Nettle Manual

For the Nettle Library version 2.7

Niels Möller

This manual is for the Nettle library version 2.7. a low-level cryptographic library.
Originally written 2, by Niels Möller, updated 2, 3.

Table of Contents

1 Introduction

Nettle is a cryptographic library that is designed to fit easily in more or less any context: In crypto toolkits for object-oriented languages (C++, Python, Pike, ...), in applications like LSH or GNUPG, or even in kernel space. In most contexts, you need more than the basic cryptographic algorithms, you also need some way to keep track of available algorithms, their properties and variants. You often have some algorithm selection process, often dictated by a protocol you want to implement.

And as the requirements of applications differ in subtle and not so subtle ways, an API that fits one application well can be a pain to use in a different context. And that is why there are so many different cryptographic libraries around.

Nettle tries to avoid this problem by doing one thing, the low-level crypto stuff, and providing a *simple* but general interface to it. In particular, Nettle doesn't do algorithm selection. It doesn't do memory allocation. It doesn't do any I/O.

The idea is that one can build several application and context specific interfaces on top of Nettle, and share the code, test cases, benchmarks, documentation, etc. Examples are the Nettle module for the Pike language, and LSH, which both use an object-oriented abstraction on top of the library.

This manual explains how to use the Nettle library. It also tries to provide some background on the cryptography, and advice on how to best put it to use.

2 Copyright

Nettle is distributed under the GNU Lesser General Public License (LGPL), see the file COPYING.LIB for details. A few of the individual files are in the public domain. To find the current status of particular files, you have to read the copyright notices at the top of the files.

This manual is in the public domain. You may freely copy it in whole or in part, e.g., into documentation of programs that build on Nettle. Attribution, as well as contribution of improvements to the text, is of course appreciated, but it is not required.

A list of the supported algorithms, their origins and licenses:

AES The implementation of the AES cipher (also known as rijndael) is written by Rafael Sevilla. Assembler for x86 by Rafael Sevilla and Niels Möller, Sparc assembler by Niels Möller. Released under the LGPL.

ARCFOUR
The implementation of the ARCFOUR (also known as RC4) cipher is written by Niels Möller. Released under the LGPL.

ARCTWO
The implementation of the ARCTWO (also known as RC2) cipher is written by Nikos Mavroyanopoulos and modified by Werner Koch and Simon Josefsson. Released under the LGPL.

BLOWFISH
The implementation of the BLOWFISH cipher is written by Werner Koch, copyright owned by the Free Software Foundation. Also hacked by Simon Josefsson and Niels Möller. Released under the LGPL.

CAMELLIA
The C implementation is by Nippon Telegraph and Telephone Corporation (NTT), heavily modified by Niels Möller. Assembler for x86 and x86_64 by Niels Möller. Released under the LGPL.

CAST128 The implementation of the CAST128 cipher is written by Steve Reid. Released into the public domain.

DES The implementation of the DES cipher is written by Dana L. How, and released under the LGPL.

GOSTHASH94
The C implementation of the GOST94 message digest is written by Aleksey Kravchenko and was ported from the rhash library by Nikos Mavrogiannopoulos. It is released under the MIT license.

MD2 The implementation of MD2 is written by Andrew Kuchling, and hacked some by Andreas Sigfridsson and Niels Möller. Python Cryptography Toolkit license (essentially public domain).

MD4 This is almost the same code as for MD5 below, with modifications by Marcus Comstedt. Released into the public domain.

MD5 The implementation of the MD5 message digest is written by Colin Plumb. It has been hacked some more by Andrew Kuchling and Niels Möller. Released into the public domain.

PBKDF2 The C implementation of PBKDF2 is based on earlier work for Shishi and GnuTLS by Simon Josefsson. Released under the LGPL.

R PEMD160
The implementation of RIPEMD, 6 message digest is based on the code in libgcrypt, copyright owned by the Free Software Foundation. Ported to Nettle by Andres Mejia. Released under the LGPL.

SALSA20 The C implementation of SALSA2 is based on D. J. Bernstein's reference implementation in the public domain, adapted to Nettle by Simon Josefsson, and heavily modified by Niels Möller. Assembly for x86_64 and ARM by Niels Möller. Released under the LGPL.

SERPENT
The implementation of the SERPENT cipher is based on the code in libgcrypt, copyright owned by the Free Software Foundation. Adapted to Nettle by Simon Josefsson and heavily modified by Niels Möller. Assembly for x86_64 by Niels Möller. Released under the LGPL.

SHA1 The C implementation of the SHA, message digest is written by Peter Gutmann, and hacked some more by Andrew Kuchling and Niels Möller. Released into the public domain. Assembler for x86, x86_64 and ARM by Niels Möller, released under the LGPL.

SHA2 Written by Niels Möller, using Peter Gutmann's SHA, code as a model. Released under the LGPL.

SHA3 Written by Niels Möller. Released under the LGPL.

TWOF SH
The implementation of the TWOFISH cipher is written by Ruud de Rooij. Released under the LGPL.

UMAC Written by Niels Möller. Released under the LGPL.

RSA Written by Niels Möller, released under the LGPL. Uses the GMP library for bignum operations.

DSA Written by Niels Möller, released under the LGPL. Uses the GMP library for bignum operations.

ECDSA Written by Niels Möller, released under the LGPL. Uses the GMP library for bignum operations. Development of Nettle's ECC support was funded by the .SE Internet Fund.

3 Conventions

For each supported algorithm, there is an include file that defines a *context struct*, a few constants, and declares functions for operating on the context. The context struct encapsulates all information needed by the algorithm, and it can be copied or moved in memory with no unexpected effects.

For consistency, functions for different algorithms are very similar, but there are some differences, for instance reflecting if the key setup or encryption function differ for encryption and decryption, and whether or not key setup can fail. There are also differences between algorithms that don't show in function prototypes, but which the application must nevertheless be aware of. There is no big difference between the functions for stream ciphers and for block ciphers, although they should be used quite differently by the application.

If your application uses more than one algorithm of the same type, you should probably create an interface that is tailor-made for your needs, and then write a few lines of glue code on top of Nettle.

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

In all functions, strings are represented with an explicit length, of type `unsigned`, and a pointer of type `uint8_t *` or `const uint8_t *`. For functions that transform one string to another, the argument order is length, destination pointer and source pointer. Source and destination areas are of the same length. Source and destination may be the same, so that you can process strings in place, but they *must not* overlap in any other way.

Many of the functions lack return value and can never fail. Those functions which can fail, return one on success and zero on failure.

4 Example

A simple example program that reads a file from standard input and writes its SHA1 checksum on standard output should give the flavor of Nettle.

```
#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 linked with the command line

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

5 Linking

Nettle actually consists of two libraries, `libnettle` and `libhogweed`. The `libhogweed` library contains those functions of Nettle that uses bignum operations, and depends on the GMP library. With this division, linking works the same for both static and dynamic libraries.

If an application uses only the symmetric crypto algorithms of Nettle (i.e., block ciphers, hash functions, and the like), it's sufficient to link with `-lnettle`. If an application also uses public-key algorithms, the recommended linker flags are `-lhogweed -lnettle -lgmp`. If the involved libraries are installed as dynamic libraries, it may be sufficient to link with just `-lhogweed`, and the loader will resolve the dependencies automatically.

6 Reference

This chapter describes all the Nettle functions, grouped by family.

6.1 Hash functions

A cryptographic *hash function* is a function that takes variable size strings, and maps them to strings of fixed, short, length. There are naturally lots of collisions, as there are more possible 1MB files than 20 byte strings. But the function is constructed such that is hard to find the collisions. More precisely, a cryptographic hash function `H` should have the following properties:

One-way Given a hash value `H(x)` it is hard to find a string `x` that hashes to that value.

Collision-resistant
It is hard to find two different strings, `x` and `y`, such that `H(x)` = `H(y)`.

Hash functions are useful as building blocks for digital signatures, message authentication codes, pseudo random generators, association of unique ids to documents, and many other things.

The most commonly used hash functions are MD5 and SHA1. Unfortunately, both these fail the collision-resistance requirement; cryptologists have found ways to construct colliding inputs. The recommended hash functions for new applications are SHA2 (with main variants SHA256 and SHA512). At the time of this writing (December 2012), the winner of the NIST SHA3 competition has recently been announced, and the new SHA3 (earlier known as Keccak) and other top SHA3 candidates may also be reasonable alternatives.

6.1.1 Recommended hash functions

The following hash functions have no known weaknesses, and are suitable for new applications. The SHA2 family of hash functions were specified by *NIST*, intended as a replacement for SHA1.

6.1.1.1 SHA256

SHA256 is a member of the SHA2 family. It outputs hash values of 256 bits, or 32 octets. Nettle defines SHA256 in `<nettle/sha2.h>`.

`struct sha256_ctx` [Context struct]

`SHA256_DIGEST_SIZE` [Constant]
The size of a SHA256 digest, i.e. 32.

`SHA256_DATA_SIZE` [Constant]
The internal block size of SHA256. Useful for some special constructions, in particular HMAC-SHA256.

`void sha256_init (struct sha256_ctx *ctx)` [Function]
Initialize the SHA256 state.

`void sha256_update (struct sha256_ctx *ctx, unsigned length, const uint8_t *data)` [Function]
Hash some more data.

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

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than SHA256_DIGEST_SIZE, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as sha256_init.

Earlier versions of nettle defined SHA256 in the header file <nettle/sha.h>, which is now deprecated, but kept for compatibility.

6.1.1.2 SHA224

SHA224 is a variant of SHA256, with a different initial state, and with the output truncated to 224 bits, or 28 octets. Nettle defines SHA224 in <nettle/sha2.h> (and in <nettle/sha.h>, for backwards compatibility).

struct sha224_ctx [Context struct]

SHA224_DIGEST_SIZE [Constant]

The size of a SHA224 digest, i.e. 28.

SHA224_DATA_SIZE [Constant]

The internal block size of SHA224. Useful for some special constructions, in particular HMAC-SHA224.

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

Initialize the SHA224 state.

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

Hash some more data.

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

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than SHA224_DIGEST_SIZE, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as sha224_init.

6.1.1.3 SHA512

SHA512 is a larger sibling to SHA256, with a very similar structure but with both the output and the internal variables of twice the size. The internal variables are 64 bits rather than 32, making it significantly slower on 32-bit computers. It outputs hash values of 512 bits, or 64 octets. Nettle defines SHA512 in <nettle/sha2.h> (and in <nettle/sha.h>, for backwards compatibility).

struct sha512_ctx [Context struct]

SHA512_DIGEST_SIZE [Constant]

The size of a SHA512 digest, i.e. 64.

SHA512_DATA_SIZE [Constant]
: The internal block size of SHA512. Useful for some special constructions, in particular HMAC-SHA512.

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

void sha512_update (*struct sha512_ctx* ***ctx***, *unsigned* **length**, *const uint8_t* ***data***) [Function]
: Hash some more data.

void sha512_digest (*struct sha512_ctx* ***ctx***, *unsigned* **length**, *uint8_t* ***digest***) [Function]
: Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA512_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

  This function also resets the context in the same way as `sha512_init`.

6.1.1.4 SHA384

SHA384 is a variant of SHA512, with a different initial state, and with the output truncated to 384 bits, or 48 octets. Nettle defines SHA384 in `<nettle/sha2.h>` (and in `<nettle/sha.h>`, for backwards compatibility).

struct sha384_ctx [Context struct]

SHA384_DIGEST_SIZE [Constant]
: The size of a SHA384 digest, i.e. 48.

SHA384_DATA_SIZE [Constant]
: The internal block size of SHA384. Useful for some special constructions, in particular HMAC-SHA384.

void sha384_init (*struct sha384_ctx* ***ctx***) [Function]
: Initialize the SHA384 state.

void sha384_update (*struct sha384_ctx* ***ctx***, *unsigned* **length**, *const uint8_t* ***data***) [Function]
: Hash some more data.

void sha384_digest (*struct sha384_ctx* ***ctx***, *unsigned* **length**, *uint8_t* ***digest***) [Function]
: Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA384_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

  This function also resets the context in the same way as `sha384_init`.

6.1.1.5 SHA3-224

The SHA3 hash functions were specified by NIST in response to weaknesses in SHA1, and doubts about SHA2 hash functions which structurally are very similar to SHA1. The standard is a result of a competition, where the winner, also known as Keccak, was designed by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. It is structurally very different from all widely used earlier hash functions. Like SHA2, there are several variants, with output sizes of 224, 256, 384 and 512 bits (28, 32, 48 and 64 octets, respectively).

Nettle defines SHA3-224 in `<nettle/sha3.h>`.

`struct sha3_224_ctx` [Context struct]

`SHA3_224_DIGEST_SIZE` [Constant]
: The size of a SHA3_224 digest, i.e., 28.

`SHA3_224_DATA_SIZE` [Constant]
: The internal block size of SHA3_224.

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

`void sha3_224_update (struct sha3_224_ctx *ctx, unsigned length, const uint8_t *data)` [Function]
: Hash some more data.

`void sha3_224_digest (struct sha3_224_ctx *ctx, unsigned length, uint8_t *digest)` [Function]
: Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA3_224_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context.

6.1.1.6 SHA3-256

This is SHA3 with 256-bit output size, and possibly the most useful of the SHA3 hash functions.

Nettle defines SHA3-256 in `<nettle/sha3.h>`.

`struct sha3_256_ctx` [Context struct]

`SHA3_256_DIGEST_SIZE` [Constant]
: The size of a SHA3_256 digest, i.e., 32.

`SHA3_256_DATA_SIZE` [Constant]
: The internal block size of SHA3_256.

`void sha3_256_init (struct sha3_256_ctx *ctx)` [Function]
: Initialize the SHA3-256 state.

`void sha3_256_update (struct sha3_256_ctx *ctx, unsigned length, const uint8_t *data)` [Function]
: Hash some more data.

void sha3_256_digest (*struct sha3_256_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***digest***) [Function]

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA3_256_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context.

6.1.1.7 SHA3-384

This is SHA3 with 384-bit output size.

Nettle defines SHA3-384 in `<nettle/sha3.h>`.

struct sha3_384_ctx [Context struct]

SHA3_384_DIGEST_SIZE [Constant]

The size of a SHA3_384 digest, i.e., 48.

SHA3_384_DATA_SIZE [Constant]

The internal block size of SHA3_384.

void sha3_384_init (*struct sha3_384_ctx* ***ctx***) [Function]

Initialize the SHA3-384 state.

void sha3_384_update (*struct sha3_384_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***data***) [Function]

Hash some more data.

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

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA3_384_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context.

6.1.1.8 SHA3-512

This is SHA3 with 512-bit output size.

Nettle defines SHA3-512 in `<nettle/sha3.h>`.

struct sha3_512_ctx [Context struct]

SHA3_512_DIGEST_SIZE [Constant]

The size of a SHA3_512 digest, i.e. 64.

SHA3_512_DATA_SIZE [Constant]

The internal block size of SHA3_512.

void sha3_512_init (*struct sha3_512_ctx* ***ctx***) [Function]

Initialize the SHA3-512 state.

void sha3_512_update (*struct sha3_512_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***data***) [Function]

Hash some more data.

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

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA3_512_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context.

6.1.2 Legacy hash functions

The hash functions in this section all have some known weaknesses, and should be avoided for new applications. These hash functions are mainly useful for compatibility with old applications and protocols. Some are still considered safe as building blocks for particular constructions, e.g., there seems to be no known attacks against HMAC-SHA1 or even HMAC-MD5. In some important cases, use of a "legacy" hash function does not in itself make the application insecure; if a known weakness is relevant depends on how the hash function is used, and on the threat model.

6.1.2.1 MD5

MD5 is a message digest function constructed by Ronald Rivest, and described in *RFC 1321*. It outputs message digests of 128 bits, or 16 octets. Nettle defines MD5 in `<nettle/md5.h>`.

struct md5_ctx [Context struct]

MD5_DIGEST_SIZE [Constant]

The size of an MD5 digest, i.e. 16.

MD5_DATA_SIZE [Constant]

The internal block size of MD5. Useful for some special constructions, in particular HMAC-MD5.

void md5_init (*struct md5_ctx* *ctx) [Function]

Initialize the MD5 state.

void md5_update (*struct md5_ctx* *ctx, *unsigned* length, *const uint8_t* *data) [Function]

Hash some more data.

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

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `MD5_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `md5_init`.

The normal way to use MD5 is to call the functions in order: First `md5_init`, then `md5_update` zero or more times, and finally `md5_digest`. After `md5_digest`, the context is reset to its initial state, so you can start over calling `md5_update` to hash new data.

To start over, you can call `md5_init` at any time.

6.1.2.2 MD2

MD2 is another hash function of Ronald Rivest's, described in RFC 1319. It outputs message digests of 128 bits, or 16 octets. Nettle defines MD2 in `<nettle/md2.h>`.

`struct md2_ctx` [Context struct]

`MD2_DIGEST_SIZE` [Constant]

The size of an MD2 digest, i.e. 16.

`MD2_DATA_SIZE` [Constant]

The internal block size of MD2.

`void md2_init (struct md2_ctx *ctx)` [Function]

Initialize the MD2 state.

`void md2_update (struct md2_ctx *ctx, unsigned length, const uint8_t *data)` [Function]

Hash some more data.

`void md2_digest (struct md2` [Function]

6.1.2.4 RIPEMD160

RIPEMD160 is a hash function designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel, as a strengthened version of RIPEMD (which, like MD4 and MD5, fails the collision-resistance requirement). It produces message digests of 160 bits, or 20 octets. Nettle defined RIPEMD160 in `nettle/ripemd160.h`.

`struct ripemd160_ctx` [Context struct]

`RIPEMD160_DIGEST_SIZE` [Constant]
: The size of a RIPEMD160 digest, i.e. 20.

`RIPEMD160_DATA_SIZE` [Constant]
: The internal block size of RIPEMD160.

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

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

`void ripemd160_digest (struct ripemd160_ctx *ctx, unsigned length, uint8_t *digest)` [Function]
: Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `RIPEMD160_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

  This function also resets the context in the same way as `ripemd160_init`.

6.1.2.5 SHA1

SHA1 is a hash function specified by FIPS 180-1, The U.S. National Institute for Standards and Technology. It outputs hash values of 160 bits, or 20 octets. Nettle defines SHA1 in `<nettle/sha1.h>` (and in `<nettle/sha.h>`, for backwards compatibility).

`struct sha1_ctx` [Context struct]

`SHA1_DIGEST_SIZE` [Constant]
: The size of a SHA1 digest, i.e. 20.

`SHA1_DATA_SIZE` [Constant]
: The internal block size of SHA1. Useful for some special constructions, in particular HMAC-SHA1.

`void sha1_init (struct sha1_ctx *ctx)` [Function]
: Initialize the SHA1 state.

`void sha1_update (struct sha1_ctx *ctx, unsigned length, const uint8_t *data)` [Function]
: Hash some more data.

`void sha1_digest` (*struct sha1_ctx *ctx, unsigned length, uint8_t *digest*) [Function]

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA1_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `sha1_init`.

6.1.2.6 GOSTHASH94

The GOST94 or GOST R 34.11-94 hash algorithm is a Soviet-era algorithm used in Russian government standards (see [GOST94], page 57). It outputs message digests of 256 bits, or 32 octets. Nettle defines GOSTHASH94 in `<nettle/gosthash94.h>`.

`struct gosthash94_ctx` [Context struct]

`GOSTHASH94_DIGEST_SIZE` [Constant]

The size of a GOSTHASH94 digest, i.e. 32.

`GOSTHASH94_DATA_SIZE` [Constant]

The internal block size of GOSTHASH94, i.e., 32.

`void gosthash94_init` (*struct gosthash94_ctx *ctx*) [Function]

Initialize the GOSTHASH94 state.

`void gosthash94_update` (*struct gosthash94_ctx *ctx, unsigned length, const uint8_t *data*) [Function]

Hash some more data.

`void gosthash94_digest` (*struct gosthash94_ctx *ctx, unsigned length, uint8_t *digest*) [Function]

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `GOSTHASH94_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `gosthash94_init`.

6.1.3 The nettle_hash abstraction

Nettle includes a struct including information about the supported hash functions. It is defined in `<nettle/nettle-meta.h>`, and is used by Nettle's implementation of HMAC (see Section 6.4 [Keyed hash functions], page 34).

`struct nettle_hash` *name context_size digest_size block_size init update digest* [Meta struct]

The last three attributes are function pointers, of types `nettle_hash_init_func`, `nettle_hash_update_func`, and `nettle_hash_digest_func`. The first argument to these functions is `void *` pointer to a context struct, which is of size `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]

These are all the hash functions that Nettle implements.

Nettle also exports a list of all these hashes.

`struct nettle_hash ** nettle_hashes` [Constant Array]

This list can be used to dynamically enumerate or search the supported algorithms. NULL-terminated.

6.2 Cipher functions

A *cipher* is a function that takes a message or *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 matching pairs of plaintext and ciphertext, it should be hard to find the key.

There are two main classes of ciphers: Block ciphers and stream ciphers.

A block cipher can process data only in fixed size chunks, called *blocks*. Typical block sizes are 8 or 16 octets. To encrypt arbitrary messages, you usually have to pad it to an integral number of blocks, split it into blocks, and then process each block. The simplest way is to process one block at a time, independent of each other. That mode of operation is called *ECB*, Electronic Code Book mode. However, using ECB is usually a bad idea. For a start, plaintext blocks that are equal are transformed to ciphertext blocks that are equal; that leaks information about the plaintext. Usually you should apply the cipher is some "feedback mode", *CBC* (Cipher Block Chaining) and *CTR* (Counter mode) being two of of the most popular. See See Section 6.3 [Cipher modes], page 28, for information on how to apply CBC and CTR with Nettle.

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

Caution: The first rule for this kind of cipher is the same as for a One Time Pad: *never* ever use the same key twice.

A common misconception is that encryption, by itself, implies authentication. Say that you and a friend share a secret key, and you receive an encrypted message. You apply the key, and get a plaintext message that makes sense to you. Can you then be sure that it really was your friend that wrote the message you're reading? The answer is no. For example, if you were using a block cipher in ECB mode, an attacker may pick up the message on its way, and reorder, delete or repeat some of the blocks. Even if the attacker can't decrypt the message, he 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 rather than ECB, or are using a stream cipher, the possibilities for this sort of attack are different, but the attacker can still make predictable changes to the message.

It is recommended to *always* use an authentication mechanism in addition to encrypting the messages. Popular choices are Message Authentication Codes like HMAC-SHA1 (see Section 6.4 [Keyed hash functions], page 34), or digital signatures like RSA.

Some ciphers have so called "weak keys", keys that results in undesirable structure after the key setup processing, and should be avoided. In Nettle, most key setup functions have no return value, but for ciphers with weak keys, the return value indicates whether or not the given key is weak. For good keys, key setup returns 1, and for weak keys, it returns 0. When possible, avoid algorithms that have weak keys. There are several good ciphers that don't have any weak keys.

To encrypt a message, you first initialize a cipher context for encryption or decryption with a particular key. You then use the context to process plaintext or ciphertext messages. The initialization is known as *key setup*. With Nettle, it is recommended to use each context struct for only one direction, even if some of the ciphers use a single key setup function that can be used for both encryption and decryption.

6.2.1 AES

AES is a block cipher, specified by NIST as a replacement for the older DES standard. The standard is the result of a competition between cipher designers. The winning design, also known as RIJNDAEL, was constructed by Joan Daemen and Vincent Rijnmen.

Like all the AES candidates, the winning design uses a block size of 128 bits, or 16 octets, and variable key-size, 128, 192 and 256 bits (16, 24 and 32 octets) being the allowed key sizes. It does not have any weak keys. Nettle defines AES in `<nettle/aes.h>`.

`struct aes_ctx` [Context struct]

`AES_BLOCK_SIZE` [Constant]

The AES block-size, 16.

`AES_MIN_KEY_SIZE` [Constant]

`AES_MAX_KEY_SIZE` [Constant]

`AES_KEY_SIZE` [Constant]

Default AES key size, 32.

`void aes_set_encrypt_key` (*struct aes_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*key`) [Function]

`void aes_set_decrypt_key` (*struct aes_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*key`) [Function]

Initialize the cipher, for encryption or decryption, respectively.

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

Given a context *src* initialized for encryption, initializes the context struct *dst* for decryption, using the same key. If the same context struct is passed for both `src` and `dst`, it is converted in place. Calling `aes_set_encrypt_key` and `aes_invert_key` is more efficient than calling `aes_set_encrypt_key` and `aes_set_decrypt_key`. This function is mainly useful for applications which needs to both encrypt and decrypt using the *same* key.

void **aes_encrypt** (*struct aes_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*dst`, *const uint8_t* `*src`) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

void **aes_decrypt** (*struct aes_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*dst`, *const uint8_t* `*src`) [Function]

Analogous to `aes_encrypt`

6.2.2 ARCFOUR

ARCFOUR is a stream cipher, also known under the trade marked name RC4, and it is one of the fastest ciphers around. A problem is that the key setup of ARCFOUR is quite weak, you should never use keys with structure, keys that are ordinary passwords, or sequences of keys like "secret:1", "secret:2", ... If you have keys that don't look like random bit strings, and you want to use ARCFOUR, always hash the key before feeding it to ARCFOUR. Furthermore, the initial bytes of the generated key stream leak information about the key; for this reason, it is recommended to discard the first 512 bytes of the key stream.

```
/* A more robust key setup function for ARCFOUR */
void
arcfour_set_key_hashed(struct arcfour_ctx *ctx,
                       unsigned 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);
}
```

Nettle defines ARCFOUR in `<nettle/arcfour.h>`.

struct **arcfour_ctx** [Context struct]

ARCFOUR_MIN_KEY_SIZE [Constant]

Minimum key size, 1.

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, *unsigned* length, *const uint8_t* *key) [Function]

Initialize the cipher. The same function is used for both encryption and decryption.

void **arcfour_crypt** (*struct arcfour_ctx* *ctx, *unsigned* length, *uint8_t* *dst, *const uint8_t* *src) [Function]

Encrypt some data. The same function is used for both encryption and decryption. Unlike the block ciphers, this function modifies the context, so you can split the data into arbitrary chunks and encrypt them one after another. The result is the same as if you had called `arcfour_crypt` only once with all the data.

6.2.3 ARCTWO

ARCTWO (also known as the trade marked name RC2) is a block cipher specified in RFC 2268. Nettle also include a variation of the ARCTWO set key operation that lack one step, to be compatible with the reverse engineered RC2 cipher description, as described in a Usenet post to `sci.crypt` by Peter Gutmann.

ARCTWO uses a block size of 64 bits, and variable key-size ranging from 1 to 128 octets. Besides the key, ARCTWO also has a second parameter to key setup, the number of effective key bits, `ekb`. This parameter can be used to artificially reduce the key size. In practice, `ekb` is usually set equal to the input key size. Nettle defines ARCTWO in `<nettle/arctwo.h>`.

We do not recommend the use of ARCTWO; the Nettle implementation is provided primarily for interoperability with existing applications and standards.

struct **arctwo_ctx** [Context struct]

ARCTWO_BLOCK_SIZE [Constant]

The ARCTWO block-size, 8.

ARCTWO_MIN_KEY_SIZE [Constant]

ARCTWO_MAX_KEY_SIZE [Constant]

ARCTWO_KEY_SIZE [Constant]

Default ARCTWO key size, 8.

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

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

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

Initialize the cipher. The same function is used for both encryption and decryption. The first function is the most general one, which lets you provide both the variable size key, and the desired effective key size (in bits). The maximum value for *ekb* is 1024, and for convenience, `ekb = 0` has the same effect 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 (*s c arc o_c* ***ctx***, *ns gn d* ***length***, [Function]
*n 8_* ***dst***, *cons n 8_* ***src***)

Encryption function. *ng* must be an integral multiple of the block size. If it is more than one block the data is processed in ECB mode. `src` and `dst` may be equal but they must not overlap in any other way.

void arctwo_decrypt (*s c arc o_c* ***ctx***, *ns gn d* ***length***, [Function]
*n 8_* ***dst***, *cons n 8_* ***src***)

Analogous to `arctwo_encrypt`

6.2.4 BLOWFISH

BLOWFISH is a block cipher designed by Bruce Schneier. It uses a block size of 64 bits 8 octets, and a variable key size up to 448 bits. It has some weak keys. Nettle de nes BLOWFISH in `<nettle/blowfish.h>`.

`struct blowfish_ctx` [Context struct]

`BLOWFISH_BLOCK_SIZE` [Constant]

The BLOWFISH block-size 8.

`BLOWFISH_MIN_KEY_SIZE` [Constant]

Minimum BLOWFISH key size 8.

`BLOWFISH_MAX_KEY_SIZE` [Constant]

Maximum BLOWFISH key size 56.

`BLOWFISH_KEY_SIZE` [Constant]

Default BLOWFISH key size 6.

int blowfish_set_key (*s c b o s _c* ***ctx***, *ns gn d* ***length***, [Function]
*cons n 8_* ***key***)

Initialize the cipher. The same function is used for both encryption and decryption. Checks for weak keys returning for good keys and for weak keys. Applications that don t care about weak keys can ignore the return value.

`blowfish_encrypt` or `blowfish_decrypt` with a weak key will crash with an assert violation.

void blowfish_encrypt (*s c b o s _c* ***ctx***, *ns gn d* ***length***, [Function]
*n 8_* ***dst***, *cons n 8_* ***src***)

Encryption function. *ng* must be an integral multiple of the block size. If it is more than one block the data is processed in ECB mode. `src` and `dst` may be equal but they must not overlap in any other way.

void blowfish_decrypt (*s c b o s _c* ***ctx***, *ns gn d* ***length***, [Function]
*n 8_* ***dst***, *cons n 8_* ***src***)

Analogous to `blowfish_encrypt`

6.2.5 Camellia

Camellia is a block cipher developed by Mitsubishi and Nippon Telegraph and Telephone Corporation, described in RFC 3713, and recommended by some Japanese and European authorities as an alternative to AES. The algorithm is patented. The implementation in Nettle is derived from the implementation released by NTT under the GNU LGPL (v2 or later), and relies on the implicit patent license of the LGPL. There is also a statement of royalty-free licensing for Camellia at `http://www.ntt.co.jp/news/news01e/0104/010417.html`, but this statement has some limitations which seem problematic for free software.

Camellia uses a the same block size and key sizes as AES: The block size is 128 bits (16 octets), and the supported key sizes are 128, 192, and 256 bits. Nettle defines Camellia in `<nettle/camellia.h>`.

`struct camellia_ctx` [Context struct]

`CAMELLIA_BLOCK_SIZE` [Constant]
The CAMELLIA block-size, 16.

`CAMELLIA_MIN_KEY_SIZE` [Constant]

`CAMELLIA_MAX_KEY_SIZE` [Constant]

`CAMELLIA_KEY_SIZE` [Constant]
Default CAMELLIA key size, 32.

`void camellia_set_encrypt_key (struct camellia_ctx *ctx, unsigned length, const uint8_t *key)` [Function]
`void camellia_set_decrypt_key (struct camellia_ctx *ctx, unsigned length, const uint8_t *key)` [Function]
Initialize the cipher, for encryption or decryption, respectively.

`void camellia_invert_key (struct camellia_ctx *dst, const struct camellia_ctx *src)` [Function]
Given a context *src* initialized for encryption, initializes the context struct *dst* for decryption, using the same key. If the same context struct is passed for both `src` and `dst`, it is converted in place. Calling `camellia_set_encrypt_key` and `camellia_invert_key` is more efficient than calling `camellia_set_encrypt_key` and `camellia_set_decrypt_key`. This function is mainly useful for applications which needs to both encrypt and decrypt using the *same* key.

`void camellia_crypt (struct camellia_ctx *ctx, unsigned length, uint8_t *dst, const uint8_t *src)` [Function]
The same function is used for both encryption and decryption. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

6.2.6 CAST128

CAST-128 is a block cipher, specified in RFC 2144. It uses a 64 bit (8 octets) block size, and a variable key size of up to 128 bits. Nettle defines cast128 in `<nettle/cast128.h>`.

`struct cast128_ctx` [Context struct]

`CAST128_BLOCK_SIZE` [Constant]

The CAST128 block-size, 8.

`CAST128_MIN_KEY_SIZE` [Constant]

Minimum CAST128 key size, 5.

`CAST128_MAX_KEY_SIZE` [Constant]

Maximum CAST128 key size, 16.

`CAST128_KEY_SIZE` [Constant]

Default CAST128 key size, 16.

`void cast128_set_key` (*struct cast128_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*key`) [Function]

Initialize the cipher. The same function is used for both encryption and decryption.

`void cast128_encrypt` (*struct cast128_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*dst`, *const uint8_t* `*src`) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void cast128_decrypt` (*struct cast128_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*dst`, *const uint8_t* `*src`) [Function]

Analogous to `cast128_encrypt`

6.2.7 DES

DES is the old Data Encryption Standard, specified by NIST. It uses a block size of 64 bits (8 octets), and a key size of 56 bits. However, the key bits are distributed over 8 octets, where the least significant bit of each octet may be used for parity. A common way to use DES is to generate 8 random octets in some way, then set the least significant bit of each octet to get odd parity, and initialize DES with the resulting key.

The key size of DES is so small that keys can be found by brute force, using specialized hardware or lots of ordinary work stations in parallel. One shouldn't be using plain DES at all today; if one uses DES at all one should be using "triple DES", see DES3 below.

DES also has some weak keys. Nettle defines DES in `<nettle/des.h>`.

`struct des_ctx` [Context struct]

`DES_BLOCK_SIZE` [Constant]

The DES block-size, 8.

`DES_KEY_SIZE` [Constant]

DES key size, 8.

`int des_set_key` (*struct des_ctx* `*ctx`, *const uint8_t* `*key`) [Function]

Initialize the cipher. The same function is used for both encryption and decryption. Parity bits are ignored. Checks for weak keys, returning 1 for good keys and 0 for weak keys. Applications that don't care about weak keys can ignore the return value.

void des_encrypt (*struct des_ctx* *ctx, *unsigned* length, *uint8_t* *dst, *const uint8_t* *src) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. src and dst may be equal, but they must not overlap in any other way.

void des_decrypt (*struct des_ctx* *ctx, *unsigned* length, *uint8_t* *dst, *const uint8_t* *src) [Function]

Analogous to des_encrypt

int des_check_parity (*unsigned* length, *const uint8_t* *key); [Function]

Checks that the given key has correct, odd, parity. Returns 1 for correct parity, and 0 for bad parity.

void des_fix_parity (*unsigned* length, *uint8_t* *dst, *const uint8_t* *src) [Function]

Adjusts the parity bits to match DES's requirements. You need this function if you have created a random-looking string by a key agreement protocol, and want to use it as a DES key. *dst* and *src* may be equal.

6.2.8 DES3

The inadequate key size of DES has already been mentioned. One way to increase the key size is to pipe together several DES boxes with independent keys. It turns out that using two DES ciphers is not as secure as one might think, even if the key size of the combination is a respectable 112 bits.

The standard way to increase DES's key size is to use three DES boxes. The mode of operation is a little peculiar: the middle DES box is wired in the reverse direction. To encrypt a block with DES3, you encrypt it using the first 56 bits of the key, then *decrypt* it using the middle 56 bits of the key, and finally encrypt it again using the last 56 bits of the key. This is known as "ede" triple-DES, for "encrypt-decrypt-encrypt".

The "ede" construction provides some backward compatibility, as you get plain single DES simply by feeding the same key to all three boxes. That should help keeping down the gate count, and the price, of hardware circuits implementing both plain DES and DES3.

DES3 has a key size of 168 bits, but just like plain DES, useless parity bits are inserted, so that keys are represented as 24 octets (192 bits). As a 112 bit key is large enough to make brute force attacks impractical, some applications uses a "two-key" variant of triple-DES. In this mode, the same key bits are used for the first and the last DES box in the pipe, while the middle box is keyed independently. The two-key variant is believed to be secure, i.e. there are no known attacks significantly better than brute force.

Naturally, it's simple to implement triple-DES on top of Nettle's DES functions. Nettle includes an implementation of three-key "ede" triple-DES, it is defined in the same place as plain DES, <nettle/des.h>.

struct des3_ctx [Context struct]

DES3_BLOCK_SIZE [Constant]

The DES3 block-size is the same as DES_BLOCK_SIZE, 8.

DES3_KEY_SIZE [Constant]

DES key size 24.

int des3_set_key (*s c des _c* *ctx, *cons n 8_* *key) [Function]

Initialize the cipher. The same function is used for both encryption and decryption. Parity bits are ignored. Checks for weak keys returning if all three keys are good keys and if one or more key is weak. Applications that don t care about weak keys can ignore the return value.

For random-looking strings you can use des_fix_parity to adjust the parity bits before calling des3_set_key.

void des3_encrypt (*s c des _c* *ctx, *ns gned* length, *n 8_* *dst, *cons n 8_* *src) [Function]

Encryption function. *ng* must be an integral multiple of the block size. If it is more than one block the data is processed in ECB mode. src and dst may be equal but they must not overlap in any other way.

void des3_decrypt (*s c des _c* *ctx, *ns gned* length, *n 8_* *dst, *cons n 8_* *src) [Function]

Analogous to des_encrypt

6.2.9 Salsa20

Salsa2 is a fairly recent stream cipher designed by D. J. Bernstein. It is built on the observation that a cryptographic hash function can be used for encryption . Form the hash input from the secret key and a counter xor the hash output and the first block of the plaintext then increment the counter to process the next block similar to CTR mode see see Section 6.3.2 [CTR] page 3 . Bernstein defined an encryption algorithm Snuffle in this way to ridicule United States export restrictions which treated hash functions as nice and harmless but ciphers as dangerous munitions.

Salsa2 uses the same idea but with a new specialized hash function to mix key block counter and a couple of constants. It s also designed for speed on x86_64 it is currently the fastest cipher offered by nettle. It uses a block size of 5 2 bits 64 octets and there are two specified key sizes 28 and 256 bits 6 and 32 octets .

Caution: The hash function used in Salsa2 is *not* directly applicable for use as a general hash function. It s *not* collision resistant if arbitrary inputs are allowed and furthermore the input and output is of fixed size.

When using Salsa2 to process a message one specifies both a key and a *nonce* the latter playing a similar role to the initialization vector IV used with CBC or CTR mode. For this reason Nettle uses the term IV to refer to the Salsa2 nonce. One can use the same key for several messages provided one uses a unique random iv for each message. The iv is 64 bits 8 octets . The block counter is initialized to zero for each message and is also 64 bits 8 octets . Nettle defines Salsa2 in <nettle/salsa20.h>.

struct salsa20_ctx [Context struct]

SALSA20_MIN_KEY_SIZE [Constant]
SALSA20_MAX_KEY_SIZE [Constant]

The two supported key sizes 6 and 32 octets.

SALSA20_KEY_SIZE [Constant]
Recommended key size, 32.

SALSA20_BLOCK_SIZE [Constant]
Salsa20 block size, 64.

SALSA20_IV_SIZE [Constant]
Size of the IV, 8.

`void salsa20_set_key (struct salsa20_ctx *ctx, unsigned length, const uint8_t *key)` [Function]
Initialize the cipher. The same function is used for both encryption and decryption. Before using the cipher, you *must* also call `salsa20_set_iv`, see below.

`void salsa20_set_iv (struct salsa20_ctx *ctx, const uint8_t *iv)` [Function]
Sets the IV. It is always of size `SALSA20_IV_SIZE`, 8 octets. This function also initializes the block counter, setting it to zero.

`void salsa20_crypt (struct salsa20_ctx *ctx, unsigned length, uint8_t *dst, const uint8_t *src)` [Function]
Encrypts or decrypts the data of a message, using salsa20. When a message is encrypted using a sequence of calls to `salsa20_crypt`, all but the last call *must* use a length that is a multiple of `SALSA20_BLOCK_SIZE`.

The full salsa20 cipher uses 20 rounds of mixing. Variants of Salsa20 with fewer rounds are possible, and the 12-round variant is specified by eSTREAM, see `http://www.ecrypt.eu.org/stream/finallist.html`. Nettle calls this variant `salsa20r12`. It uses the same context struct and key setup as the full salsa20 cipher, but a separate function for encryption and decryption.

`void salsa20r12_crypt (struct salsa20_ctx *ctx, unsigned length, uint8_t *dst, const uint8_t *src)` [Function]
Encrypts or decrypts the data of a message, using salsa20 reduced to 12 rounds.

6.2.10 SERPENT

SERPENT is one of the AES finalists, designed by Ross Anderson, Eli Biham and Lars Knudsen. Thus, the interface and properties are similar to AES). One peculiarity is that it is quite pointless to use it with anything but the maximum key size, smaller keys are just padded to larger ones. Nettle defines SERPENT in `<nettle/serpent.h>`.

`struct serpent_ctx` [Context struct]

SERPENT_BLOCK_SIZE [Constant]
The SERPENT block-size, 16.

SERPENT_MIN_KEY_SIZE [Constant]
Minimum SERPENT key size, 16.

SERPENT_MAX_KEY_SIZE [Constant]
Maximum SERPENT key size, 32.

SERPENT_KEY_SIZE [Constant]

Default SERPENT key size, 32.

void serpent_set_key (*struct serpent_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

Initialize the cipher. The same function is used for both encryption and decryption.

void serpent_encrypt (*struct serpent_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***dst***, *const uint8_t* ***src***) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

void serpent_decrypt (*struct serpent_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***dst***, *const uint8_t* ***src***) [Function]

Analogous to `serpent_encrypt`

6.2.11 TWOFISH

Another AES finalist, this one designed by Bruce Schneier and others. Nettle defines it in `<nettle/twofish.h>`.

struct twofish_ctx [Context struct]

TWOFISH_BLOCK_SIZE [Constant]

The TWOFISH block-size, 16.

TWOFISH_MIN_KEY_SIZE [Constant]

Minimum TWOFISH key size, 16.

TWOFISH_MAX_KEY_SIZE [Constant]

Maximum TWOFISH key size, 32.

TWOFISH_KEY_SIZE [Constant]

Default TWOFISH key size, 32.

void twofish_set_key (*struct twofish_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

Initialize the cipher. The same function is used for both encryption and decryption.

void twofish_encrypt (*struct twofish_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***dst***, *const uint8_t* ***src***) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

void twofish_decrypt (*struct twofish_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***dst***, *const uint8_t* ***src***) [Function]

Analogous to `twofish_encrypt`

6.2.12 `struct nettle_cipher`

Nettle includes a struct including information about some of the more regular cipher functions. It should be considered a little experimental, but can be useful for applications that need a simple way to handle various algorithms. Nettle defines these structs in `<nettle/nettle-meta.h>`.

`struct nettle_cipher` *name context_size block_size key_size set_encrypt_key set_decrypt_key encrypt decrypt* [Meta struct]

The last four attributes are function pointers, of types `nettle_set_key_func` and `nettle_crypt_func`. The first argument to these functions is a `void *` pointer to a context struct, which is of size `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]

Nettle includes such structs for all the *regular* ciphers, i.e. ones without weak keys or other oddities.

Nettle also exports a list of all these ciphers without weak keys or other oddities.

`struct nettle_cipher ** nettle_ciphers` [Constant Array]

This list can be used to dynamically enumerate or search the supported algorithms. NULL-terminated.

6.3 Cipher modes

Cipher modes of operation specifies the procedure to use when encrypting a message that is larger than the cipher's block size. As explained in See Section 6.2 [Cipher functions], page 7, splitting the message into blocks and processing them independently with the block cipher (Electronic Code Book mode, ECB) leaks information. Besides ECB, Nettle provides three other modes of operation: Cipher Block Chaining (CBC), Counter mode (CTR), and Galois Counter mode (GCM). CBC is widely used, but there are a few subtle issues of information leakage, see, e.g., SSH CBC vulnerability http://www.kb.cert.org/vuls/id/958563 . CTR and GCM were standardized more recently and are believed to be more secure. GCM includes message authentication for the other modes one should always use a MAC see Section 6.4 [Keyed hash functions], page 34 or signature to authenticate the message.

6.3.1 Cipher Block Chaining

When using CBC mode plaintext blocks are not encrypted independently of each other like in Electronic Cook Book mode. Instead when encrypting a block in CBC mode the previous ciphertext block is XORed with the plaintext before it is fed to the block cipher. When encrypting the first block a random block called an , or Initialization Vector is used as the previous ciphertext block". The IV should be chosen randomly but it need not be kept secret and can even be transmitted in the clear together with the encrypted data.

In symbols if `E_k` is the encryption function of a block cipher and `IV` is the initialization vector then `n` plaintext blocks `M_1` . . . `M_n` are transformed into `n` ciphertext blocks `C_1` . . . `C_n` as follows .

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

...

C_n = E_k(C_(n-1) XOR M_n)
```

Nettle's includes two functions for applying a block cipher in Cipher Block Chaining CBC mode one for encryption and one for decryption. These functions uses `void *` to pass cipher contexts around.

void cbc_encrypt (*void *ctx*, *nettle_crypt_func f*, *unsigned block_size*, *uint8_t *iv*, *unsigned length*, *uint8_t *dst*, *const uint8_t *src*) [Function]

void cbc_decrypt (*void *ctx*, *void (*f)()*, *unsigned block_size*, *uint8_t *iv*, *unsigned length*, *uint8_t *dst*, *const uint8_t *src*) [Function]

Applies the encryption or decryption function in CBC mode. The final ciphertext block processed is copied into before returning so that large message be processed be a sequence of calls to `cbc_encrypt`. The function is of type

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

and the `cbc_encrypt` and `cbc_decrypt` functions pass their argument *c* on to .

There are also some macros to help use these functions correctly.

CBC_CTX (*context_type*, *block_size*) [Macro]

Expands to

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

It can be used to define a CBC context struct, either directly,

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

or to give it a struct tag,

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

CBC_SET_IV (*ctx*, *iv*) [Macro]

First argument is a pointer to a context struct as defined by CBC_CTX, and the second is a pointer to an Initialization Vector (IV) that is copied into that context.

CBC_ENCRYPT (*ctx*, *f*, *length*, *dst*, *src*) [Macro]

CBC_DECRYPT (*ctx*, *f*, *length*, *dst*, *src*) [Macro]

A simpler way to invoke cbc_encrypt and cbc_decrypt. The first argument is a pointer to a context struct as defined by CBC_CTX, and the second argument is an encryption or decryption function following Nettle's conventions. The last three arguments define the source and destination area for the operation.

These macros use some tricks to make the compiler display a warning if the types of *f* and *ctx* don't match, e.g. if you try to use an struct aes_ctx context with the des_encrypt function.

6.3.2 Counter mode

Counter mode (CTR) uses the block cipher as a keyed pseudo-random generator. The output of the generator is XORed with the data to be encrypted. It can be understood as a way to transform a block cipher to a stream cipher.

The message is divided into n blocks M_1,... M_n, where M_n is of size m which may be smaller than the block size. Except for the last block, all the message blocks must be of size equal to the cipher's block size.

If E_k is the encryption function of a block cipher, IC is the initial counter, then the n plaintext blocks are transformed into n ciphertext blocks C_1,... C_n as follows:

```
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
```

The IC is the initial value for the counter, it plays a similar role as the IV for CBC. When adding, IC + x, IC is interpreted as an integer, in network byte order. For the last block, E_k(IC + n - 1) [1..m] means that the cipher output is truncated to m bytes.

void ctr_crypt (*void *ctx*, *nettle_crypt_func f*, *unsigned block_size*, *uint8_t *ctr*, *unsigned length*, *uint8_t *dst*, *const uint8_t *src*) [Function]

Applies the encryption function *f* in CTR mode. Note that for CTR mode, encryption and decryption is the same operation, and hence *f* should always be the encryption function for the underlying block cipher.

When a message is encrypted using a sequence of calls to ctr_crypt, all but the last call *must* use a length that is a multiple of the block size.

Like for CBC there are also a couple of helper macros.

`CTR_CTX` (*context_type*, *block_size*) [Macro]

Expands to

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

`CTR_SET_COUNTER` (*ctx*, *iv*) [Macro]

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

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

A simpler way to invoke `ctr_crypt`. The first argument is a pointer to a context struct as defined by `CTR_CTX` and the second argument is an encryption function following Nettle s conventions. The last three arguments define the source and destination area for the operation.

6.3.3 Galois counter mode

Galois counter mode is the combination of counter mode with message authentication based on universal hashing. The main objective of the design is to provide high performance for hardware implementations where other popular MAC algorithms see Section 6.4 [Keyed hash functions], page 34 becomes a bottleneck for high-speed hardware implementations. It was proposed by David A. McGrew and John Viega in 2 5 and recommended by NIST in 2 7 NIST Special Publication 8 -38D `http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`. It is constructed on top of a block cipher which must have a block size of 28 bits.

GCM is applied to messages of arbitrary length. The inputs are

- A key which can be used for many messages.
- An initialization vector IV which *m st* be unique for each message.
- Additional authenticated data which is to be included in the message authentication but not encrypted. May be empty.
- The plaintext. Maybe empty.

The outputs are a ciphertext of the same length as the plaintext and a message digest of length 28 bits. Nettle s support for GCM consists of a low-level general interface some convenience macros and specific functions for GCM using AES as the underlying cipher. These interfaces are defined in `<nettle/gcm.h>`

6.3.3.1 General GCM interface

`struct gcm_key` [Context struct]

Message independent hash sub-key and related tables.

`struct gcm_ctx` [Context struct]

Holds state corresponding to a particular message.

`GCM_BLOCK_SIZE` [Constant]

GCM s block size , 6.

`GCM_IV_SIZE` [Constant]

Recommended size of the IV, 2. Other sizes are allowed.

void **gcm_set_key** (*s c gc _ *`*key`, *o d* `*cipher`, *n _ _ nc* `*f`) [Function]

Initializes . *c* gives a context struct for the underlying cipher which must have been previously initialized for encryption and is the encryption function.

void **gcm_set_iv** (*s c gc _c* `*ctx`, *cons s c gc _* `*key`, *ns gn d* `length`, *cons n 8_* `*iv`) [Function]

Initializes *c* using the given IV. The argument is actually needed only if *ng* ers from `GCM_IV_SIZE`.

void **gcm_update** (*s c gc _c* `*ctx`, *cons s c gc _* `*key`, *ns gn d* `length`, *cons n 8_* `*data`) [Function]

Provides associated data to be authenticated. If used must be called before `gcm_encrypt` or `gcm_decrypt`. All but the last call for each message *m st* use a length that is a multiple of the block size.

void **gcm_encrypt** (*s c gc _c* `*ctx`, *cons s c gc _* `*key` *o d* `*cipher`, *n _ _ nc* `*f`, *ns gn d* `length`, *n 8_* `*dst`, *cons n 8_* `*src`) [Function]

void **gcm_decrypt** (*s c gc _c* `*ctx`, *cons s c gc _* `*key`, *o d* `*cipher`, *n _ _ nc* `*f`, *ns gn d* `length`, *n 8_* `*dst`, *cons n 8_* `*src`) [Function]

Encrypts or decrypts the data of a message. *c* is the context struct for the underlying cipher and is the encryption function. All but the last call for each message *m st* use a length that is a multiple of the block size.

void **gcm_digest** (*s c gc _c* `*ctx`, *cons s c gc _* `*key`, *o d* `*cipher`, *n _ _ nc* `*f`, *ns gn d* `length`, *n 8_* `*digest`) [Function]

Extracts the message digest also known authentication tag" . This is the final operation when processing a message. *ng* is usually equal to `GCM_BLOCK_SIZE` but if you provide a smaller value only the first *ng* octets of the digest are written.

To encrypt a message using GCM first initialize a context for the underlying block cipher with a key to use for encryption. Then call the above functions in the following order . `gcm_set_key` `gcm_set_iv` `gcm_update` `gcm_encrypt` `gcm_digest`. The decryption procedure is analogous just calling `gcm_decrypt` instead of `gcm_encrypt` note that GCM decryption still uses the encryption function of the underlying block cipher . To process a new message using the same key call `gcm_set_iv` with a new iv.

6.3.3.2 GCM helper macros

The following macros are defined.

GCM_CTX (*context_type*) [Macro]

This defines an all-in-one context struct, including the context of the underlying cipher, the hash sub-key, and the per-message state. It expands to

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

Example use:

```
struct gcm_aes_ctx GCM_CTX(struct aes_ctx);
```

The following macros operate on context structs of this form.

GCM_SET_KEY (*ctx*, *set_key*, *encrypt*, *length*, *data*) [Macro]

First argument, *ctx*, is a context struct as defined by GCM_CTX. *set_key* and *encrypt* are functions for setting the encryption key and for encrypting data using the underlying cipher. *length* and *data* give the key.

GCM_SET_IV (*ctx*, *length*, *data*) [Macro]

First argument is a context struct as defined by GCM_CTX. *length* and *data* give the initialization vector (IV).

GCM_UPDATE (*ctx*, *length*, *data*) [Macro]

Simpler way to call gcm_update. First argument is a context struct as defined by GCM_CTX

GCM_ENCRYPT (*ctx*, *encrypt*, *length*, *dst*, *src*) [Macro]
GCM_DECRYPT (*ctx*, *encrypt*, *length*, *dst*, *src*) [Macro]
GCM_DIGEST (*ctx*, *encrypt*, *length*, *digest*) [Macro]

Simpler way to call gcm_encrypt, gcm_decrypt or gcm_digest. First argument is a context struct as defined by GCM_CTX. Second argument, *encrypt*, is a pointer to the encryption function of the underlying cipher.

6.3.3.3 GCM-AES interface

The following functions implement the common case of GCM using AES as the underlying cipher.

struct gcm_aes_ctx [Context struct]

The context struct, defined using GCM_CTX.

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

Initializes *ctx* using the given key. All valid AES key sizes can be used.

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

Initializes the per-message state, using the given IV.

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

Provides associated data to be authenticated. If used, must be called before `gcm_aes_encrypt` or `gcm_aes_decrypt`. All but the last call for each message *must* use a length that is a multiple of the block size.

void gcm_aes_encrypt (*struct gcm_aes_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***dst**, *const uint8_t* ***src**) [Function]

void gcm_aes_decrypt (*struct gcm_aes_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***dst**, *const uint8_t* ***src**) [Function]

Encrypts or decrypts the data of a message. All but the last call for each message *must* use a length that is a multiple of the block size.

void gcm_aes_digest (*struct gcm_aes_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***digest**) [Function]

Extracts the message digest (also known "authentication tag"). This is the final operation when processing a message. *length* is usually equal to `GCM_BLOCK_SIZE`, but if you provide a smaller value, only the first *length* octets of the digest are written.

6.4 Keyed Hash Functions

A *keyed hash function*, or *Message Authentication Code* (MAC) is a function that takes a key and a message, and produces fixed size MAC. It should be hard to compute a message and a matching MAC without knowledge of the key. It should also be hard to compute the key given only messages and corresponding MACs.

Keyed hash functions are useful primarily for message authentication, when Alice and Bob shares a secret: The sender, Alice, computes the MAC and attaches it to the message. The receiver, Bob, also computes the MAC of the message, using the same key, and compares that to Alice's value. If they match, Bob can be assured that the message has not been modified on its way from Alice.

However, unlike digital signatures, this assurance is not transferable. Bob can't show the message and the MAC to a third party and prove that Alice sent that message. Not even if he gives away the key to the third party. The reason is that the *same* key is used on both sides, and anyone knowing the key can create a correct MAC for any message. If Bob believes that only he and Alice knows the key, and he knows that he didn't attach a MAC to a particular message, he knows it must be Alice who did it. However, the third party can't distinguish between a MAC created by Alice and one created by Bob.

Keyed hash functions are typically a lot faster than digital signatures as well.

6.4.1 HMAC

One can build keyed hash functions from ordinary hash functions. Older constructions simply concatenate secret key and message and hashes that, but such constructions have weaknesses. A better construction is HMAC, described in *RFC 2104*.

For an underlying hash function `H`, with digest size `l` and internal block size `b`, HMAC-H is constructed as follows: From a given key `k`, two distinct subkeys `k_i` and `k_o` are constructed, both of length `b`. The HMAC-H of a message `m` is then computed as `H(k_o | H(k_i | m))`, where `|` denotes string concatenation.

HMAC keys can be of any length, but it is recommended to use keys of length `l`, the digest size of the underlying hash function `H`. Keys that are longer than `b` are shortened to length `l` by hashing with `H`, so arbitrarily long keys aren't very useful.

Nettle's HMAC functions are defined in `<nettle/hmac.h>`. There are abstract functions that use a pointer to a `struct nettle_hash` to represent the underlying hash function and `void *` pointers that point to three different context structs for that hash function. There are also concrete functions for HMAC-MD5, HMAC-RIPEMD 6, HMAC-SHA, HMAC-SHA256, and56

`HMAC_DIGEST` (*ctx*, *H*, *length*, *digest*) [Macro]

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. The last two arguments specify where the digest is written.

Note that there is no `HMAC_UPDATE` macro; simply call `hmac_update` function directly, or the update function of the underlying hash function.

6.4.2 Concrete HMAC functions

Now we come to the specialized HMAC functions, which are easier to use than the general HMAC functions.

6.4.2.1 HMAC-MD5

`struct hmac_md5_ctx` [Context struct]

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

Initializes the context with the key.

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

Process some more data.

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

Extracts the MAC, writing it to *digest*. *length* may be smaller than `MD5_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.2.2 HMAC-RIPEMD160

`struct hmac_ripemd160_ctx` [Context struct]

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

Initializes the context with the key.

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

Process some more data.

`void hmac_ripemd160_digest` (*struct hmac_ripemd160_ctx *ctx*, *unsigned length*, *uint8_t *digest*) [Function]

Extracts the MAC, writing it to *digest*. *length* may be smaller than `RIPEMD160_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.2.3 HMAC-SHA1

`struct hmac_sha1_ctx` [Context struct]

`void hmac_sha1_set_key` (*struct hmac_sha1_ctx* `*ctx`, *unsigned* `key_length`, *const uint8_t* `*key`) [Function]

Initializes the context with the key.

`void hmac_sha1_update` (*struct hmac_sha1_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*data`) [Function]

Process some more data.

`void hmac_sha1_digest` (*struct hmac_sha1_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*digest`) [Function]

Extracts the MAC, writing it to *digest*. *length* may be smaller than `SHA1_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.2.4 HMAC-SHA256

`struct hmac_sha256_ctx` [Context struct]

`void hmac_sha256_set_key` (*struct hmac_sha256_ctx* `*ctx`, *unsigned* `key_length`, *const uint8_t* `*key`) [Function]

Initializes the context with the key.

`void hmac_sha256_update` (*struct hmac_sha256_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*data`) [Function]

Process some more data.

`void hmac_sha256_digest` (*struct hmac_sha256_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*digest`) [Function]

Extracts the MAC, writing it to *digest*. *length* may be smaller than `SHA256_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.2.5 HMAC-SHA512

`struct hmac_sha512_ctx` [Context struct]

`void hmac_sha512_set_key` (*struct hmac_sha512_ctx* `*ctx`, *unsigned* `key_length`, *const uint8_t* `*key`) [Function]

Initializes the context with the key.

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

Process some more data.

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

Extracts the MAC, writing it to *digest*. *length* may be smaller than `SHA512_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.3 UMAC

UMAC is a message authentication code based on universal hashing, and designed for high performance on modern processors (in contrast to GCM, See Section 6.3.3 [GCM], page 31, which is designed primarily for hardware performance). On processors with good integer multiplication performance, it can be 10 times faster than SHA256 and SHA512. UMAC is specified in RFC 4418.

The secret key is always 128 bits (16 octets). The key is used as an encryption key for the AES block cipher. This cipher is used in counter mode to generate various internal

`UMAC128_DIGEST_SIZE` [Constant]

The size of an UMAC128 digest, 16.

`UMAC128_DATA_SIZE` [Constant]

The internal block size of UMAC.

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]

These functions initialize the UMAC context struct. They also initialize the nonce to zero (with length 16, for auto-increment).

void **umac32_set_nonce** (*struct umac32_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*nonce`) [Function]
void **umac64_set_nonce** (*struct umac64_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*nonce`) [Function]
void **umac96_set_nonce** (*struct umac96_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*nonce`) [Function]
void **umac128_set_nonce** (*struct umac128_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*nonce`) [Function]

Sets the nonce to be used for the next message. In general, nonces should be set before processing of the message. This is not strictly required for UMAC (the nonce only affects the final processing generating the digest), but it is nevertheless recommended that this function is called *before* the first `_update` call for the message.

void **umac32_update** (*struct umac32_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*data`) [Function]
void **umac64_update** (*struct umac64_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*data`) [Function]
void **umac96_update** (*struct umac96_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*data`) [Function]
void **umac128_update** (*struct umac128_ctx* `*ctx`, *unsigned* `length`, *const uint8_t* `*data`) [Function]

These functions are called zero or more times to process the message.

void **umac32_digest** (*struct umac32_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*digest`) [Function]
void **umac64_digest** (*struct umac64_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*digest`) [Function]
void **umac96_digest** (*struct umac96_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*digest`) [Function]
void **umac128_digest** (*struct umac128_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*digest`) [Function]

Extracts the MAC of the message, writing it to *digest*. *length* is usually equal to the specified output size, but if you provide a smaller value, only the first *length* octets of the MAC are written. These functions reset the context for processing of a new message with the same key. The nonce is incremented as described above, the new value is used unless you call the `_set_nonce` function explicitly for each message.

6.5 Key derivation Functions

A *key derivation function* (KDF) is a function that from a given symmetric key derives other symmetric keys. A sub-class 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 it and protect against certain pre-computation attacks by using salting and expensive computation.

6.5.1 PBKDF2

The most well known PBKDF is the `PKCS #5 PBKDF2` described in *RFC 2898* which uses a pseudo-random function such as HMAC-SHA1.

Nettle's PBKDF2 functions are defined in `<nettle/pbkdf2.h>`. There is an abstract function that operate on any PRF implemented via the `nettle_hash_update_func`, `nettle_hash_digest_func` interfaces. There is also helper macros and concrete functions PBKDF2-HMAC-SHA1 and PBKDF2-HMAC-SHA256. First, the abstract function:

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

Derive symmetric key from a password according to PKCS #5 PBKDF2. The PRF is assumed to have been initialized and this function will call the *update* and *digest* functions passing the *mac_ctx* context parameter as an argument in order to compute digest of size *digest_size*. Inputs are the salt *salt* of length *salt_length*, the iteration counter *iterations* (> 0), and the desired derived output length *length*. The output buffer is *dst* which must have room for at least *length* octets.

Like for CBC and HMAC, there is a macro to help use the function correctly.

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

ctx is a pointer to a context struct passed to the *update* and *digest* functions (of the types `nettle_hash_update_func` and `nettle_hash_digest_func` respectively) to implement the underlying PRF with digest size of *digest_size*. Inputs are the salt *salt* of length *salt_length*, the iteration counter *iterations* (> 0), and the desired derived output length *length*. The output buffer is *dst* which must have room for at least *length* octets.

6.5.2 Concrete PBKDF2 functions

Now we come to the specialized PBKDF2 functions, which are easier to use than the general PBKDF2 function.

6.5.2.1 PBKDF2-HMAC-SHA1

`void pbkdf2_hmac_sha1 (unsigned key_length, const uint8_t *key, unsigned iterations, unsigned salt_length, const uint8_t *salt, unsigned length, uint8_t *dst)` [Function]

PBKDF2 with HMAC-SHA1. Derive *length* bytes of key into buffer *dst* using the password *key* of length *key_length* and salt *salt* of length *salt_length*, with iteration counter *iterations* (> 0). The output buffer is *dst* which must have room for at least *length* octets.

6.5.2.2 PBKDF2-HMAC-SHA256

```
void pbkdf2_hmac_sha256 (unsigned key_length, const uint8_t *key,      [Function]
        unsigned iterations, unsigned salt_length, const uint8_t *salt,
        unsigned length, uint8_t *dst)
```

PBKDF2 with HMAC-SHA256. Derive *length* bytes of key into buffer *dst* using the password *key* of length *key_length* and salt *salt* of length *salt_length*, with iteration counter *iterations* (> 0). The output buffer is *dst* which must have room for at least *length* octets.

6.6 Public-key algorithms

Nettle uses GMP, the GNU bignum library, for all calculations with large numbers. In order to use the public-key features of Nettle, you must install GMP, at least version 3.0, before compiling Nettle, and you need to link your programs with `-lhogweed -lnettle -lgmp`.

The concept of *Public-key* encryption and digital signatures was discovered by Whitfield Diffie and Martin E. Hellman and described in a paper 1976. In traditional, "symmetric", cryptography, sender and receiver share the same keys, and these keys must be distributed in a secure way. And if there are many users or entities that need to communicate, each *pair* needs a shared secret key known by nobody else.

Public-key cryptography uses trapdoor one-way functions. A *one-way function* is a function `F` such that it is easy to compute the value `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 certain groups.

A *trapdoor one-way function* is a function `F` that is one-way, unless one knows some secret information about `F`. If one knows the secret, it is easy to compute both `F` and it's inverse. If this sounds strange, look at the RSA example below.

Two important uses for one-way functions with trapdoors are public-key encryption, and digital signatures. The public-key encryption functions in Nettle are not yet documented; the rest of this chapter is about digital signatures.

To use a digital signature algorithm, one must first create a *key-pair*: A public key and a corresponding private key. The private key is used to sign messages, while the public key is used for verifying that that signatures and messages match. Some care must be taken when distributing the public key; it need not be kept secret, but if a bad guy is able to replace it (in transit, or in some user's list of known public keys), bad things may happen.

There are two operations one can do with the keys. The signature operation takes a message and a private key, and creates a signature for the message. A signature is some string of bits, usually at most a few thousand bits or a few hundred octets. Unlike paper-and-ink signatures, the digital signature depends on the message, so one can't cut it out of context and glue it to a different message.

The verification operation takes a public key, a message, and a string that is claimed to be a signature on the message, and returns true or false. If it returns true, that means that the three input values matched, and the verifier can be sure that someone went through with the signature operation on that very message, and that the “someone” also knows the private key corresponding to the public key.

The desired properties of a digital signature algorithm are as follows: Given the public key and pairs of messages and valid signatures on them, it should be hard to compute the private key, and it should also be hard to create a new message and signature that is accepted by the verification operation.

Besides signing meaningful messages, digital signatures can be used for authorization. A server can be configured with a public key, such that any client that connects to the service is given a random nonce message. If the server gets a reply with a correct signature matching the nonce message and the configured public key, the client is granted access. So the configuration of the server can be understood as “grant access to whoever knows the private key corresponding to this particular public key, and to no others”.

6.6.1 RSA

The RSA algorithm was the first practical digital signature algorithm that was constructed. It was described 1978 in a paper by Ronald Rivest, Adi Shamir and L.M. Adleman, and the technique was also patented in the USA in 1983. The patent expired on September 20, 2000, and since that day, RSA can be used freely, even in the USA.

It's remarkably simple to describe the trapdoor function behind RSA. The “one-way”-function used is

```
F(x) = x^e mod n
```

I.e. raise x to the e'th power, while discarding all multiples of `n`. The pair of numbers `n` and `e` is the public key. `e` can be quite small, even `e = 3` has been used, although slightly larger numbers are recommended. `n` should be about 1000 bits or larger.

If `n` is large enough, and properly chosen, the inverse of F, the computation of e'th roots modulo `n`, is very difficult. But, where's the trapdoor?

Let's first look at how RSA key-pairs are generated. First `n` is chosen as the product of two large prime numbers `p` and `q` of roughly the same size (so if `n` is 1000 bits, `p` and `q` are about 500 bits each). One also computes the number `phi = (p-1)(q-1)`, in mathematical speak, `phi` is the order of the multiplicative group of integers modulo n.

Next, `e` is chosen. It must have no factors in common with `phi` (in particular, it must be odd), but can otherwise be chosen more or less randomly. `e = 65537` is a popular choice, because it makes raising to the e'th power particularly efficient, and being prime, it usually has no factors common with `phi`.

Finally, a number `d`, `d < n` is computed such that `e d mod phi = 1`. It can be shown that such a number exists (this is why `e` and `phi` must have no common factors), and that for all x,

```
(x^e)^d mod n = x^(ed) mod n = (x^d)^e mod n = x
```

Using Euclid's algorithm, `d` can be computed quite easily from `phi` and `e`. But it is still hard to get `d` without knowing `phi`, which depends on the factorization of `n`.

So `d` is the trapdoor: if we know `d` and `y = F(x)`, we can recover x as `y^d mod n`. `d` is also the private half of the RSA key-pair.

The most common signature operation for RSA is defined in *PKCS#1*, a specification by RSA Laboratories. The message to be signed is first hashed using a cryptographic hash function (e.g. MD5 or SHA1). Next, some padding, the ASN.1 "Algorithm Identifier" for the hash function, and the message digest itself, are concatenated and converted to a number `x`. The signature is computed from `x` and the private key as `s = x^d mod n`[1]. The signature, `s` is a number of about the same size of `n`, and it usually encoded as a sequence of octets, most significant octet first.

The verification operation is straight-forward, `x` is computed from the message in the same way as above. Then `s^e mod n` is computed, the operation returns true if and only if the result equals `x`.

6.6.2 Nettle's RSA support

Nettle represents RSA keys using two structures that contain large numbers (of type `mpz_t`).

`rsa_public_key` *size n e* [Context struct]

: `size` is the size, in octets, of the modulo, and is used internally. `n` and `e` is the public key.

`rsa_private_key` *size d p q a b c* [Context struct]

: `size` is the size, in octets, of the modulo, and is used internally. `d` is the secret exponent, but it is not actually used when signing. Instead, the factors `p` and `q`, and the parameters `a`, `b` and `c` are used. They are computed from `p`, `q` and `e` such that `a e mod (p - 1) = 1, b e mod (q - 1) = 1, c q mod p = 1`.

Before use, these structs must be initialized by calling one of

`void rsa_public_key_init (struct rsa_public_key *pub)` [Function]
`void rsa_private_key_init (struct rsa_private_key *key)` [Function]

: Calls `mpz_init` on all numbers in the key struct.

and when finished with them, the space for the numbers must be deallocated by calling one of

`void rsa_public_key_clear (struct rsa_public_key *pub)` [Function]
`void rsa_private_key_clear (struct rsa_private_key *key)` [Function]

: Calls `mpz_clear` on all numbers in the key struct.

In general, Nettle's RSA functions deviates from Nettle's "no memory allocation"-policy. Space for all the numbers, both in the key structs above, and temporaries, are allocated dynamically. For information on how to customize allocation, see See Section "GMP Allocation" in *GMP Manual*.

When you have assigned values to the attributes of a key, you must call

`int rsa_public_key_prepare (struct rsa_public_key *pub)` [Function]
`int rsa_private_key_prepare (struct rsa_private_key *key)` [Function]

: Computes the octet size of the key (stored in the `size` attribute, and may also do other basic sanity checks. Returns one if successful, or zero if the key can't be used, for instance if the modulo is smaller than the minimum size needed for RSA operations specified by PKCS#1.

[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.

Before signing or verifying a message, you first hash it with the appropriate hash function. You pass the hash function's context struct to the RSA signature function, and it will extract the message digest and do the rest of the work. There are also alternative functions that take the hash digest as argument.

There is currently no support for using SHA224 or SHA384 with RSA signatures, since there's no gain in either computation time nor message size compared to using SHA256 and SHA512, respectively.

Creation and verification of signatures is done with the following functions:

`int` **`rsa_md5_sign`** (*const struct rsa_private_key* `*key`, *struct md5_ctx* `*hash`, *mpz_t* `signature`) [Function]

`int` **`rsa_sha1_sign`** (*const struct rsa_private_key* `*key`, *struct sha1_ctx* `*hash`, *mpz_t* `signature`) [Function]

`int` **`rsa_sha256_sign`** (*const struct rsa_private_key* `*key`, *struct sha256_ctx* `*hash`, *mpz_t* `signature`) [Function]

`int` **`rsa_sha512_sign`** (*const struct rsa_private_key* `*key`, *struct sha512_ctx* `*hash`, *mpz_t* `signature`) [Function]

The signature is stored in *signature* (which must have been `mpz_init`'ed earlier). The hash context is reset so that it can be used for new messages. Returns one on success, or zero on failure. Signing fails if the key is too small for the given hash size, e.g., it's not possible to create a signature using SHA512 and a 512-bit RSA key.

`int` **`rsa_md5_sign_digest`** (*const struct rsa_private_key* `*key`, *const uint8_t* `*digest`, *mpz_t* `signature`) [Function]

`int` **`rsa_sha1_sign_digest`** (*const struct rsa_private_key* `*key`, *const uint8_t* `*digest`, *mpz_t* `signature`); [Function]

`int` **`rsa_sha256_sign_digest`** (*const struct rsa_private_key* `*key`, *const uint8_t* `*digest`, *mpz_t* `signature`); [Function]

`int` **`rsa_sha512_sign_digest`** (*const struct rsa_private_key* `*key`, *const uint8_t* `*digest`, *mpz_t* `signature`); [Function]

Creates a signature from the given hash digest. *digest* should point to a digest of size `MD5_DIGEST_SIZE`, `SHA1_DIGEST_SIZE`, or `SHA256_DIGEST_SIZE`, respectively. The signature is stored in *signature* (which must have been `mpz_init`:ed earlier). Returns one on success, or zero on failure.

`int` **`rsa_md5_verify`** (*const struct rsa_public_key* `*key`, *struct md5_ctx* `*hash`, *const mpz_t* `signature`) [Function]

`int` **`rsa_sha1_verify`** (*const struct rsa_public_key* `*key`, *struct sha1_ctx* `*hash`, *const mpz_t* `signature`) [Function]

`int` **`rsa_sha256_verify`** (*const struct rsa_public_key* `*key`, *struct sha256_ctx* `*hash`, *const mpz_t* `signature`) [Function]

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

Returns 1 if the signature is valid, or 0 if it isn't. In either case, the hash context is reset so that it can be used for new messages.

int rsa_md5_verify_digest (*cons s c sa_ b c_* *key, *cons n 8_* *digest, *cons z_* signature) [Function]

int rsa_sha1_verify_digest (*cons s c sa_ b c* [Function]

For DSA, the underlying mathematical problem is the computation of discrete logarithms. The public key consists of a large prime `p`, a small prime `q` which is a factor of `p-1`, a number `g` which generates a subgroup of order `q` modulo `p`, and an element `y` in that subgroup.

In the original DSA, the size of `q` is fixed to 160 bits, to match with the SHA1 hash algorithm. The size of `p` is in principle unlimited, but the standard specifies only nine specific sizes: `512 + l*64`, where `l` is between 0 and 8. Thus, the maximum size of `p` is 1024 bits, and sizes less than 1024 bits are considered obsolete and not secure.

The subgroup requirement means that if you compute

```
g^t mod p
```

for all possible integers `t`, you will get precisely `q` distinct values.

The private key is a secret exponent `x`, such that

```
g^x = y mod p
```

In mathematical speak, `x` is the *discrete logarithm* of `y` mod `p`, with respect to the generator `g`. The size of `x` will also be about the same size as `q`. The security of the DSA algorithm relies on the difficulty of the discrete logarithm problem. Current algorithms to compute discrete logarithms in this setting, and hence crack DSA, are of two types. The first type works directly in the (multiplicative) group of integers mod `p`. The best known algorithm of this type is the Number Field Sieve, and it's complexity is similar to the complexity of factoring numbers of the same size as `p`. The other type works in the smaller `q`-sized subgroup generated by `g`, which has a more difficult group structure. One good algorithm is Pollard-rho, which has complexity `sqrt(q)`.

The important point is that security depends on the size of *both* `p` and `q`, and they should be chosen so that the difficulty of both discrete logarithm methods are comparable. Today, the security margin of the original DSA may be uncomfortably small. Using a `p` of 1024 bits implies that cracking using the number field sieve is expected to take about the same time as factoring a 1024-bit RSA modulo, and using a `q` of size 160 bits implies that cracking using Pollard-rho will take roughly `2^80` group operations. With the size of `q` fixed, tied to the SHA1 digest size, it may be tempting to increase the size of `p` to, say, 4096 bits. This will provide excellent resistance against attacks like the number field sieve which works in the large group. But it will do very little to defend against Pollard-rho attacking the small subgroup; the attacker is slowed down at most by a single factor of 10 due to the more expensive group operation. And the attacker will surely choose the latter attack.

The signature generation algorithm is randomized; in order to create a DSA signature, you need a good source for random numbers (see Section 6.7 [Randomness], page 52). Let us describe the common case of a 160-bit `q`.

To create a signature, one starts with the hash digest of the message, `h`, which is a 160 bit number, and a random number `k, 0<k<q`, also 160 bits. Next, one computes

```
r = (g^k mod p) mod q
s = k^-1 (h + x r) mod q
```

The signature is the pair `(r, s)`, two 160 bit numbers. Note the two different mod operations when computing `r`, and the use of the secret exponent `x`.

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

```
w = s^-1 mod q
v = (g^(w h) y^(w r) mod p) mod q
```

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

```
g^(w h) y^(w r) = g^(w h) (g^x)^(w r) = g^(w (h + x r)) = g^k
```

When reducing mod `q` this yields `r`. Note that when verifying a signature, we don't know either `k` or `x`: those numbers are secret.

If you can choose between RSA and DSA, which one is best? Both are believed to be secure. DSA gained popularity in the late 1990s, as a patent free alternative to RSA. Now that the RSA patents have expired, there's no compelling reason to want to use DSA. Today, the original DSA key size does not provide a large security margin, and it should probably be phased out together with RSA keys of 1024 bits. Using the revised DSA algorithm with a larger hash function, in particular, SHA256, a 256-bit `q`, and `p` of size 2048 bits or more, should provide for a more comfortable security margin, but these variants are not yet in wide use.

DSA signatures are smaller than RSA signatures, which is important for some specialized applications.

From a practical point of view, DSA's need for a good randomness source is a serious disadvantage. If you ever use the same `k` (and `r`) for two different message, you leak your private key.

6.6.4 Nettle's DSA support

Like for RSA, Nettle represents DSA keys using two structures, containing values of type `mpz_t`. For information on how to customize allocation, see See Section "GMP Allocation" in GMP Manual.

Most of the DSA functions are very similar to the corresponding RSA functions, but there are a few differences pointed out below. For a start, there are no functions corresponding to `rsa_public_key_prepare` and `rsa_private_key_prepare`.

dsa_public_key *p q g y* [Context struct]

The public parameters described above.

dsa_private_key *x* [Context struct]

The private key `x`.

Before use, these structs must be initialized by calling one of

void dsa_public_key_init (*struct dsa_public_key *pub*) [Function]

void dsa_private_key_init (*struct dsa_private_key *key*) [Function]

Calls `mpz_init` on all numbers in the key struct.

When finished with them, the space for the numbers must be deallocated by calling one of

void dsa_public_key_clear (*struct dsa_public_key *pub*) [Function]

void dsa_private_key_clear (*struct dsa_private_key *key*) [Function]

Calls `mpz_clear` on all numbers in the key struct.

Signatures are represented using the structure below, and need to be initialized and cleared in the same way as the key structs.

`dsa_signature` *r, s* [Context struct]

`void dsa_signature_init` (*struct dsa_signature* `*signature`) [Function]
`void dsa_signature_clear` (*struct dsa_signature* `*signature`) [Function]

You must call `dsa_signature_init` before creating or using a signature, and call `dsa_signature_clear` when you are finished with it.

For signing, you need to provide both the public and the private key (unlike RSA, where the private key struct includes all information needed for signing), and a source for random numbers. Signatures can use the SHA1 or the SHA256 hash function, although the implementation of DSA with SHA256 should be considered somewhat experimental due to lack of official test vectors and interoperability testing.

`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]

Creates a signature from the given hash context or digest. *random_ctx* and *random* is a randomness generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. For advice, see See Section 6.7 [Randomness], page 52. Returns one on success, or zero on failure. Signing fails if the key size and the hash size don't match.

Verifying signatures is a little easier, since no randomness generator is needed. The functions are

`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`) [Function]
`int dsa_sha256_verify_digest` (*const struct dsa_public_key* `*key`, *const uint8_t* `*digest`, *const struct dsa_signature* `*signature`) [Function]

Verifies a signature. Returns 1 if the signature is valid, otherwise 0.

Key generation uses mostly the same parameters as the corresponding RSA function.

int dsa_generate_keypair (*struct dsa_public_key *pub, struct dsa_private_key *key, void *random_ctx, nettle_random_func* [Function]

- Attacks using memory caches Assume you have some secret data on a multi-user system and that this data is properly protected so that other users get no direct access to it. If you have a process operating on the secret data and this process does memory accesses depending on the data e.g an internal lookup table in some cryptographic algorithm an attacker running a separate process on the same system may use behavior of internal CPU caches to get information about your secrets.

Nettle's ECC implementation is designed to be *s d-c anne s n.* and not leak any information to these attacks. Timing and memory accesses depend only on the size of the input data and its location in memory not on the actual data bits. This implies a performance penalty in several of the building blocks.

6.6.6 ECDSA

ECDSA is a variant of the DSA digital signature scheme see Section 6.6.3 [DSA] page 45, which works over an elliptic curve group rather than over a subgroup of integers modulo p. Like DSA creating a signature requires a unique random nonce repeating the nonce with two di erent messages reveals the private key and any leak or bias in the generation of the nonce also leaks information about the key .

Unlike DSA signatures are in general not tied to any particular hash function or even hash size. Any hash function can be used and the hash value is truncated or padded as needed to get a size matching the curve being used. It is recommended to use a strong cryptographic hash function with digest size close to the bit size of the curve e.g. SHA256 is a reasonable choice when using ECDSA signature over the curve secp256r . A protocol or application using ECDSA has to specify which curve and which hash function to use or provide some mechanism for negotiating.

Nettle de nes ECDSA in `<nettle/ecdsa.h>`. We rst need to de ne the data types used to represent public and private keys.

`struct ecc_point` [struct]

Represents a point on an elliptic curve. In particular it is used to represent an ECDSA public key.

`void ecc_point_init` (*s c cc_ o n *p, cons s c cc_c e *ecc*) [Function]

Initializes to represent points on the given curve *cc*. Allocates storage for the coordinates using the same allocation functions as GMP.

`void ecc_point_clear` (*s c cc_ o n *p*) [Function]

Deallocate storage.

`int ecc_point_set` (*s c cc_ o n *p, cons z_ x, cons z_ y*) [Function]

Check that the given coordinates represent a point on the curve. If so the coordinates are copied and converted to internal representation and the function returns . Otherwise it returns . Currently the in nity point or zero point with additive notation i snot allowed.

`void ecc_point_get` (*cons s c cc_ o n *p, z_ x, z_ y*) [Function]

Extracts the coordinate of the point . The output parameters or may be NULL if the caller doesn't want that coordinate.

`struct ecc_scalar` [struct]

Represents an integer in the range $0 < x < group\_order$, where the “group order” refers to the order of an ECC group. In particular, it is used to represent an ECDSA private key.

`void ecc_scalar_init` (*struct ecc_scalar* `*s`, *const struct ecc_curve* `*ecc`) [Function]

Initializes *s* to represent a scalar suitable for the given curve *ecc*. Allocates storage using the same allocation functions as GMP.

`void ecc_scalar_clear` (*struct ecc_scalar* `*s`) [Function]

Deallocate storage.

`int ecc_scalar_set` (*struct ecc_scalar* `*s`, *const mpz_t* `z`) [Function]

Check that *z* is in 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, in GMP `mpz_t` representation.

To create and verify ECDSA signatures, the following functions are used.

`void ecdsa_sign` (*const struct ecc_scalar* `*key`, *void* `*random_ctx`, *nettle_random_func* `*random`, *unsigned* `digest_length`, *const uint8_t* `*digest`, *struct dsa_signature* `*signature`) [Function]

Uses the private key *key* to create a signature on *digest*. *random_ctx* and *random* is a randomness generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. The signature is stored in *signature*, in the same was as for plain DSA.

`int ecdsa_verify` (*const struct ecc_point* `*pub`, *unsigned* `length`, *const uint8_t* `*digest`, *const struct dsa_signature* `*signature`) [Function]

Uses the public key *pub* to verify that *signature* is a valid signature for the message digest *digest* (of *length* octets). Returns 1 if the signature is valid, otherwise 0.

Finally, to generation of new an ECDSA key pairs

`void ecdsa_generate_keypair` (*struct ecc_point* `*pub`, *struct ecc_scalar* `*key`, *void* `*random_ctx`, *nettle_random_func* `*random`); [Function]

pub and *key* is where the resulting key pair is stored. The structs should be initialized, for the desired ECC curve, before you call this function.

random_ctx and *random* is a randomness generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. For advice, see See Section 6.7 [Randomness], page 52.

6.7 Randomness

A crucial ingredient in many cryptographic contexts is randomness: Let `p` be a random prime, choose a random initialization vector `iv`, a random key `k` and a random exponent `e`, etc. In the theories, it is assumed that you have plenty of randomness around. If this assumption is not true in practice, systems that are otherwise perfectly secure, can be broken. Randomness has often turned out to be the weakest link in the chain.

In non-cryptographic applications, such as games as well as scientific simulation, a good randomness generator usually means a generator that has good statistical properties, and is seeded by some simple function of things like the current time, process id, and host name.

However, such a generator is inadequate for cryptography, for at least two reasons:

- It's too easy for an attacker to guess the initial seed. Even if it will take some 2^32 tries before he guesses right, that's far too easy. For example, if the process id is 16 bits, the resolution of "current time" is one second, and the attacker knows what day the generator was seeded, there are only about 2^32 possibilities to try if all possible values for the process id and time-of-day are tried.
- The generator output reveals too much. By observing only a small segment of the generator's output, its internal state can be recovered, and from there, all previous output and all future output can be computed by the attacker.

A randomness generator that is used for cryptographic purposes must have better properties. Let's first look at the seeding, as the issues here are mostly independent of the rest of the generator. The initial state of the generator (its seed) must be unguessable by the attacker. So what's unguessable? It depends on what the attacker already knows. The concept used in information theory to reason about such things is called "entropy", or "conditional entropy" (not to be confused with the thermodynamic concept with the same name). A reasonable requirement is that the seed contains a conditional entropy of at least some 80-100 bits. This property can be explained as follows: Allow the attacker to ask `n` yes-no-questions, of his own choice, about the seed. If the attacker, using this question-and-answer session, as well as any other information he knows about the seeding process, still can't guess the seed correctly, then the conditional entropy is more than `n` bits.

Let's look at an example. Say information about timing of received network packets is used in the seeding process. If there is some random network traffic going on, this will contribute some bits of entropy or "unguessability" to the seed. However, if the attacker can listen in to the local network, or if all but a small number of the packets were transmitted by machines that the attacker can monitor, this additional information makes the seed easier for the attacker to figure out. Even if the information is exactly the same, the conditional entropy, or unguessability, is smaller for an attacker that knows some of it already before the hypothetical question-and-answer session.

Seeding of good generators is usually based on several sources. The key point here is that the amount of unguessability that each source contributes, depends on who the attacker is. Some sources that have been used are:

High resolution timing of i/o activities

Such as completed blocks from spinning hard disks, network packets, etc. Getting access to such information is quite system dependent, and not all systems include suitable hardware. If available, it's one of the better randomness source one can find in a digital, mostly predictable, computer.

User activity

Timing and contents of user interaction events is another popular source that is available for interactive programs even if I suspect that it is sometimes used in order to make the user feel good, not because the quality of the input is needed or used properly. Obviously, not available when a machine is unattended. Also beware of networks. User interaction that happens across a long serial cable, TELNET session, or even SSH session may be visible to an attacker, in full or partially.

Audio input

Any room, or even a microphone input that's left unconnected, is a source of some random background noise, which can be fed into the seeding process.

Specialized hardware

Hardware devices with the sole purpose of generating random data have been designed. They range from radioactive samples with an attached Geiger counter, to amplification of the inherent noise in electronic components such as diodes and resistors, to low-frequency sampling of chaotic systems. Hashing successive images of a Lava lamp is a spectacular example of the latter type.

Secret information

Secret information, such as user passwords or keys, or private files stored on disk, can provide some unguessability. A problem is that if the information is revealed at a later time, the unguessability vanishes. Another problem is that this kind of information tends to be fairly constant, so if you rely on it and seed your generator regularly, you risk constructing almost similar seeds or even constructing the same seed more than once.

For all practical sources, it's difficult but important to provide a reliable lower bound on the amount of unguessability that it provides. Two important points are to make sure that the attacker can't observe your sources so if you like the Lava lamp idea, remember that you have to get your own lamp, and not put it by a window or anywhere else where strangers can see it, and that hardware failures are detected. What if the bulb in the Lava lamp, which you keep locked into a cupboard following the above advice, breaks after a few months?

So let's assume that we have been able to find an unguessable seed, which contains at least 8 bits of conditional entropy, relative to all attackers that we care about typically, we must at the very least assume that no attacker has root privileges on our machine.

How do we generate output from this seed, and how much can we get? Some generators notably the Linux `/dev/random` generator tries to estimate available entropy and restrict the amount of output. The goal is that if you read 128 bits from `/dev/random`, you should get 128 truly random" bits. This is a property that is useful in some specialized circumstances, for instance when generating key material for a one time pad, or when working with unconditional blinding, but in most cases, it doesn't matter much. For most application, there's no limit on the amount of useful random" data that we can generate from a small seed, what matters is that the seed is unguessable and that the generator has good cryptographic properties.

At the heart of all generators lies its internal state. Future output is determined by the internal state alone. Let's call it the generator's key. The key is initialized from the unguessable seed. Important properties of a generator are:

Key-hiding
: An attacker observing the output should not be able to recover the generator's key.

Independence of outputs
: Observing some of the output should not help the attacker to guess previous or future output.

Forward secrecy
: Even if an attacker compromises the generator's key, he should not be able to guess the generator output *before* the key compromise.

Recovery from key compromise
: If an attacker compromises the generator's key, he can compute *all* future output. This is inevitable if the generator is seeded only once, at startup. However, the generator can provide a reseeding mechanism, to achieve recovery from key compromise. More precisely: If the attacker compromises the key at a particular time `t_1`, there is another later time `t_2`, such that if the attacker observes all output generated between `t_1` and `t_2`, he still can't guess what output is generated after `t_2`.

Nettle includes one randomness generator that is believed to have all the above properties, and two simpler ones.

ARCFOUR, like any stream cipher, can be used as a randomness generator. Its output should be of reasonable quality, if the seed is hashed properly before it is used with `arcfour_set_key`. There's no single natural way to reseed it, but if you need reseeding, you should be using Yarrow instead.

The "lagged Fibonacci" generator in `<nettle/knuth-lfib.h>` is a fast generator with good statistical properties, but is **not** for cryptographic use, and therefore not documented here. It is included mostly because the Nettle test suite needs to generate some test data from a small seed.

The recommended generator to use is Yarrow, described below.

6.7.1 Yarrow

Yarrow is a family of pseudo-randomness generators, designed for cryptographic use, by John Kelsey, Bruce Schneier and Niels Ferguson. Yarrow-160 is described in a paper at `http://www.counterpane.com/yarrow.html`, and it uses SHA1 and triple-DES, and has a 160-bit internal state. Nettle implements Yarrow-256, which is similar, but uses SHA256 and AES to get an internal state of 256 bits.

Yarrow was an almost finished project, the paper mentioned above is the closest thing to a specification for it, but some smaller details are left out. There is no official reference implementation or test cases. This section includes an overview of Yarrow, but for the details of Yarrow-256, as implemented by Nettle, you have to consult the source code. Maybe a complete specification can be written later.

Yarrow can use many sources at least two are needed for proper reseeding, and two randomness pools", referred to as the slow pool" and the fast pool". Input from the sources is fed alternatingly into the two pools. When one of the sources has contributed, bits of entropy to the fast pool, a fast reseed" happens and the fast pool is mixed into the internal state. When at least two of the sources have contributed at least 6 bits each to the slow pool, a slow reseed" takes place. The contents of both pools are mixed into the internal state. These procedures should ensure that the generator will eventually recover after a key compromise.

The output is generated by using AES to encrypt a counter, using the generator's current key. After each request for output, another 256 bits are generated which replace the key. This ensures forward secrecy.

Yarrow can also use a *seed file* to save state across restarts. Yarrow is seeded by either feeding it the contents of the previous seed-file, or feeding it input from its sources until a slow reseed happens.

Nettle defines Yarrow-256 in `<nettle/yarrow.h>`.

`struct yarrow256_ctx` [Context struct]

`struct yarrow_source` [Context struct]

Information about a single source.

`YARROW256_SEED_FILE_SIZE` [Constant]

Recommended size of the Yarrow-256 seed-file.

`void yarrow256_init` (*struct yarrow256_ctx* `*ctx`, *unsigned* `nsources`, *struct yarrow_source* `*sources`) [Function]

Initializes the yarrow context, and its *nsources* sources. It's possible to call it with *nsources* and *sources* NULL, if you don't need the update features.

`void yarrow256_seed` (*struct yarrow256_ctx* `*ctx`, *unsigned* `length`, *uint8_t* `*seed_file`) [Function]

Seeds Yarrow-256 from a previous seed-file. *length* should be at least `YARROW256_SEED_FILE_SIZE`, but it can be larger.

The generator will trust you that the *seed_file* data really is unguessable. After calling this function, you *must* overwrite the old seed-file with newly generated data from `yarrow256_random`. If it's possible for several processes to read the seed-file at about the same time, access must be coordinated using some locking mechanism.

`int yarrow256_update` (*struct yarrow256_ctx* `*ctx`, *unsigned* `source`, *unsigned* `entropy`, *unsigned* `length`, *const uint8_t* `*data`) [Function]

Updates the generator with data from source *SOURCE* an index that must be smaller than the number of sources. *entropy* is your estimated lower bound for the entropy in the data, measured in bits. Calling update with zero *entropy* is always safe, no matter if the data is random or not.

Returns, if a reseed happened, in which case an application using a seed-file may want to generate new seed data with `yarrow256_random` and overwrite the seed-file. Otherwise, the function returns .

void yarrow256_random (*s c aro 256_c* *ctx, *ns gned* length, *n 8_* *dst) [Function]

Generates *ng* octets of output. The generator must be seeded before you call this function.

If you don't need forward secrecy e.g. if you need non-secret randomness for initialization vectors or padding you can gain some e ciency by b ering calling this function for reasonably large blocks of data say, - octets at a time.

int yarrow256_is_seeded (*s c aro 256_c* *ctx) [Function]

Returns, if the generator is seeded and ready to generate output otherwise .

unsigned yarrow256_needed_sources (*s c aro 256_c* *ctx) [Function]

Returns the number of sources that must reach the threshold before a slow reseed will happen. Useful primarily when the generator is unseeded.

void yarrow256_fast_reseed (*s c aro 256_c* *ctx) [Function]

void yarrow256_slow_reseed (*s c aro 256_c* *ctx) [Function]

Causes a fast or slow reseed to take place immediately regardless of the current entropy estimates of the two pools. Use with care.

Nettle includes an entropy estimator for one kind of input source . User keyboard input.

struct yarrow_key_event_ctx [Context struct]

Information about recent key events.

void yarrow_key_event_init (*s c aro _ _ n _c* *ctx) [Function]

Initializes the context.

unsigned yarrow_key_event_estimate (*s c aro _ _ n _c* *ctx, *ns gned* key, *ns gned* time) [Function]

is the id of the key ASCII value hardware key code X keysym . . . it doesn't matter, and is the timestamp of the event. The time must be given in units matching the resolution by which you read the clock. If you read the clock with microsecond precision should be provided in units of microseconds. But if you use gettimeofday on a typical Unix system where the clock ticks, or so microseconds at a time should be given in units of, microseconds.

Returns an entropy estimate in bits suitable for calling yarrow256_update. Usually , or 2 bits.

6.8 ASCII encoding

Encryption will transform your data from text into binary format and that may be a problem if you want for example to send the data as if it was plain text in an email or store it along with descriptive text in a file . You may then use an encoding from binary to text . each binary byte is translated into a number of bytes of plain text.

A base-N encoding of data is one representation of data that only uses N di erent symbols instead of the 256 possible values of a byte .

The base64 encoding will always use alphanumeric upper and lower case characters and the' +, ' ' and' ' symbols to represent the data. Four output characters are generated for each three bytes of input. In case the length of the input is not a multiple of three, padding characters are added at the end.

The base16 encoding, also known as "hexadecimal", uses the decimal digits and the letters from A to F. Two hexadecimal digits are generated for each input byte. Base16 may be useful if you want to use the data for filenames or URLs, for example.

Nettle supports both base64 and base16 encoding and decoding.

Encoding and decoding uses a context struct to maintain its state (with the exception of base16 encoding, which doesn't need any). To encode or decode the your data, first initialize the context, then call the update function as many times as necessary, and complete the operation by calling the final function.

The following functions can be used to perform base64 encoding and decoding. They are defined in `<nettle/base64.h>`.

`struct base64_encode_ctx` [Context struct]

`void base64_encode_init` (*struct base64_encode_ctx* `*ctx`) [Function]

Initializes a base64 context. This is necessary before starting an encoding session.

`unsigned base64_encode_single` (*struct base64_encode_ctx* `*ctx`, *uint8_t* `*dst`, *uint8_t* `src`) [Function]

Encodes a single byte. Returns amount of output (always 1 or 2).

`BASE64_ENCODE_LENGTH` (*length*) [Macro]

The maximum number of output bytes when passing *length* input bytes to `base64_encode_update`.

`unsigned base64_encode_update` (*struct base64_encode_ctx* `*ctx`, *uint8_t* `*dst`, *unsigned* `length`, *const uint8_t* `*src`) [Function]

After *ctx* is initialized, this function may be called to encode *length* bytes from *src*. The result will be placed in *dst*, and the return value will be the number of bytes generated. Note that *dst* must be at least of size BASE64_ENCODE_LENGTH(*length*).

`BASE64_ENCODE_FINAL_LENGTH` [Constant]

The maximum amount of output from `base64_encode_final`.

`unsigned base64_encode_final` (*struct base64_encode_ctx* `*ctx`, *uint8_t* `*dst`) [Function]

After calling base64_encode_update one or more times, this function should be called to generate the final output bytes, including any needed paddding. The return value is the number of output bytes generated.

`struct base64_decode_ctx` [Context struct]

`void base64_decode_init` (*struct base64_decode_ctx* `*ctx`) [Function]

Initializes a base64 decoding context. This is necessary before starting a decoding session.

`int base64_decode_single` (*struct base64_decode_ctx* `*ctx`, *uint8_t* `*dst`, *uint8_t* `src`) [Function]

Decodes a single byte (*src*) and stores the result in *dst*. Returns amount of output (0 or 1), or -1 on errors.

`BASE64_DECODE_LENGTH (length)` [Macro]

The maximum number of output bytes when passing *length* input bytes to `base64_decode_update`.

`void base64_decode_update (struct base64_decode_ctx *ctx, unsigned *dst_length, uint8_t *dst, unsigned src_length, const uint8_t *src)` [Function]

After *ctx* is initialized, this function may be called to decode *src_length* bytes from *src*. *dst* should point to an area of size at least BASE64_DECODE_LENGTH(*length*), and for sanity checking, *dst_length* should be initialized to the size of that area before the call. *dst_length* is updated to the amount of decoded output. The function will return 1 on success and 0 on error.

`int base64_decode_final (struct base64_decode_ctx *ctx)` [Function]

Check that final padding is correct. Returns 1 on success, and 0 on error.

Similarly to the base64 functions, the following functions perform base16 encoding, and are defined in `<nettle/base16.h>`. Note that there is no encoding context necessary for doing base16 encoding.

`void base16_encode_single (uint8_t *dst, uint8_t src)` [Function]

Encodes a single byte. Always stores two digits in *dst*[0] and *dst*[1].

`BASE16_ENCODE_LENGTH (length)` [Macro]

The number of output bytes when passing *length* input bytes to `base16_encode_update`.

`void base16_encode_update (uint8_t *dst, unsigned length, const uint8_t *src)` [Function]

Always stores BASE16_ENCODE_LENGTH(*length*) digits in *dst*.

`struct base16_decode_ctx` [Context struct]

`void base16_decode_init (struct base16_decode_ctx *ctx)` [Function]

Initializes a base16 decoding context. This is necessary before starting a decoding session.

`int base16_decode_single (struct base16_decode_ctx *ctx, uint8_t *dst, uint8_t src)` [Function]

Decodes a single byte from *src* into *dst*. Returns amount of output (0 or 1), or -1 on errors.

`BASE16_DECODE_LENGTH (length)` [Macro]

The maximum number of output bytes when passing *length* input bytes to `base16_decode_update`.

`int base16_decode_update (struct base16_decode_ctx *ctx, unsigned *dst_length, uint8_t *dst, unsigned src_length, const uint8_t *src)` [Function]

After *ctx* is initialized, this function may be called to decode *src_length* bytes from *src*. *dst* should point to an area of size at least BASE16_DECODE_LENGTH(*length*), and for sanity checking, *dst_length* should be initialized to the size of that area before the call. *dst_length* is updated to the amount of decoded output. The function will return 1 on success and 0 on error.

int base16_decode_final (*s c bas 16_d cod _c* *ctx*) [Function]

Checks that the end of data is correct i.e. an even number of hexadecimal digits have been seen . Returns, on success and on error.

6.9 Miscellaneous functions

uint8_t * memxor (*n 8_* *dst*, *cons n 8_* *src*, *s z_* *n*) [Function]

XORs the source area on top of the destination area. The interface doesn't follow the Nettle conventions because it is intended to be similar to the ANSI-C memcpy function.

memxor is declared in <nettle/memxor.h>.

6.10 Compatibility functions

For convenience Nettle includes alternative interfaces to some algorithms for compatibility with some other popular crypto toolkits. These are not fully

7 Traditional Nettle Soup

For the serious nettle hacker here is a recipe for nettle soup 4 servings

- liter fresh nettles urtica dioica
- 2 tablespoons butter
- 3 tablespoons our
- liter stock meat or vegetable
- 2 teaspoon salt
- a tad white pepper
- some cream or milk

Gather liter fresh nettles Use gloves! Small tender shoots are preferable but the tops of larger nettles can also be used

Rinse the nettles very well Boil them for minutes in lightly salted water Strain the nettles and save the water Hack the nettles Melt the butter and mix in the our Dilute with stock and the nettle-water you saved earlier Add the hacked nettles If you wish you can add some milk or cream at this stage Bring to a boil and let boil for a few minutes Season with salt and pepper

Serve with boiled egg-halves

8 Installation

Nettle uses autoconf. To build it, unpack the source and run

```
./configure
make
make check
make install
```

to install in under the default prefix, /usr/local.

To get a list of configure options, use ./configure --help.

By default, both static and shared libraries are built and installed. To omit building the shared libraries, use the --disable-shared option to ./configure.

Using GNU make is recommended. For other make programs, in particular BSD make, you may have to use the --disable-dependency-tracking option to ./configure.

Function and Concept Index

G

H

K

M

O

P

R