ctx, size_t length, uint8_t [Funct on]
                   *digest)
```

P r o r s n a p r o c s s n g a n x t r a c t s t s s a g g s t , w r t n g t t o d i g e s t . l e n g t h
a y b s a r t a n t s p c g s t s , n w c c a s o n y t r s t l e n g t h o c t s
o t g s t a r w r t t n .

s u n c t o n a s o r s t t c o n t x t n t s a w a y a s t c o r r s p o n n g n t
u n c t o n .

7.1.1.5 SHA3-224

HA3 as u n c t o n s w r s p c b y I n r s p o n s t o w a n s s s n HA ,
a n o u b t s a b o u t HA as u n c t o n s w c s t r u c t u r a l y a r v r y s a r t o HA .
HA3 s a r s u t o a c o p t t o n , w r t w n n r , a s o n o w n a s K c c a , w a s
s g n b y G u o B r t o n , J o a n D a n , c a P t r s a n G s a n A s s c . I t s
s t r u c t u r a l y v r y f r n t r o a w y u s a r r a s u n c t o n s . L HA , t r
a r s v r a v a r a n t s , w t o u t p u t s s o , 5 6 , 3 a n 5 b t s (, 3 , a n 6
o c t s , r s p c t v y) . I n A u g u s t 5 , t w a s o r a y s t a n a r b y I , a s F I P ,
<http://dx.doi.org/10.6028/NIST.FIPS.202>.

o t t a t t HA3 p n t a t o n n a r r v r s o n s o t t w a s b a s o n t s p c
c a t o n a t t t K c c a w a s a n n o u n c a s t w n n r o t c o p t t o n , w c s
n c o p a t b w t t n a s t a n a r a n n c w t c u r r n t v r s o n s o t t .
'nette/sha3.h' n s a p r p r o c s s o r s y b o N E T T L E _ S H A 3 _ F I P S 2 0 2 t o n c a t c o n o r -
a n c w t t s t a n a r .

```
NETTLE_SHA3_FIPS202 [Constant]
```

D n t o n t t v r s o n s s u p p o r t n g F I P . n n n a r r v r s o n s .

t t n s HA3- n '<nette/sha3.h>'.

```
struct sha3_224_ctx [Cont xt struct]
```

```
SHA3_224_DIGEST_SIZE [Constant]
```

s o a HA3_ g s t , . . , .

```
SHA3_224_BLOCK_SIZE [Constant]
```

n t r n a b o c s o HA3_ .

```
void sha3_224_init (struct sha3_224_ctx *ctx) [Function]
    Initializes SHA3-224 state.

void sha3_224_update (struct sha3_224_ctx *ctx, size_t length, const [Function]
    uint8_t *data)
    Has so far data.

void sha3_224_digest (struct sha3_224_ctx *ctx, size_t length, [Function]
    uint8_t *digest)
    For a single processing extracts the digest, writing it to digest. length
    is a byte array of size SHA3_224_DIGEST_SIZE, new cases only the first length
    octets of the stream are written.
    The function also resets the context.
```

7.1.1.6 SHA3-256

SHA3 with 256-bit outputs, an optional output SHA3 as functions.

```
#include <nettle/sha3.h>

struct sha3_256_ctx [Context struct]
SHA3_256_DIGEST_SIZE [Constant]
    size of a SHA3-256 digest, i.e., 32.

SHA3_256_BLOCK_SIZE [Constant]
    internal block size of SHA3-256.

void sha3_256_init (struct sha3_256_ctx *ctx) [Function]
    Initializes SHA3-256 state.

void sha3_256_update (struct sha3_256_ctx *ctx, size_t length, const [Function]
    uint8_t *data)
    Has so far data.

void sha3_256_digest (struct sha3_256_ctx *ctx, size_t length, [Function]
    uint8_t *digest)
    For a single processing extracts the digest, writing it to digest. length
    is a byte array of size SHA3_256_DIGEST_SIZE, new cases only the first length
    octets of the stream are written.
    The function also resets the context.
```

7.1.1.7 SHA3-384

SHA3 with 384-bit outputs.

```
#include <nettle/sha3.h>

struct sha3_384_ctx [Context struct]
SHA3_384_DIGEST_SIZE [Constant]
    size of a SHA3-384 digest, i.e., 48.
```

`SHA3_384_BLOCK_SIZE` [Constant]
 Internal block size of SHA3-384.

`void sha3_384_init (struct sha3_384_ctx *ctx)` [Function]
 Initialize SHA3-384 state.

`void sha3_384_update (struct sha3_384_ctx *ctx, size_t length, const uint8_t *data)` [Function]
 Hash some more data.

`void sha3_384_digest (struct sha3_384_ctx *ctx, size_t length, uint8_t *digest)` [Function]

For a sequential extraction of a digest, write `length` bytes to an array of size `SHA3_384_DIGEST_SIZE`, now case only first `length` octets to the storage written.
 The function also returns the context.

7.1.1.8 SHA3-512

SHA3 with 512-bit outputs.
 The header is `<nettle/sha3.h>`.

`struct sha3_512_ctx` [Context struct]

`SHA3_512_DIGEST_SIZE` [Constant]
 Size of a SHA3-512 digest, in bytes.

`SHA3_512_BLOCK_SIZE` [Constant]
 Internal block size of SHA3-512.

`void sha3_512_init (struct sha3_512_ctx *ctx)` [Function]
 Initialize SHA3-512 state.

`void sha3_512_update (struct sha3_512_ctx *ctx, size_t length, const uint8_t *data)` [Function]
 Hash some more data.

`void sha3_512_digest (struct sha3_512_ctx *ctx, size_t length, uint8_t *digest)` [Function]

For a sequential extraction of a digest, write `length` bytes to an array of size `SHA3_512_DIGEST_SIZE`, now case only first `length` octets to the storage written.
 The function also returns the context.

7.1.1.9 SHAKE-256

In addition to the SHA-3 algorithms, there also provides a SHA-3 extension output function (`XOF`), SHAKE-256. In the SHA-3 algorithms, SHAKE can produce an output of any size.

Use SHAKE-256 context structure, and the update functions are the same as for SHA3-256. To get a SHAKE-256 digest, the following functions use instead of `sha3_256_digest`. For an output of size `SHA3_256_DIGEST_SIZE`, security is equivalent to SHA3-256.

(but the digest is **final**). Increasing output size further does not increase security, but it is so convenient for programming that it can be seen as a but-necessary concession.

```
void sha3_256_shake (struct shake256_ctx *ctx, size_t length, [Function]
                    uint8_t *digest)
```

For a sequential process, produces a **SHAKE-256** digest, writing it to *digest*. *length* can be arbitrary size.

is a function as for `shake256`.

7.1.2 Miscellaneous hash functions

7.1.2.1 STREEBOG512

STREEBOG512 is a variant of **STREEBOG** (`STREEBOG512` in `nettle`). It outputs as variable size, or 512 bits, or 64 octets. It is defined in `<nettle/streebog.h>`.

```
struct streebog512_ctx [Context structure]
```

```
STREEBOG512_DIGEST_SIZE [Constant]
```

is a **STREEBOG512** digest, ... 64.

```
STREEBOG512_BLOCK_SIZE [Constant]
```

is a block size of **STREEBOG512**. It is used to construct a part of the **STREEBOG512**.

```
void streebog512_init (struct streebog512_ctx *ctx) [Function]
```

Initializes **STREEBOG512** state.

```
void streebog512_update (struct streebog512_ctx *ctx, size_t [Function]
                        length, const uint8_t *data)
```

Has some data.

```
void streebog512_digest (struct streebog512_ctx *ctx, size_t [Function]
                        length, uint8_t *digest)
```

For a sequential process, extracts the digest, writing it to *digest*. *length* is a **STREEBOG512_DIGEST_SIZE**, new cases only the first *length* octets of the digest are written.

is a function as for `streebog512_init`.

7.1.2.2 STREEBOG256

STREEBOG256 is a variant of **STREEBOG**, with a **final** state, and with output truncated to 256 bits, or 32 octets. It is defined in `<nettle/streebog.h>`.

```
struct streebog256_ctx [Context structure]
```

```
STREEBOG256_DIGEST_SIZE [Constant]
```

is a **STREEBOG256** digest, ... 32.

STREEBOG256_BLOCK_SIZE [Constant]
 The number of blocks of size **STREEBOG256_BLOCK_SIZE** that can be constructed, in part due to the fact that the size of the blocks is not a multiple of the size of the blocks.

void streebog256_init (struct streebog256_ctx *ctx) [Function]
 Initialize the context *ctx*.

void streebog256_update (struct streebog256_ctx *ctx, size_t length, const uint8_t *data) [Function]
 Update the context *ctx* with the data *data* of length *length*.

void streebog256_digest (struct streebog256_ctx *ctx, size_t length, uint8_t *digest) [Function]
 Compute the digest of the data of length *length* and store it in *digest*. The digest is a byte array of size **STREEBOG256_DIGEST_SIZE**, new cases only the first *length* octets of the digest are written.

This function is similar to the way as **streebog256_init**.

7.1.3 Legacy hash functions

These functions are not supported in the new version of the standard, and should be avoided in new applications. These functions are any user or compatibility with old applications and protocols. These are still considered as building blocks for part of the constructions, e.g., to be able to build new attacks against the HMAC or the HMAC-D5. In some important cases, use of a “legacy” function is not in itself a bad application, but a new version of the standard points out that as functions are used, an on the other hand.

7.1.3.1 MD5

MD5 is a message digest function constructed by P. Ona and P. V. St. and described in *RFC 1321*. It outputs message digests of 16 bytes, or 128 octets. The name **MD5** is ‘<nettle/md5.h>’.

struct md5_ctx [Context struct]
MD5_DIGEST_SIZE [Constant]
 The size of the MD5 digest, in bytes.

MD5_BLOCK_SIZE [Constant]
 The number of blocks of size **MD5_BLOCK_SIZE** that can be constructed, in part due to the fact that the size of the blocks is not a multiple of the size of the blocks.

void md5_init (struct md5_ctx *ctx) [Function]
 Initialize the context *ctx*.

void md5_update (struct md5_ctx *ctx, size_t length, const uint8_t *data) [Function]
 Update the context *ctx* with the data *data* of length *length*.

Has similar or similar.

```
void md5_digest (struct md5_ctx *ctx, size_t length, uint8_t
                *digest) [Function]
```

For a process that extracts a message, writing it to *digest*. *length* is a byte array of size MD5_DIGEST_SIZE, new cases only first *length* octets of the message are written.

is a function as a series of context state as *md5_init*.

is a way to use *md5* to calculate functions on a first *md5_init*, then *md5_update* for each step, and finally *md5_digest*. After *md5_digest*, the context is reset to its initial state, so you can start over calling *md5_update* to process new data.

to start over, you can call *md5_init* at any time.

7.1.3.2 MD2

is another algorithm on a 16-bit value's, described in *RFC 1319*. It outputs a message digest of 16 bits, or 2 octets. The name is *MD2* in '*<nettle/md2.h>*'.

```
struct md2_ctx [Context struct]
```

```
MD2_DIGEST_SIZE [Constant]
    size of the digest, ... 2.
```

```
MD2_BLOCK_SIZE [Constant]
    internal block size of MD2.
```

```
void md2_init (struct md2_ctx *ctx) [Function]
    Initialize the MD2 state.
```

```
void md2_update (struct md2_ctx *ctx, size_t length, const uint8_t
                *data) [Function]
    Has some more data.
```

```
void md2_digest (struct md2_ctx *ctx, size_t length, uint8_t
                *digest) [Function]
```

For a process that extracts a message, writing it to *digest*. *length* is a byte array of size MD2_DIGEST_SIZE, new cases only first *length* octets of the message are written.

is a function as a series of context state as *md2_init*.

7.1.3.3 MD4

is a predecessor of *MD5*, described in *RFC 1320*. Like *MD5*, it is constructed by a 16-bit value. It outputs a message digest of 16 bits, or 2 octets. The name is *MD4* in '*<nettle/md4.h>*'. Since *MD4* is not recommended, but it is so that some or compatible with existing applications and protocols.

```
struct md4_ctx [Context struct]
```

```
MD4_DIGEST_SIZE [Constant]
    size of the digest, ... 2.
```

`MD4_BLOCK_SIZE` [Constant]
 Internal block size of MD4.

`void md4_init (struct md4_ctx *ctx)` [Function]
 Initialize MD4 state.

`void md4_update (struct md4_ctx *ctx, size_t length, const uint8_t *data)` [Function]
 Hash some more data.

`void md4_digest (struct md4_ctx *ctx, size_t length, uint8_t *digest)` [Function]

For some processing, extracts the message, writes it to *digest*. *length* may be as large as `MD4_DIGEST_SIZE`, new cases only the first *length* octets of the message are written.

is implemented as follows: call `md4_init`.

7.1.3.4 RIPEMD160

`RIPEMD160` is a standard function designed by Hans Dobbertin, Antoon Bossaers, and Bart Preneel, as a stronger version of `RIPEMD` (which, like `MD5`, has a collision resistance requirement). It produces message digests of 160 bits, or 20 octets. It is in `RIPEMD160` in ‘`nettle/ripemd160.h`’.

`struct ripemd160_ctx` [Context struct]

`RIPEMD160_DIGEST_SIZE` [Constant]
 Size of a `RIPEMD160` digest, in bytes.

`RIPEMD160_BLOCK_SIZE` [Constant]
 Internal block size of `RIPEMD160`.

`void ripemd160_init (struct ripemd160_ctx *ctx)` [Function]
 Initialize `RIPEMD160` state.

`void ripemd160_update (struct ripemd160_ctx *ctx, size_t length, const uint8_t *data)` [Function]
 Hash some more data.

`void ripemd160_digest (struct ripemd160_ctx *ctx, size_t length, uint8_t *digest)` [Function]

For some processing, extracts the message, writes it to *digest*. *length* may be as large as `RIPEMD160_DIGEST_SIZE`, new cases only the first *length* octets of the message are written.

is implemented as follows: call `ripemd160_init`.

7.1.3.5 SHA1

`SHA1` is a standard function specified by NIST (National Institute of Standards and Technology). It outputs a value of 160 bits, or 20 octets. It is in `SHA1` in ‘`nettle/sha1.h`’ (and in ‘`nettle/sha.h`’, or backwards compatibility).

```

struct sha1_ctx [Cont xt struct]
SHA1_DIGEST_SIZE [Constant]
    s o a SHA1 digest, . . . .

SHA1_BLOCK_SIZE [Constant]
    ntrnal block s o SHA1. s u or so sp ca construct ons, n part cu ar
    H AC-SHA1.

void sha1_init (struct sha1_ctx *ctx) [Funct on]
    Int a t SHA1 stat .

void sha1_update (struct sha1_ctx *ctx, size_t length, const uint8_t [Funct on]
    *data)
    Has so or ata.

void sha1_digest (struct sha1_ctx *ctx, size_t length, uint8_t [Funct on]
    *digest)
    P r or s na proc ss ng an xtracts t ssag g st, wr t ng t to digest. length
    ay b s a r t an SHA1_DIGEST_SIZE, n w c cas on y t rst length oct ts o
    t g st ar wr tt n.
    s unct on a so r s ts t cont xt nt sa way as sha1_init.
    
```

7.1.3.6 GOSTHASH94 and GOSTHASH94CP

GOST94 or GOST R 34.31-94 as a gort s a ov t-ra a gort us n P us-
 san gov rn nt stan ar s (s RFC 4357). It outputs ssag g sts o 56 b ts, or 3
 oct ts. stan ar ts o s not xt box us by t as a gort, so t r ar two
 popu ar var ants (t t st ng box ro t stan ar ts an t box n by Cryp-
 to Pro co pany, s RFC 35). tt prov s support or t or r box nt or o
 GOST-HASH94 as a gort an ort att r nt or o GOST-HASH94 CP as a go-
 r t. tt n s GOST-HASH94 an GOST-HASH94 CP n '<nettle/gosthash94.h>'.

```

struct gosthash94_ctx [Cont xt struct]
GOSTHASH94_DIGEST_SIZE [Constant]
    s o a GOST-HASH94 digest, . . . 3 .

GOSTHASH94_BLOCK_SIZE [Constant]
    ntrnal block s o GOST-HASH94, . . ., 3 .

void gosthash94_init (struct gosthash94_ctx *ctx) [Funct on]
    Int a t GOST-HASH94 stat .

void gosthash94_update (struct gosthash94_ctx *ctx, size_t length, [Funct on]
    const uint8_t *data)
    Has so or ata.

void gosthash94_digest (struct gosthash94_ctx *ctx, size_t length, [Funct on]
    uint8_t *digest)
    P r or s na proc ss ng an xtracts t ssag g st, wr t ng t to digest. length
    ay b s a r t an GOSTHASH94_DIGEST_SIZE, n w c cas on y t rst length
    oct ts o t g st ar wr tt n.
    
```

```

    struct on a so r s t s t   cont xt nt   sa   way as gosthash94_init.

struct gosthash94cp_ctx                                     [Cont xt struct]
GOSTHASH94CP_DIGEST_SIZE                                  [Constant]
    s   o a GOSTHASH94CP_DIGEST_SIZE, . . 3 .

GOSTHASH94CP_BLOCK_SIZE                                    [Constant]
    nt rna b oc s   o GOSTHASH94CP_BLOCK_SIZE, . . , 3 .

void gosthash94cp_init (struct gosthash94cp_ctx *ctx)      [Funct on]
    In t a   t   GOSTHASH94CP_DIGEST_SIZE .

void gosthash94cp_update (struct gosthash94cp_ctx *ctx, size_t [Funct on]
    length, const uint8_t *data)
    Has so   or   ata.

void gosthash94cp_digest (struct gosthash94cp_ctx *ctx, size_t [Funct on]
    length, uint8_t *digest)
    P r o c   s   na p r o c   s   ng an   x t r a c t s   t   ssag   g   st, wr t ng   t to digest. length
    ay b   s   a   r t an GOSTHASH94CP_DIGEST_SIZE, n w   c   cas   on y t   rst length
    oct ts o t   g   st ar   wr tt n.
    struct on a so r s t s t   cont xt nt   sa   way as gosthash94cp_init.
    
```

7.1.4 The struct nettle_hash abstraction

tt nc u s a struct nc u ng n or at on about t support as unct ons. It s
 n n 'nettle/nettle-meta.h', an s us by tt 's p ntat on o H AC
 (s ct on 5 [K y as unct ons], pag 6).

```

struct nettle_hash name context_size digest_size block_size init [   ta struct]
    update digest
    ast   t   r   attr   but   s   ar   unct   on   po nt   rs, o   typ   s nettle_hash_init_func *,
    nettle_hash_update_func *, an   nettle_hash_digest_func *.   rst   argu   nt
    to   t   s   unct   ons   s   void *po nt   r to a cont xt struct, w   c   s   o   s   context_size.

struct nettle_hash nettle_md2                               [Constant struct]
struct nettle_hash nettle_md4                               [Constant struct]
struct nettle_hash nettle_md5                               [Constant struct]
struct nettle_hash nettle_ripemd160                         [Constant struct]
struct nettle_hash nettle_sha1                              [Constant struct]
struct nettle_hash nettle_sha224                            [Constant struct]
struct nettle_hash nettle_sha256                            [Constant struct]
struct nettle_hash nettle_sha384                            [Constant struct]
struct nettle_hash nettle_sha512                            [Constant struct]
struct nettle_hash nettle_sha3_256                          [Constant struct]
struct nettle_hash nettle_gosthash94                        [Constant struct]
struct nettle_hash nettle_gosthash94cp                      [Constant struct]
    s   ar   a   t   as   unct   ons   t   at   tt   p   nt   s.
    
```

tt a so x ports a st o a t s as s.

`const struct nettle_hash ** nettle_get_hashes (void)` [Function]
 This turns a `libnettle` native structure to support as functions. It can be used to dynamically rate or support a given set.

`nettle_hashes` [acro]
 An acronym for a collection of hashes, so that on the whole, e.g., `nettle_hashes[0]->name` or the name of the first hash function on the list. In the previous versions, this was not an acronym but the actual array of pointers. However, regarding the type of the array as a set of pointers to the ABI in some cases.

7.2 Cipher functions

A *cipher* is a function that takes a *plaintext* and a *secret key* and transforms it to a *ciphertext*. Given only the ciphertext, but not the key, it should be hard to find the plaintext. Given the ciphertext and the key, it should be easy to find the plaintext.

There are two main classes of ciphers: *Block ciphers* and *stream ciphers*. A block cipher can process data on a byte-by-byte basis, called *blocks*. Typically, a block size is 128 or 192 bits. To encrypt arbitrary messages, you usually have to partition the message into blocks, split them into bytes, and then process each block. The simplest way is to process on a block at a time, in parallel to each other. The most common operation is called *ECB*, *Electronic Code Book*. However, using ECB is usually a bad idea. For a start, plaintext blocks that are equal transform to ciphertext blocks that are equal, that is, information about the plaintext is leaked. Usually, you should apply the cipher to the message in a “*block*” mode, *CBC* (*Cipher Block Chaining*) and *CTR* (*Counter*) being two of the most popular. See [Ciphers], page 3, or the introduction on how to apply CBC and CTR with the library.

A stream cipher can be used for messages of arbitrary length. A typical stream cipher is a *pseudo-random generator*. To encrypt a plaintext message into ciphertext, you use the generator, generate ciphertext as a pseudo-random stream, and XOR it with the plaintext. To decrypt, generate the same stream using the key, XOR it to the ciphertext, and the plaintext is recovered.

Caution: The first rule is never to use a cipher as a *one-time pad*. It is never secure.

A common misconception is that encryption, by itself, is sufficient. Say that you have a message and a secret key, and you receive an encrypted message. You apply the key, and get a plaintext message that is sent to you. Can you then be sure that the message was your friend that wrote the message you’re reading? The answer is no. For example, you’re using a block cipher in ECB mode, an attacker can intercept the message in the middle, and for each block, they can replace it with any other block. Even if the attacker can’t decrypt the message, they can change it so that you are not reading the same message as your friend wrote. If you are using a block cipher in CBC mode or an ECB, or are using a stream cipher, it is possible for the attacker to intercept the message, but the attacker can still change the message to the attacker’s liking.

It is recommended to *always* use an authentication code in addition to encryption. The popular choices are *Authenticated Code* (HMAC) (see [Key Functions], page 6), or *authenticated encryption* (AEAD).

To compare, we can say that the “way” of the algorithm is not a structural attribute of the algorithm, but a property of the algorithm. In other words, the algorithm is not a structural attribute of the algorithm, but a property of the algorithm. For example, the algorithm is not a structural attribute of the algorithm, but a property of the algorithm. It is possible, however, to say that the algorithm is a structural attribute of the algorithm.

To encrypt a message, you first need a cipher context or encryption key. You then use the context to process the plaintext or ciphertext message.

The context is known as *key setup*. It is the source of the context structure on the encryption, verification, or decryption of the ciphertext. It can be used for both encryption and decryption.

7.2.1 AES

AES is a block cipher, specified by NIST as a replacement for DES. It is a standard for a computer to be used in a secure manner. It is known as FIPS 197, was constructed by Joan Daemen and Vincent Rijmen.

The AES can be used with a key size of 128, 192, or 256 bits, or 128, 192, or 256 bytes. It is not available in any way. The AES is defined in the header file `<openssl/aes.h>`, and the structure for the context is `struct aes_ctx`. (Earlier versions of the library used the structure `struct aes_ctx`, or a similar structure.)

```

struct aes128_ctx [Context struct]
struct aes192_ctx [Context struct]
struct aes256_ctx [Context struct]
struct aes_ctx [Context struct]
    
```

A thread-safe structure, or the AES is not a race condition.

```

AES_BLOCK_SIZE [Constant]
    AES block size, 16.
    
```

```

AES128_KEY_SIZE [Constant]
AES192_KEY_SIZE [Constant]
AES256_KEY_SIZE [Constant]
AES_MIN_KEY_SIZE [Constant]
AES_MAX_KEY_SIZE [Constant]
AES_KEY_SIZE [Constant]
    Default AES key size, 128.
    
```

```

void aes128_set_encrypt_key (struct aes128_ctx *ctx, const uint8_t *key) [Function]
void aes128_set_decrypt_key (struct aes128_ctx *ctx, const uint8_t *key) [Function]
void aes192_set_encrypt_key (struct aes192_ctx *ctx, const uint8_t *key) [Function]
void aes192_set_decrypt_key (struct aes192_ctx *ctx, const uint8_t *key) [Function]
    
```

```
void aes256_set_encrypt_key (struct aes256_ctx *ctx, const uint8_t *key) [Funct on]
```

```
void aes256_set_decrypt_key (struct aes256_ctx *ctx, const uint8_t *key) [Funct on]
```

```
void aes_set_encrypt_key (struct aes_ctx *ctx, size_t length, const uint8_t *key) [Funct on]
```

```
void aes_set_decrypt_key (struct aes_ctx *ctx, size_t length, const uint8_t *key) [Funct on]
```

In t a t c p r, o r n c r y p t o n o r c r y p t o n, r s p c t v y.

```
void aes128_invert_key (struct aes128_ctx *dst, const struct aes128_ctx *src) [Funct on]
```

```
void aes192_invert_key (struct aes192_ctx *dst, const struct aes192_ctx *src) [Funct on]
```

```
void aes256_invert_key (struct aes256_ctx *dst, const struct aes256_ctx *src) [Funct on]
```

```
void aes_invert_key (struct aes_ctx *dst, const struct aes_ctx *src) [Funct on]
```

G v n a c o n t x t s r c n t a o r n c r y p t o n, n t a s t c o n t x t s t r u c t d s t o r c r y p t o n, u s n g t s a y. I t s a c o n t x t s t r u c t s p a s s o r b o t s r c a n d s t, t s c o n v r t n p a c. s u n c t o n s a r a n y u s u o r a p p c a t o n s w c n s t o b o t n c r y p t a n c r y p t u s n g t s a m e y, b c a u s c a n g, .g., aes128_set_encrypt_key a n aes128_invert_key, s o r e n t t a n c a n g aes128_set_encrypt_key a n aes128_set_decrypt_key.

```
void aes128_encrypt (struct aes128_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

```
void aes192_encrypt (struct aes192_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

```
void aes256_encrypt (struct aes256_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

```
void aes_encrypt (struct aes_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

E n c r y p t o n u n c t o n. l e n g t h u s t b a n n t g r a u t p o t b o c s. I t s o r t a n o n b o c, t a t a s p r o c s s n E C B o. s r c a n d s t a y b q u a, b u t t y u s t n o t o v r a p n a n y o t r w a y.

```
void aes128_decrypt (struct aes128_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

```
void aes192_decrypt (struct aes192_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

```
void aes256_decrypt (struct aes256_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

```
void aes_decrypt (struct aes_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src) [Funct on]
```

A n a l o g o u s t o t n c r y p t o n u n c t o n s a b o v.

7.2.2 ARCFOUR

ARCFOUR is a stream cipher, also known under the trademark name RC4, and it is one of the fastest ciphers around. A problem with the setup of ARCFOUR is that, quite often, you should never use the same structure, the same arbitrary passwords, or sequences of bytes like "secret", "secret", If you have the same password or arbitrary strings, and you want to use ARCFOUR, always use the same password or arbitrary strings. Furthermore, the same byte sequence is generated by the same password or arbitrary strings, or the same reason, the same sequence is generated by the same password or arbitrary strings.

```
/* A more robust key setup function for ARCFOUR */
void
arcfour_set_key_hashed(struct arcfour_ctx *ctx,
                      size_t length, const uint8_t *key)
{
    struct sha256_ctx hash;
    uint8_t digest[SHA256_DIGEST_SIZE];
    uint8_t buffer[0x200];

    sha256_init(&hash);
    sha256_update(&hash, length, key);
    sha256_digest(&hash, SHA256_DIGEST_SIZE, digest);

    arcfour_set_key(ctx, SHA256_DIGEST_SIZE, digest);
    arcfour_crypt(ctx, sizeof(buffer), buffer, buffer);
}

#ifdef HAVE_ARCFOUR
#include <nettle/arcfour.h>

struct arcfour_ctx [Context struct]
ARCFOUR_MIN_KEY_SIZE [Constant]
    minimum key size, 16.

ARCFOUR_MAX_KEY_SIZE [Constant]
    maximum key size, 256.

ARCFOUR_KEY_SIZE [Constant]
    default ARCFOUR key size, 16.

void arcfour_set_key (struct arcfour_ctx *ctx, size_t length, const [Function]
    uint8_t *key)
    Initialize cipher state using or both encryption and decryption.

void arcfour_crypt (struct arcfour_ctx *ctx, size_t length, uint8_t [Function]
    *dst, const uint8_t *src)
    Encrypt source data using or both encryption and decryption.
    The block ciphers function operates on context, so you can split data
    into arbitrary chunks and encrypt them in arbitrary order. The result
    is the same as you would get from arcfour_crypt on a single block of data.
```

7.2.3 ARCTWO

ARCTWO (also known as the Traub–Arne–Naor (TAN) subblock cipher) is a block cipher designed by Matt Aschmann, a variation on the ARCTIC subblock cipher operation that acts on state, to be compatible with the reverse engineering of the cipher script on, as described in a recent post to `sci.crypt` by Peter Gutmann.

ARCTWO uses a block size of 64 bits, an variable-sized ranging from 1 to 1024 octets. Besides state, ARCTWO also has a state parameter to the setup, the number of fixed bytes, `ekb`. This parameter can be used to calculate the output bytes. In practice, `ekb` is usually set equal to the input bytes. The TAN ARCTWO is named ‘<nettle/arctwo.h>’.

It is not recommended to use ARCTWO, though, as the implementation is provably or not robust with existing applications and standards.

`struct arctwo_ctx` [Content struct]

`ARCTWO_BLOCK_SIZE` [Constant]

ARCTWO block size, in bytes.

`ARCTWO_MIN_KEY_SIZE` [Constant]

`ARCTWO_MAX_KEY_SIZE` [Constant]

`ARCTWO_KEY_SIZE` [Constant]

Default ARCTWO key size, in bytes.

`void arctwo_set_key_ekb (struct arctwo_ctx *ctx, size_t length, const uint8_t *key, unsigned ekb)` [Function]

`void arctwo_set_key (struct arctwo_ctx *ctx, size_t length, const uint8_t *key)` [Function]

`void arctwo_set_key_gutmann (struct arctwo_ctx *ctx, size_t length, const uint8_t *key)` [Function]

In the TAN cipher, the same function is used for both encryption and decryption. The first function sets the state and range, while the second provides both the variable-sized state and the fixed bytes (in bytes). The maximum value for `ekb` is 1024, and for conversion, `ekb = 0` as the same fixed bytes as `ekb = 1024`.

`arctwo_set_key(ctx, length, key)` is equivalent to `arctwo_set_key_ekb(ctx, length, key, 8*length)`, and `arctwo_set_key_gutmann(ctx, length, key)` is equivalent to `arctwo_set_key_ekb(ctx, length, key, 1024)`.

`void arctwo_encrypt (struct arctwo_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src)` [Function]

Encryption function. `length` is the number of bytes to be encrypted. It is important to note that the data is processed in ECB mode. `src` and `dst` may be the same, but they must not overlap in any other way.

`void arctwo_decrypt (struct arctwo_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src)` [Function]

Analogous to `arctwo_encrypt`.

7.2.4 BLOWFISH

BL01F1FH s a b o c c p r s g n by Bruc c n r. It us s a b o c s o b t s
(o c t s), an a var ab y s , up to b t s. It as so w a y s. tt n s
BL01F1FH n '<nettle/blowfish.h>'.

```
struct blowfish_ctx
```

[Cont xt struct]

BLOWFISH_BLOCK_SIZE	[Constant]
---------------------	------------

BL 041717H b o c -s , .

BLOWFISH_MIN_KEY_SIZE	[Constant]
-----------------------	------------

in the BLFTH system, .

BLOWFISH_MAX_KEY_SIZE	[Constant]
-----------------------	------------

ax u BL 0 4 FTH y s , 56.

BLOWFISH_KEY_SIZE	[Constant]
-------------------	------------

Daut BL, O'Flaherty J. *Phys*, 6.

```
int blowfish_set_key (struct blowfish_ctx *ctx, size_t length, const [Funct on]
                    uint8_t *key)
```

In a text processing, a function is either a built-in function or a user-defined function. Built-in functions are those functions that are already defined in the Python interpreter. User-defined functions are those functions that are defined by the user. Applications that don't care about what a function can do are called user-defined functions.

`blowfish_encrypt` or `blowfish_decrypt` w t a w a y w cras w t an ass rt
v o at on.

```
void blowfish_encrypt (struct blowfish_ctx *ctx, size_t length, [Funct on]
                      uint8_t *dst, const uint8_t *src)
```

Encrypt on function, *length* must be an integer greater than 0. Its
 or function block, its data process in ECB mode. *src* and *dst* may be
 but they must not overlap in any other way.

```
void blowfish_decrypt (struct blowfish_ctx *ctx, size_t length, [Funct on]
                      uint8_t *dst, const uint8_t *src)
```

Analogous to blowfish_encrypt

```
int blowfish_bcrypt_hash (char *dst, size_t lenkey, const char [Funct on]
                          *key, size_t lenscheme, const char *scheme, int log2rounds, const uint8_t
                          *salt)
```

Co put t bcrpt passwor as . unct on w r turn 0 t as cannot b
co put u to nva nput. unct on w r turn 1 an stor t co put
as n t array pont to by *dst*. as s co put bas on t c os n
scheme, nu b r o roun s *log2rounds* an sp c *salt*.

dst must point to a character array of at least BLOWFISH_BCRYPT_HASH_SIZE bytes.

$$\text{key_contains_parent_password_string_only_lenkey.}$$

schema so *schema* and contains trust constraints (variables are 2a, 2b, 2x or 2y), or (to prove) an existing password (typically \$2b\$10\$...).

log2rounds contains the logarithm of the number of encryption rounds that must be used to compute the result. It is a constant value which is extracted from the *scheme*.

salt is a pointer to an array of `BLOWFISH_BCRYPT_BINSALT_SIZE` random bytes to be used to perturb the computation. It is a `NULL` if the salt was extracted from the *scheme*.

Example code to generate a bcrypt hash:

```
char cleartxtpassword[] = "ExamplePassword";
char scheme[] = "2b";
uint8_t salt[BLOWFISH_BCRYPT_BINSALT_SIZE];
...
/* Make sure that salt is filled with random bytes */
...
char hashedresult[BLOWFISH_BCRYPT_HASH_SIZE];
int result = blowfish_bcrypt(hashedresult,
                             sizeof(cleartxtpassword) - 1, cleartxtpassword,
                             sizeof(scheme) - 1, scheme, 10, salt);

if (result)
    printf("%s\n", hashedresult);
```

`int blowfish_bcrypt_verify (size_t lenkey, const char *key, size_t lenhashed, const char *hashed)` [Function]

Verify a bcrypt password against the supplied plaintext password. Returns 0 if the password is not a match, and 1 if the password matches.

key contains the plaintext password string of size *lenkey*.

hashed contains the hashed string of size *lenhashed* to compare with.

Example code to verify a bcrypt hash:

```
char cleartxtpassword[] = "ExamplePassword";
char existinghashed[] =
    "$2y$" /* Hash algorithm version */
    "10" /* 2^10 hash rounds (strength) */
    "$" /* separator */
    "1b2lPgo4XumibnJGN3r3s0" /* base64 encoded 16-byte salt */
    "u7wE7xNfYDKlAxZffJDCJdVfFTAyevu"; /* Hashed part */
if (blowfish_bcrypt_verify(sizeof(cleartxtpassword) - 1, cleartxtpassword,
                           sizeof(existinghashed) - 1, existinghashed))
```

r as by un rt G LGPL (v . or at r), an r s on t p ct pat nt
 c ns o t LGPL. r s as o a stat nt o roya ty- r c ns ng or Ca a at
<http://www.ntt.co.jp/news/news01e/0104/010417.html>, but t s stat nt as so
 tat ons w c s prob at c or r so twar .

Ca a u s s a t s a b o c s a n y s s a s A E b o c s s b t s
(o c t s), a n t s u p p o r t y s s a r , 9 , a n b t s. v a r a n t s w t
9 a n b t y s a r n t c a , x c p t o r t y s t u p. t t n s C a a n
'[nettle/camellia.h](#)>', a n t r s o n c o n t x t s t r u c t o r a c y s . (E a r r v r s o n s
o t t u s a s n g c o n t x t s t r u c t, `struct camellia_ctx`, o r a y s s. s n t r a c
p t o r b a c w a r s c o p a t b t y).

```
struct camellia128_ctx [Cont xt struct]
struct camellia192_ctx [Cont xt struct]
struct camellia256_ctx [Cont xt struct]
Cont xts structs. Actual y, camellia192_ctx s an a as or camellia256_ctx.
```

struct camellia_ctx [Cont xt struct]
 At runtime, structure is allocated on the stack.

CAMELLIA_BLOCK_SIZE [Constant]

CAMELLIA128_KEY_SIZE	[Constant]
CAMELLIA192_KEY_SIZE	[Constant]
CAMELLIA256_KEY_SIZE	[Constant]
CAMELLIA_MIN_KEY_SIZE	[Constant]
CAMELLIA_MAX_KEY_SIZE	[Constant]
CAMELLIA_KEY_SIZE	[Constant]

<code>void camellia128_set_encrypt_key (struct camellia128_ctx *ctx, const uint8_t *key)</code>	[Funct on]
<code>void camellia128_set_decrypt_key (struct camellia128_ctx *ctx, const uint8_t *key)</code>	[Funct on]
<code>void camellia192_set_encrypt_key (struct camellia192_ctx *ctx, const uint8_t *key)</code>	[Funct on]
<code>void camellia192_set_decrypt_key (struct camellia192_ctx *ctx, const uint8_t *key)</code>	[Funct on]
<code>void camellia256_set_encrypt_key (struct camellia256_ctx *ctx, const uint8_t *key)</code>	[Funct on]
<code>void camellia256_set_decrypt_key (struct camellia256_ctx *ctx, const uint8_t *key)</code>	[Funct on]
<code>void camellia_set_encrypt_key (struct camellia_ctx *ctx, size_t length, const uint8_t *key)</code>	[Funct on]
<code>void camellia_set_decrypt_key (struct camellia_ctx *ctx, size_t length, const uint8_t *key)</code>	[Funct on]

In the context of encryption or decryption, respectively.

```
void camellia128_invert_key (struct camellia128_ctx *dst, const struct camellia128_ctx *src) [Funct on]
void camellia192_invert_key (struct camellia192_ctx *dst, const struct camellia192_ctx *src) [Funct on]
void camellia256_invert_key (struct camellia256_ctx *dst, const struct camellia256_ctx *src) [Funct on]
void camellia_invert_key (struct camellia_ctx *dst, const struct camellia_ctx *src) [Funct on]
```

Given a context src and a function $encrypt$ on src to construct dst or $encrypt$ on src using sa . It sa constructs $pass$ or both src and dst , then convert $pass$ to sa . The functions are any use or applications with sa to both $encrypt$ and $decrypt$ using sa .

```
void camellia128_crypt (struct camellia128_ctx *ctx, size_t length, [Funct on]
                        uint8_t *dst, const uint8_t *src)
void camellia192_crypt (struct camellia192_ctx *ctx, size_t length, [Funct on]
                        uint8_t *dst, const uint8_t *src)
void camellia256_crypt (struct camellia256_ctx *ctx, size_t length, [Funct on]
                        uint8_t *dst, const uint8_t *src)
void camellia_crypt (struct camellia_ctx *ctx, size_t length, uint8_t [Funct on]
                    *dst, const uint8_t *src)
```

sa unct on s us or bot ncrypt on an crypt on. *length* ust b an
 nt gra ut p o t boc s . I t s or t an on boc , ata s proc ss
 n ECB o . **src** an **dst** ay b qua , but t y ust not ov rap n any ot r
 way.

7.2.6 CAST128

CA[™] - s a b o c c p r, s p c n *RFC 2144*. It u s s a c b t (o c t s) b o c s , a n
a y s o b t s. It s p o s s b , b t s c o u r a g , t o u s t s a a g o r t w t s o r t r
y s. t t r r s t o t v a r a n t w t v a r a b y s a s C A[™] - 5. K y s o r C A[™] - 5 a r
r o p a t o b t s, a n w t v r y s o r t y s, s s t a n b t s, n c r y p t o n a s o u s s
w r r o u n s t a n C A[™] , t t n s c a s t n ' < n e t t l e / c a s t 1 2 8 . h > '.

```
struct cast128_ctx [Cont xt struct]
```

CAST128_BLOCK_SIZE	[Constant]
CA	block size, .

CAST128_KEY_SIZE	[Constant]
CAST128_KEY_SIZE	16

CAST5_MIN_KEY_SIZE	[Constant]
n u C A 5 y s , 5 .	

CAST5_MAX_KEY_SIZE [Constant]
 Maximum key size in bytes, 16 octets (128 bits), CA⁵-S standard as CA⁵-S.

`void cast128_set_key (struct cast128_ctx *ctx, const uint8_t *key)` [Function]
 Int a t c p r. sa unct on s us or bot ncrypt on an crypt on.

`void cast128_encrypt (struct cast128_ctx *ctx, size_t length, uint8_t [Funct on]
*dst, const uint8_t *src)`

Encrypt on unct on. *length* ust b an nt gra utp o t boc s . I t s
or t an on boc , t ata s proc ss n ECB o . *src* an *dst* ay b qua ,
but t y ust not ov rap n any ot r way.

`void cast128_decrypt (struct cast128_ctx *ctx, size_t length, uint8_t [Funct on]
*dst, const uint8_t *src)`

Ana ogous to `cast128_encrypt`

`void cast5_set_key (struct cast128_ctx *ctx, size_t length, const [Funct on]
uint8_t *key)`

In t a t c p r. s var ant o t y s tup ta s t y s as argu nt.
sa unct on s us or bot ncrypt on an crypt on.

7.2.7 ChaCha

C aC a s a var ant o t str a c p r a sa , a so s gn

7.2.7.1 32-bit counter variant

This is the original 32-bit counter variant defined in *RFC 8439* using 32-bit counter values. This variant is particularly useful for stream ciphers. [Cacaya-Poy 35], page 5 AEAD construction, which supports octet nonces.

`CHACHA_NONCE96_SIZE` [Constant]
Octet nonce, 96.

`CHACHA_COUNTER32_SIZE` [Constant]
Octet counter, 32.

`void chacha_set_nonce96 (struct chacha_ctx *ctx, const uint8_t *nonce)` [Function]
Sets the nonce. This is similar to the above `chacha_set_nonce`, but the input is always of size `CHACHA_NONCE96_SIZE`, octets.

`void chacha_set_counter32 (struct chacha_ctx *ctx, const uint8_t *counter)` [Function]
Sets the block counter. This is similar to the above `chacha_set_counter`, but the input is always of size `CHACHA_COUNTER32_SIZE`, octets.

`void chacha_crypt32 (struct chacha_ctx *ctx, size_t length, uint8_t *dst, const uint8_t *src)` [Function]
Encrypts or decrypts data of a size, using ChaCha. This is similar to the above `chacha_crypt`, but the assumed initial counter value is 32-bit ongoing nonce stream.

7.2.8 DES

DES is the Data Encryption Standard, specified by *FIPS 46-3*. It uses a block size of 64 bits (octets), and a key size of 56 bits. However, the key bits are distributed over 16 rounds, with the last significant bit of each octet always being used for parity. A common way to use DES is to generate random octets in some way, then set the last significant bit of each octet to get the parity, and then use DES with the resulting key.

Since DES is so slow, it can be broken by brute force, using special hardware or lots of ordinary workstations in parallel. One should not be using plain DES at all today, one uses DES at all one should be using “trip DES”, or DES³ below.

DES also has some weaknesses. The main one is DES is in ‘<nettle/des.h>’.

`struct des_ctx` [Context struct]
`DES_BLOCK_SIZE` [Constant]
DES block size, 64.

`DES_KEY_SIZE` [Constant]
DES key size, 56.

`int des_set_key (struct des_ctx *ctx, const uint8_t *key)` [Function]
Initializes the context for encryption or decryption. Parity bits are ignored. Ciphers or weaknesses, returning error codes or weaknesses. Applications that don’t care about weaknesses can ignore the return value.

```
void des_encrypt (struct des_ctx *ctx, size_t length, uint8_t *dst, [Function]
                  const uint8_t *src)
    Encrypt on unct on. length ust b an nt gra utp o t boc s . I t s
    or t an on boc , t ata s proc ss n ECB o . src an dst ay b qua ,
    but t y ust not ov r ap n any ot r w . 65t x
```

`DES3_KEY_SIZE` [Constant]
 DEFSYSS, .

`int des3_set_key (struct des3_ctx *ctx, const uint8_t *key)` [Function]
 Intake the cipher's state and set up for encryption or decryption.
 Parity bits are ignored. Ciphers or ways, returning a true value if good
 yes, and an error or error value. Applications that don't care about ways
 can ignore the return value.

For randomizing strings, you can use `des_fix_parity` to adjust parity bits before
 calling `des3_set_key`.

`void des3_encrypt (struct des3_ctx *ctx, size_t length, uint8_t *dst, [Function]
 const uint8_t *src)`
 Encrypt on function. `length` is the number of bytes to process. It's
 for the entire block, data is processed in ECB mode. `src` and `dst` may be the same,
 but they must not overlap in any other way.

`void des3_decrypt (struct des3_ctx *ctx, size_t length, uint8_t *dst, [Function]
 const uint8_t *src)`
 Analogous to `des3_encrypt`

7.2.10 Salsa20

Salsa is a very recent stream cipher designed by D. J. Bernstein. It's built on the
 observation that a cryptographic function can be used for encryption. For the
 input, rotate the secret key and a counter, xor the result with the output and the
 previous state, then increment the counter to process the next block (see [Chapter 7](#),
 section 3.3. [C1], page 3). Bernstein's implementation of encryption is a good
 example of the way to structure the code to process the next block as a function of the
 previous state, but the code is a bit more complex.

Salsa uses a 256-bit key, but with a new space as a function to XOR, block
 counter, and a couple of constants. It's a so-called "sponge", on x1, the current
 state is a 256-bit key and a 256-bit counter. It uses a block of 512 bits (4 octets) and the
 two spaces are 256 bits, and 512 bits (4 octets).

Caution: as a function, use the Salsa *not* directly as a function or use as a
 function. It's *not* constant on random inputs and a few, and it's not a
 function and output is a function.

Using Salsa to process a message, on space, both a key and a nonce, the
 attribute of a state is a random value (I) used with CBC or CTR. On
 can use the state for a variable message, providing a unique random value
 for the message. The state is 512 bits (4 octets). The block counter is a
 message, and a 512-bit state. The state is Salsa in '<nettle/salsa20.h>'.
 The state is Salsa in '<nettle/salsa20.h>'.

`struct salsa20_ctx` [Context struct]
`SALSA20_128_KEY_SIZE` [Constant]
`SALSA20_256_KEY_SIZE` [Constant]
 two support keys, 512 and 3 octets.

SALSA20_KEY_SIZE [Constant]
 P co n y s , 3 .

SALSA20_BLOCK_SIZE [Constant]
 a sa b oc s , 6 .

SALSA20_NONCE_SIZE [Constant]
 o t nonc , .

void salsa20_128_set_key (struct salsa20_ctx *ctx, const uint8 [Funct on]

```

struct serpent_ctx [Cont xt struct]
SERPENT_BLOCK_SIZE [Constant]
    #define SERPENT_BLOCK_SIZE 64

SERPENT_MIN_KEY_SIZE [Constant]
    #define SERPENT_MIN_KEY_SIZE 16

SERPENT_MAX_KEY_SIZE [Constant]
    #define SERPENT_MAX_KEY_SIZE 32

SERPENT_KEY_SIZE [Constant]
    #define SERPENT_KEY_SIZE 32

void serpent_set_key (struct serpent_ctx *ctx, size_t length, const
    uint8_t *key) [Funct on]
    Initialize serpent_ctx with key. Can be used for both encryption and decryption.

void serpent_encrypt (struct serpent_ctx *ctx, size_t length, uint8_t
    *dst, const uint8_t *src) [Funct on]
    Encrypt on ctx. length must be an integer multiple of 64. It's
    for in-place encryption. src and dst may overlap, but they must not overlap in any other way.

void serpent_decrypt (struct serpent_ctx *ctx, size_t length, uint8_t
    *dst, const uint8_t *src) [Funct on]
    Analogous to serpent_encrypt

```

7.2.12 TWOFISH

Another AEAD algorithm designed by Bruce Schneier and others. It's in `<nettle/twofish.h>`.

```

struct twofish_ctx [Cont xt struct]
TWOFISH_BLOCK_SIZE [Constant]
    #define TWOFISH_BLOCK_SIZE 128

TWOFISH_MIN_KEY_SIZE [Constant]
    #define TWOFISH_MIN_KEY_SIZE 16

TWOFISH_MAX_KEY_SIZE [Constant]
    #define TWOFISH_MAX_KEY_SIZE 32

TWOFISH_KEY_SIZE [Constant]
    #define TWOFISH_KEY_SIZE 32

void twofish_set_key (struct twofish_ctx *ctx, size_t length, const
    uint8_t *key) [Funct on]
    Initialize twofish_ctx with key. Can be used for both encryption and decryption.

```

```
void twofish_encrypt (struct twofish_ctx *ctx, size_t length, uint8_t [Func on]
                    *dst, const uint8_t *src)
```

Encrypt on unct on. *length* ust b an nt gra utp o t boc s . I t s
or t an on boc , t ata s proc ss n ECB o . *src* an *dst* ay b qua ,
but t y ust not ov r ap n any ot r way.

```
void twofish_decrypt (struct twofish_ctx *ctx, size_t length, uint8_t [Func on]
                    *dst, const uint8_t *src)
```

Ana ogous to `twofish_encrypt`

7.2.13 The struct `nettle_cipher` abstraction

tt nc u s a struct nc u ng n or at on about so o t or r guar c p r unc-
t ons. It can b us u or app cat ons t at n a s p way to an var ous a gor t s.
tt n s t s structs n ‘<nettle/nettle-meta.h>’.

```
struct nettle_cipher name context_size block_size key_size [ ta struct]
    set_encrypt_key set_decrypt_key encrypt decrypt
    ast our attr but s ar unct on po nt rs, o typ s nettle_set_key_func * an
    nettle_cipher_func *. rst argu nt to t s unct ons s a const void *
    po nt r to a cont xt struct, w c s o s context_size.
```

```
struct nettle_cipher nettle_aes128 [Constant struct]
struct nettle_cipher nettle_aes192 [Constant struct]
struct nettle_cipher nettle_aes256 [Constant struct]
struct nettle_cipher nettle_arctwo40 [Constant struct]
struct nettle_cipher nettle_arctwo64 [Constant struct]
struct nettle_cipher nettle_arctwo128 [Constant struct]
struct nettle_cipher nettle_arctwo_gutmann128 [Constant struct]
struct nettle_cipher nettle_arcfour128 [Constant struct]
struct nettle_cipher nettle_camellia128 [Constant struct]
struct nettle_cipher nettle_camellia192 [Constant struct]
struct nettle_cipher nettle_camellia256 [Constant struct]
struct nettle_cipher nettle_cast128 [Constant struct]
struct nettle_cipher nettle_serpent128 [Constant struct]
struct nettle_cipher nettle_serpent192 [Constant struct]
struct nettle_cipher nettle_serpent256 [Constant struct]
struct nettle_cipher nettle_twofish128 [Constant struct]
struct nettle_cipher nettle_twofish192 [Constant struct]
struct nettle_cipher nettle_twofish256 [Constant struct]
```

tt nc u s suc structs or a t *regular* c p rs, . . on s w t out w a ys or
ot r o t s.

tt a so xports a st o a t s c p rs w t out w a ys or ot r o t s.

```
const struct nettle_cipher ** nettle_get_ciphers (void) [Func on]
```

turns a LL-t r nat st o po nt rs to support boc c p rs. s st can
b us to yna ca y nu rat or s arc t support a gor t s.

`nettle_ciphers` [acro]
 An acronym spanning to a caution telling the compiler. In array versions, this was not an acronym but the actual array of pointers.

7.3 Cipher modes

Compiler's operation specifies the procedure to use when encrypting a message at the argument's block size. As spanning the encryption. [Cipher functions], page 1, specifying the message into blocks and processing the plaintext with the block cipher (Electronic Code Book mode, ECB), as an operation.

Besides ECB, the provided standard operation Cipher Block Chaining (CBC), Counter mode (CTR), Cipher Feedback (CFB and CFB), XEX-based twofish codebook with ciphertext stealing (XECB) and a couple of AEAD modes (seal and [Authenticated encryption], page 1). CBC is widely used, but the rare alternative is an operation called, say, .g., the CBC vulnerability (<http://www.kb.cert.org/vuls/id/958563>). Okay, CTR is usually preferred over CBC.

Of course, CBC, CTR, CFB and CFB provide no message authentication, and so always be used together with a MAC (see [Key management], page 1) or signature to authenticate the message.

7.3.1 Cipher Block Chaining

When using CBC mode, plaintext blocks are not encrypted independently of each other, in Electronic Code Book mode. Instead, when encrypting a block in CBC mode, the previous ciphertext blocks are XORed with the plaintext blocks to the block cipher. When encrypting the first block, a random block called an IV, or Initialization vector, is used as the "previous ciphertext block". It is usually random, but it is not necessarily secret, and can be transmitted in the clear together with the encrypted data.

In symbols, E_k is the encryption function of a block cipher, an IV is the initialization vector, the n th plaintext blocks $M_1, \dots, M_n$ are transformed into ciphertext blocks $C_1, \dots, C_n$ as follows

$$C_1 = E_k(IV \oplus M_1)$$

$$C_2 = E_k(C_1 \oplus M_2)$$

...

$$C_n = E_k(C_{(n-1)} \oplus M_n)$$

It's nice to see two functions for applying a block cipher in Cipher Block Chaining (CBC) mode, one for encryption and one for decryption. These functions use `void *` to pass cipher contexts around.

`void cbc_encrypt (const void *ctx, nettle_cipher_func *f, size_t [Funct on]
 block_size, uint8_t *iv, size_t length, uint8_t *dst, const uint8_t *src)`
`void cbc_decrypt (const void *ctx, nettle_cipher_func *f, size_t [Funct on]
 block_size, uint8_t *iv, size_t length, uint8_t *dst, const uint8_t *src)`
 App s t n c r y p t o n o r c r y p t o n u n c t o n f n C B C o . n a c p r t x t
 b o c p r o c s s s c o p n t o i v b o r r t u r n n g , s o t a t a a r g s s a g c a n b
 p r o c s s b y a s q u n c o c a s t o c b c _ e n c r y p t . u n c t o n f s o t y p
`void f (void *ctx, size_t length, uint8_t dst, const uint8_t *src),`
 a n t c b c _ e n c r y p t a n c b c _ d e c r y p t u n c t o n s p a s s t r a r g u m t c t x o n t o f .
 r a r a s o s o a c r o s t o p u s t s u n c t o n s c o r r c t y .

`CBC_CTX (context_type, block_size)` [acro]
 Exp a n s t o

```
{
    context_type ctx;
    uint8_t iv[block_size];
}
```

It can b u s t o n a C B C c o n t x t s t r u c t , t r r c t y ,

```
struct CBC_CTX(struct aes_ctx, AES_BLOCK_SIZE) ctx;
```

o r t o g v t a s t r u c t t a g ,

```
struct aes_cbc_ctx CBC_CTX (struct aes_ctx, AES_BLOCK_SIZE);
```

`CBC_SET_IV (ctx, iv)` [acro]
 F r s t a r g u m t s a p o n t r t o a c o n t x t s t r u c t a s n b y C B C _ C T X , a n t s c o n
 s a p o n t r t o a n I n t a a t o n c t o r (I) t a t s c o p n t o t a t c o n t x t .

`CBC_ENCRYPT (ctx, f, length, dst, src)` [acro]
`CBC_DECRYPT (ctx, f, length, dst, src)` [acro]

A s p r w a y t o n v o c b c _ e n c r y p t a n c b c _ d e c r y p t . f r s t a r g u m t s a
 p o n t r t o a c o n t x t s t r u c t a s n b y C B C _ C T X , a n t s c o n a r g u m t s a n
 n c r y p t o n o r c r y p t o n u n c t o n o o w n g t t ' s c o n v n t o n s . a s t t r a r
 g u m t s n t s o u r c a n s t n a t o n a r a o r t o p r a t o n .

s a c r o s u s s o t r c s t o a t c o p r s p a y a w a r n n g t t y p s o f a n
 c t x o n ' t a t c , . g . y o u t r y t o u s a n s t r u c t a e s _ c t x c o n t x t w t t d e s _ e n c r y p t
 u n c t o n .

7.3.2 Counter mode

C o u n t e r m o d e (C P) u s s t b o c c p r a s a y p s u o - r a n o g n r a t o r .

o u t p u t o t g n r a t o r s X **P** w t t a t a t o b n c r y p t . I t c a n b u n r s t o o a s
 a w a y t o t r a n s o r a b o c c p r t o a s t r a c p r .

s s a g s v n t o n b o c s M _ 1 , . . . M _ n , w r M _ n s o s m w c a y b
 s a r t a n t b o c s . E x c p t o r t a s t b o c , a t s s a g b o c s u s t b o
 s q u a t o t c p r ' s b o c s .

I E _ k s t n c r y p t o n u n c t o n o a b o c c p r , I C s t n t a c o u n t e r , t n t n
 p a n t x t b o c s a r t r a n s o r n t o n c p r t x t b o c s C _ 1 , . . . C _ n a s o o w s

```
C_1 = E_k(IC) XOR M_1
C_2 = E_k(IC + 1) XOR M_2
```

```
...
```

```
C_(n-1) = E_k(IC + n - 2) XOR M_(n-1)
C_n = E_k(IC + n - 1) [1..m] XOR M_n
```

IC is the initial value or counter, typically as an array as in CBC. $\blacktriangle$ In a single, IC + x, IC is incremented as an integer, not a byte or word. For the last block, $E_k(IC + n - 1) [1..m]$ is the ciphertext output truncated to m bytes.

```
void ctr_crypt (const void *ctx, nettle_cipher_func *f, size_t [Funct on]
                block_size, uint8_t *ctr, size_t length, uint8_t *dst, const uint8_t *src)
App s t ncrypt on unct on f n C P o . ot t at or C P o , ncrypt on
an crypt on s t s a o p r a t o n , an n c f s o u a w a y s b t ncrypt on
unct on or t un r y n g b o c c p r .
```

$\blacktriangle$ In a single, ncrypt using a sequence of calls to `ctr_crypt`, a but t ast ca must use a single output block.

Like CBC, the array is a counter across.

```
CTR_CTX (context_type, block_size) [acro]
```

Expands to

```
{
    context_type ctx;
    uint8_t ctr[block_size];
}
```

```
CTR_SET_COUNTER (ctx, iv) [acro]
```

First argument is a pointer to a context structure as defined by `CTR_CTX`, and the second is a pointer to an integer counter that is copied into the context.

```
CTR_CRYPT (ctx, f, length, dst, src) [acro]
```

As a way to use `ctr_crypt`. First argument is a pointer to a context structure as defined by `CTR_CTX`, and the second argument is an encryption function of the type's conventions. The third argument is the source and destination for the operation.

7.3.3 Cipher Feedback mode

Cipher Feedback (CFB) and Cipher Feedback (CFB) block cipher modes are similar to both CBC and CTR, but borrow some characteristics from stream ciphers.

For CFB the single value is the block $M_1, \dots, M_n$, where M_n is the message block. Except for the last block, a single block is used to produce the next block's ciphertext.

In `E_k` is the encryption function of a block cipher, `IV` is the initial vector, then

```

C_1 = E_k(IV) XOR M_1
C_2 = E_k(C_1) XOR M_2

```

```

...

```

```

C_(n-1) = E_k(C_(n - 2)) XOR M_(n-1)
C_n = E_k(C_(n - 1)) [1..m] XOR M_n

```

Cipher feedback (CFB) transforms a block cipher into a stream cipher. It encrypts one byte at a time, not requiring any padding.

If E_k is a block encryption function on a block cipher, E_k block size, IV is an initialization vector, then the plaintext bytes are transformed into ciphertext bytes $C_1, \dots, C_n$ as follows:

```

I_1 = IV
C_1 = E_k(I_1) [1..8] XOR M_1
I_2 = I_1 [9..b] << 8 | C_1
C_2 = E_k(I_2) [1..8] XOR M_2

```

```

...

```

```

I_(n-1) = I_(n-2) [9..b] << 8 | C_(n-2)
C_(n-1) = E_k(I_(n-1)) [1..8] XOR M_(n-1)
I_n = I_(n-1) [9..b] << 8 | C_(n-1)
C_n = E_k(I_n) [1..8] XOR M_n

```

It's nice to see the operations for applying a block cipher in CFB and Cipher Feedback (CFB) mode. The operations use `void *` to pass cipher contexts around.

```

void cfb_encrypt (const void *ctx, nettle_cipher_func *f, size_t [Function]
    block_size, uint8_t *iv, size_t length, uint8_t *dst, const uint8_t *src)
void cfb_decrypt (const void *ctx, nettle_cipher_func *f, size_t [Function]
    block_size, uint8_t *iv, size_t length, uint8_t *dst, const uint8_t *src)

```

Apply the encryption or decryption function f in CFB mode. The ciphertext block processes copy into `iv` before turning, so that a single byte can be processed by a sequence of calls to `cfb_encrypt`. Note that for CFB mode, the encryption and decryption functions are the same, always block encryption or decryption.

★ In a single encryption using a sequence of calls to `cfb_encrypt`, a buffer must be used to store the output blocks.

```

void cfb8_encrypt (const void *ctx, nettle_cipher_func *f, size_t [Function]
    block_size, uint8_t *iv, size_t length, uint8_t *dst, const uint8_t *src)
void cfb8_decrypt (const void *ctx, nettle_cipher_func *f, size_t [Function]
    block_size, uint8_t *iv, size_t length, uint8_t *dst, const uint8_t *src)

```

Apply the encryption or decryption function f in CFB mode. In block processing, copy into `iv` before turning, so that a single byte can be processed by a sequence of calls to `cfb8_encrypt`. Note that for CFB mode, the encryption and decryption functions are the same.

encryption operation and decryption operation or
 the underlying block cipher.

Like CBC, the array is also copied across.

CFB_CTX (*context_type*, *block_size*) [acro]

Expands to

```
{
    context_type ctx;
    uint8_t iv[block_size];
}
```

CFB_SET_IV(*ctx*, *iv*) [acro]

First argument is a pointer to a context structure as defined by CFB_CTX, and the second is a pointer to an initial value vector that is copied into the context.

CFB_ENCRYPT (*ctx*, *f*, *length*, *dst*, *src*) [acro]

As a way to invoke `cfb_encrypt`. First argument is a pointer to a context structure as defined by CFB_CTX, and the second argument is an encryption operation owning the conversions. The third argument is the source and destination array or operation.

CFB_DECRYPT (*ctx*, *f*, *length*, *dst*, *src*) [acro]

As a way to invoke `cfb_decrypt`. First argument is a pointer to a context structure as defined by CFB_CTX, and the second argument is an encryption operation owning the conversions. The third argument is the source and destination array or operation.

CFB8_CTX (*context_type*, *block_size*) [acro]

Expands to

```
{
    context_type ctx;
    uint8_t iv[block_size];
}
```

CFB8_SET_IV (*ctx*, *iv*) [acro]

First argument is a pointer to a context structure as defined by CFB8_CTX, and the second is a pointer to an initial value vector that is copied into the context.

CFB8_ENCRYPT (*ctx*, *f*, *length*, *dst*, *src*) [acro]

As a way to invoke `cfb8_encrypt`. First argument is a pointer to a context structure as defined by CFB8_CTX, and the second argument is an encryption operation owning the conversions. The third argument is the source and destination array or operation.

CFB8_DECRYPT (*ctx*, *f*, *length*, *dst*, *src*) [acro]

As a way to invoke `cfb8_decrypt`. First argument is a pointer to a context structure as defined by CFB8_CTX, and the second argument is an encryption operation owning the conversions. The third argument is the source and destination array or operation.


```

T_(n-1) = T_(n-2) MUL a
CC_(n-1) = E_k1(P_(n-1) XOR T_(n-1)) XOR T_(n-1)

T_n = T_(n-1) MUL a
PP = [1..m]Pn | [m+1..128]CC_(n-1)
C_(n-1) = E_k1(PP XOR T_n) XOR T_n

C_n = [1..m]CC_(n-1)

```

7.3.4.1 General (XTS) interface.

two generic functions to encrypt and decrypt using XTS block cipher are as follows:

```

void xts_encrypt_message (const void *enc_ctx, const void [Function]
                        *twk_ctx, nettle_cipher_func *encf, const uint8_t *tweak, size_t length,
                        uint8_t *dst, const uint8_t *src)
void xts_decrypt_message (const void *dec_ctx, const void [Function]
                        *twk_ctx, nettle_cipher_func *decf, nettle_cipher_func *encf, const uint8_t
                        *tweak, size_t length, uint8_t *dst, const uint8_t *src)

```

Applying encryption function *encf* or decryption function *decf* on XTS blocks. At least one block (64 bytes) worth of data must be available to process any given block and 64 bytes size.

Functions *encf* *decf* are of type

```
void f (const void *ctx, size_t length, uint8_t *dst, const uint8_t *src),
```

and *xts_encrypt_message* and *xts_decrypt_message* functions pass their arguments *enc_ctx*, *twk_ctx* and *dec_ctx* to the functions *encf*, *decf* as *ctx*.

7.3.4.2 XTS-AES interface

AEAD functions provide an API for using XTS mode with AEAD block ciphers. Parameters available as arguments are *nt_rac*, *enc_ctx*, *dec_ctx*, *twk_ctx*, *encf* and *decf* are replaced with an AEAD context structure *ctx*, an appropriate function *ustb_cba* block using any other encryption or decryption functions *nt_rac*.

```
struct xts_aes128_key [Context struct]
```

Holds state corresponding to AEAD block cipher.

```
struct xts_aes256_key [Context struct]
```

Holds state corresponding to AEAD block cipher.

```
void xts_aes128_set_encrypt_key (struct xts_aes128_key *ctx, [Function]
                                const uint8_t *key)
```

```
void xts_aes256_set_encrypt_key (struct xts_aes256_key *ctx, [Function]
                                const uint8_t *key)
```

```
void xts_aes128_set_decrypt_key (struct xts_aes128_key *ctx, [Function]
                                const uint8_t *key)
```


owever AEAD functions produce authenticated tag as a separate output or both encryption and decryption.

Both associated data and message (plaintext or ciphertext) can be processed incrementally. In general, an associated data must be processed before message, and a case but the last one must use a next tag state output blocks, although so AEAD may permit other conventions. CC does a bit special in that it requires message next step, other AEAD constructions don't have this restriction.

support AEAD constructions are Galois Counter Mode (GCM), EAX, CAC-Poly 3.5, and Counter with CBC-AC (CC). The reason we want to see GCM authentication, see <http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/CWC-GCM/Ferguson2> CC and EAX use the same building blocks, but the EAX signature and randomization is a couple of nonconventional so CC. For example, EAX is a good conservative choice.

For recent CAC-Poly 3.5 may also be an attractive but for adventurous alternative, in particular for performance is important.

7.4.1 EAX

EAX is an AEAD mode with combined symmetric encryption, authentication. [C 1], page 3, with a message authentication based on CBC, authentication. [CBC], page 3. The permutation is the structure to process with blocks of bits (64 octets). EAX was designed as a reaction to the CC mode, authentication. [CC], page 5, which uses the same primitives but also includes randomization and integrity properties.

EAX supports arbitrary nonce, it's very possible to use any nonce in case of using message encryption or decryption.

It's support for EAX consists of a low-level general interface, so conversion across an specific functions for EAX using AE as a universal cipher. The interface is defined in `<nettle/eax.h>`

7.4.1.1 General EAX interface

struct eax_key [Context struct]
EAX state with cipher key only on the key, but not on the nonce or the message.

struct eax_ctx [Context struct]
Handle state corresponding to a particular message.

EAX_BLOCK_SIZE [Constant]
EAX's blocks, 64.

EAX_DIGEST_SIZE [Constant]
The output EAX digest, also 64.

void eax_set_key (*struct eax_key *key*, *const void *cipher*, [Function]
*nettle_cipher_func *f*)

Initialize *key*. *cipher* gives a context structure for the universal cipher, which must be an previous native encryption, and *f* is the encryption function.

```
void eax_set_nonce (struct eax_ctx *eax, const struct eax_key *key, [Func on]
                    const void *cipher, nettle_cipher_func *f, size_t nonce_length, const
                    uint8_t *nonce)
```

Initializes `ctx` for processing a new message, using `key` and `nonce`.

```
void eax_update (struct eax_ctx *eax, const struct eax_key *key, const [Func on]
                 void *cipher, nettle_cipher_func *f, size_t data_length, const uint8_t
                 *data)
```

Processes associated data or authentication. Although associated data *must* be authenticated, it is not necessary to compute the AEAD constructions, or EAX's not necessary to compute the processing associated data before encryption or decryption.

```
void eax_encrypt (struct eax_ctx *eax, const struct eax_key *key, [Func on]
                  const void *cipher, nettle_cipher_func *f, size_t length, uint8_t *dst, const
                  uint8_t *src)
```

```
void eax_decrypt (struct eax_ctx *eax, const struct eax_key *key, [Func on]
                  const void *cipher, nettle_cipher_func *f, size_t length, uint8_t *dst, const
                  uint8_t *src)
```

Encrypts or decrypts associated data. `cipher` is the context structure underlying the encryption or decryption. Although associated data *must* be authenticated, it is not necessary to compute the processing associated data before encryption or decryption.

```
void eax_digest (struct eax_ctx *eax, const struct eax_key *key, const [Func on]
                 void *cipher, nettle_cipher_func *f, size_t length, uint8_t *digest);
```

Extracts the associated data (also known as "authentication tag"). The operation with processing associated data. If `length` is at least `EAX_DIGEST_SIZE`, only the first `length` octets of the tag are written.

7.4.1.2 EAX helper macros

Following macros are defined.

```
EAX_CTX (context_type) [acro]
```

Defines a new context structure, including the underlying encryption or decryption algorithm. It expands to

```
{
    struct eax_key key;
    struct eax_ctx eax;
    context_type cipher;
}
```

For all macros, `ctx`, is a context structure as defined by `EAX_CTX`, an `encrypt` is the encryption operation and `decrypt` is the decryption operation.

```
EAX_SET_KEY (ctx, set_key, encrypt, key) [acro]
```

`set_key` is the function or setting the encryption or decryption algorithm, and `key` is the key.

```
EAX_SET_NONCE (ctx, encrypt, length, nonce) [acro]
```

`nonce` is the nonce to be used for the processing.

EAX_UPDATE (*ctx*, *encrypt*, *length*, *data*) [acro]
 Process associated data or authentication.

EAX_ENCRYPT (*ctx*, *encrypt*, *length*, *dst*, *src*) [acro]

EAX_DECRYPT (*ctx*, *encrypt*, *length*, *dst*, *src*) [acro]
 Process associated data or encryption or decryption.

EAX_DIGEST (*ctx*, *encrypt*, *length*, *digest*) [acro]

Extract the authentication tag or tag associated.

7.4.1.3 EAX-AES128 interface

Following functions provide EAX using AES as the underlying cipher.

struct eax_aes128_ctx [Context struct]

Context struct, initialized using EAX_CTX.

void eax_aes128_set_key (*struct eax_aes128_ctx *ctx*, *const uint8_t *key*) [Function]

Initialize *ctx* using the given *key*.

void eax_aes128_set_nonce (*struct eax_aes128_ctx *ctx*, *size_t length*, *const uint8_t *iv*) [Function]

Initialize the processing state, using the given nonce.

void eax_aes128_update (*struct eax_aes128_ctx *ctx*, *size_t length*, *const uint8_t *data*) [Function]

Process associated data or authentication. Although associated data *must* be a contiguous subsequence of blocks.

void eax_aes128_encrypt (*struct eax_aes128_ctx *ctx*, *size_t length*, *uint8_t *dst*, *const uint8_t *src*) [Function]

void eax_aes128_decrypt (*struct eax_aes128_ctx *ctx*, *size_t length*, *uint8_t *dst*, *const uint8_t *src*) [Function]

Encrypts or decrypts the associated data. Although associated data *must* be a contiguous subsequence of blocks.

void eax_aes128_digest (*struct eax_aes128_ctx *ctx*, *size_t length*, *uint8_t *digest*); [Function]

Extracts the associated tag (also known as “authentication tag”). The size of the output will be processing associated data. If *length* is at least **EAX_DIGEST_SIZE**, only the first *length* octets of the tag are written.

7.4.2 Galois counter mode

Galois counter mode is an AEAD constructions combining counter mode with associated authentication based on universal hashing. An objective of this design is to provide good performance for applications, without requiring AC algorithms (Section 5.1 [Key as functions], page 6) because a bottleneck for high-speed applications. It was proposed by David A. McGraw and Jonathan G. M. and, and by I. N., I. N. p. ca. Publication -3 D

(<http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>). It is constructed on top of a block cipher, usually available as a block of bits.

aut nt cat on n GC as so nown w a n ss s, s <http://csrc.nist.gov/groups/ST/toolk>.
In part cu ar, on't us GC w t s ort aut nt cat on tags.

tt's support for GC consists of a low-level generic interface, so conversion across, and specific functions for GC using AE² or C++ as a target underlying compiler. The interface is in the file `<nettle/gcm.h>`

7.4.2.1 General GCM interface

struct gcm_key [Cont xt struct]
 ssag n p n nt as sub- y, an r at tab s.

struct gcm_ctx [Cont xt struct]
 Ho s stat corr spon ng to a part cu ar ssag .

GCM_BLOCK_SIZE [Constant]
GC's block size, 6.

GCM_DIGEST_SIZE [Constant]

GCM_IV_SIZE [Constant]

```
void gcm_set_key (struct gcm_key *key, const void *cipher, [Func on]
                 nettle_cipher_func *f)
In a s key. cipher g v s a cont xt struct or t un r y ng c p r, w c ust
av b n pr v ous y n t a or n c r y p t o n, an f s t n c r y p t o n u n c t o n.
```

```
void gcm_set_iv (struct gcm_ctx *ctx, const struct gcm_key *key, [Funct on]
                size_t length, const uint8_t *iv)
In a s ctx us ng t g v n I . key argu nt s actua y n on y length
# f rs ro GCM_IV_SIZE.
```

```
void gcm_update (struct gcm_ctx *ctx, const struct gcm_key *key, [Funct on]
                size_t length, const uint8_t *data)
Prov s assoc at ata to b aut nt cat . I us , ust b ca b or gcm_
encrypt or gcm_decrypt. A but t ast ca or ac ssag must us a ngt
t at sa utp ot boc s .
```

```
void gcm_encrypt (struct gcm_ctx *ctx, const struct gcm_key *key,      [Funct on]
                  const void *cipher, nettle_cipher_func *f, size_t length, uint8_t *dst, const
                  uint8_t *src)
```

```
void gcm_decrypt (struct gcm_ctx *ctx, const struct gcm_key *key,      [Funct on]
                 const void *cipher, nettle_cipher_func *f, size_t length, uint8_t *dst, const
                 uint8_t *src)
```

Encrypts or decrypts data on a message. cipher is the context structure
 un_r yng cp us(a) ngt 35 an) - 6 3 () - 3 96 J P 5 .9 9 - 9

void gcm_digest (*struct gcm_ctx *ctx, const struct gcm_key *key,* [Function]
*const void *cipher, nettle_cipher_func *f, size_t length, uint8_t *digest*)
 Extracts the digest (also known “authentication tag”). *digest* is a
 operation with processing a digest. It’s strongly recommended that *length* is GCM_
 DIGEST_SIZE, but you provide a smaller value, only the first *length* octets of
 digest are written.

to encrypt a digest using GCM, first take a context or a unary block cipher
 with a key to use for encryption. Then call above functions not allowing for **gcm_
 set_key, gcm_set_iv, gcm_update, gcm_encrypt, gcm_decrypt**. **gcm_encrypt** and **gcm_
 decrypt** are analogous, using **gcm_decrypt** instead of **gcm_encrypt** (not that GCM
 is symmetric encryption operation on a unary block cipher). **gcm_update** and **gcm_
 digest** are also unary, call **gcm_set_iv** with an *iv*.

7.4.2.2 GCM helper macros

allowing macros are defined.

GCM_CTX (*context_type*) [acro]
 is a non-atomic context struct, containing context of a unary
 cipher, *type* as sub-type, and pointer to digest state. It expands to

```
{
    struct gcm_key key;
    struct gcm_ctx gcm;
    context_type cipher;
}
```

Example:

```
struct gcm_aes128_ctx GCM_CTX(struct aes128_ctx);
```

allowing macros operate on context structs of the form.

GCM_SET_KEY (*ctx, set_key, encrypt, key*) [acro]
 First argument, *ctx*, is a context struct as defined by **GCM_CTX**. *set_key* and *en-
 crypt* are functions or static encryption and decryption data using the
 unary cipher.

GCM_SET_IV (*ctx, length, data*) [acro]
 First argument is a context struct as defined by **GCM_CTX**. *length* and *data* give
 the authentication vector (IV).

GCM_UPDATE (*ctx, length, data*) [acro]
 is a preprocessor to call **gcm_update**. First argument is a context struct as
 defined by **GCM_CTX**.

GCM_ENCRYPT (*ctx, encrypt, length, dst, src*) [acro]

GCM_DECRYPT (*ctx, encrypt, length, dst, src*) [acro]

GCM_DIGEST (*ctx, encrypt, length, digest*) [acro]

is a preprocessor to call **gcm_encrypt, gcm_decrypt** or **gcm_digest**. First argument is a
 context struct as defined by **GCM_CTX**. Second argument, *encrypt*, is the encryption
 operation on a unary cipher.

7.4.2.3 GCM-AES interface

Following functions provide the common case of GCM using AES as the underlying cipher. Variants with a specific AES flavor are recommended, with the following struct gcm_aes_ctx are provided to work with the relevant operation.

```
struct gcm_aes128_ctx [Context struct]
struct gcm_aes192_ctx [Context struct]
struct gcm_aes256_ctx [Context struct]
```

Context structs, named using GCM_CTX.

```
struct gcm_aes_ctx [Context struct]
```

Abstract context struct, using the AES operation.

```
void gcm_aes128_set_key (struct gcm_aes128_ctx *ctx, const uint8_t *key) [Function]
```

```
void gcm_aes192_set_key (struct gcm_aes192_ctx *ctx, const uint8_t *key) [Function]
```

```
void gcm_aes256_set_key (struct gcm_aes256_ctx *ctx, const uint8_t *key) [Function]
```

Initialize ctx using the given key.

```
void gcm_aes_set_key (struct gcm_aes_ctx *ctx, size_t length, const uint8_t *key) [Function]
```

Corresponding function, using the AES operation. Always AES operation can be used.

```
void gcm_aes128_set_iv (struct gcm_aes128_ctx *ctx, size_t length, const uint8_t *iv) [Function]
```

```
void gcm_aes192_set_iv (struct gcm_aes192_ctx *ctx, size_t length, const uint8_t *iv) [Function]
```

```
void gcm_aes256_set_iv (struct gcm_aes256_ctx *ctx, size_t length, const uint8_t *iv) [Function]
```

```
void gcm_aes_set_iv (struct gcm_aes_ctx *ctx, size_t length, const uint8_t *iv) [Function]
```

Initialize the processing state, using the given IV.

```
void gcm_aes128_update (struct gcm_aes128_ctx *ctx, size_t length, const uint8_t *data) [Function]
```

```
void gcm_aes192_update (struct gcm_aes192_ctx *ctx, size_t length, const uint8_t *data) [Function]
```

```
void gcm_aes256_update (struct gcm_aes256_ctx *ctx, size_t length, const uint8_t *data) [Function]
```

```
void gcm_aes_update (struct gcm_aes_ctx *ctx, size_t length, const uint8_t *data) [Function]
```

Provides associated data to be authenticated. In use, must be called before gcm_aes_encrypt or gcm_aes_decrypt. A but the associated data must be used as the output of the block.

```

void gcm_aes128_encrypt (struct gcm_aes128_ctx *ctx, size_t length,
                        uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes192_encrypt (struct gcm_aes192_ctx *ctx, size_t length,
                        uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes256_encrypt (struct gcm_aes256_ctx *ctx, size_t length,
                        uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes_encrypt (struct gcm_aes_ctx *ctx, size_t length,
                     uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes128_decrypt (struct gcm_aes128_ctx *ctx, size_t length,
                        uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes192_decrypt (struct gcm_aes192_ctx *ctx, size_t length,
                        uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes256_decrypt (struct gcm_aes256_ctx *ctx, size_t length,
                        uint8_t *dst, const uint8_t *src) [Funct on]
void gcm_aes_decrypt (struct gcm_aes_ctx *ctx, size_t length,
                     uint8_t *dst, const uint8_t *src) [Funct on]

```

Encrypts or decrypts data of a `ssag`. Although `dst` can be `src`, `ssag` must be a `uint8_t` array of `length` bytes.

```

void gcm_aes128_digest (struct gcm_aes128_ctx *ctx, size_t length,
                      uint8_t *digest) [Funct on]
void gcm_aes192_digest (struct gcm_aes192_ctx *ctx, size_t length,
                      uint8_t *digest) [Funct on]
void gcm_aes256_digest (struct gcm_aes256_ctx *ctx, size_t length,
                      uint8_t *digest) [Funct on]
void gcm_aes_digest (struct gcm_aes_ctx *ctx, size_t length, uint8_t
                    *digest) [Funct on]

```

Extracts the `ssag` digest (also known “authentication tag”). `ssag` is a `uint8_t` array of `length` bytes. It’s strongly recommended that `length` be `GCM_DIGEST_SIZE`, but you provide a `ssag` value, only the first `length` octets of the `ssag` are written.

7.4.2.4 GCM-Camellia interface

Following functions provide the interface to GCM using Camellia as the underlying cipher.

```

struct gcm_camellia128_ctx [Cont xt struct]
struct gcm_camellia256_ctx [Cont xt struct]
    Cont xt structs, using GCM_CTX.

```

```

void gcm_camellia128_set_key (struct gcm_camellia128_ctx *ctx,
                             const uint8_t *key) [Funct on]
void gcm_camellia256_set_key (struct gcm_camellia256_ctx *ctx,
                             const uint8_t *key) [Funct on]

```

Initializes `ctx` using the given `key`.

```

void gcm_camellia128_set_iv (struct gcm_camellia128_ctx *ctx,      [Funct on]
    size_t length, const uint8_t *iv)
void gcm_camellia256_set_iv (struct gcm_camellia256_ctx *ctx,      [Funct on]
    size_t length, const uint8_t *iv)
    Into a state processing state, using given I .

void gcm_camellia128_update (struct gcm_camellia128_ctx *ctx,      [Funct on]
    size_t length, const uint8_t *data)
void gcm_camellia256_update (struct gcm_camellia256_ctx *ctx,      [Funct on]
    size_t length, const uint8_t *data)
    Provides associated authentication. I use, must be called before gcm_
    camellia_encrypt or gcm_camellia_decrypt. A but t ast ca or ac ssag
    must use a ngt t at sa utp ot boc s .

void gcm_camellia128_encrypt (struct gcm_camellia128_ctx *ctx,      [Funct on]
    size_t length, uint8_t *dst, const uint8_t *src)
void gcm_camellia256_encrypt (struct gcm_camellia256_ctx *ctx,      [Funct on]
    size_t length, uint8_t *dst, const uint8_t *src)
void gcm_camellia128_decrypt (struct gcm_camellia128_ctx *ctx,      [Funct on]
    size_t length, uint8_t *dst, const uint8_t *src)
void gcm_camellia256_decrypt (struct gcm_camellia256_ctx *ctx,      [Funct on]
    size_t length, uint8_t *dst, const uint8_t *src)
    Encrypts or decrypts data of a ssag . A but t ast ca or ac ssag
    must use a ngt t at sa utp ot boc s .

void gcm_camellia128_digest (struct gcm_camellia128_ctx *ctx,      [Funct on]
    size_t length, uint8_t *digest)
void gcm_camellia192_digest (struct gcm_camellia192_ctx *ctx,      [Funct on]
    size_t length, uint8_t *digest)
void gcm_camellia256_digest (struct gcm_camellia256_ctx *ctx,      [Funct on]
    size_t length, uint8_t *digest)
void gcm_camellia_digest (struct gcm_camellia_ctx *ctx, size_t      [Funct on]
    length, uint8_t *digest)
    Extracts the ssag g st (also known “authentication tag”). s st na
    operation with processing a ssag . It’s strongly recommended that length s GCM_
    DIGEST_SIZE, but you provide a s a r va u , only the first length octets of
    g st ar wr tt n.

```

7.4.3 Counter with CBC-MAC mode

CC is a combination of counter and ssag authentication based on cipher block chaining, and using blocks as EAX, see Section 7.4.3 [EAX], page 5. It is constructed on top of a block cipher with a block size of 128 bits. CC is recommended by IETF in IETF RFC 7178, “Authenticated Encryption with Associated Data” (http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf). It supports CC consists of a low-level generic interface, a ssag encryption and authentication interface, and specific functions for CC using AES as the underlying block cipher. The source code is in the file ‘<nettle/ccm.h>’.

In CCM, the length of the associated data is known before processing. The associated data is padded to a nonce, since the associated data is not encrypted, the padding is not encrypted. For example, with a nonce of 10 octets, the associated data of 10 octets is padded to 10 octets, the associated data is padded to $2^{10} - 10$ octets.

CCM encryption operations as follows

- nonce and associated data are concatenated to create $B_0 = \text{flags} \mid \text{nonce} \mid \text{mlength}$
- concatenate associated data and padding to form string $B = L(\text{adata}) \mid \text{adata} \mid \text{padding} \mid \text{plaintext} \mid \text{padding}$ with padding being the sort of string of 16 bytes such that the length of string is a multiple of 16 bytes, and $L(\text{adata})$ is an integer of the length of adata .
- string B is padded to blocks $B_1 \dots B_n$
- concatenate tag T such that as $T=0$, for $i=0$ to n , do $T = E_k(B_i \text{ XOR } T)$
- An integer counter s is initialized to create $IC = \text{flags} \mid \text{nonce} \mid \text{padding}$, with padding of the sort of string of 16 bytes such that IC is exactly on block length.
- concatenate tag is encrypted using CCM to $MAC = E_k(IC) \text{ XOR } T$
- plaintext is encrypted using CCM to form an integer counter $IC+1$.

CCM encryption operations are, except that the ciphertext and MAC are not encrypted using CCM to reverse the plaintext and concatenate tag. Authentication tag can be calculated from the authentication, and compared to the value of the associated data to check authenticity.

7.4.3.1 General CCM interface

For all functions of the CCM interface, *cipher* is the context structure returned by the cipher's first encryption function. The cipher's encryption function is a function of the CCM functions. The cipher's encryption function is a function of the CCM functions.

struct ccm_ctx [Context structure]

Holder of state corresponding to a particular associated data.

CCM_BLOCK_SIZE [Constant]

CCM's block size, 16.

CCM_DIGEST_SIZE [Constant]

Length of CCM digest, 16.

CCM_MIN_NONCE_SIZE [Constant]

CCM_MAX_NONCE_SIZE [Constant]

The minimum and maximum size of a CCM nonce, and, respectively.

CCM_MAX_MSG_SIZE (nonce_size) [acro]

Argument of plaintext length, when using CCM with a nonce of given size.

```
void ccm_set_nonce (struct ccm_ctx *ctx, const void *cipher, [Funct on]
    nettle_cipher_func *f, size_t noncelen, const uint8_t *nonce, size_t
    authlen, size_t msglen, size_t taglen)
```

Initializes the context with the given nonce and sets the authentication tag size, the associated authentication process.

```
void ccm_update (struct ccm_ctx *ctx, const void *cipher, [Funct on]
    nettle_cipher_func *f, size_t length, const uint8_t *data)
```

Provides associated authentication tag. Must be called after `ccm_set_nonce`, and before `ccm_encrypt`, `ccm_decrypt`, or `ccm_digest`.

```
void ccm_encrypt (struct ccm_ctx *ctx, const void *cipher, [Funct on]
    nettle_cipher_func *f, size_t length, uint8_t *dst, const uint8_t *src)
```

```
void ccm_decrypt (struct ccm_ctx *ctx, const void *cipher, [Funct on]
    nettle_cipher_func *f, size_t length, uint8_t *dst, const uint8_t *src)
```

Encrypts or decrypts the message. Must be called after `ccm_set_nonce` and before `ccm_digest`. A buffer must be provided to store the output. The output buffer must be at least as large as the input buffer.

```
void ccm_digest (struct ccm_ctx *ctx, const void *cipher, [Funct on]
    nettle_cipher_func *f, size_t length, uint8_t *digest)
```

Extracts the authentication tag (also known as “authentication tag”). The size of the output is determined by the `length` parameter. The output must be at least as large as the output of `ccm_set_nonce`, but you provide a larger value, only the first `length` octets of the tag are written.

To encrypt a message using the given context, set the nonce and call `ccm_set_nonce` and then call `ccm_update` to generate the authentication tag. After the authentication tag is generated, call `ccm_encrypt` to encrypt the plaintext. Finally, call `ccm_digest` to return the authentication tag.

To decrypt a message, use `ccm_set_nonce` and `ccm_update` to set the context as you would for encryption, and then call `ccm_decrypt` to decrypt the ciphertext. After decrypting the ciphertext, call `ccm_digest` to return the authentication tag. The authentication tag must be at least as large as the authentication tag generated by `ccm_encrypt`.

7.4.3.2 CCM message interface

The CCM message interface provides a simple interface for encrypting and decrypting messages. The interface consists of a set of functions that take a context, a message, and a nonce as input and return the ciphertext, the authentication tag, and the plaintext. The functions are `ccm_encrypt_message`, `ccm_decrypt_message`, and `ccm_digest_message`. The `ccm_encrypt_message` function takes a context, a message, a nonce, and a destination buffer as input and returns the ciphertext, the authentication tag, and the plaintext. The `ccm_decrypt_message` function takes a context, a ciphertext, a nonce, and a destination buffer as input and returns the plaintext, the authentication tag, and the ciphertext. The `ccm_digest_message` function takes a context, a ciphertext, a nonce, and a destination buffer as input and returns the authentication tag.

```
void ccm_encrypt_message (void *cipher, nettle_cipher_func *f, [Funct on]
    size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata,
    size_t tlength, size_t clength, uint8_t *dst, const uint8_t *src)
```

Computes the authentication tag for the message. The `adata` and `src` parameters are the ciphertext and the plaintext, respectively. The `dst` parameter is the destination buffer for the authentication tag.

```
int ccm_decrypt_message (void *cipher, nettle_cipher_func *f, size_t [Func on]
    nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata, size_t
    tlength, size_t mlength, uint8_t *dst, const uint8_t *src)
```

Decrypts the ciphertext *src*, outputs the plaintext to *dst*, concatenates *adata* and the plaintext, and compares it to the *nlength* bytes of *src*. If the value of the received concatenation of *adata* and the ciphertext is *nlength* bytes, then the function will return *src*.

7.4.3.3 CCM-AES interface

AEAD functions provide an API for using CCM with AEAD blocks. The parameters are the same as the ones for the *ssag* interface, except that *cipher*, *f*, and *ctx* parameters are replaced with an AEAD context structure, and a stateful function must be called before using any of the other functions in the interface.

```
struct ccm_aes128_ctx [Context struct]
    Holds state corresponding to a particular ssag encrypt using AEAD block cipher.
```

```
struct ccm_aes192_ctx [Context struct]
    Holds state corresponding to a particular ssag encrypt using AEAD 96 block cipher.
```

```
struct ccm_aes256_ctx [Context struct]
    Holds state corresponding to a particular ssag encrypt using AEAD 128 block cipher.
```

```
void ccm_aes128_set_key (struct ccm_aes128_ctx *ctx, const uint8_t [Func on]
    *key)
```

```
void ccm_aes192_set_key (struct ccm_aes192_ctx *ctx, const uint8_t [Func on]
    *key)
```

```
void ccm_aes256_set_key (struct ccm_aes256_ctx *ctx, const uint8_t [Func on]
    *key)
```

Initializes the encryption for the AEAD block cipher. The other functions must be called before any of the other functions in the AEAD interface.

```
void ccm_aes128_set_nonce (struct ccm_aes128_ctx *ctx, size_t [Func on]
    noncelen, const uint8_t *nonce, size_t authlen, size_t msglen, size_t
    taglen)
```

```
void ccm_aes192_set_nonce (struct ccm_aes192_ctx *ctx, size_t [Func on]
    noncelen, const uint8_t *nonce, size_t authlen, size_t msglen, size_t
    taglen)
```

```
void ccm_aes256_set_nonce (struct ccm_aes256_ctx *ctx, size_t [Func on]
    noncelen, const uint8_t *nonce, size_t authlen, size_t msglen, size_t
    taglen)
```

Similar to *ccm_set_nonce*, except that *cipher*, *f*, and *ctx* are replaced with the context structure.

```

void ccm_aes128_update (struct ccm_aes128_ctx *ctx, size_t length, [Funct on]
    const uint8_t *data)
void ccm_aes192_update (struct ccm_aes192_ctx *ctx, size_t length, [Funct on]
    const uint8_t *data)
void ccm_aes256_update (struct ccm_aes256_ctx *ctx, size_t length, [Funct on]
    const uint8_t *data)
    s a r   n t c a t o c c m _ s e t _ u p d a t e , x c p t t a t c i p h e r , f , a n   c t x a r   r p a c
    w t   a c o n t x t s t r u c t u r .

void ccm_aes128_encrypt (struct ccm_aes128_ctx *ctx, size_t [Funct on]
    length, uint8_t *dst, const uint8_t *src)
void ccm_aes192_encrypt (struct ccm_aes192_ctx *ctx, size_t [Funct on]
    length, uint8_t *dst, const uint8_t *src)
void ccm_aes256_encrypt (struct ccm_aes256_ctx *ctx, size_t [Funct on]
    length, uint8_t *dst, const uint8_t *src)
void ccm_aes128_decrypt (struct ccm_aes128_ctx *ctx, size_t [Funct on]
    length, uint8_t *dst, const uint8_t *src)
void ccm_aes192_decrypt (struct ccm_aes192_ctx *ctx, size_t [Funct on]
    length, uint8_t *dst, const uint8_t *src)
void ccm_aes256_decrypt (struct ccm_aes256_ctx *ctx, size_t [Funct on]
    length, uint8_t *dst, const uint8_t *src)
    s a r   n t c a t o c c m _ s e t _ e n c r y p t a n   c c m _ s e t _ d e c r y p t , x c p t t a t c i p h e r ,
    f , a n   c t x a r   r p a c   w t   a c o n t x t s t r u c t u r .

void ccm_aes128_digest (struct ccm_aes128_ctx *ctx, size_t length, [Funct on]
    uint8_t *digest)
void ccm_aes192_digest (struct ccm_aes192_ctx *ctx, size_t length, [Funct on]
    uint8_t *digest)
void ccm_aes256_digest (struct ccm_aes256_ctx *ctx, size_t length, [Funct on]
    uint8_t *digest)
    s a r   n t c a t o c c m _ s e t _ d i g e s t , x c p t t a t c i p h e r , f , a n   c t x a r   r p a c
    w t   a c o n t x t s t r u c t u r .

void ccm_aes128_encrypt_message (struct ccm_aes128_ctx *ctx, [Funct on]
    size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata,
    size_t tlength, size_t clength, uint8_t *dst, const uint8_t *src)
void ccm_aes192_encrypt_message (struct ccm_aes192_ctx *ctx, [Funct on]
    size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata,
    size_t tlength, size_t clength, uint8_t *dst, const uint8_t *src)
void ccm_aes256_encrypt_message (struct ccm_aes256_ctx *ctx, [Funct on]
    size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata,
    size_t tlength, size_t clength, uint8_t *dst, const uint8_t *src)
int ccm_aes128_decrypt_message (struct ccm_aes128_ctx *ctx, size_t [Funct on]
    nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata, size_t
    tlength, size_t mlength, uint8_t *dst, const uint8_t *src)
int ccm_aes192_decrypt_message (struct ccm_aes192_ctx *ctx, size_t [Funct on]
    nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata, size_t
    tlength, size_t mlength, uint8_t *dst, const uint8_t *src)

```

```
int ccm_aes192_decrypt_message (struct ccm_aes256_ctx *ctx, size_t [Func on]
    nlength, const uint8_t *nonce, size_t alength, const uint8_t *adata, size_t
    tlength, size_t mlength, uint8_t *dst, const uint8_t *src)
    s ar      nt ca to ccm_encrypt_message an ccm_decrypt_message xc pt t at
    cipher an f ar r p ac w t a cont xt structur .
```

7.4.4 ChaCha-Poly1305

C a C a-Poy 3 5 s a c o b n a t o n o t C a C a s t r a c p r a n t p o y 3 5 s s a g
a u t n t c a t o n c o (s e c t o n 5.5 [Poy 3 5], p a g e). I t o r g n a t s r o t a C
c r y p t o g r a p h c b r a r y b y D. J. B r n s t n t a , w c n s a s a r c o n s t r u c t o n b u t w t
a s a n s t a o C a C a .

The presentation of CACa-Poly 3 shows RFC 8439, with CACa c-
 presentation with a -byte nonce and a -byte block counter. It shows up to 56
 gigabytes of data to be encrypted using the same key and nonce.

For C aC a-**po**y 3 **5**, t C aC a c p r s n t a w t a y, o **56** b ts, an a
p r- ssag nonc . rst boc o t y str a (count r a ro) s s t as or
t aut nt cat on sub ys. **0** t s **6** -oct t boc , t rst **6** oct ts sp c y t po y 3 **5**
va uat on po nt, an t n xt **6** byt s sp c y t va u to a n or t na g st.

na 3 byt s o t s b o c a r u n u s . o t t a t u n p o y 3 5-a s, t v a u a t o n
p o n t p n s o n t n o n c . s s p r a b , b c a u s t a s s s n o r a t o n n c a s t
a t t a c r o r s o r a s o n s u c y n o u g t o o r g a v a a u t n t c a t o n t a g, a n o b s r v
(r o t r c v r's b a v o u r) t a t t o r g r y s u c c .

C aC a y str a , start ng w t count r va u , s t n us to ncrypt
t ssag . For aut nt cat on, po y 3 5 s app to t concat nat on o t
assoc at ata, t cryptot xt, an t ng t s o t assoc at ata an t ssag ,
ac a 6 -b t nu b r (g t oct ts, tt - n an). tt n s C aC a-Po y 3 5 n
'<nettle/chacha-poly1305.h>'.

CHACHA_POLY1305_BLOCK_SIZE [Constant]

CHACHA_POLY1305_KEY_SIZE	[Constant]
ChaChaPoly3512, 32	

CHACHA_POLY1305_NONCE_SIZE [Constant]

CHACHA_POLY1305_DIGEST_SIZE [Constant]

```
struct chacha_poly1305_ctx
```

```
void chacha_poly1305_set_key (struct chacha_poly1305_ctx *ctx,      [Funct on]
                             const uint8_t *key)
```

In the `ctx_using_tv_gvn` y. But using the context, you *must* also call `chacha_poly1305_set_nonce`, so be aware.

```
void chacha_poly1305_set_nonce (struct chacha_poly1305_ctx *ctx, [Funct on]
                               const uint8_t *nonce)
```

In the statement, using the given nonc.

7.4.5.2 SIV-CMAC-AES interface

AE[✓] $\mathcal{V}$ -C AC unct ons prov an API or us ng $\mathcal{V}$ -C AC o w t t AE[✓] boc cp rs. para trs a av t sa an ng as t g n ra an ssag nt rac s, xc pt t at t *cipher*, *f*, an *ctx* para trs ar r pac w t an AE[✓] cont xt structur , an a st- y unct on ust b ca b or us ng any o t ot r unct ons n t s nt rac .

```
struct siv_cmac_aes128_ctx [Context struct]
    Ho s stat corr spon ng to a part cu ar ssag ncrypt us ng t AE b oc
    c p r.
```

```

struct siv_cmac_aes256_ctx                                     [Context struct]
    Ho s stat corr spon ng to a part cu ar ssag ncrypt us ng t AE- 56 b oc
    c p r.

```

```
void siv_cmac_aes128_set_key (struct siv_cmac_aes128_ctx *ctx, [Funct on]
                             const uint8_t *key)
```

```
void siv_cmac_aes256_set_key (struct siv_cmac_aes256_ctx *ctx, [Funct on]
                             const uint8_t *key)
```

In a stream cipher, the output sequence of bits is generated by combining the input sequence of bits with a secret key sequence of bits. The output sequence of bits is then encrypted using the same key sequence.

```
void siv_cmac_aes128_encrypt_message (struct siv_cmac_aes128_ctx [Funct on]
                                     *ctx, size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t
                                     *adata, size_t clength, uint8_t *dst, const uint8_t *src)
```

```
void siv_cmac_aes256_encrypt_message (struct siv_cmac_aes256_ctx [Funct on]
    *ctx, size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t
    *adata, size_t clength, uint8_t *dst, const uint8_t *src)
```

```

Co put s t      ssag g st ro t  adata an src para t rs, ncrpts t  p a n-
t xt ro  src, pr p n s t      n t a  at on v ctor to t  c p  rt xt an  outputs t to
dst.      clength var ab  ust b  qua to t      ngt o src p us SIV_DIGEST_SIZE.

```

```
int siv_cmac_aes128_decrypt_message (struct siv_cmac_aes128_ctx [Funct on]
    *ctx, size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t
    *adata, size_t mlength, uint8_t *dst, const uint8_t *src)
```

```
int siv_cmac_aes256_decrypt_message (struct siv_cmac_aes128_ctx [Funct on]
    *ctx, size_t nlength, const uint8_t *nonce, size_t alength, const uint8_t
    *adata, size_t mlength, uint8_t *dst, const uint8_t *src)
```

D decrypts the ciphertext `src`, outputs the plaintext to `dst`, recalculate the authentication vector from `adata` and the plaintext. It uses the received authentication vector `ar` to check, then return the authentication value and authentication message. Otherwise, the function will return `ro`.

7.4.6 The struct nettle aead abstraction

tt nc u s a struct nc u ng n or at on about t support as unct ons. It s
n n '<nettle/nettle-meta.h>'.

```
struct nettle_aead name context_size block_size key_size nonce_size [  ta struct]
digest_size set_encrypt_key set_decrypt_key set_nonce update encrypt decrypt
digest
```

ast s v n attr but s ar unct on po nt rs.

```
struct nettle_aead nettle_gcm_aes128 [Constant struct]
struct nettle_aead nettle_gcm_aes192 [Constant struct]
struct nettle_aead nettle_gcm_aes256 [Constant struct]
struct nettle_aead nettle_gcm_camellia128 [Constant struct]
struct nettle_aead nettle_gcm_camellia256 [Constant struct]
struct nettle_aead nettle_eax_aes128 [Constant struct]
struct nettle_aead nettle_chacha_poly1305 [Constant struct]
```

s ar ost o t AEAD construct ons t at tt p nts. ot t at CC
s ssng, t r qu r nt t at t ssag s s sp c n a vanc a s t
nco pat b w t t nettle_aead abstract on.

tt a so xports a st o a t s construct ons.

```
const struct nettle_aead ** nettle_get_aeads (void) [Funct on]
```

P turns a LL-t r nat st o po nt rs to support a gor t s. s st can b
us to yna ca y nu rat or s arc t support a gor t s.

```
nettle_aeads [acro]
```

Aacro xpan ng to a ca to n tt g t a a s. In ar r v rs ons, t s was not a
acro but t actua array o po nt rs.

7.5 Keyed Hash Functions

A *keyed hash function*, or *Message Authentication Code* (MAC) s a unct on t at ta s a
y an a ssag, an pro uc s x s AC. It s ou b ar to co put a ssag
an a atc ng AC w t out now g o t y. It s ou a so b ar to co put t
y g v n on y ssag s an corr spon ng ACs.

K y as unct ons ar us u pr ar y or ssag aut nt cat on, w n A c an
Bob s ar s a s cr t s n r, A c, co put s t AC an attac s t to t ssag.
r c v r, Bob, a so co put s t AC o t ssag, us ng t sa y, an co par s
t at to A c's va u. I t y atc, Bob can b assur t at t ssag as not b n
o on ts way ro A c.

How v r, un g ta s gnatur s, t s assuranc s not trans rab. Bob can't s ow
t ssag an t AC to a t r party an prov t at A c s nt t at ssag. ot
v n g v s away t y to t t r party. r ason s t at t same y s us on
bot s s, an anyon now ng t y can cr at a corr ct AC or any ssag. I Bob
b v s t at on y an A c nows t y, an nows t at n't attac a AC
to a part cu ar ssag, nows t ust b A c w o t. How v r, t t r party
can't st ngu s b tw n a AC cr at by A c an on cr at by Bob.

K y as unct ons ar typ ca y a ot ast r t an g ta s gnatur s as w.

7.5.1 HMAC

One can build many hash functions from ordinary hash functions. One can construct ones simply concatenating secret key and message and hashing that, but such constructions are weak. A better construction is HMAC, described in *RFC 2104*.

For an underlying hash function H , with digest size l and initial value b , HMAC-H constructs as follows. From a given key k , two distinct subkeys k_i and k_o are constructed, both of length b . HMAC-H on a message m is then computed as $H(k_o \parallel H(k_i \parallel m))$, where $\parallel$ denotes string concatenation.

HMAC keys can be of any length, but this reduces to us only one length, the digest size of the underlying hash function H . Keys that are longer than b are sorted into length b by hashing with H , so arbitrary long keys aren't very useful.

The HMAC functions are defined in `<nettle/hmac.h>`. Here are abstract functions that use a pointer to a `struct nettle_hash`

It can be used to create an HMAC context struct, thereby,

```
struct HMAC_CTX(struct md5_ctx) ctx;
```

or to give it a struct tag,

```
struct hmac_md5_ctx HMAC_CTX (struct md5_ctx);
```

HMAC_SET_KEY (*ctx*, *H*, *length*, *key*) [acro]

ctx is a pointer to a context struct as defined by HMAC_CTX, *H* is a pointer to a **const struct nettle_hash** describing the underlying hash function (so that static type of the components of *ctx*). Last two arguments specify the secret key.

HMAC_DIGEST (*ctx*, *H*, *length*, *digest*) [acro]

ctx is a pointer to a context struct as defined by HMAC_CTX, *H* is a pointer to a **const struct nettle_hash** describing the underlying hash function. Last two arguments specify where to store the digest.

That is, there is no HMAC_UPDATE macro, simply call **hmac_update** function directly, or the update function of the underlying hash function.

7.5.2 Concrete HMAC functions

Now we come to the specific HMAC functions, which are as follows and generate HMAC functions.

7.5.2.1 HMAC-MD5

struct hmac_md5_ctx [Context struct]

void hmac_md5_set_key (*struct hmac_md5_ctx *ctx*, *size_t key_length*, *const uint8_t *key*) [Function]

Initializes the context with the key.

void hmac_md5_update (*struct hmac_md5_ctx *ctx*, *size_t length*, *const uint8_t *data*) [Function]

Processes the data.

void hmac_md5_digest (*struct hmac_md5_ctx *ctx*, *size_t length*, *uint8_t *digest*) [Function]

Extracts the HMAC, writing it to *digest*. *length* may be as small as MD5_DIGEST_SIZE, now the case on the first *length* octets of the HMAC are written.

This function also stores the context for processing new messages, without saving.

7.5.2.2 HMAC-RIPEMD160

struct hmac_ripemd160_ctx [Context struct]

void hmac_ripemd160_set_key (*struct hmac_ripemd160_ctx *ctx*, *size_t key_length*, *const uint8_t *key*) [Function]

Initializes the context with the key.

void hmac_ripemd160_update (*struct hmac_ripemd160_ctx *ctx*, *size_t length*, *const uint8_t *data*) [Function]

Processes the data.

void hmac_ripemd160_digest (*struct hmac_ripemd160_ctx *ctx, size_t length, uint8_t *digest*) [Funct on]

Extracts t AC, wr t ng t to *digest*. *length* ay b s a r t an RIPEMD160_DIGEST_SIZE, n w c cas on y t rst *length* oct ts o t AC ar wr tt n.

s unct on a so r s ts t cont xt or proc ss ng n w ssag s, w t t sa y.

7.5.2.3 HMAC-SHA1

struct hmac_sha1_ctx [Cont xt struct]

void hmac_sha1_set_key (*struct hmac_sha1_ctx *ctx, size_t key_length, const uint8_t *key*) [Funct on]

In t a s t cont xt w t t y.

void hmac_sha1_update (*struct hmac_sha1_ctx *ctx, size_t length, const uint8_t *data*) [Funct on]

Proc ss so or ata.

void hmac_sha1_digest (*struct hmac_sha1_ctx *ctx, size_t length, uint8_t *digest*) [Funct on]

Extracts t AC, wr t ng t to *digest*. *length* ay b s a r t an SHA1_DIGEST_SIZE, n w c cas on y t rst *length* oct ts o t AC ar wr tt n.

s unct on a so r s ts t cont xt or proc ss ng n w ssag s, w t t sa y.

7.5.2.4 HMAC-SHA256

struct hmac_sha256_ctx [Cont xt struct]

void hmac_sha256_set_key (*struct hmac_sha256_ctx *ctx, size_t key_length, const uint8_t *key*) [Funct on]

In t a s t cont xt w t t y.

void hmac_sha256_update (*struct hmac_sha256_ctx *ctx, size_t length, const uint8_t *data*) [Funct on]

Proc ss so or ata.

void hmac_sha256_digest (*struct hmac_sha256_ctx *ctx, size_t length, uint8_t *digest*) [Funct on]

Extracts t AC, wr t ng t to *digest*. *length* ay b s a r t an SHA256_DIGEST_SIZE, n w c cas on y t rst *length* oct ts o t AC ar wr tt n.

s unct on a so r s ts t cont xt or proc ss ng n w ssag s, w t t sa y.

7.5.2.5 HMAC-SHA512

struct hmac_sha512_ctx [Cont xt struct]

void hmac_sha512_set_key (*struct hmac_sha512_ctx *ctx, size_t key_length, const uint8_t *key*) [Funct on]

In t a s t cont xt w t t y.

```
void hmac_sha512_update (struct hmac_sha512_ctx *ctx, size_t length, const uint8_t *data) [Function]
```

Processes `data`.

```
void hmac_sha512_digest (struct hmac_sha512_ctx *ctx, size_t length, uint8_t *digest) [Function]
```

Extracts the HMAC, writing it to `digest`. `length` may be as large as `SHA512_DIGEST_SIZE`, new cases only the first `length` octets of the HMAC are written.

See function `hmac_sha512_init` for context processing new messages, with the same.

7.5.3 UMAC

MAC is a message authentication code based on universal hashing, an algorithm for producing a fixed-size output (in contrast to GCM, which produces a variable-size output). MACs are used for integrity and authentication. MACs are used in *RFC 4418*.

MACs are always `len` bytes (octets). They are used as an input to the AEAD block cipher. The block cipher is used to generate various intermediate values from the MAC. MACs are arbitrary size, and are used as a source of a unique nonce. One value must not be reused for two MACs with the same key, but they may not be padded.

nonce must be at least `len` octets, and at most `len`, nonces sort of at least `len` octets are required. The padding of the MAC is not used to generate the intermediate values, so the padding of the MAC is not used. The intermediate values are used to generate the final output. For example, if the nonce is "abc" (3 octets), the value is padded to 4 octets or the first padding. For the next padding, the nonce is padded to "ab ", and the next value is padded to 4 octets.

MACs are in four variants, or 32-bit outputs (32 octets), 64-bit (64 octets), 96-bit (96 octets) and 128-bit (128 octets), corresponding to 32-bit, 64-bit, 96-bit and 128-bit security. The size of the output is so small (but not always) given the size of the input using a long output and truncating the result. It is important to use the right variant. For consistency with other standard MAC functions, the `_digest` functions for MAC accept a length parameter so that the output can be truncated to any size, but the size of the output is so small that the output is not truncated to any size.

Intermediate values are used to generate the final output. The intermediate values are used to generate the final output. The intermediate values are used to generate the final output.

The standard MAC is `<nettle/umac.h>`.

```
struct umac32_ctx [Context struct]
struct umac64_ctx [Context struct]
struct umac96_ctx [Context struct]
struct umac128_ctx [Context struct]
```

Each MAC variant uses its own context struct.

UMAC_KEY_SIZE [Constant]
 The size of the UMAC key, in bytes.

UMAC_MIN_NONCE_SIZE [Constant]
 The minimum size of the UMAC nonce, in bytes.

UMAC_MAX_NONCE_SIZE [Constant]
 The maximum size of the UMAC nonce, in bytes.

UMAC32_DIGEST_SIZE [Constant]
 The size of the 32-bit UMAC digest, in bytes.

UMAC64_DIGEST_SIZE [Constant]
 The size of the 64-bit UMAC digest, in bytes.

UMAC96_DIGEST_SIZE [Constant]
 The size of the 96-bit UMAC digest, in bytes.

UMAC128_DIGEST_SIZE [Constant]
 The size of the 128-bit UMAC digest, in bytes.

UMAC_BLOCK_SIZE [Constant]
 The size of the UMAC block, in bytes.

void umac32_set_key (struct umac32_ctx *ctx, const uint8_t *key) [Function]
 void umac64_set_key (struct umac64_ctx *ctx, const uint8_t *key) [Function]
 void umac96_set_key (struct umac96_ctx *ctx, const uint8_t *key) [Function]
 void umac128_set_key (struct umac128_ctx *ctx, const uint8_t *key) [Function]
 Set the UMAC key for the context. The key must be a valid UMAC key of the appropriate size. The key is used to calculate the UMAC value for the data. The key is also used to calculate the UMAC value for the data.

void umac32_set_nonce (struct umac32_ctx *ctx, size_t length, const uint8_t *nonce) [Function]
 void umac64_set_nonce (struct umac64_ctx *ctx, size_t length, const uint8_t *nonce) [Function]
 void umac96_set_nonce (struct umac96_ctx *ctx, size_t length, const uint8_t *nonce) [Function]
 void umac128_set_nonce (struct umac128_ctx *ctx, size_t length, const uint8_t *nonce) [Function]
 Set the UMAC nonce for the context. The nonce must be a valid UMAC nonce of the appropriate size. The nonce is used to calculate the UMAC value for the data. The nonce is also used to calculate the UMAC value for the data.

void umac32_update (struct umac32_ctx *ctx, size_t length, const uint8_t *data) [Function]
 void umac64_update (struct umac64_ctx *ctx, size_t length, const uint8_t *data) [Function]
 void umac96_update (struct umac96_ctx *ctx, size_t length, const uint8_t *data) [Function]
 void umac128_update (struct umac128_ctx *ctx, size_t length, const uint8_t *data) [Function]
 Update the UMAC value for the data. The data must be a valid UMAC data of the appropriate size. The data is used to calculate the UMAC value for the data. The data is also used to calculate the UMAC value for the data.

```

void umac32_digest (struct umac32_ctx *ctx, size_t length, uint8_t
    *digest) [Funct on]
void umac64_digest (struct umac64_ctx *ctx, size_t length, uint8_t
    *digest) [Funct on]
void umac96_digest (struct umac96_ctx *ctx, size_t length, uint8_t
    *digest) [Funct on]
void umac128_digest (struct umac128_ctx *ctx, size_t length,
    uint8_t *digest) [Funct on]

```

Extracts the MAC of the message, writing it to *digest*. *length* is usually equal to the size of the output, but you provide a range, only the first *length* octets of the MAC are written. The function returns the context or processing of a new message with the same key. The nonce is not ascribed above, then you use the `_set_nonce` function explicitly or the message.

7.5.4 CMAC

CMAC is a message authentication code based on CBC encryption. It is suitable for systems where block processors are prevalent and provide better than as functions. CMAC is specified in *RFC4493*. Block size is always 64 bits (6 octets). CMAC is specified by IEEE in the Publication 3-B (https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38B.pdf). Block size is always 64 bits (6 octets).

It provides the functions for CMAC with the AEAD block cipher or CMAC with the rpp-DEAD block cipher.

It is in the CMAC namespace '`<nettle/cmac.h>`'.

```

struct cmac_aes128_ctx [Cont xt struct]
struct cmac_aes256_ctx [Cont xt struct]
CMAC128_DIGEST_SIZE [Constant]
    size of an CMAC digest, 6.

```

```

void cmac_aes128_set_key (struct cmac_aes128_ctx *ctx, const
    uint8_t *key) [Funct on]
    function that sets the CMAC context structure or AEAD.

```

```

void cmac_aes128_update (struct cmac_aes128_ctx *ctx, size_t
    length, const uint8_t *data) [Funct on]
    function that processes the data to process the message.

```

```

void cmac_aes128_digest (struct cmac_aes128_ctx *ctx, size_t
    length, uint8_t *digest) [Funct on]
    Extracts the MAC of the message, writing it to digest. length is usually equal to the size of the output, but you provide a range, only the first length octets of the MAC are written. The function returns the context or processing of a new message with the same key.

```

```

void cmac_aes256_set_key (struct cmac_aes256_ctx *ctx, const
    uint8_t *key) [Funct on]
    function that sets the CMAC context structure or AEAD.

```

`void cmac_aes256_update (struct cmac_aes256_ctx *ctx, size_t length, const uint8_t *data)` [Function]

Updates the context `ctx` with `length` bytes of data `data`.

`void cmac_aes256_digest (struct cmac_aes256_ctx *ctx, size_t length, uint8_t *digest)` [Function]

Computes the MAC of `length` bytes of data `data` using the context `ctx`. The result is stored in `digest`. The length `length` is usually equal to the size of the data, but you provide a value, only the first `length` octets of the data are written. The context `ctx` is not modified.

`struct cmac_des3_ctx` [Context structure]

`CMAC64_DIGEST_SIZE` [Constant]

Size of the MAC in bytes.

`void cmac_des3_set_key (struct cmac_des3_ctx *ctx, const uint8_t *key)` [Function]

Initializes the context `ctx` with the key `key`.

`void cmac_des3_update (struct cmac_des3_ctx *ctx, size_t length, const uint8_t *data)` [Function]

Updates the context `ctx` with `length` bytes of data `data`.

`void cmac_des3_digest (struct cmac_des3_ctx *ctx, size_t length, uint8_t *digest)` [Function]

Computes the MAC of `length` bytes of data `data` using the context `ctx`. The result is stored in `digest`. The length `length` is usually equal to the size of the data, but you provide a value, only the first `length` octets of the data are written. The context `ctx` is not modified.

7.5.5 Poly1305

Poly1305-AEAD is a message authentication code (MAC) algorithm designed by D. J. Bernstein. It treats the message as a polynomial of degree 13 .

The message, m bytes, consists of two parts, where the first 16 bytes, and the second 16 bytes, are the polynomial coefficients. The first 16 bytes are the polynomial coefficients, and the second 16 bytes are the polynomial coefficients. The polynomial is evaluated at the point r . For each message, one must also provide a nonce. The nonce is encrypted using the AEAD key, and then the result is XORed with the polynomial result.

The message is split into 16 -bit chunks (with the last chunk possibly being shorter), and each chunk is treated as a polynomial coefficient. The polynomial is evaluated at the point r . Finally, the result is XORed with the nonce (as a polynomial) to produce the final authentication code (MAC).

See <http://cr.yp.to/mac/poly1305-20050329.pdf> for further details.

Currently, variants using a fixed constant r are not recommended. Another variant is the **CAC-Poly1305 AEAD** construction (see Section 7.5.6 [CAC-Poly1305], page 7). The function `poly1305_aead_encrypt` in `'nettle/poly1305.h'`.

<code>POLY1305_AES_KEY_SIZE</code>	[Constant]
Key size, 3 octets.	
<code>POLY1305_AES_DIGEST_SIZE</code>	[Constant]
Digest size for “authenticated”, 6 octets.	
<code>POLY1305_AES_NONCE_SIZE</code>	[Constant]
Nonce size, 6 octets.	
<code>struct poly1305_aes_ctx</code>	[Context struct]
poly1305-aes context struct.	
<code>void poly1305_aes_set_key (struct poly1305_aes_ctx *ctx, const uint8_t *key)</code>	[Function]
Initializes context struct. Also sets the nonce to zero.	
<code>void poly1305_aes_set_nonce (struct poly1305_aes_ctx *ctx, const uint8_t *nonce)</code>	[Function]
Sets the nonce. Calling this function is optional, since the nonce is never used automatically or accessed.	
<code>void poly1305_aes_update (struct poly1305_aes_ctx *ctx, size_t length, const uint8_t *data)</code>	[Function]
Processes <code>data</code> .	
<code>void poly1305_aes_digest (struct poly1305_aes_ctx *ctx, size_t length, uint8_t *digest)</code>	[Function]
Extracts the digest. If <code>length</code> is smaller than <code>POLY1305_AES_DIGEST_SIZE</code> , only the first <code>length</code> octets are written. Also clears the nonce, and prepares the context for processing a new message.	

7.6 Key derivation Functions

A *key derivation function* (KDF) is a function that takes a given symmetric key and some other symmetric keys. A subclass of KDFs is the *password-based key derivation functions* (PBKDFs), which take as input a password or passphrase, and its purpose is typically to strengthen the protection against certain pre-computation attacks by using salt and expensive computation.

7.6.1 HKDF: HMAC-based Extract-and-Expand

HKDF is a key derivation function used as a building block of group-v protocols [LW03]. It is a derivation function based on HMAC described in RFC 5869, and splits into two operations, called ‘extract’ and ‘expand’. The extract operation takes a secret and a random salt to “extract” a fixed-length pseudorandom key (PRK). The second stage takes as input the previous PRK and some information (e.g., text) and expands it into output bytes.

The HKDF functions are defined in the header file ‘<nettle/hkdf.h>’. There are two abstract functions for the extract and expand operations that operate on any HMAC primitive: `nettle_hash_update_func`, and `nettle_hash_digest_func` (not recursive).

```
void hkdf_extract (void *mac_ctx, nettle_hash_update_func *update, [Func on]
    nettle_hash_digest_func *digest, size_t digest_size, size_t secret_size, const
    uint8_t *secret, uint8_t *dst)
```

Extract a **PS** u r a n o **K** y (**PPK**) r o a s c r t a n a s a t a c c o r n g t o H K D F .
 H A C u s t a v b n n t a , w t t s y b n g t s a t o r t E x t r a c t
 o p r a t o n . s u n c t o n w c a t u p d a t e a n d i g e s t u n c t o n s p a s s n g t m a c _ c t x
 c o n t x t p a r a t r a s a n a r g u n t n o r r t o c o p u t g s t o s d i g e s t _ s i z e .
 I n p u t s a r t s c r t s e c r e t o n g t s e c r e t _ l e n g t h . o u t p u t n g t s x t o
 d i g e s t _ s i z e o c t e t s , t u s t o u t p u t b u f f e r d s t u s t a v r o o r o r a t a s t d i g e s t _ s i z e
 o c t e t s .

```
void hkdf_expand (void *mac_ctx, nettle_hash_update_func *update, [Func on]
    nettle_hash_digest_func *digest, size_t digest_size, size_t info_size, const uint8_t
    *info, size_t length, uint8_t *dst)
```

E x p a n a **PS** u r a n o **K** y (**PPK**) t o a n a r b t r a r y s a c c o r n g t o H K D F .
 H A C u s t a v b n n t a , w t t s y b n g t **PPK** r o t E x t r a c t
 o p r a t o n . s u n c t o n w c a t u p d a t e a n d i g e s t u n c t o n s p a s s n g t m a c _ c t x
 c o n t x t p a r a t r a s a n a r g u n t n o r r t o c o p u t g s t o s d i g e s t _ s i z e .
 I n p u t s a r t n o i n f o o n g t i n f o _ l e n g t h , a n t s r r v o u t p u t n g t
 l e n g t h . o u t p u t b u f f e r s d s t w c u s t a v r o o r o r a t a s t l e n g t h o c t e t s .

7.6.2 PBKDF2

o s t w n o w n **PBKDF** s t **PKCS #5 PBKDF2** s c r b n *RFC 2898* w c u s s a
 p s u o - r a n o u n c t o n s u c a s H A C - H A .

t t ' s **PBKDF** u n c t o n s a r n n ' < nettle / p b k d f 2 . h > ' . r s a n a b s t r a c t
 u n c t o n t a t o p r a t o n a n y **PPF** p n t v a t **nettle_hash_update_func**,
nettle_hash_digest_func n t r a c s . r s a s o p r a c r o s a n c o n c r t u n c t o n s
PBKDF -H AC- HA , **PBKDF -H AC- HA** , **PBKDF -H AC- HA3** a n
PBKDF -H AC- HA . F r s t , t a b s t r a c t u n c t o n

```
void pbkdf2 (void *mac_ctx, nettle_hash_update_func *update, [Func on]
    nettle_hash_digest_func *digest, size_t digest_size, unsigned iterations, size_t
    salt_length, const uint8_t *salt, size_t length, uint8_t *dst)
```

D r v s y t r c y r o a p a s s w o r a c c o r n g t o **PKC #5 PBKDF** . **PPF**
 s a s s u t o a v b n n t a a n t s u n c t o n w c a t u p d a t e a n d i g e s t
 u n c t o n s p a s s n g t m a c _ c t x c o n t x t p a r a t r a s a n a r g u n t n o r r t o c o p u t
 g s t o s d i g e s t _ s i z e . I n p u t s a r t s a t s a l t o n g t s a l t _ l e n g t h , t t r a t o n
 c o u n t r i t e r a t i o n s (>) , a n t s r r v o u t p u t n g t l e n g t h . o u t p u t
 b u f f e r s d s t w c u s t a v r o o r o r a t a s t l e n g t h o c t e t s .

L o r C B C a n H A C , t r s a a c r o t o p u s t u n c t o n c o r r e c t y .

```
PBKDF2 (ctx, update, digest, digest_size, iterations, [acro]
    salt_length, salt, length, dst)
```

c t x s a p o n t r t o a c o n t x t s t r u c t p a s s t o t u p d a t e a n d i g e s t u n c t o n s (o
 t t y p s **nettle_hash_update_func** a n **nettle_hash_digest_func** r s p c t v y)
 t o p n t t u n r y n g **PPF** w t g s t s o d i g e s t _ s i z e . I n p u t s a r t
 s a t s a l t o n g t s a l t _ l e n g t h , t t r a t o n c o u n t r i t e r a t i o n s (>) , a n t s r

returning output of length *length*. The output buffer *dst* will contain at least *length* octets.

7.6.3 Concrete PBKDF2 functions

Following are the specific PBKDF2 functions, which are as straightforward as the PBKDF2 function.

7.6.3.1 PBKDF2-HMAC-SHA1

```
void pbkdf2_hmac_sha1 (size_t key_length, const uint8_t *key, [Function]
                      unsigned iterations, size_t salt_length, const uint8_t *salt, size_t
                      length, uint8_t *dst)
    PBKDF2 with HMAC-SHA1. Derive length bytes into buffer dst using the
    password key of length key_length and salt salt of length salt_length, with iteration
    count iterations (> 0). The output buffer dst will contain at least
    length octets.
```

7.6.3.2 PBKDF2-HMAC-SHA256

```
void pbkdf2_hmac_sha256 (size_t key_length, const uint8_t *key, [Function]
                        unsigned iterations, size_t salt_length, const uint8_t *salt, size_t
                        length, uint8_t *dst)
    PBKDF2 with HMAC-SHA256. Derive length bytes into buffer dst using the
    password key of length key_length and salt salt of length salt_length, with iteration
    count iterations (> 0). The output buffer dst will contain at least
    length octets.
```

7.6.3.3 PBKDF2-HMAC-SHA384

```
void pbkdf2_hmac_sha384 (size_t key_length, const uint8_t *key, [Function]
                        unsigned iterations, size_t salt_length, const uint8_t *salt, size_t
                        length, uint8_t *dst)
    PBKDF2 with HMAC-SHA384. Derive length bytes into buffer dst using the
    password key of length key_length and salt salt of length salt_length, with iteration
    count iterations (> 0). The output buffer dst will contain at least
    length octets.
```

7.6.3.4 PBKDF2-HMAC-SHA512

```
void pbkdf2_hmac_sha512 (size_t key_length, const uint8_t *key, [Function]
                        unsigned iterations, size_t salt_length, const uint8_t *salt, size_t
                        length, uint8_t *dst)
    PBKDF2 with HMAC-SHA512. Derive length bytes into buffer dst using the
    password key of length key_length and salt salt of length salt_length, with iteration
    count iterations (> 0). The output buffer dst will contain at least
    length octets.
```

7.7 Public-key algorithms

to use the public-ly available GNU library, or a calculation with arguments. In order to use the public-ly available GNU library, you must install GNU **P**, at least version 3. , before compiling it, and you need to run your program with `-lhogweed -lnettle -lgmp`.

concept of *Public-key* encryption was discovered by Diffie and Martin E. Hellman and published in a paper [9]. In that paper, “symmetric” cryptography, symmetric versus asymmetric, and its uses were described in a new way. An important new sort of system to communicate, a *pair* of systems, is described by nobody.

Public-key cryptography uses trapdoor one-way functions. A *one-way function* is a function F such that it is easy to compute $F(x)$ for any x , but given a value y , it is hard to compute a corresponding x such that $y = F(x)$. Two examples are cryptographic hash functions, and exponentiation in cyclic groups.

A *trapdoor one-way function* is a function F that is one-way, unless one knows some secret information about F . In other words, it is easy to compute both F and F^{-1} given x . It is somewhat strange, too, that $P \neq P^A$ is a plausible assumption.

two important us s or on -way unct ons w t trap oors ar pub c- y ncrypt on, an gta s gnatur s. pub c- y ncrypt on unct ons n tt ar not y t ocu nt , t r st o t s c apt r s about gta s gnatur s.

ous a gta sgnatur a gor t , on ust rst cr at a *key-pair* A pub c y an
a corr spon ng pr vat y. pr vat y s us to s gn ssag s, w t pub c y
s us or v r y ng t at t at sgnatur s an ssag s atc . To car ust b ta n
w n str but ng t pub c y, t n not b pt s cr t, but a ba guy s ab to
r pac t (n trans t, or n so us r's st o nown pub c ys), ba t ngs ay app n.

r ar two op rat ons on can o wt t ys. s gnatur op rat on ta s a
 ssag an a pr vat y, an cr at s a s gnatur or t ssag . A s gnatur s so
 str ng o b ts, usua y at ost a w t ousan b ts or a w un r oct ts. n pap r-
 an - n s gnatur s, t g ta s gnatur p n s on t ssag , so on can't cut t out o
 cont xt an gu t to a fr nt ssag .

v r cat on op rat on ta s a pub c y, a ssag , an a strng t at sca to
 b a s gnatur on t ssag , an r turns tru or a s . I t r turns tru , t at ans t at
 t t r nput va u s atc , an t v r r can b sur t at so on w nt t roug
 w t t s gnatur op rat on on t at v ry ssag , an t at t “so on ” a so nows t
 pr vat y corr spon ng to t pub c y.

s r prop rt s o a g ta s gnatu r a g o r t ar as o ows G v n t pub c
 y an pa rs o ssag s an va s gnatu r s on t , t s ou b ar to co put
 t pr vat y, an t s ou a so b ar to cr at a n w ssag an s gnatu r t at s
 acc pt by t v r cat on op rat on.

Besides signing an unsigned tag, signatures can be used for authentication. A server can be configured with a public key, such that any content that connects to the service is given a random nonce signed. If the server gets a reply with a correct signature, it can be confident that the connection is legitimate. The configuration of the public key, the content's grant access. The configuration of the server can be understood as "grant access to whoever knows the private key corresponding to the particular public key, and to no others".

7.7.1 RSA

The RSA algorithm was the first practical digital signature algorithm that was constructed. It was described in a paper by Rivest, Shamir, and Adleman, and the technique was a so-called “trapdoor” function [1]. The patent expired in 1996, and since that day, RSA can be used freely, even in the US.

It’s rare to see a paper that describes a trapdoor function based on RSA. “oneway”-function is

$$F(x) = x^e \bmod n$$

Let’s take $x=2$ as an example, with $e=3$ and $n=65537$. The output is 8 , although the argument is 2 . The output is 8 , although the argument is 2 .

In this argument, the property is that F is a permutation on the set of roots of unity. But, with the trapdoor

Let’s first look at the RSA algorithm. First, choose two primes p and q (so $n = pq$), and compute $\phi = (p-1)(q-1)$, the totient of n . Then, choose e such that $\gcd(e, \phi) = 1$. The output is $y = x^e \bmod n$.

Next, choose d such that $ed \equiv 1 \pmod{\phi}$ (particular, d is the inverse of e modulo ϕ), but can’t be chosen for $e=65537$ as a popular choice, because it is a small prime. The output is $y = x^e \bmod n$, and the input is x .

Finally, a number d , $d < n$ is computed such that $ed \equiv 1 \pmod{\phi}$. It can be shown that such a number exists (the same way as ϕ is the inverse of e modulo ϕ), and that for

$$(x^e)^d \bmod n = x^{ed} \bmod n = (x^d)^e \bmod n = x$$

using Euler’s theorem, d can be computed quite easily modulo ϕ . But this is a trapdoor, with n as the trapdoor.

To do the trapdoor, we now have $y = F(x)$, we can recover x as $y^d \bmod n$. d is a so-called private key.

Most common signature operations for RSA are in PKCS#1, a specification by the RSA Laboratories. The signature is first computed using a cryptographic hash function, e.g. SHA-1. Next, the padding is added. The algorithm is then applied to the padded message, and the signature is concatenated and converted to a number x . The signature is computed as $s = x^d \bmod n$. The signature, s , is a number about the same size as n , and is usually encoded as a sequence of octets, often as a sequence of octets.

Verification of the signature is straightforward, x is computed from s as above. Then $s^e \bmod n$ is computed, and the operation returns true if the result equals x .

The RSA algorithm can also be used for encryption. The RSA encryption uses the public key (n, e) to compute the ciphertext $m^e \bmod n$. PKCS#1 padding scheme was used

¹ Actually, the computation is not done like this, it is done more efficiently using p , q and the Chinese remainder theorem (CRT). But the result is the same.

ast ran o an non- ro oct ts, us ng *m o t* or [00 02 padding 00 plaintext]. It s r qu r t at $m < n$, an t r or t p ant xt ust b s a r t ant oct ts o t o u o n, w t so arg n.

o crypt t ssag , on n s t pr vat y to co put $m = c^e \bmod n$ o ow by c c ng an r ov ng t pa ng.

7.7.1.1 Nettle's RSA support

tt r pr s nts **P**A ys us ng two structur s t at conta n arg nu b rs (o typ *mpz_t*).

rsa_public_key size *n e* [Cont xt struct]
size s t s , n oct ts, o t o u o, an s us nt rna y. n an e s t pub c y.

rsa_private_key size *d p q a b c* [Cont xt struct]
size s t s , n oct ts, o t o u o, an s us nt rna y. d s t s cr t xpon nt, but t s not actua y us w n s gn ng. Inst a , t actors p an q, an t para t rs a, b an c ar us . y ar co put ro p, q an e suc t at a e mod $(p - 1) = 1$, b e mod $(q - 1) = 1$, c q mod $p = 1$.

B or us , t s structs ust b n t a by ca ng on o

*void rsa_public_key_init (struct rsa_public_key *pub)* [Funct on]
*void rsa_private_key_init (struct rsa_private_key *key)* [Funct on]
Ca s *mpz_init* on a nu b rs n t y struct.

an w n n s w t t , t spac or t nu b rs ust b a ocat by ca ng on o

*void rsa_public_key_clear (struct rsa_public_key *pub)* [Funct on]
*void rsa_private_key_clear (struct rsa_private_key *key)* [Funct on]
Ca s *mpz_clear* on a nu b rs n t y struct.

In g n ra , tt 's **P**A unct ons v at s ro tt 's "no ory a ocat on"-po cy. pac or a t nu b rs, bot n t y structs abov , an t porar s, ar a ocat yna ca y. For n or at on on ow to custo a ocat on, s ct on "G **P**A ocat on" n *GMP Manual*.

✦ n you av ass gn va u s to t attr but s o a y, you ust ca

*int rsa_public_key_prepare (struct rsa_public_key *pub)* [Funct on]
*int rsa_private_key_prepare (struct rsa_private_key *key)* [Funct on]
Co put s t oct ts o t y (stor nt size attr but , an ay a so o ot r bas c san ty c c s. **P** turns on succ ss u, or ro t y can't b us , or nstanc t o u o s s a r t ant n u s n or **P**A op rat ons sp c by **PKC**# .

For ac op rat on us ng t pr vat y, t r ar two var ants, .g., *rsa_sha256_sign* an *rsa_sha256_sign_tr*. or r unct on s o r, an t s ou b avo , b caus t prov s no ns s aga nst s -c ann attac s. attr unct on us ran o **P**A b n ng, w c n s aga nst t ng attac s us ng c os n-c p rt xt, an t a so

correctness of private computation using public key, which is against software or hardware errors which could affect privacy.

For signing or verifying a message, you first select the appropriate algorithm. You pass the algorithm's context structure to the `PKCS_Signature` function, and then extract the message and output it.

are 90% of the total (y) 0.5% of the total (t) 0.5% of the total (n) 0.5% of the total (r) 0.5% of the total (o) 0.5% of the total (s) 0.3% of the total (t) 0.5%

```

int rsa_pkcs1_sign_tr(const struct rsa_public_key *pub, const [Funct on]
    struct rsa_private_key *key, void *random_ctx, nettle_random_func
    *random, size_t length, const uint8_t *digest_info, mpz_t signature)
    /*
    *   ar to t abov _sign_digest_tr unct ons, but t nput s not t p an as
    *   g st, but a PKCS# "DigestInfo", an A. DEP- nco ng o t g st tog t r
    *   w t an ob ct nt r or t us as a gort .
    */

int rsa_md5_sign (const struct rsa_private_key *key, struct md5_ctx [Funct on]
    *hash, mpz_t signature)
int rsa_sha1_sign (const struct rsa_private_key *key, struct sha1_ctx [Funct on]
    *hash, mpz_t signature)
int rsa_sha256_sign (const struct rsa_private_key *key, struct [Funct on]
    sha256_ctx *hash, mpz_t signature)
int rsa_sha512_sign (const struct rsa_private_key *key, struct [Funct on]
    sha512_ctx *hash, mpz_t signature)
    s gnatur s stor n signature (w c ust av b n mpz_init' ar r).
    as cont xt s r s t so t at t can b us or n w ssag s. P turns on on succ ss,
    or ro on a ur . gn ng a s t y stoo s a or t g v n as s , g., t's
    not poss b to cr at a s gnatur us ng HA5 an a 5 -b t P A y.

int rsa_md5_sign_digest (const struct rsa_private_key *key, const [Funct on]
    uint8_t *digest, mpz_t signature)
int rsa_sha1_sign_digest (const struct rsa_private_key *key, const [Funct on]
    uint8_t *digest, mpz_t signature);
int rsa_sha256_sign_digest (const struct rsa_private_key *key, const [Funct on]
    uint8_t *digest, mpz_t signature);
int rsa_sha512_sign_digest (const struct rsa_private_key *key, const [Funct on]
    uint8_t *digest, mpz_t signature);
    Cr at s a s gnatur ro t g v n as g st, ot rws ana oguous to t abov
    s gn ng unct ons. digest s ou po nt to a g st o s MD5_DIGEST_SIZE, SHA1_
    DIGEST_SIZE, SHA256_DIGEST_SIZE, or SHA512_DIGEST_SIZE, r sp ct v y. s g-
    natur s stor n signature (w c ust av b n mpz_init ar r). P turns
    on on succ ss, or ro on a ur .

int rsa_pkcs1_sign(const struct rsa_private_key *key, size_t [Funct on]
    length, const uint8_t *digest_info, mpz_t s)
    /*
    *   ar to t abov _s gn_ g st unct ons, but t nput s not t p an as g st,
    *   but a PKCS# "DigestInfo", an A. DEP- nco ng o t g st tog t r w t an
    *   ob ct nt r or t us as a gort .
    */

    r y ng an P A s gnatur s on w t on o t o ow ng unct ons

int rsa_md5_verify (const struct rsa_public_key *key, struct md5_ctx [Funct on]
    *hash, const mpz_t signature)
int rsa_sha1_verify (const struct rsa_public_key *key, struct sha1_ctx [Funct on]
    *hash, const mpz_t signature)
int rsa_sha256_verify (const struct rsa_public_key *key, struct [Funct on]
    sha256_ctx *hash, const mpz_t signature)
    
```

```
int rsa_sha512_verify (const struct rsa_public_key *key, struct sha512_ctx *hash, const mpz_t signature) [Function]
```

P turns the signature value, or `signature`, into a boolean value. In the case, it is a constant value that can be used or not as a signature.

```
int rsa_md5_verify_digest (const struct rsa_public_key *key, const uint8_t *digest, const mpz_t signature) [Function]
```

```
int rsa_sha1_verify_digest (const struct rsa_public_key *key, const uint8_t *digest, const mpz_t signature) [Function]
```

```
int rsa_sha256_verify_digest (const struct rsa_public_key *key, const uint8_t *digest, const mpz_t signature) [Function]
```

```
int rsa_sha512_verify_digest (const struct rsa_public_key *key, const uint8_t *digest, const mpz_t signature) [Function]
```

P turns the signature value, or `signature`, into a boolean value. `digest` is a pointer to a digest of size MD5_DIGEST_SIZE.

SHA384_DIGEST_SIZE, or SHA512_DIGEST_SIZE respectively. The signature is stored in *signature* (which must have been initialized by `mpz_init`). `P` turns on on success, or returns an error.

Returning an `P` signature with the `P` padding scheme is on with on the following functions

```
int rsa_pss_sha256_verify_digest (const struct rsa_public_key [Funct on]
    *key, size_t salt_length, const uint8_t *digest, const mpz_t signature)
int rsa_pss_sha384_verify_digest (const struct rsa_public_key [Funct on]
    *key, size_t salt_length, const uint8_t *digest, const mpz_t signature)
int rsa_pss_sha512_verify_digest (const struct rsa_public_key [Funct on]
    *key, size_t salt_length, const uint8_t *digest, const mpz_t signature)
P turns on the signature value, or returns an error. digest should point to a digest of
SHA256_DIGEST_SIZE, SHA384_DIGEST_SIZE, or SHA512_DIGEST_SIZE respectively.
```

Following functions use the `nettle` library to encrypt a ciphertext using `P`.

```
int rsa_encrypt (const struct rsa_public_key *key, void *random_ctx, [Funct on]
    nettle_random_func *random, size_t length, const uint8_t *cleartext,
    mpz_t ciphertext)
P turns on success, or returns an error. If the signature is too long to fit in the
array.
```

Following functions use the `nettle` library to decrypt a ciphertext using `P`.

```
int rsa_decrypt (const struct rsa_private_key *key, size_t *length, [Funct on]
    uint8_t *cleartext, const mpz_t ciphertext)
P turns on success, or returns an error. Causes an error if the ciphertext is
not a valid signature. The signature is too long to fit in the
array. At the end of the ciphertext, the length will be updated with the
signature.
```

It is also a good idea to use the `nettle` library to decrypt a ciphertext using `P`
before using it.

```
int rsa_decrypt_tr (const struct rsa_public_key *pub, const struct [Funct on]
    rsa_private_key *key, void *random_ctx, nettle_random_func *random, size_t
    *length, uint8_t *message, const mpz_t ciphertext)
P turns on success, or returns an error.
```

If you need to use the `P` trapdoor, the private key, in a way that isn't supported by
the above functions, then you can use a function that computes $x^d \bmod n$ and not
necessarily using the `C` option.

```
int rsa_compute_root_tr (const struct rsa_public_key *pub, const [Funct on]
    struct rsa_private_key *key, void *random_ctx, nettle_random_func
    *random, mpz_t x, const mpz_t m)
Compute  $x = m^d$ . P turns on on success, or returns an error if the computation
was not successful.
```

```
void rsa_compute_root (struct rsa_private_key *key, mpz_t x, const [Function]
                        mpz_t m)
    Compute  $x = m^d$ .
```

At last, now you can calculate y s

```
int rsa_generate_keypair (struct rsa_public_key *pub, struct [Function]
                          rsa_private_key *key, void *random_ctx, nettle_random_func random, void
                          *progress_ctx, nettle_progress_func progress, unsigned n_size, unsigned
                          e_size);
```

random_ctx is a parameter. pub and key are the resulting pair of structures, but you don't need to call rsa_public_key_prepare or rsa_private_key_prepare at any time.

random_ctx and random are random number generators. random(random_ctx, length, dst) will generate length random octets and store them at dst. For a vector, s is a pointer to a [Function], page 9.

progress and progress_ctx can be used to get callbacks during the generation process, in order to update an ongoing progress. progress can be NULL, in that case the library will not call back.

size_n is the size of the output, in bits. I size_e is non-zero, then the size of the public exponent and the random exponent of the secret. But size_e is zero, then the size of the public exponent is a variable, and store it in pub. P turns on on success, and returns a value. unct can be a pointer to a function, or size_e is zero and pub->e is a variable.

7.7.2 DSA

DSA is a signature algorithm for computing a signature. It was specified in RFC 1321, and in 1991 IETF published FIPS 186-2, which is the standard for DSA. The acronym DSA, or Digital Signature Algorithm, is used to refer to the specification, FIPS 186-2, was issued in 1999, and it also supports the algorithm as a function in the library.

For DSA, the underlying mathematical computation is the discrete logarithm problem. The public key consists of a prime p , a primitive root g of $p-1$, a number x which generates a subgroup of order q of p , and an element y of the subgroup.

In the original DSA, the size of x is 160 bits, to match with the SHA-1 as a generator. The size of p is 160 bits, but the standard specifies only 160 bits, $512 + 1 \cdot 64$, where 1 is the number of bits, and 64 is the number of bits, and the standard is obsolete and not current.

subgroup requires that you compute

$$g^t \bmod p$$

or a possible generator, you will get a primitive root value.

private key is a secret exponent x , such that

$$g^x = y \bmod p$$

In the standard specification, x is the discrete logarithm of y to g modulo p , with respect to the generator g . The size of x is also about the same as q . The security of DSA

algorithm runs on the security of the secret logarithm problem. Current algorithms to compute secret logarithms in the setting, and hence crack DSA, are of two types.

The first type works by finding a (multiplicative) subgroup of $\mathbb{Z}_p^*$ of size $p-1$. The best known algorithm to find such a subgroup is the Pollard rho algorithm, which is a probabilistic algorithm. The second type works by finding a subgroup of $\mathbb{Z}_p^*$ of size $p-1$. The best known algorithm to find such a subgroup is the Pollard rho algorithm, which is a probabilistic algorithm.

Important point is that security depends on the size of both p and q , and thus one must choose p and q such that $p-1$ and $q-1$ are both large. The standard way to choose p and q is to choose p and q such that $p-1$ and $q-1$ are both large. The standard way to choose p and q is to choose p and q such that $p-1$ and $q-1$ are both large.

The signature generation algorithm runs on the secret logarithm, and thus to create a DSA signature, you need a good source of random numbers (see Section 1.2.1 [Pan et al., 1996]). Let us describe the process of creating a signature.

To create a signature, one starts with the message m , which is a bit string, and a random number k , $0 < k < q$, as a seed. Then, one computes

$$r = (g^k \bmod p) \bmod q$$

$$s = k^{-1} (h + x r) \bmod q$$

The signature is the pair (r, s) , two bit strings. Note that two different messages will produce the same r , and thus the signature is not unique.

To verify a signature, one first checks that $0 < r, s < q$, and then one computes backwards,

$$w = s^{-1} \bmod q$$

$$v = (g^{w h} y^{w r} \bmod p) \bmod q$$

The signature is valid if $v = r$. This works out because $w = s^{-1} \bmod q = k^{-1} (h + x r)^{-1} \bmod q$, so that

$$g^{w h} y^{w r} = g^{w h} (g^x)^{w r} = g^{w (h + x r)} = g^k$$

and thus the signature is valid. Note that when verifying a signature, we don't need to know the random number k used to generate the signature.

If you can choose both p and q , then DSA is secure. But in practice, p is chosen first, and then q is chosen. The standard way to choose p and q is to choose p and q such that $p-1$ and $q-1$ are both large. The standard way to choose p and q is to choose p and q such that $p-1$ and $q-1$ are both large.

DSA signatures are a sort of asymmetric signatures, which are so important for so many applications.

From a practical point of view, DSA's are either a good random source as a source of advantage. If you verify the signature (and/or) or two different signatures, you are your private key.

7.7.2.1 Nettle's DSA support

Like for RSA, the library provides DSA by using two structures, containing various types of mpz_t. For information on how to customize a location, see the section "GMP A location" in *GMP Manual*. The library's DSA interface is defined in `<nettle/dsa.h>`.

A DSA group is represented using the following structure.

`dsa_params` *p q g* [Content struct]
 Parameters of the DSA group.

`void dsa_params_init (struct dsa_params *params)` [Function]
 Call `mpz_init` on the number `n` in the structure.

`void dsa_params_clear (struct dsa_params *params)` [Function]
 Call `mpz_clear` on the number `n` in the structure.

`int dsa_generate_params (struct dsa_params *params, void [Function]
 *random_ctx, nettle_random_func *random, void *progress_ctx,
 nettle_progress_func *progress, unsigned p_bits, unsigned q_bits)`
 Generate parameters for a new group. The `params` structure is built and/or you call the function.

`random_ctx` and `random` is a random source generator. `random(random_ctx, length, dst)` is a function that generates `length` random octets and stores them at `dst`. For a value, see the section "Randomness", page 9.

`progress` and `progress_ctx` can be used to get callbacks during the generation process, in order to update an ongoing progress. `progress` can be NULL, in that case the callback is not called.

`p_bits` and `q_bits` are the sizes of `p` and `q`. The generation is that conform to the original DSA standard, you must use `q_bits = 160` and select `p_bits` of the form `p_bits = 512 + l*64`, where `0 <= l <= 8`, with the same size as the original, so you should choose the value of `p_bits = 1024`. The standard is also possible, in particular `p_bits` argument and, although DSA parameters can not be generated to support such values. A side note that using very large `p_bits`, with `q_bits` at 160, doesn't make sense, because the security is so high by the size of the prime. The generation of DSA keys or values with SHA-1, using `q_bits = 256` and, e.g., `p_bits = 2048`.

The library turns on on success, and returns a value. The function will return `q_bits` is too small, or too close to `p_bits`.

Signatures are represented using the structure below.

`dsa_signature` *rs* [Cont xt struct]

`void dsa_signature_init (struct dsa_signature *signature)` [Func on]

`void dsa_signature_clear (struct dsa_signature *signature)` [Func on]

You ust ca `dsa_signature_init` b or cr at ng or us ng a s gnatur , an ca `dsa_signature_clear` w n you ar n s w t t.

K ys ar r pr s nt as b gnu s, o typ `mpz_t`. A pub c y r pr s nts a group nt, an s o t sa s as p, w a pr vat y s an xpon nt, o t sa s as q.

`int dsa_sign (const struct dsa_params *params, const mpz_t x, void` [Func on]

`*random_ctx, nettle_random_func *random, size_t digest_size, const`

`uint8_t *digest, struct dsa_signature *signature)`

Cr at s a s gnatur ro t g v n as g st, us ng t pr vat y x. `random_ctx` an `random` s a ran o n ss g n rator. `random(random_ctx, length, dst)` s ou g n rat `length` ran o oct ts an stor t at `dst`. For a v c , s ct on . [Pan o n ss], pag 9. `P` turns on on succ ss, or ro on a ur . gn ng can a on y t y s nva , so t at nv rs on o u o q a s.

`int dsa_verify (const struct dsa_params *params, const mpz_t y,` [Func on]

`size_t digest_size, const uint8_t *digest, const struct dsa_signature`

`*signature)`

r s a s gnatur , us ng t pub c y y. `P` turns t s gnatur s va , ot rws .

o g n rat a ypar, rst g n rat a DSA group us ng `dsa_generate_params`. A ypar nt s group st n cr at us ng

`void dsa_generate_keypair (const struct dsa_params *params, mpz_t` [Func on]

`pub, mpz_t key, void *random_ctx, nettle_random_func *random)`

G n rat s a n w ypar, us ng t group `params`. pub c y s stor n `pub`, an t pr vat y n `key`. Bot var ab s ust b n ta us ng `mpz_init` b or t s ca .

`random_ctx` an `random` s a ran o n ss g n rator. `random(random_ctx, length, dst)` s ou g n rat `length` ran o oct ts an stor t at `dst`. For a v c , s ct on . [Pan o n ss], pag 9.

7.7.2.2 Old, deprecated, DSA interface

rs ons b or n tt -3. us a fr nt nt rac or DSA s gnatur s, w r t group para t rs an t pub c y was pac tog t r as `struct dsa_public_key`. ost o t s nt rac s pt or bac war s co pat b ty, an car n 'nettle/dsa-compat.h'. B ow st o ocu ntaton. o an n w nt rac us st nct na s an on't con ct, w t on xc pt on y g n rat on unct on. 'nettle/dsa-compat.h' r -

n s `dsa_generate_keypair` as an a as or `dsa_compat_generate_keypair`, co pat b w t t o nt rac an ocu nt b ow.

o DSA unct ons ar v ry s ar to t corr spon ng PSA unct ons, but t r ar a w fr nc s po nt out b ow. For a start, t r ar no unct ons corr spon ng to `rsa_public_key_prepare` an `rsa_private_key_prepare`.

`dsa_public_key` $p q g y$ [Cont xt struct]
 pub c para t rs scr b abov .

`dsa_private_key` x [Cont xt struct]
 pr vat y x .

Below, these structs are built by calling

`void dsa_public_key_init (struct dsa_public_key *pub)` [Function]

`void dsa_private_key_init (struct dsa_private_key *key)` [Function]

Call `mpz_init` on `a` and `b` first.

Then, to set the space for the buffers, call

`void dsa_public_key_clear (struct dsa_public_key *pub)` [Function]

`void dsa_private_key_clear (struct dsa_private_key *key)` [Function]

Call `mpz_clear` on `a` and `b` first.

Signature is represented using `struct dsa_signature`, which is

For signing, you need to provide both the public and private y (using P , where the private y is the structure used for signing), and a source random number. Signature can use `SHA` or `SHA256` as function, although the point is that `DSA` with `SHA256` is somewhat experimental to achieve a set of vectors and not a random type of signing.

`int dsa_sha1_sign (const struct dsa_public_key *pub, const struct dsa_private_key *key, void *random_ctx, nettle_random_func random, struct sha1_ctx *hash, struct dsa_signature *signature)` [Function]

`int dsa_sha1_sign_digest (const struct dsa_public_key *pub, const struct dsa_private_key *key, void *random_ctx, nettle_random_func random, const uint8_t *digest, struct dsa_signature *signature)` [Function]

`int dsa_sha256_sign (const struct dsa_public_key *pub, const struct dsa_private_key *key, void *random_ctx, nettle_random_func random, struct sha256_ctx *hash, struct dsa_signature *signature)` [Function]

`int dsa_sha256_sign_digest (const struct dsa_public_key *pub, const struct dsa_private_key *key, void *random_ctx, nettle_random_func random, const uint8_t *digest, struct dsa_signature *signature)` [Function]

Creating a signature is given as context or `random_ctx` and `random` is a random number generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. For a vector, see [Pan on ss], page 9. P turns on on success, or returns a signature as a set of bytes and as a set of octets.

Returning signature is a set of bytes, since no random number generator is used.

`int dsa_sha1_verify (const struct dsa_public_key *key, struct sha1_ctx *hash, const struct dsa_signature *signature)` [Function]

`int dsa_sha1_verify_digest (const struct dsa_public_key *key, const uint8_t *digest, const struct dsa_signature *signature)` [Function]

```
int dsa_sha256_verify (const struct dsa_public_key *key, struct sha256_ctx *hash, const struct dsa_signature *signature) [Funct on]
int dsa_sha256_verify_digest (const struct dsa_public_key *key, const uint8_t *digest, const struct dsa_signature *signature) [Funct on]
    r s a s g n a t u r . P t u r n s t s g n a t u r s v a , o t r w s .
```

Key generation uses a fast pseudo-random number generator.

```
int dsa_compat_generate_keypair (struct dsa_public_key *pub, struct dsa_private_key *key, void *random_ctx, nettle_random_func random, void *progress_ctx, nettle_progress_func progress, unsigned p_bits, unsigned q_bits) [Funct on]
    pub a n k e y s w r t r s u t n g y p a r s s t o r . s t r u c t s s o u b n t a
    b o r y o u c a t s u n c t o n .
```

random_ctx and *random* is a random number generator. *random(random_ctx, length, dst)* stores *length* random octets into *dst*. For a vector, *s* is a *nettle* object. [P. 9].

progress and *progress_ctx* can be used to get callbacks during key generation process, normally to update progress. *progress* can be *NULL*, not at all, not a callback.

p_bits and *q_bits* are the size of *p* and *q*. *dsa_generate_keypair* or *dsa_generate_keypair_compat*.

7.7.3 Elliptic curves

For cryptographic purposes, an elliptic curve is a set of points, an abelian group. For a given elliptic curve, the set of points is a group. If *P* and *Q* are two points, and *k* is an integer, the point *kP*, *P + Q*, and the point *kP* can be computed. The point *kP* is the point *P* added to itself *k* times. The point *kP* is the point *P* added to itself *k* times.

It supports standard curves with equation $y^2 = x^3 - 3x + b \pmod{p}$, where *p* is a prime. The points are coordinates (x, y) , both considered modulo *p*. Curves are represented as a *struct ecc_curve*. It also supports curves with equation $y^2 = x^3 + ax + b \pmod{p}$, where *p* is a prime. Support curves are defined in `<nettle/ecc-curve.h>`, e.g., `nettle_get_secp_256r1` or a standard curve using the *secp256r1* parameters. The contents of the structure *ecc_curve* are defined in `nettle/ecc-curve.h`. The "bits of curve" is used as a sort of identifier for the curve, e.g., *secp256r1* or *ECDSA* curve.

7.7.3.1 Side-channel silence

It's important to note that elliptic curve operations are not side-channel resistant. Side-channel attacks can be performed.

- Using attacks like timing or power analysis, an attacker can learn information about your secrets, e.g., a private signature key.

struct ecc_scalar [struct]
 $\mathbb{P}$ presents an integer ranging $0 \leq x < \text{grouporder}$, with the “group order” referring to the order of an ECC group. In particular, this structure represents an ECDH private key.

void ecc_scalar_init (*struct ecc_scalar *s, const struct ecc_curve *ecc*) [Function]
 Initializes the scalar structure *s* for the given curve *ecc*. Allocates storage using the same allocation functions as $\mathbb{P}$.

void ecc_scalar_clear (*struct ecc_scalar *s*) [Function]
 Deallocates storage.

int ecc_scalar_set (*struct ecc_scalar *s, const mpz_t z*) [Function]
 Converts the integer *z* to the correct range. If so, copies the value to *s* and returns 1, otherwise returns 0.

void ecc_scalar_get (*const struct ecc_scalar *s, mpz_t z*) [Function]
 Extracts the scalar, *n* $\mathbb{P}$ -*mpz_t* representation.

For each and every ECDH signatures, the following functions are used.

void ecdsa_sign (*const struct ecc_scalar *key, void *random_ctx, nettle_random_func *random, size_t digest_length, const uint8_t *digest, struct dsa_signature *signature*) [Function]
 Signs the private key to create a signature on *digest*. *random_ctx* and *random* is a random number generator. *random(random_ctx, length, dst)* outputs *length* random octets and stores them at *dst*. The signature is stored in *signature*, and *signature* was allocated by DSA .

int ecdsa_verify (*const struct ecc_point *pub, size_t length, const uint8_t *digest, const struct dsa_signature *signature*) [Function]
 Signs the public key *pub* to verify that *signature* is a valid signature for the message *digest* (of *length* octets). $\mathbb{P}$ returns 1 if signature is valid, otherwise returns 0.

Finally, generating a new ECDH key pair

void ecdsa_generate_keypair (*struct ecc_point *pub, struct ecc_scalar *key, void *random_ctx, nettle_random_func *random*); [Function]
pub and *key* will be returned by *pair* structure. *random_ctx* and *random* is a random number generator. *random(random_ctx, length, dst)* outputs *length* random octets and stores them at *dst*. For a *vec*, *s* $\mathbb{P}$ *vec* on $\mathbb{P}$ [an *on* ss], pag 9.

7.7.3.3 GOSTDSA

GOSTDSA ($\text{GOST} \mathbb{P} 310$, $\text{GOST} \mathbb{P} 310$) is a variant of DSA (see [DHA], pag 3) and ECDH (see [ECDH], pag 3) signature scheme, with a different elliptic curve group. GOST algorithms are written in

These signing operations are provided in *RFC 5832* and *RFC 7091*. The implementation does not stop on choosing GOST DSA over any curve, it is only over $\mathbb{F}_p$ and $\mathbb{F}_q$ -based curves. Like DSA and ECDSA, creating a signature requires a unique random nonce (repeating the nonce with two different messages results in a private key, and any other base point generation of the nonce also results in a non-repeating nonce).

GOST $\mathbb{F}_3$ - was used to us GOST $\mathbb{F}_3$ - 9 as a function (GOST $\mathbb{H}_9$ and GOST $\mathbb{H}_9$ CP; *RFC 5831*). GOST $\mathbb{F}_3$ - is used to us GOST $\mathbb{F}_3$ - as a function (Trilog, *RFC 6986*) of corresponding (5 or 5) point on curves.

It is GOST DSA in '`<nettle/gostdsa.h>`'. GOST DSA uses ECDSA data types (`struct ecc_point`, `struct ecc_scalar`) to represent public and private keys. As to generate a new GOST DSA key pair as to us `ecdsa_generate_keypair()` function. The created private key GOST DSA signatures, the following functions are used.

```
void gostdsa_sign (const struct ecc_scalar *key, void *random_ctx, [Function]
    nettle_random_func *random, size_t digest_length, const uint8_t *digest,
    struct dsa_signature *signature)
    sign private key to create a signature on digest. random_ctx and random is a
    random number generator. random(random_ctx, length, dst) outputs length
    random octets and stores it at dst. signature stores signature, not
    signature was as a pair of DSA.
```

```
int gostdsa_verify (const struct ecc_point *pub, size_t length, const [Function]
    uint8_t *digest, const struct dsa_signature *signature)
    sign public key pub to verify that signature is a valid signature or not
    sign digest (length octets). It turns the signature value, otherwise.
```

For storage reasons over curve IDs (OIDs) are corresponding to a sign curve generator combination. Following standards correspond to each other in the 'view on curves' and actual names in *RFC 4357* and *RFC 7836*.

```
const struct ecc_curve nettle_get_gost_gc256b(void) [Function]
    It turns curve corresponding to following names
    • -Gost $\mathbb{F}_3$  - -CryptoPro-A-Para  $\mathbb{F}_3$  (RFC 4357)
    • -Gost $\mathbb{F}_3$  - -CryptoPro-Xc-A-Para  $\mathbb{F}_3$  (RFC 4357)
    • -tc  $\mathbb{F}_3$ -gost-3 - - $\mathbb{F}_3$ -para  $\mathbb{F}_3$ tB
```

```
const struct ecc_curve nettle_get_gost_gc512a(void) [Function]
    It turns curve corresponding to following names
    • -tc  $\mathbb{F}_3$ -gost-3 - - $\mathbb{F}_3$ -para  $\mathbb{F}_3$ tA (RFC 7836)
```

For GOST key pairs generation algorithm function (`K $\mathbb{F}_3$`) is used in *RFC 4357* and *RFC 7836*. Basically are the same as (cofactor $\mathbb{F}_3$ private public). The library provides a function that outputs a pair of public and private keys (two base point GOST $\mathbb{F}_3$ - 9) (`[GOST  $\mathbb{H}_9$  CP]`, page 9) or GOST $\mathbb{F}_3$ - 3, Trilog, with no part of the library (y).

```
void gostdsa_vko (const struct ecc_scalar *priv, const struct ecc_point [Funct on]
                  *pub, size_t ukm_length, const uint8_t *ukm, uint8_t *out)
    s s p r v a t y p r i v , p u b c t p u b a n s a r y a t r a u k m t o g n r a t s a r
    s c r t , w r t t n t o b u f f e r o u t . b u f f e r s o u b o t s q u a t o p r v a t y
    n g t s 6 b y t s o r 56 b t c u r v s a n b y t s o r 5 b t o n s . K s a s a r
    y a t r a , u s u a y t r a n s r r n c a r t x t . I t o s n o t a v t o b s c r t .
```

7.7.3.4 Curve25519 and Curve448

Curve 25519 is an elliptic curve of Montgomery type, $y^2 = x^3 + 6x^2 + x - 9$, with $p = 2^{255} - 19$. Montgomery curves have a advantage of a point at infinity on the x-coordinate. The special curve was proposed by D. J. Bernstein in 2006, or as D. J. Bernstein and Y. Zeng, and also described in RFC 7748. The group generator is G by $x = 9$ (the actual two points with $x = 9$, forming by the sign of the y-coordinate, but that doesn't affect the curve 25519 operations with the x-coordinate).

Curve 25519 operations are done as operations on octet strings, representing 255-bit scalars or x-coordinates, in little-endian byte order. The most significant input bit, ..., the most significant bit of the octet, is always ignored.

For scalars, namely on the last significant bits are ignored, and treated as zero, and the most significant bits are ignored too, and treated as one. The scalar input string always represents the same number in the range $2^{254} < s < 2^{255}$.

On all possible input strings, on the about a correspondence to x-coordinates of points on curve 25519, ..., a value x or w of the curve equation can be solved for y . The other a correspondence to points on the other "twisted curve". The function `curve25519_mul` uses a Montgomery arithmetic scalar multiplication, as suggested in the curve 25519 literature, and required by RFC 7748. The output string or word is for all possible inputs, no matter the input string represents a valid point on the curve or not.

On the other curve 25519 point at infinity is not a valid point. The value s is given by RFC 7748, not at bit 255 of the x-coordinate of the point input to curve 25519. It was not ignored. The file 'nettle/curve25519.h' contains a preprocessor symbol `NETTLE_CURVE25519_RFC7748` to indicate the correct behavior with the standard.

The name of Curve 25519 is 'nettle/curve25519.h'.

`NETTLE_CURVE25519_RFC7748` [Constant]

Defined to the value of the constant `NETTLE_CURVE25519_RFC7748`. The name of the constant.

`CURVE25519_SIZE` [Constant]

The size of the strings representing curve 25519 points and scalars, 32.

`void curve25519_mul_g (uint8_t *q, const uint8_t *n)` [Function]

Computes $q = nG$, where G is the group generator and N is an integer. The input argument n and the output argument q are both 32-byte strings of size `CURVE25519_SIZE`.

The function is intended to be compatible with the function `crypto_scalar_mult_base` in the library.

```
void curve25519_mul (uint8_t *q, const uint8_t *n, const uint8_t *p) [Function]
    Computes  $Q = nP$ , where  $P$  is an input point on  $N$  and  $n$  is an input
    argument.  $n$  is an integer,  $p$  is a point on the curve, and  $q$  is a
    point on the curve. The output is a point  $q$  on the curve. The
    function is implemented to be compatible with the function crypto_scalar_mult
    in the libcrypto library.
```

Curves, the so-called *Curve*, are elliptic curves of the form $y^2 = x^3 + ax + b$ (mod p), where p is a prime. The first curve was proposed by Haber and Damgård [1], and is now known as *Curve25519*. It is defined by the equation $y^2 = x^3 + 486662x + 486662$ (mod $2^{255} - 19$).

The function `curve25519_mul` is defined in `<nettle/curve25519.h>`.

```
CURVE448_SIZE [Constant]
    The size of the curve, in bytes, for the curve curve448.
```

```
void curve448_mul_g (uint8_t *q, const uint8_t *n) [Function]
    Computes  $Q = nG$ , where  $G$  is the generator point on  $N$  and  $n$  is an input
    argument.  $n$  is an integer,  $p$  is a point on the curve, and  $q$  is a
    point on the curve. The output is a point  $q$  on the curve. The
    function is implemented to be compatible with the function crypto_scalar_mult
    in the libcrypto library.
```

```
void curve448_mul (uint8_t *q, const uint8_t *n, const uint8_t *p) [Function]
    Computes  $Q = nP$ , where  $P$  is an input point on  $N$  and  $n$  is an input
    argument.  $n$  is an integer,  $p$  is a point on the curve, and  $q$  is a
    point on the curve. The output is a point  $q$  on the curve. The
    function is implemented to be compatible with the function crypto_scalar_mult
    in the libcrypto library.
```

7.7.3.5 EdDSA

EdDSA is a signature scheme proposed by D. J. Bernstein [1]. It is based on the use of a “twisted Edwards curve”, of the form $-x^2 + y^2 = dx^2y^2$. The signature scheme uses a curve with a constant d to create two groups using the same curve. The signature is computed using the coordinates of the point, so that the signature is a pair of coordinates.

The signature scheme is implemented in the `libcrypto` library. The function `eddsa_sign` is used to compute the signature of a message. The function `eddsa_verify` is used to verify the signature of a message. The function `eddsa_sign_det` is used to compute the signature of a message, with a deterministic nonce. The function `eddsa_verify_det` is used to verify the signature of a message, with a deterministic nonce. The function `eddsa_sign_det` is used to compute the signature of a message, with a deterministic nonce. The function `eddsa_verify_det` is used to verify the signature of a message, with a deterministic nonce.

`ED25519_KEY_SIZE` [Constant]
 size of a private or public E₂₅₅₁₉ key, 32 octets.

`ED25519_SIGNATURE_SIZE` [Constant]
 size of an E₂₅₅₁₉ signature, 64 octets.

`void ed25519_sha512_public_key (uint8_t *pub, const uint8_t *priv)` [Function]
 Compute the public key corresponding to the given private key. Both input and output are of size `ED25519_KEY_SIZE`.

`void ed25519_sha512_sign (const uint8_t *pub, const uint8_t *priv, size_t length, const uint8_t *msg, uint8_t *signature)` [Function]
 Sign a message using the provided private key.

`int ed25519_sha512_verify (const uint8_t *pub, size_t length, const uint8_t *msg, const uint8_t *signature)` [Function]
 Verify a message using the provided public key. $\mathcal{P}$ returns 1 if signature is valid, 0 otherwise.

The above provides E₂₅₅₁₉, an Edwards signature scheme based on an Edwards curve equivalent to Curve25519.

`ED448_KEY_SIZE` [Constant]
 size of a private or public E₄₄₈ key, 56 octets.

`ED448_SIGNATURE_SIZE` [Constant]
 size of an E₄₄₈ signature, 112 octets.

`void ed448_shake256_public_key (uint8_t *pub, const uint8_t *priv)` [Function]
 Compute the public key corresponding to the given private key. Both input and output are of size `ED448_KEY_SIZE`.

`void ed448_shake256_sign (const uint8_t *pub, const uint8_t *priv, size_t length, const uint8_t *msg, uint8_t *signature)` [Function]
 Sign a message using the provided private key.

`int ed448_shake256_verify (const uint8_t *pub, size_t length, const uint8_t *msg, const uint8_t *signature)` [Function]
 Verify a message using the provided public key. $\mathcal{P}$ returns 1 if signature is valid, 0 otherwise.

7.8 Randomness

A crucial ingredient in any cryptographic context is randomness. Let b be a random prime, c a random integer, a a random vector, v a random vector, k a random exponent, etc. In the following, it is assumed that you have plenty of randomness around. It is assumed that not true in practice, systems that are not truly secure, can be broken. $\mathcal{P}$ randomness is often turned out to be a waste of time.

In non-cryptographic applications, such as gathering statistics on a group of users, a group of users is usually assigned a group identifier that is good statistical properties, and is usually by some punctuation to the current time, process, and host name.

How ver, such a generator is not a quantum cryptograph, or at least two reasons

- It's too easy for an attacker to guess the plaintext. Even without the so-called "traps" or "guessing" steps, the attacker is too easy. For example, the process is so simple, the reason for "current time" is on the screen, and the attacker knows what the generator was, the attacker only has about 2^3 possibilities to try a possible value or the process and the -o- output.
- The generator output is too easy. By observing only a single output of the generator's output, the entire state can be recovered, and therefore, a previous output and a future output can be computed by the attacker.

A random string generator that is used for cryptographic purposes must have the following properties. Let's first look at the string, as the string is the most important part of the random generator. In a statistical random generator (the string) must be unbiased

by t attac r. o w at's ungu ssab It p n s on w at t attac r a r a y n ows.
 conc pt us n n or at on t ory to r ason about suc t ngs s ca " ntropy",
 or "con t on a ntropy" (not to b con us w t t t r o yna c conc pt w t t
 sa na). A r asonab r qu r nt s t at t s conta ns a con t on a ntropy o
 at ast so - b ts. s prop rty can b xp an as o ows A ow t attac r
 to as n y s-no-qu st ons, o s own c o c , about t s . I t attac r, us ng t s
 qu st on-an -answ r s ss on, as w as any ot r n or at on n ows about t s ng
 proc ss, st can't gu ss t s corr ct y, t n t con t on a ntropy s or t an n b ts.

Let's look at an example. Say you're not at all into two-particle systems and the processing. It's so random that the trajectory is going on, the system is not in equilibrium or "unstable" to the system. However, the attack can still be not a two-particle, or a but as a number of particles will be transferred by the system at the attack can be not, the system is not a stable system as for the attack to be out. Even though the system is exactly the same, the continuous entropy, or unstable, is a random attack that now is so far away from the hypothesis that it is not an answer to the question.

ngo goo g n rators s usua y bas on s v ra sourc s. y po nt r st at
t a ount o ungu ssab ty t at ac sourc contr but s, p n s on w o t attac r s.
o sourc st at av b n us ar

Highlighting important concepts

tuc as co p t b oc s ro sp n n g ar s s, n twor pac ts, tc. G t
 t ng acc ss to suc n or at on s qu t syst p n nt, an not a syst s
 nc u su tab ar war . I ava ab , t's on o t b tt r ran o n ss sourc
 on can n n a g ta , ost y pr ctab , co put r.

s r act v ty

ngan cont nts o us r nt ract on v nts sanot r popu ar sourc t at s
ava ab or nt ract v progra s (v n I susp ct t at t sso t s us n
or r to a t us r goo , not b caus t qua ty o t nput s n
or us prop r y). obv ous y, not ava ab w na ac n sunatt n . A so

Independence of outputs

observing so that outputs are not put attacker to guess previous or future output.

Forward secrecy

Even an attacker who knows the generator's y , is unable to guess the generator output *before* the y is chosen.

Recovery from key compromise

In an attack on the proposed generator's γ , an attacker can output *all* output bits of the generator sequentially on any one γ , at startup. However, the generator can provide a response to a receiver covering only a portion of the process. In the attack on the proposed γ at a particular t_1 , the receiver at t_2 , such that the attacker observes an output of the generator between t_1 and t_2 , still can't guess what output the generator will output at t_2 .

tt ncu s on ran o n ss g n rator t at s b v to av a t abov prop r-
t s, an two s p r on s.

APCF **0 P**, any string can be used as a random string generator. Its output is obviously of reasonable quality, it's as good as propably the same with **arcfour_setkey**. There's no single natural way to resist, but you're resisting, you should be using Yarrow instead.

“agg F bonacc” g n rator n ‘<nettle/knuth-lfib.h>’ s a ast g n rator wt
goo stat st ca prop rt s, but s **not** or cryptograp c us , an t r or not ocu nt
r . It s nc u ost y b caus t tt t st sut n s to g n rat so t st ata
ro a s a s .

reconnecting to us as Yarrow, scrub below.

7.8.1 Yarrow

Yarrow s a a y o p s u o-ran o n s s g n raters, s gn or cryptograp c us , by Jo n K s y, Bruc c n r an s F rguson. Yarrow- s scr b n a pap r at <http://www.counterpane.com/yarrow.html>, an t us s HA an tr p -DE, an as a -b t nt rna stat . tt p nts Yarrow- 56, w c s s ar, but us s HA 56 an AE to g t an nt rna stat o 56 b ts.

Yarrow was an a ost n s pro ct, t pap r nt on abov st cos st t ng to a sp c cat on or t, but so s a r ta s ar t out. r s no o ear r nc p ntat on or t st cas s. s s ct on nc u s an ov rv wo Yarrow, but or t ta s o Yarrow- 56, as p nt by tt , you av to consu t t sourc co . ayb a co p t sp c cat on can b wr tt n at r.

Yarrow can use any source s (at least two are necessary or possibly none), and two random seeds “pools”, referred to as the “slow pool” and the “fast pool”. Input random source s is alternately used in the two pools.  In one of the source s as contrast bits of entropy to the fast pool, a “fast s ” appears and the fast pool s is used in the random state.  In at least two of the source s as contrast at least  bits act to the slow pool, a “slow s ” takes place.  contents of both pools are  used in the random state.  processor s is  used in the generator with  virtual  coverage  at a  proportion.

output sign ratio by using AE to encrypt a counter, using the generator's current value. A trace ratio for output, another 56 bits are generated with respect to the source's security.

Yarrow can also use a *seed file* to save state across restarts. Yarrow is initialized by the following contents of the previous source, or the input roots source until a source is applied.

```
typedef struct Yarrow-56 { char '<nettle/yarrow.h>'.
```

```
struct yarrow256_ctx [Context struct]
```

```
struct yarrow_source [Context struct]
```

In order to obtain about a source.

```
YARROW256_SEED_FILE_SIZE [Constant]
```

```
typedef struct Yarrow-56 { char '<nettle/yarrow.h>'.
```

```
void yarrow256_init (struct yarrow256_ctx *ctx, unsigned nsources, [Function]
                    struct yarrow_source *sources)
```

In a state yarrow context, an nsources source. It's possible to call with nsources and sources. LL, you don't need to put it up at the source.

```
void yarrow256_seed (struct yarrow256_ctx *ctx, size_t length, [Function]
                    uint8_t *seed_file)
```

Yarrow-56 is a previous source. length is the number of bytes at least YARROW256_SEED_FILE_SIZE, but it can be larger.

generator will trust you that the seed file is a random source. A trace ratio function, you must overwrite the source with a new generator value at yarrow256_random. It's possible for a source process to store a state about the state, access the state by coordinate using so-called source.

```
int yarrow256_update (struct yarrow256_ctx *ctx, unsigned source, [Function]
                    unsigned entropy, size_t length, const uint8_t *data)
```

parameter generator will be a source SOURCE (an integer at the source and the number of source). entropy is your state of the source's entropy. The state, the source's bits. Calling up at the source entropy is always safe, no matter the state's random or not.

The turns are applied, new cases are applied using a source. want to generate new state with yarrow256_random and overwrite the state. The turns, the function turns.

```
void yarrow256_random (struct yarrow256_ctx *ctx, size_t length, [Function]
                    uint8_t *dst)
```

Generator's length of the output. generator will be the source or you call the function.

If you don't need the source's security, e.g. you need a non-secure random source or a state vector or padding, you can generate a source by buffering, calling the function or reasonably argue about the state, say the state's state.

`int yarrow256_is_seeded (struct yarrow256_ctx *ctx)` [Function]
 Returns 1 if the generator is seeded, 0 otherwise.

`unsigned yarrow256_needed_sources (struct yarrow256_ctx *ctx)` [Function]
 Returns the number of sources that must be read before a source is available for use by the generator.

`void yarrow256_fast_reseed (struct yarrow256_ctx *ctx)` [Function]

`void yarrow256_slow_reseed (struct yarrow256_ctx *ctx)` [Function]
 Causes a fast or slow reseed to take place at the next opportunity. The entropy source is not used for the fast reseed.

The function `yarrow256_slow_reseed` is only used for the slow reseed.

`struct yarrow_key_event_ctx` [Context structure]

Information about the key event.

`void yarrow_key_event_init (struct yarrow_key_event_ctx *ctx)` [Function]

Initializes the context.

`unsigned yarrow_key_event_estimate (struct yarrow_key_event_ctx *ctx, unsigned key, unsigned time)` [Function]

key is the key value, ar is the array, X is the array, ..., time is the time, and time is the time. The function returns the number of sources that must be read before a source is available for use by the generator. The function returns the number of sources that must be read before a source is available for use by the generator. The function returns the number of sources that must be read before a source is available for use by the generator.

The function `yarrow_key_event_estimate` is only used for the key event.

7.9 ASCII encoding

Encryption with a transposition cipher is a simple way to encrypt data, but it is not secure. If you want to send data as a plain text, you can use a transposition cipher. You can use a transposition cipher to encrypt data. The function `yarrow256_encrypt` is used for this purpose.

A basic transposition cipher is a simple way to encrypt data. The function `yarrow256_encrypt` is used for this purpose.

The basic transposition cipher is a simple way to encrypt data. The function `yarrow256_encrypt` is used for this purpose.

The basic transposition cipher is a simple way to encrypt data. The function `yarrow256_encrypt` is used for this purpose.

The basic transposition cipher is a simple way to encrypt data. The function `yarrow256_encrypt` is used for this purpose.

Enco ng an co ng us s a cont xt struct to anta n ts stat (w t t xc pt on o bas 0 nco ng, w c o sn't n any). o nco or co t ata, rst n ta t cont xt, t n ca t up at unct on as any t s as n c ssary, an co p t t op rat on by ca ng t na unct on.

allowing functions can be used to provide basic encoding and decoding. For example, the following function encodes a string using the `base64` module:

```
struct base64_encode_ctx [Cont xt struct]
```

```
void base64_encode_init (struct base64_encode_ctx *ctx) [Funct on]
```

```
void base64url_encode_init (struct base64_encode_ctx *ctx) [Funct on]
```

In a `base64` context, `base64_encode_init` sets the standard `base64` alphabet, while `base64url_encode_init` sets the `URL` safe alphabet.

```
size_t base64_encode_single (struct base64_encode_ctx *ctx, [Funct on]
                             uint8_t *dst, uint8_t src)
```

Encoding by $\mathbf{P}$ turns a count of output (a ways or).

BASE64_ENCODE_LENGTH (*length*) [acro]

```
ax_u_nu_bro_output_bytes_w_npassng_length_nput_bytes_to_base64_
encode_update.
```

```
size_t base64_encode_update (struct base64_encode_ctx *ctx, [Funct on]
                             uint8_t *dst, size_t length, const uint8_t *src)
```

```

A t r c t x s n t a      , t s u n c t i o n   a y b c a      t o n c o   l e n g t h   b y t s r o   s r c .
    r s u t w   b p a c   n d s t , a n t   r t u r n v a u w   b t   n u b r o b y t s g n r a t .
    o t t a t d s t   u s t b a t a s t o s   B A E _ E C _ D E _ L E   G H ( l e n g t h ) .

```

BASE64_ENCODE_FINAL_LENGTH	[Constant]
----------------------------	------------

```
ax u a ount o output ro  base64_encode_final.
```

```
size_t base64_encode_final (struct base64_encode_ctx *ctx, uint8_t [Funct on]
                           *dst)
```

At read time, the encoder outputs a sequence of tokens, which are then processed by the decoder to produce the final output. The decoder also takes the encoder's output as input and produces the final output. The final output is then compared with the ground truth output to calculate the loss.

```
struct base64_decode_ctx [Cont xt struct]
```

```
void base64_decode_init (struct base64_decode_ctx *ctx) [Funct on]
```

```
void base64url_decode_init (struct base64_decode_ctx *ctx) [Funct on]
```

In a `base64_decode_init` context, the necessary buffer for storing the decoded data is obtained by calling `base64_decode_init` with the `PL` flag.

```
int base64_decode_single (struct base64_decode_ctx *ctx, uint8_t  
                          *dst, uint8_t src) [Funct on]
```

Decoding: The decoder takes a source string `src` and a storage `dst`. It turns a count of output (or), or - on errors.

BASE64_DECODE_LENGTH (*length*) [acro]
 ax u nu b r o output byt s w n pass ng *length* nput byt s to **base64_decode_update**.

void base64_decode_update (*struct base64_decode_ctx *ctx*, *size_t* [Func on]
**dst_length*, *uint8_t *dst*, *size_t src_length*, *const uint8_t *src*)
 At r ctx s n t a , t s unct on ay b ca to co *src_length* byt s ro *src*.
dst s ou p nt to an ar a o s at ast BA E _DEC _DE LE G H(*src_length*).
 a ount o ata g n rat s r turn n *dst_length*. P turns on succ ss an
 on rror.

int base64_decode_final (*struct base64_decode_ctx *ctx*) [Func on]
 C c t at na pa ng s corr ct. P turns on succ ss, an on rror.

ary to t bas unct ons, t o ow ng unct ons p r or bas nco ng, an
 ar n n '<nettle/base16.h>'. ot t at t r s no nco ng cont xt n c ssary or
 on g bas nco ng.

void base16_encode_single (*uint8_t *dst*, *uint8_t src*) [Func on]
 Enco s a s ng byt . A ways stor s two g ts n *dst*[] an *dst*[].

BASE16_ENCODE_LENGTH (*length*) [acro]
 nu b r o output byt s w n pass ng *length* nput byt s to **base16_encode_update**.

void base16_encode_update (*uint8_t *dst*, *size_t length*, *const* [Func on]
*uint8_t *src*)
 A ways stor s BA E _E C _DE LE G H(*length*) g ts n *dst*.

struct base16_decode_ctx [Cont xt struct]

void base16_decode_init (*struct base16_decode_ctx *ctx*) [Func on]
 In t a s a bas co ng cont xt. s s n c ssary b or start ng a co ng
 s ss on.

int base16_decode_single (*struct base16_decode_ctx *ctx*, *uint8_t* [Func on]
**dst*, *uint8_t src*)
 D co s a s ng byt ro *src* nto *dst*. P turns a ount o output (or), or - on
 rrors.

BASE16_DECODE_LENGTH (*length*) [acro]
 ax u nu b r o output byt s w n pass ng *length* nput byt s to **base16_decode_update**.

int base16_decode_update (*struct base16_decode_ctx *ctx*, *size_t* [Func on]
**dst_length*, *uint8_t *dst*, *size_t src_length*, *const uint8_t *src*)
 At r ctx s n t a , t s unct on ay b ca to co *src_length* byt s ro *src*.
dst s ou p nt to an ar a o s at ast BA E _DEC _DE LE G H(*src_length*).
 a ount o ata g n rat s r turn n *dst_length*. P turns on succ ss an
 on rror.

```
int base16_decode_final (struct base16_decode_ctx *ctx) [Funct on]
C c s t a t t n o a t a s c o r r c t ( . . , a n v n n u b r o x a c a g t s
a v b n s n). P turns on succ ss, an on rror.
```

7.10 Miscellaneous functions

```
void * memxor (void *dst, const void *src, size_t n) [Funct on]
X o P s t s o u r c a r a o n t o p o t s t n a t o n a r a. n t r a c o s n ' t o o w
t t t t c o n v n t o n s, b c a u s t s n t n t o b s a r t o t A I-C m e m c p y
u n c t o n.
```

```
void * memxor3 (void *dst, const void *a, const void *b, size_t n) [Funct on]
L m e m x o r, b u t t a s t w o s o u r c a r a s a n s p a r a t s t n a t o n a r a.
```

```
int memeq1_sec (const void *a, const void *b, size_t n) [Funct on]
- c a n n s n t c o p a r s o n o t n b y t s a t a a n b. I. ., n s t r u c t o n s x c u t
a n o r y a c c s s a r n t c a n o a t t r w r t a r a s f r, s c t o n . 3.
[ - c a n n s n c ], p a g 3. P t u r n n o n- r o t a r a s a r q u a, a n r o
t y f r.
```

s u n c t o n s a r c a r n '<nettle/memops.h>'. F o r c o p a t b t y w t a r r
v r s o n s o t t , m e m x o r a n m e m x o r 3 a r a s o c a r n '<nettle/memxor.h>'.

7.11 Compatibility functions

F o r c o n v n n c , t t n c u s a t r n a t v n t r a c s t o s o a g o r t s, o r c o p a t b t y
w t s o o t r p o p u a r c r y p t o t o o t s. s a r n o t u y o c u n t r, r r t o t
s o u r c o r t o t o c u n t a t o n o r t o r g n a p n t a t o n.

D s n n [P F C 3], w c n c u s a r r n c p n t a t o n. t t n s
a c o p a t b n t r a c t o D n '<nettle/md5-compat.h>'. s n s t t y p
M D 5 _ C T X, a n c a r s t u n c t o n s M D 5 I n i t, M D 5 U p d a t e a n M D 5 F i n a l.

8 Traditional Nettle Soup

For t s r o u s n t t a c r , r s a r c p o r n t t s o u p . s r v n g s .

t r r s n t t s (u r t c a o c a)

t a b s p o o n s b u t t r

3 t a b s p o o n s o u r

t r s t o c (a t o r v g t a b)

t a s p o o n s a t

a t a w t p p p r

s o c r a o r

G a t r t r r s n t t s . s g o v s   a , t n r s o o t s a r p r r a b b u t t t o p s
o a r g r n t t s c a n a s o b u s .

P n s t n t t s v r y w . B o t o r n u t s n g t y s a t w a t r .   t r a n t
n t t s a n s a v t w a t r . H a c t n t t s . t t b u t t r a n x n t o u r . D u t
w t s t o c a n t n t t - w a t r y o u s a v a r r . A t a c n t t s . I y o u w s y o u
c a n a s o o r c r a a t t s s t a g . B r n g t o a b o a n t b o o r a w n u t s .
  a s o n w t s a t a n p p p r .

  r v w t b o g g - a v s .

9 Installation

```

tt us s autoconf. o bu t, unpac t sourc an run
    ./configure
    make
    make check
    make install
to nsta t un r t aut pr x, '/usr/local'. s ng G a s strong y r co -
n . By aut, bot statc an s ar brar s ar bu t an nsta .
og t a sto con gur opt ons, us ./configure --help. o o t or nt r st ng
ar

'--enable-fat'
    Incu ut p v rs ons o c rta n unct ons nt brary, an s ct t on s
to us at run-t , p n ng on ava ab proc ssor atur s. upport or
AP an x .

'--enable-mini-gmp'
    s t s a r an s ow r " n-g p" p ntat on o t b gnu unct ons
n or pub c- y cryptograp y, nst a o t r a G G brary. s
opt on s nt n pr ar y or s a r b syst s. ot t at bu s
us ng n-g p ar not b nary co pat b wt r gu ar bu s o tt , an
or y to a s -c ann n or at on.

'--disable-shared'
    o t bu ng t s ar brar s.

'--disable-dependency-tracking'
    Dsab t auto at c p n ncy trac ng. You w y n t s opt on to
b ab to bu wt B D a .

```

Function and Concept Index

3

3DES 32

A

ABI compatibility 8
 AEAD 44
 aes_decrypt 23
 aes_encrypt 23
 aes_invert_key 23
 aes_set_decrypt_key 23
 aes_set_encrypt_key 23
 aes128_decrypt 23
 aes128_encrypt 23
 aes128_invert_key 23
 aes128_set_decrypt_key 22
 aes128_set_encrypt_key 22
 aes192_decrypt 23
 aes192_encrypt 23
 aes192_invert_key 23
 aes192_set_decrypt_key 22
 aes192_set_encrypt_key 22
 aes256_decrypt 23
 aes256_encrypt 23
 aes256_invert_key 23
 aes256_set_decrypt_key 23
 aes256_set_encrypt_key 22
 AES 22
 API compatibility 8
 Arcfour 24
 arcfour_crypt 24
 arcfour_set_key 24
 Arctwo 25
 arctwo_decrypt 25
 arctwo_encrypt 25
 arctwo_set_key 25
 arctwo_set_key_ekb 25
 arctwo_set_key_gutmann 25
 Authenticated encryption 44

B

base16_decode_final 97
 base16_decode_init 96
 base16_decode_single 96
 base16_decode_update 96
 base16_encode_single 96
 base16_encode_update 96
 base64_decode_final 96
 base64_decode_init 95
 base64_decode_single 95
 base64_decode_update 96
 base64_encode_final 95
 base64_encode_init 95

base64_encode_single 95
 base64_encode_update 95
 base64url_decode_init 95
 base64url_encode_init 95
 BASE16_DECODE_LENGTH 96
 BASE16_ENCODE_LENGTH 96
 BASE64_DECODE_LENGTH 96
 BASE64_ENCODE_LENGTH 95
 Binary compatibility 8
 Block Cipher 21
 Blowfish 26
 blowfish_bcrypt_hash 26
 blowfish_bcrypt_verify 27
 blowfish_decrypt 26
 blowfish_encrypt 26
 blowfish_set_key 26

C

Camellia 27
 camellia_crypt 29
 camellia_invert_key 29
 camellia_set_decrypt_key 28
 camellia_set_encrypt_key 28
 camellia128_crypt 29
 camellia128_invert_key 29
 camellia128_set_decrypt_key 28
 camellia128_set_encrypt_key 28
 camellia192_crypt 29
 camellia192_invert_key 29
 camellia192_set_decrypt_key 28
 camellia192_set_encrypt_key 28
 camellia256_crypt 29
 camellia256_invert_key 29
 camellia256_set_decrypt_key 28
 camellia256_set_encrypt_key 28
 cast128_decrypt 30
 cast128_encrypt 30
 cast128_set_key 29
 cast5_set_key 30
 CAST 29
 cbc_decrypt 38
 cbc_encrypt 38
 CBC Mode 37
 CBC_CTX 38
 CBC_DECRYPT 38
 CBC_ENCRYPT 38
 CBC_SET_IV 38
 ccm_aes128_decrypt 56
 ccm_aes128_decrypt_message 56
 ccm_aes128_digest 56
 ccm_aes128_encrypt 56
 ccm_aes128_encrypt_message 56
 ccm_aes128_set_key 55

ccm_aes128_set_nonce	55	cmac_aes128_digest	66
ccm_aes128_update	56	cmac_aes128_set_key	66
ccm_aes192_decrypt	56	cmac_aes128_update	66
ccm_aes192_decrypt_message	56	cmac_aes256_digest	67
ccm_aes192_digest	56	cmac_aes256_set_key	66
ccm_aes192_encrypt	56	cmac_aes256_update	67
ccm_aes192_encrypt_message	56	cmac_des3_digest	67
ccm_aes192_set_key	55	cmac_des3_set_key	67
ccm_aes192_set_nonce	55	cmac_des3_update	67
ccm_aes192_update	56	CMAC	66
ccm_aes256_decrypt	56	CMAC-128	66
ccm_aes256_digest	56	CMAC-64	66
ccm_aes256_encrypt	56	Collision-resistant	9
ccm_aes256_encrypt_message	56	Conditional entropy	90
ccm_aes256_set_key	55	Counter Mode	38
ccm_aes256_set_nonce	55	Counter with CBC-MAC Mode	52
ccm_aes256_update	56	ctr_crypt	39
ccm_decrypt	54	CTR Mode	38
ccm_decrypt_message	55	CTR_CRYPT	39
ccm_digest	54	CTR_CTX	39
ccm_encrypt	54	CTR_SET_COUNTER	39
ccm_encrypt_message	54	Curve 25519	87
ccm_set_nonce	54	Curve 448	87
ccm_update	54	curve25519_mul	88
CCM Mode	52	curve25519_mul_g	87
CCM_MAX_MSG_SIZE	53	curve448_mul	88
cfb_decrypt	40	curve448_mul_g	88
cfb_encrypt	40		
cfb8_decrypt	40	D	
cfb8_encrypt	40	des_check_parity	32
CFB Modes	39	des_decrypt	32
CFB_CTX	41	des_encrypt	32
CFB_DECRYPT	41	des_fix_parity	32
CFB_ENCRYPT	41	des_set_key	31
CFB_SET_IV(ctx,)	41	des3_decrypt	33
CFB8 Mode	39	des3_encrypt	33
CFB8_CTX	41	des3_set_key	33
CFB8_DECRYPT	41	DES	31
CFB8_ENCRYPT	41	DES3	32
CFB8_SET_IV	41	dsa_compat_generate_keypair	83
ChaCha	30	dsa_generate_keypair	81
chacha_crypt	30	dsa_generate_params	80
chacha_crypt32	31	dsa_params_clear	80
chacha_poly1305_decrypt	58	dsa_params_init	80
chacha_poly1305_digest	58	dsa_private_key_clear	82
chacha_poly1305_encrypt	58	dsa_private_key_init	82
chacha_poly1305_set_key	57	dsa_public_key_clear	82
chacha_poly1305_set_nonce	57	dsa_public_key_init	82
chacha_poly1305_update	58	dsa_shal_sign	82
chacha_set_counter	30	dsa_shal_sign_digest	82
chacha_set_counter32	31	dsa_shal_verify	82
chacha_set_key	30	dsa_shal_verify_digest	82
chacha_set_nonce	30	dsa_sha256_sign	82
chacha_set_nonce96	31	dsa_sha256_sign_digest	82
Cipher	21	dsa_sha256_verify	82
Cipher Block Chaining	37	dsa_sha256_verify_digest	83
Cipher Feedback 8-bit Mode	39	dsa_sign	81
Cipher Feedback Mode	39		

dsa_signature_clear	81
dsa_signature_init	81
dsa_verify	81

E

eax_aes128_decrypt	47
eax_aes128_digest	47
eax_aes128_encrypt	47
eax_aes128_set_key	47
eax_aes128_set_nonce	47
eax_aes128_update	47
eax_decrypt	46
eax_digest	46
eax_encrypt	46
eax_set_key	45
eax_set_nonce	46
eax_update	46
EAX_CTX	46
EAX_DECRYPT	47
EAX_DIGEST	47
EAX_ENCRYPT	47
EAX_SET_KEY	46
EAX_SET_NONCE	46
EAX_UPDATE	47
ecc_point_clear	84
ecc_point_get	84
ecc_point_init	84
ecc_point_set	84
ecc_scalar_clear	85
ecc_scalar_get	85
ecc_scalar_init	85
ecc_scalar_set	85
ecdsa_generate_keypair	85
ecdsa_sign	85
ecdsa_verify	85
ed25519_sha512_public_key	89
ed25519_sha512_sign	89
ed25519_sha512_verify	89
ed448_shake256_public_key	89
ed448_shake256_sign	89
ed448_shake256_verify	89
eddsa	88
Entropy	90

G

Galois Counter Mode	47
gcm_aes_decrypt	51
gcm_aes_digest	51
gcm_aes_encrypt	51
gcm_aes_set_iv	50
gcm_aes_set_key	50
gcm_aes_update	50
gcm_aes128_decrypt	51
gcm_aes128_digest	51
gcm_aes128_encrypt	51
gcm_aes128_set_iv	50

gcm_aes128_set_key	50
gcm_aes128_update	50
gcm_aes192_decrypt	51
gcm_aes192_digest	51
gcm_aes192_encrypt	51
gcm_aes192_set_iv	50
gcm_aes192_set_key	50
gcm_aes192_update	50
gcm_aes256_decrypt	51
gcm_aes256_digest	51
gcm_aes256_encrypt	51
gcm_aes256_set_iv	50
gcm_aes256_set_key	50
gcm_aes256_update	50
gcm_camellia_digest	52
gcm_camellia128_decrypt	52
gcm_camellia128_digest	52
gcm_camellia128_encrypt	52
gcm_camellia128_set_iv	52
gcm_camellia128_set_key	51
gcm_camellia128_update	52
gcm_camellia192_digest	52
gcm_camellia256_decrypt	52
gcm_camellia256_digest	52
gcm_camellia256_encrypt	52
gcm_camellia256_set_iv	52
gcm_camellia256_set_key	51
gcm_camellia256_update	52
gcm_decrypt	48
gcm_digest	49
gcm_encrypt	48
gcm_set_iv	48
gcm_set_key	48
gcm_update	48
GCM	47
GCM_CTX	49
GCM_DECRYPT	49
GCM_DIGEST	49
GCM_ENCRYPT	49
GCM_SET_IV	49
GCM_SET_KEY	49
GCM_UPDATE	49
GOST DSA	85
GOST hash	19
gostdsa_sign	86
gostdsa_verify	86
gostdsa_vko	87
gosthash94_digest	19
gosthash94_init	19
gosthash94_update	19
gosthash94cp_digest	20
gosthash94cp_init	20
gosthash94cp_update	20

H

Hash function	9
hkdf_expand	69

hkdf_extract	69
HKDF	68
hmac_digest	61
hmac_md5_digest	62
hmac_md5_set_key	62
hmac_md5_update	62
hmac_ripemd160_digest	63
hmac_ripemd160_set_key	62
hmac_ripemd160_update	62
hmac_set_key	61
hmac_sha1_digest	63
hmac_sha1_set_key	63
hmac_sha1_update	63
hmac_sha256_digest	63
hmac_sha256_set_key	63
hmac_sha256_update	63
hmac_sha512_digest	64
hmac_sha512_set_key	63
hmac_sha512_update	64
hmac_update	61
HMAC	61
HMAC_CTX	61
HMAC_DIGEST	62
HMAC_SET_KEY	62

K

KDF	69
Key Derivation Function	68
Keyed Hash Function	60

M

MAC	60
md2_digest	17
md2_init	17
md2_update	17
md4_digest	18
md4_init	18
md4_update	18
md5_digest	17
md5_init	16
md5_update	16
memeql_sec	97
memxor	97
memxor3	97
Message Authentication Code	60

N

nettle_aead	59
nettle_aeads	60
nettle_cipher	36
nettle_ciphers	37
nettle_get_aeads	60
nettle_get_ciphers	36
nettle_get_gost_gc256b(void)	86
nettle_get_gost_gc512a(void)	86

nettle_get_hashes	21
nettle_hash	20
nettle_hashes	21

O

One-way	9
One-way function	71

P

Password Based Key Derivation Function	69
pbkdf2	69
pbkdf2_hmac_sha1	70
pbkdf2_hmac_sha256	70
pbkdf2_hmac_sha384	70
pbkdf2_hmac_sha512	70
PBKDF	69
PBKDF2	69
PKCS #5	69
poly1305_aes_digest	68
poly1305_aes_set_key	68
poly1305_aes_set_nonce	68
poly1305_aes_update	68
Public Key Cryptography	71

R

Randomness	89
RC2	25
RC4	24
ripemd160_digest	18
ripemd160_init	18
ripemd160_update	18
rsa_compute_root	78
rsa_compute_root_tr(const	77
rsa_decrypt	77
rsa_decrypt_tr	77
rsa_encrypt	77
rsa_generate_keypair	78
rsa_md5_sign	75
rsa_md5_sign_digest	75
rsa_md5_sign_digest_tr(const	74
rsa_md5_sign_tr(const	74
rsa_md5_verify	75
rsa_md5_verify_digest	76
rsa_pkcs1_sign(const	75
rsa_pkcs1_sign_tr(const	75
rsa_pkcs1_verify(const	76
rsa_private_key_clear	73
rsa_private_key_init	73
rsa_private_key_prepare	73
rsa_pss_sha256_sign_digest_tr(const	76
rsa_pss_sha256_verify_digest	77
rsa_pss_sha384_sign_digest_tr(const	76
rsa_pss_sha384_verify_digest	77
rsa_pss_sha512_sign_digest_tr(const	76
rsa_pss_sha512_verify_digest	77

rsa_public_key_clear	73
rsa_public_key_init	73
rsa_public_key_prepare	73
rsa_sha1_sign	75
rsa_sha1_sign_digest	75
rsa_sha1_sign_digest_tr(const	74
rsa_sha1_sign_tr(const	74
rsa_sha1_verify	75
rsa_sha1_verify_digest	76
rsa_sha256_sign	75
rsa_sha256_sign_digest	75
rsa_sha256_sign_digest_tr(const	74
rsa_sha256_sign_tr(const	74
rsa_sha256_verify	75
rsa_sha256_verify_digest	76
rsa_sha512_sign	75
rsa_sha512_sign_digest	75
rsa_sha512_sign_digest_tr(const	74
rsa_sha512_sign_tr(const	74
rsa_sha512_verify	75
rsa_sha512_verify_digest	76

S

Salsa20	33
salsa20_128_set_key	34
salsa20_256_set_key	34
salsa20_crypt	34
salsa20_set_key	34
salsa20_set_nonce	34
salsa20r12_crypt	34
Serpent	34
serpent_decrypt	35
serpent_encrypt	35
serpent_set_key	35
sha1_digest	19
sha1_init	19
sha1_update	19
sha224_digest	10
sha224_init	10
sha224_update	10
sha256_digest	10
sha256_init	9
sha256_update	9
sha3_224_digest	13
sha3_224_init	13
sha3_224_update	13
sha3_256_digest	13
sha3_256_init	13
sha3_256_shake	15
sha3_256_update	13
sha3_384_digest	14
sha3_384_init	14
sha3_384_update	14
sha3_512_digest	14
sha3_512_init	14
sha3_512_update	14
sha384_digest	12

sha384_init	11
sha384_update	12
sha512_224_digest	12
sha512_224_init	11
sha512_224_update	12
sha512_256_digest	12
sha512_256_init	11
sha512_256_update	12
sha512_digest	11
sha512_init	11
sha512_update	11
SHA3	12
SHAKE	14
Side-channel attack	83
siv_cmac_aes128_decrypt_message	59
siv_cmac_aes128_encrypt_message	59
siv_cmac_aes128_set_key	59
siv_cmac_aes256_decrypt_message	59
siv_cmac_aes256_encrypt_message	59
siv_cmac_aes256_set_key	59
SIV mode	58
SIV-CMAC mode	58
Stream Cipher	21
streebog256_digest	16
streebog256_init	16
streebog256_update	16
streebog512_digest	15
streebog512_init	15
streebog512_update	15

T

Triple-DES	32
Twofish	35
twofish_decrypt	36
twofish_encrypt	36
twofish_set_key	35

U

umac128_digest	66
umac128_set_key	65
umac128_set_nonce	65
umac128_update	65
umac32_digest	66
umac32_set_key	65
umac32_set_nonce	65
umac32_update	65
umac64_digest	66
umac64_set_key	65
umac64_set_nonce	65
umac64_update	65
umac96_digest	66
umac96_set_key	65
umac96_set_nonce	65
umac96_update	65
UMAC	64

X

XEX-based tweaked-codebook mode with ciphertext stealing	42
<code>xts_aes128_decrypt_message(struct</code>	44
<code>xts_aes128_encrypt_message(struct</code>	44
<code>xts_aes128_set_decrypt_key</code>	43
<code>xts_aes128_set_encrypt_key</code>	43
<code>xts_aes256_decrypt_message(struct</code>	44
<code>xts_aes256_encrypt_message(struct</code>	44
<code>xts_aes256_set_decrypt_key</code>	43
<code>xts_aes256_set_encrypt_key</code>	43
<code>xts_decrypt_message</code>	43
<code>xts_encrypt_message</code>	43

XTS Mode	42
----------------	----

Y

<code>yarrow_key_event_estimate</code>	94
<code>yarrow_key_event_init</code>	94
<code>yarrow256_fast_reseed</code>	94
<code>yarrow256_init</code>	93
<code>yarrow256_is_seeded</code>	94
<code>yarrow256_needed_sources</code>	94
<code>yarrow256_random</code>	93
<code>yarrow256_seed</code>	93
<code>yarrow256_slow_reseed</code>	94
<code>yarrow256_update</code>	93