Cifra Documentation

Release 0.1

Joseph Birr-Pixton

Nov 12, 2018

Contents

Contents:

CHAPTER 1

Library configuration

1.1 Macros

CF_SIDE_CHANNEL_PROTECTION
Define this as 1 if you need all available side channel protections. **This option may alter the ABI**.

This has a non-trivial performance penalty. Where a side-channel free option is cheap or free (like checking a MAC) this is always done in a side-channel free way.

The default is **on** for all available protections.

CF_TIME_SIDE_CHANNEL_PROTECTION
Define this as 1 if you need timing/branch prediction side channel protection.

You probably want this. The default is on.

CF_CACHE_SIDE_CHANNEL_PROTECTION
Define this as 1 if you need cache side channel protection.

If you have a microcontroller with no cache, you can turn this off without negative effects.

The default is on. This will have some performance impact, especially on AES.

CHAPTER 2

General block cipher description

This allows us to implement block cipher modes which can work with different block ciphers.

2.1 Macros

CF_MAXBLOCK
The maximum block cipher blocksize we support, in bytes.

2.2 Types

cf_prp_block
Block processing function type.

The *in* and *out* blocks may alias.

Return type void

Parameters

- **ctx** – block cipher-specific context object.
- **in** – input block.
- **out** – output block.

cf_prp
Describes an PRP in a general way.

cf_prp.blocksz

Block size in bytes. Must be no more than *CF_MAXBLOCK*.

cf_prp.encrypt

Block encryption function.

cf_prp.decrypt

Block decryption function.

CHAPTER 3

General hash function description

This allows us to make use of hash functions without depending on a specific one. This is useful in implementing, for example, *HMAC*.

3.1 Macros

CF_CHASH_MAXCTX
The maximum size of a *cf_chash_ctx*. This allows use to put a structure in automatic storage that can store working data for any supported hash function.

CF_CHASH_MAXBLK
Maximum hash function block size (in bytes).

CF_MAXHASH
Maximum hash function output (in bytes).

3.2 Types

cf_chash_init
Hashing initialisation function type.

Functions of this type should initialise the context in preparation for hashing a message with *cf_chash_update* functions.

Return type void

Parameters

- **ctx** – hash function-specific context structure.

cf_chash_update
Hashing data processing function type.

Functions of this type hash *count* bytes of data at *data*, updating the contents of *ctx*.

Return type void

Parameters

- **ctx** – hash function-specific context structure.
- **data** – input data to hash.
- **count** – number of bytes to hash.

cf_chash_digest

Hashing completion function type.

Functions of this type complete a hashing operation, writing `cf_chash.hashsz` bytes to *hash*.

This function does not change *ctx* – any padding which needs doing must be done seperately (in a copy of *ctx*, say).

This means you can interlave *_update* and *_digest* calls to learn *H(A)* and *H(A || B)* without hashing *A* twice.

Return type void

Parameters

- **ctx** – hash function-specific context structure.
- **hash** – location to write hash result.

cf_chash

This type describes an incremental hash function in an abstract way.

cf_chash.hashsz

The hash function's output, in bytes.

cf_chash.blocksz

The hash function's internal block size, in bytes.

cf_chash.init

Context initialisation function.

cf_chash:update

Data processing function.

cf_chash:digest

Completion function.

cf_chash_ctx

A type usable with any *cf_chash* as a context.

3.3 Functions

void **cf_hash**(const *cf_chash* *h, const void *m, size_t *nm*, uint8_t *out)

One shot hashing: *out = h(m)*.

Using the hash function *h*, *nm* bytes at *m* are hashed and *h->hashsz* bytes of result is written to the buffer *out*.

Parameters

- **h** – hash function description.
- **m** – message buffer.

- **nm** – message length.
- **out** – hash result buffer (written).

CHAPTER 4

The AES block cipher

This is a small, simple implementation of AES. Key expansion is done first, filling in a *cf_aes_context*. Then encryption and decryption can be performed as desired.

Usually you don't want to use AES directly; you should use it via a *block cipher mode*.

4.1 Macros

AES_BLOCKSZ
AES has a 128-bit block size. This quantity is in bytes.

AES128_ROUNDS

AES192_ROUNDS

AES256_ROUNDS
Round counts for different key sizes.

CF_AES_MAXROUNDS
You can reduce the maximum number of rounds this implementation supports. This reduces the storage needed by *cf_aes_context*.

The default is *AES256_ROUNDS* and is good for all key sizes.

CF_AES_ENCRYPT_ONLY
Define this to 1 if you don't need to decrypt anything. This saves space. *cf_aes_decrypt()* calls *abort(3)*.

4.2 Types

cf_aes_context
This type represents an expanded AES key. Create one using *cf_aes_init()*, make use of one using *cf_aes_encrypt()* or *cf_aes_decrypt()*.

The contents of this structure are equivalent to the original key material. You should clean the contents of this structure with *cf_aes_finish()* when you're done.

cf_aes_context.rounds

Number of rounds to use, set by *cf_aes_init()*.

This depends on the original key size, and will be *AES128_ROUNDS*, *AES192_ROUNDS* or *AES256_ROUNDS*.

cf_aes_context.ks

Expanded key material. Filled in by *cf_aes_init()*.

4.3 Functions

void **cf_aes_init**(*cf_aes_context* **ctx*, const uint8_t **key*, size_t *nkey*)

This function does AES key expansion. It destroys existing contents of `ctx`.

Parameters

- **ctx** – expanded key context, filled in by this function.
- **key** – pointer to key material, of `nkey` bytes.
- **nkey** – length of key material. Must be *16*, *24* or *32*.

void **cf_aes_encrypt**(const *cf_aes_context* **ctx*, const uint8_t *in[AES_BLOCKSZ]*, uint8_t *out[AES_BLOCKSZ]*)

Encrypts the given block, from `in` to `out`. These may alias.

Fails at runtime if `ctx` is invalid.

Parameters

- **ctx** – expanded key context
- **in** – input block (read)
- **out** – output block (written)

void **cf_aes_decrypt**(const *cf_aes_context* **ctx*, const uint8_t *in[AES_BLOCKSZ]*, uint8_t *out[AES_BLOCKSZ]*)

Decrypts the given block, from `in` to `out`. These may alias.

Fails at runtime if `ctx` is invalid.

Parameters

- **ctx** – expanded key context
- **in** – input block (read)
- **out** – output block (written)

void **cf_aes_finish**(*cf_aes_context* **ctx*)

Erase scheduled key material.

Call this when you're done to erase the round keys.

4.4 Values

const *cf_prp* **cf_aes**
: Abstract interface to AES. See *cf_prp* for more information.

CHAPTER 5

The NORX AEAD cipher

This is an implementation of NORX32-4-1 with a one-shot interface. NORX is a CAESAR candidate with a core similar to ChaCha20 and a sponge structure like Keccak.

This is NORX v2.0. It is not compatible with earlier versions.

NORX32 uses a 128-bit key. Each encryption requires a 64-bit nonce. An encryption processes one sequence of additional data ('header'), followed by encryption of the plaintext, followed by processing a second sequence of additional data ('trailer'). It outputs a 128-bit tag.

5.1 Functions

void **cf_norx32_encrypt**(const uint8_t *key[16]*, const uint8_t *nonce[8]*, const uint8_t **header*, size_t *nheader*, const uint8_t **plaintext*, size_t *nbytes*, const uint8_t **trailer*, size_t *ntrailer*, uint8_t **ciphertext*, uint8_t *tag[16]*)

NORX32-4-1 one-shot encryption interface.

Parameters

- **key** – key material.
- **nonce** – per-message nonce.
- **header** – header buffer.
- **nheader** – number of header bytes.
- **plaintext** – plaintext bytes to be encrypted.
- **nbytes** – number of plaintext/ciphertext bytes.
- **trailer** – trailer buffer.
- **ntrailer** – number of trailer bytes.
- **ciphertext** – ciphertext output buffer, nbytes in length.
- **tag** – authentication tag output buffer.

int **cf_norx32_decrypt**(const uint8_t *key[16]*, const uint8_t *nonce[8]*, const uint8_t **header*, size_t *nheader*, const uint8_t **ciphertext*, size_t *nbytes*, const uint8_t **trailer*, size_t *ntrailer*, const uint8_t *tag[16]*, uint8_t **plaintext*)

NORX32-4-1 one-shot decryption interface.

Returns 0 on success, non-zero on error. Plaintext is zeroed on error.

Parameters

- **key** – key material.
- **nonce** – per-message nonce.
- **header** – header buffer.
- **nheader** – number of header bytes.
- **ciphertext** – ciphertext bytes to be decrypted.
- **nbytes** – number of plaintext/ciphertext bytes.
- **trailer** – trailer buffer.
- **ntrailer** – number of trailer bytes.
- **plaintext** – plaintext output buffer, nbytes in length.
- **tag** – authentication tag output buffer.

CHAPTER 6

The Salsa20/Chacha20 stream ciphers

These are similar stream ciphers by djb.

A reduced round variant of Salsa20 (Salsa20/12) was selected as a finalist of the eSTREAM stream cipher competition. This implementation does the full 20 rounds.

ChaCha20 is fundamentally like Salsa20, but has a tweaked round function to improve security margin without damaging performance.

6.1 Types

cf_salsa20_ctx

Incremental interface to Salsa20.

cf_salsa20_ctx.key0

Half of key material.

cf_salsa20_ctx.key1

Half of key material.

cf_salsa20_ctx.nonce

Nonce and counter block.

cf_salsa20_ctx.constant

Per-key-length constants.

cf_salsa20_ctx.block

Buffer for unused key stream material.

cf_salsa20_ctx.nblock

Number of bytes at end of *block* that can be used as key stream.

cf_chacha20_ctx
Incremental interface to Chacha20. This structure is identical to *cf_salsa20_ctx*.

6.2 Functions

void **cf_salsa20_init**(*cf_salsa20_ctx* *ctx, const uint8_t *key, size_t nkey, const uint8_t nonce[8])
Salsa20 initialisation function.

Parameters

- **ctx** – salsa20 context.
- **key** – key material.
- **nkey** – length of key in bytes, either 16 or 32.
- **nonce** – per-message nonce.

void **cf_chacha20_init**(*cf_chacha20_ctx* *ctx, const uint8_t *key, size_t nkey, const uint8_t nonce[8])
Chacha20 initialisation function.

Parameters

- **ctx** – chacha20 context (written).
- **key** – key material.
- **nkey** – length of key in bytes, either 16 or 32.
- **nonce** – per-message nonce.

void **cf_chacha20_init_custom**(*cf_chacha20_ctx* *ctx, const uint8_t *key, size_t nkey, const uint8_t nonce[16], size_t ncounter)
Chacha20 initialisation function. This version gives full control over the whole initial nonce value, and the size of the counter. The counter is always at the front of the nonce.

Parameters

- **ctx** – chacha20 context (written).
- **key** – key material.
- **nkey** – length of key in bytes, either 16 or 32.
- **nonce** – per-message nonce. *ncounter* bytes at the start are the block counter.
- **ncounter** – length, in bytes, of the counter portion of the nonce.

void **cf_salsa20_cipher**(*cf_salsa20_ctx* *ctx, const uint8_t *input, uint8_t *output, size_t count)
Salsa20 encryption/decryption function.

Parameters

- **ctx** – salsa20 context.
- **input** – input data buffer (read), *count* bytes long.
- **output** – output data buffer (written), *count* bytes long.

void **cf_chacha20_cipher**(*cf_chacha20_ctx* *ctx, const uint8_t *input, uint8_t *output, size_t count)
Chacha20 encryption/decryption function.

Parameters

- **ctx** – chacha20 context.

- **input** – input data buffer (read), *count* bytes long.
- **output** – output data buffer (written), *count* bytes long.

CHAPTER 7

Block cipher modes

7.1 CBC mode

This implementation allows encryption or decryption of whole blocks in CBC mode. It does not offer a byte-wise incremental interface, or do any padding.

This mode provides no useful integrity and should not be used directly.

7.1.1 Types

cf_cbc
: This structure binds together the things needed to encrypt/decrypt whole blocks in CBC mode.

  cf_cbc.prp

  How to encrypt or decrypt blocks. This could be, for example, *cf_aes*.

  cf_cbc.prpctx

  Private data for prp functions. For a *prp* of *cf_aes*, this would be a pointer to a *cf_aes_context* instance.

  cf_cbc.block

  The IV or last ciphertext block.

7.1.2 Functions

void **cf_cbc_init** (*cf_cbc* *ctx, const *cf_prp* *prp, void *prpctx, const uint8_t *iv[CF_MAXBLOCK]*)
: Initialise CBC encryption/decryption context using selected prp, prp context and IV.

void **cf_cbc_encrypt** (*cf_cbc* *ctx, const uint8_t *input, uint8_t *output, size_t *blocks*)
: Encrypt blocks in CBC mode. input and output must point to blocks * ctx->prp->blocksz bytes of storage (and may alias).

void **cf_cbc_decrypt** (*cf_cbc* *ctx, const uint8_t *input, uint8_t *output, size_t *blocks*)
Decrypt blocks in CBC mode. input and output must point to blocks * ctx->prp->blocksz bytes of storage (and may alias).

7.2 Counter mode

This implementation allows incremental encryption/decryption of messages. Encryption and decryption are the same operation.

The counter is always big-endian, but has configurable location and size within the nonce block. The counter wraps, so you should make sure the length of a message with a given nonce doesn't cause nonce reuse.

This mode provides no integrity and should not be used directly.

7.2.1 Types

cf_ctr

cf_ctr.prp

How to encrypt or decrypt blocks. This could be, for example, *cf_aes*.

cf_ctr.prpctx

Private data for prp functions. For a *prp* of *cf_aes*, this would be a pointer to a *cf_aes_context* instance.

cf_ctr.nonce

The next block to encrypt to get another block of key stream.

cf_ctr.keymat

The current block of key stream.

cf_ctr.nkeymat

The number of bytes at the end of `keymat` that are so-far unused. If this is zero, all the bytes are used up and/or of undefined value.

cf_ctr.counter_offset

The offset (in bytes) of the counter block within the nonce.

cf_ctr.counter_width

The width (in bytes) of the counter block in the nonce.

7.2.2 Functions

void **cf_ctr_init** (*cf_ctr* *ctx, const *cf_prp* *prp, void *prpctx, const uint8_t *nonce[CF_MAXBLOCK]*)
Initialise CTR encryption/decryption context using selected prp and nonce. (nb, this only increments the whole nonce as a big endian block)

void **cf_ctr_custom_counter** (*cf_ctr* *ctx, size_t *offset*, size_t *width*)
Set the location and width of the nonce counter.

eg. offset = 12, width = 4 means the counter is mod 2^32 and placed at the end of the nonce.

void **cf_ctr_cipher**(*cf_ctr* *ctx, const uint8_t *input, uint8_t *output, size_t bytes)
Encrypt or decrypt bytes in CTR mode. input and output may alias and must point to specified number of bytes.

void **cf_ctr_discard_block**(*cf_ctr* *ctx)
Discards the rest of this block of key stream.

7.3 CBC-MAC

This is a incremental interface to computing a CBC-MAC tag over a message.

It optionally pads the message with PKCS#5/PKCS#7 padding – if you don't do this, messages must be an exact number of blocks long.

You shouldn't use this directly because it isn't secure for variable-length messages. Use CMAC instead.

7.3.1 Types

cf_cbcmac_stream
Stream interface to CBC-MAC signing.

cf_cbcmac.prp

How to encrypt or decrypt blocks. This could be, for example, `cf_aes`.

cf_cbcmac.prpctx

Private data for prp functions. For a *prp* of *cf_aes*, this would be a pointer to a `cf_aes_context` instance.

cf_cbcmac.cbc

CBC data.

cf_cbcmac.buffer

Buffer for data which can't be processed until we have a full block.

cf_cbcmac.used

How many bytes at the front of `buffer` are valid.

7.3.2 Functions

void **cf_cbcmac_stream_init**(*cf_cbcmac_stream* *ctx, const *cf_prp* *prp, void *prpctx)
Initialise CBC-MAC signing context using selected prp.

void **cf_cbcmac_stream_reset**(*cf_cbcmac_stream* *ctx)
Reset the streaming signing context, to sign a new message.

void **cf_cbcmac_stream_update**(*cf_cbcmac_stream* *ctx, const uint8_t *data, size_t ndata)
Process ndata bytes at data.

void **cf_cbcmac_stream_finish_block_zero**(*cf_cbcmac_stream* *ctx)
Finish the current block of data by adding zeroes. Does nothing if there are no bytes awaiting processing.

void **cf_cbcmac_stream_nopad_final**(*cf_cbcmac_stream* *ctx, uint8_t *out[CF_MAXBLOCK]*)
Output the MAC to ctx->prp->blocksz bytes at out. ctx->used must be zero: the inputed message must be an exact number of blocks.

void **cf_cbcmac_stream_pad_final** (*cf_cbcmac_stream* *ctx, uint8_t *out[CF_MAXBLOCK]*)
Output the MAC to ctx->prp->blocksz bytes at out.

The message is padded with PKCS#5 padding.

7.4 CMAC

This is both a one-shot and incremental interface to computing a CMAC tag over a message.

The one-shot interface separates out the per-key computation, so if you need to compute lots of MACs with one key you don't pay that cost more than once.

CMAC is a good choice for a symmetric MAC.

7.4.1 Types

cf_cmac
One-shot interface to CMAC signing.

cf_cmac.prp

How to encrypt or decrypt blocks. This could be, for example, *cf_aes*.

cf_cmac.prpctx

Private data for prp functions. For a *prp* of *cf_aes*, this would be a pointer to a *cf_aes_context* instance.

cf_cmac.B

The XOR offset for the last message block if it is a complete block (also known as K_1).

cf_cmac.P

The XOR offset for the last message block if it is a partial block (also known as K_2).

cf_cmac_stream
Stream interface to CMAC signing.

Input data in arbitrary chunks using *cf_cmac_stream_update()*. The last bit of data must be signalled with the *isfinal* flag to that function, and the data cannot be zero length unless the whole message is empty.

cf_cmac_stream.cmac

CMAC one-shot data.

cf_cmac_stream.cbc

CBC block encryption data.

cf_cmac_stream.buffer

Buffer for data which can't be processed until we have a full block.

cf_cmac_stream.used

How many bytes at the front of `buffer` are valid.

cf_cmac_stream.processed

How many bytes in total we've processed. This is used to correctly process empty messages.

cf_cmac_stream.finalised

A flag set when the final chunk of the message has been processed. Only when this flag is set can you get the MAC out.

7.4.2 Functions

void **cf_cmac_init** (*cf_cmac* *ctx, const *cf_prp* *prp, void *prpctx)
Initialise CMAC signing context using selected prp.

void **cf_cmac_sign** (*cf_cmac* *ctx, const uint8_t *data, size_t bytes, uint8_t out[CF_MAXBLOCK])
CMAC sign the given data. The MAC is written to ctx->prp->blocksz bytes at out. This is a one-shot function.

void **cf_cmac_stream_init** (*cf_cmac_stream* *ctx, const *cf_prp* *prp, void *prpctx)
Initialise CMAC streaming signing context using selected prp.

void **cf_cmac_stream_reset** (*cf_cmac_stream* *ctx)
Reset the streaming signing context, to sign a new message.

void **cf_cmac_stream_update** (*cf_cmac_stream* *ctx, const uint8_t *data, size_t ndata, int isfinal)
Process ndata bytes at data. isfinal is non-zero if this is the last piece of data.

void **cf_cmac_stream_final** (*cf_cmac_stream* *ctx, uint8_t out[CF_MAXBLOCK])
Output the MAC to ctx->cmac->prp->blocksz bytes at out. cf_cmac_stream_update with isfinal non-zero must have been called since the last _init/_reset.

7.5 EAX

The EAX authenticated encryption mode. This is a one-shot interface.

EAX is a pretty respectable and fast AEAD mode.

7.5.1 Functions

void **cf_eax_encrypt** (const *cf_prp* *prp, void *prpctx, const uint8_t *plain, size_t nplain, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, uint8_t *cipher, uint8_t *tag, size_t ntag)
EAX authenticated encryption.

This function does not fail.

Parameters

- **prp/prpctx** – describe the block cipher to use.
- **plain** – message plaintext.
- **nplain** – length of message. May be zero.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD. May be zero.
- **nonce** – nonce. This must not repeat for a given key.
- **nnonce** – length of nonce. The nonce can be any length.
- **cipher** – ciphertext output. *nplain* bytes are written here.
- **tag** – authentication tag. *ntag* bytes are written here.
- **ntag** – authentication tag length. This must be non-zero and no greater than *prp->blocksz*.

int **cf_eax_decrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *cipher, size_t ncipher, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, const uint8_t *tag, size_t ntag, uint8_t *plain)

EAX authenticated decryption.

Returns 0 on success, non-zero on error. Nothing is written to plain on error.

Parameters

- **prp/prpctx** – describe the block cipher to use.
- **cipher** – message ciphertext.
- **ncipher** – message length.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD.
- **nonce** – nonce.
- **nnonce** – length of nonce.
- **tag** – authentication tag. *ntag* bytes are read from here.
- **ntag** – authentication tag length.
- **plain** – plaintext output. *ncipher* bytes are written here.

7.6 GCM

The GCM ('Galois counter mode') authenticated encryption mode. This is a one-shot interface.

GCM is a reasonably respectable AEAD mode. It's somewhat more complex than EAX, and side channel-free implementations can be quite slow.

7.6.1 Functions

void **cf_gcm_encrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *plain, size_t nplain, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, uint8_t *cipher, uint8_t *tag, size_t ntag)

GCM authenticated encryption.

This function does not fail.

Parameters

- **prp/prpctx** – describe the block cipher to use.
- **plain** – message plaintext.
- **nplain** – length of message. May be zero.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD. May be zero.
- **nonce** – nonce. This must not repeat for a given key.
- **nnonce** – length of nonce. The nonce can be any length, but 12 bytes is strongly recommended.
- **cipher** – ciphertext output. *nplain* bytes are written here.

- **tag** – authentication tag. *ntag* bytes are written here.
- **ntag** – authentication tag length. This must be non-zero and no greater than *prp->blocksz*.

This function does not fail.

int **cf_gcm_decrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *cipher, size_t ncipher, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, const uint8_t *tag, size_t ntag, uint8_t *plain)

GCM authenticated decryption.

Returns 0 on success, non-zero on error. Nothing is written to plain on error.

Parameters

- **prp** – describe the block cipher to use.
- **prpctx** – describe the block cipher to use.
- **cipher** – message ciphertext.
- **ncipher** – message length.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD.
- **nonce** – nonce.
- **nnonce** – length of nonce.
- **tag** – authentication tag. *ntag* bytes are read from here.
- **ntag** – authentication tag length.
- **plain** – plaintext output. *ncipher* bytes are written here.

7.7 CCM

The CCM ('Counter with CBC-MAC') authenticated encryption mode. CCM is a widely used AEAD mode (in IPSec, WPA2, Bluetooth, etc.)

It works (at a high level) by just gluing together CTR and CBC-MAC modes (in MAC-then-encrypt mode) and then fixing the problems inherent with CBC-MAC in over-complicated ways.

This is a one-shot interface, which is good because the underlying mechanism isn't actually online: you need to know the message length before you start, or do everything in two passes.

7.7.1 Functions

void **cf_ccm_encrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *plain, size_t nplain, size_t L, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, uint8_t *cipher, uint8_t *tag, size_t ntag)

CCM authenticated encryption.

This function does not fail.

Parameters

- **prp/prpctx** – describe the block cipher to use.
- **plain** – message plaintext.

- **nplain** – length of message. May be zero. Must meet the constraints placed on it by *L*.
- **L** – length of the message length encoding. This must be in the interval *[2,8]* and gives a maximum message size of 2^{8L} bytes.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD. May be zero.
- **nonce** – nonce. This must not repeat for a given key.
- **nnonce** – length of nonce. Must be exactly *15 - L* bytes for a 128-bit block cipher.
- **cipher** – ciphertext output. *nplain* bytes are written here.
- **tag** – authentication tag. *ntag* bytes are written here.
- **ntag** – authentication tag length. This must be 4, 6, 8, 10, 12, 14 or 16.

int **cf_ccm_decrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *cipher, size_t ncipher, size_t L, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, const uint8_t *tag, size_t ntag, uint8_t *plain)

CCM authenticated decryption.

Returns 0 on success, non-zero on error. Plain is cleared on error.

Parameters

- **prp** – describe the block cipher to use.
- **prpctx** – describe the block cipher to use.
- **cipher** – message ciphertext.
- **ncipher** – length of message.
- **L** – length of the message length encoding. See *cf_ccm_encrypt()*.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD.
- **nonce** – nonce.
- **nnonce** – length of nonce.
- **tag** – authentication tag. *ntag* bytes are read from here.
- **ntag** – authentication tag length. This must be 4, 6, 8, 10, 12, 14 or 16.
- **plain** – plaintext output. *ncipher* bytes are written here.

7.8 OCB

OCB is an authenticated encryption mode by Phil Rogaway.

This is version 3, as standardised in RFC7253. It's defined only for block ciphers with a 128-bit block size.

This is a one-shot interface.

7.8.1 Functions

void **cf_ocb_encrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *plain, size_t nplain, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, uint8_t *cipher, uint8_t *tag, size_t ntag)

OCB authenticated encryption.

This function does not fail.

Parameters

- **prp/prpctx** – describe the block cipher to use.
- **plain** – message plaintext.
- **nplain** – length of message. May be zero.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD. May be zero.
- **nonce** – nonce. This must not repeat for a given key.
- **nnonce** – length of nonce. Must be 15 or fewer bytes.
- **cipher** – ciphertext output. *nplain* bytes are written here.
- **tag** – authentication tag. *ntag* bytes are written here.
- **ntag** – authentication tag length. Must be 16 or fewer bytes.

int **cf_ocb_decrypt**(const *cf_prp* *prp, void *prpctx, const uint8_t *cipher, size_t ncipher, const uint8_t *header, size_t nheader, const uint8_t *nonce, size_t nnonce, const uint8_t *tag, size_t ntag, uint8_t *plain)

OCB authenticated decryption.

Returns 0 on success, non-zero on error. *plain* is cleared on error.

Parameters

- **prp** – describe the block cipher to use.
- **prpctx** – describe the block cipher to use.
- **cipher** – message ciphertext.
- **ncipher** – length of message.
- **header** – additionally authenticated data (AAD).
- **nheader** – length of AAD.
- **nonce** – nonce.
- **nnonce** – length of nonce.
- **tag** – authentication tag. *ntag* bytes are read from here.
- **ntag** – authentication tag length.
- **plain** – plaintext output. *ncipher* bytes are written here.

CHAPTER 8

HMAC

This is a one-shot and incremental interface to computing HMAC with any hash function.

(Note: HMAC with SHA3 is possible, but is probably not a sensible thing to want.)

8.1 Types

cf_hmac_ctx
HMAC incremental interface context.

cf_hmac_ctx.hash

Hash function description.

cf_hmac_ctx.inner

Inner hash computation.

cf_hmac_ctx.outer

Outer hash computation.

8.2 Functions

void **cf_hmac_init** (*cf_hmac_ctx* *ctx, const *cf_chash* *hash, const uint8_t *key, size_t nkey)
Set up ctx for computing a HMAC using the given hash and key.

void **cf_hmac_update** (*cf_hmac_ctx* *ctx, const void *data, size_t ndata)
Input data.

void **cf_hmac_finish** (*cf_hmac_ctx* *ctx, uint8_t *out)
Finish and compute HMAC. *ctx->hash->hashsz* bytes are written to *out*.

void **cf_hmac**(const uint8_t **key*, size_t *nkey*, const uint8_t **msg*, size_t *nmsg*, uint8_t **out*, const *cf_chash* **hash*)

One shot interface: compute *HMAC_hash(key, msg)*, writing the answer (which is *hash->hashsz* long) to *out*.

This function does not fail.

CHAPTER 9

Poly1305

This is an incremental interface to computing the poly1305 single shot MAC.

Note: construct Poly1305-AES with this by taking a 16 byte nonce and encrypting it, and then using the result as an input to this function.

9.1 Types

cf_poly1305
: Poly1305 incremental interface context.

  cf_poly1305.h

  Current accumulator.

  cf_poly1305.r

  Block multiplier.

  cf_poly1305.s

  Final XOR offset.

  cf_poly1305.partial

  Unprocessed input.

  cf_poly1305.npartial

  Number of bytes of unprocessed input.

9.2 Functions

void **cf_poly1305_init** (*cf_poly1305* **ctx*, const uint8_t *r[16]*, const uint8_t *s[16]*)
: Sets up *ctx* ready to compute a new MAC.

In Poly1305-AES, *r* is the second half of the 32-byte key. *s* is a nonce encrypted under the first half of the key.

Parameters

- **ctx** – context (written)
- **r** – MAC key.
- **s** – preprocessed nonce.

void **cf_poly1305_update**(*cf_poly1305* *ctx, const uint8_t *data, size_t *nbytes*)
Processes *nbytes* at *data*. Copies the data if there isn't enough to make a full block.

void **cf_poly1305_finish**(*cf_poly1305* *ctx, uint8_t *out[16]*)
Finishes the operation, writing 16 bytes to *out*.

This destroys *ctx*.

CHAPTER 10

The ChaCha20-Poly1305 AEAD construction

This is a composition of the ChaCha20 stream cipher and the Poly1305 polynomial MAC to form an AEAD. It's specified for use in TLS in the form of RFC7539.

It uses a 256-bit key and a 96-bit nonce.

This is a one-shot interface.

10.1 Functions

void **cf_chacha20poly1305_encrypt**(const uint8_t *key[32]*, const uint8_t *nonce[12]*, const uint8_t **header*, size_t *nheader*, const uint8_t **plaintext*, size_t *nbytes*, uint8_t **ciphertext*, uint8_t *tag[16]*)

ChaCha20-Poly1305 authenticated encryption.

Parameters

- **key** – key material.
- **nonce** – per-message nonce.
- **header** – header buffer.
- **nheader** – number of header bytes.
- **plaintext** – plaintext bytes to be encrypted.
- **nbytes** – number of plaintext/ciphertext bytes.
- **ciphertext** – ciphertext output buffer, nbytes in length.
- **tag** – authentication tag output buffer.

int **cf_chacha20poly1305_decrypt**(const uint8_t *key[32]*, const uint8_t *nonce[12]*, const uint8_t **header*, size_t *nheader*, const uint8_t **ciphertext*, size_t *nbytes*, const uint8_t *tag[16]*, uint8_t **plaintext*)

ChaCha20-Poly1305 authenticated decryption.

Returns 0 on success, non-zero on error. Plaintext is zeroed on error.

Parameters

- **key** – key material.
- **nonce** – per-message nonce.
- **header** – header buffer.
- **nheader** – number of header bytes.
- **ciphertext** – ciphertext bytes to be decrypted.
- **nbytes** – number of plaintext/ciphertext bytes.
- **plaintext** – plaintext output buffer, nbytes in length.
- **tag** – authentication tag output buffer.

CHAPTER 11

PBKDF2-HMAC

This is PBKDF2 as described by PKCS#5/RFC2898 with HMAC as the PRF.

11.1 Functions

void **cf_pbkdf2_hmac**(const uint8_t **pw*, size_t *npw*, const uint8_t **salt*, size_t *nsalt*, uint32_t *iterations*, uint8_t **out*, size_t *nout*, const *cf_chash* **hash*)

This computes PBKDF2-HMAC with the given hash functon.

Parameters

- **pw** – password input buffer.
- **npw** – password length.
- **salt** – salt input buffer.
- **nsalt** – salt length.
- **iterations** – non-zero iteration count. Tune this for performance/security tradeoff.
- **out** – key material output buffer. *nout* bytes are written here.
- **nout** – key material length.
- **hash** – hash function description.

CHAPTER 12

SHA1

You shouldn't use this for anything new.

12.1 Macros

CF_SHA1_HASHSZ
: The output size of SHA1: 20 bytes.

CF_SHA1_BLOCKSZ
: The block size of SHA1: 64 bytes.

12.2 Types

cf_sha1_context
: Incremental SHA1 hashing context.

  cf_sha1_context.H

  Intermediate values.

  cf_sha1_context.partial

  Unprocessed input.

  cf_sha1_context.npartial

  Number of bytes of unprocessed input.

  cf_sha1_context.blocks

  Number of full blocks processed.

12.3 Functions

void **cf_sha1_init**(*cf_sha1_context* *ctx)
Sets up *ctx* ready to hash a new message.

void **cf_sha1_update**(*cf_sha1_context* *ctx, const void *data, size_t *nbytes*)
Hashes *nbytes* at *data*. Copies the data if there isn't enough to make a full block.

void **cf_sha1_digest**(const *cf_sha1_context* *ctx, uint8_t *hash[CF_SHA1_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA1_HASHSZ* bytes to *hash*.

This leaves *ctx* unchanged.

void **cf_sha1_digest_final**(*cf_sha1_context* *ctx, uint8_t *hash[CF_SHA1_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA1_HASHSZ* bytes to *hash*.

This destroys *ctx*, but uses less stack than `cf_sha1_digest()`.

12.4 Values

cf_sha1
Abstract interface to SHA1. See `cf_chash` for more information.

CHAPTER 13

SHA224/SHA256

13.1 Macros

CF_SHA224_HASHSZ
The output size of SHA224: 28 bytes.

CF_SHA224_BLOCKSZ
The block size of SHA224: 64 bytes.

CF_SHA256_HASHSZ
The output size of SHA256: 32 bytes.

CF_SHA256_BLOCKSZ
The block size of SHA256: 64 bytes.

13.2 Types

cf_sha256_context
Incremental SHA256 hashing context.

cf_sha256_context.H

Intermediate values.

cf_sha256_context.partial

Unprocessed input.

cf_sha256_context.npartial

Number of bytes of unprocessed input.

cf_sha256_context.blocks

Number of full blocks processed.

13.3 Functions

void **cf_sha256_init** (*cf_sha256_context* *ctx)
Sets up *ctx* ready to hash a new message.

void **cf_sha256_update** (*cf_sha256_context* *ctx, const void *data, size_t nbytes)
Hashes *nbytes* at *data*. Copies the data if there isn't enough to make a full block.

void **cf_sha256_digest** (const *cf_sha256_context* *ctx, uint8_t *hash[CF_SHA256_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA256_HASHSZ* bytes to *hash*.

This leaves *ctx* unchanged.

void **cf_sha256_digest_final** (*cf_sha256_context* *ctx, uint8_t *hash[CF_SHA256_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA256_HASHSZ* bytes to *hash*.

This destroys *ctx*, but uses less stack than `cf_sha256_digest()`.

void **cf_sha224_init** (*cf_sha256_context* *ctx)
Sets up *ctx* ready to hash a new message.

nb. SHA224 uses SHA256's underlying types.

void **cf_sha224_update** (*cf_sha256_context* *ctx, const void *data, size_t nbytes)
Hashes *nbytes* at *data*. Copies the data if there isn't enough to make a full block.

void **cf_sha224_digest** (const *cf_sha256_context* *ctx, uint8_t *hash[CF_SHA224_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA224_HASHSZ* bytes to *hash*.

This leaves *ctx* unchanged.

void **cf_sha224_digest_final** (*cf_sha256_context* *ctx, uint8_t *hash[CF_SHA224_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA224_HASHSZ* bytes to *hash*.

This destroys *ctx*, but uses less stack than `cf_sha224_digest()`.

13.4 Values

cf_sha224
Abstract interface to SHA224. See `cf_chash` for more information.

cf_sha256
Abstract interface to SHA256. See `cf_chash` for more information.

CHAPTER 14

SHA384/SHA512

14.1 Macros

CF_SHA384_HASHSZ
The output size of SHA384: 48 bytes.

CF_SHA384_BLOCKSZ
The block size of SHA384: 128 bytes.

CF_SHA512_HASHSZ
The output size of SHA512: 64 bytes.

CF_SHA512_BLOCKSZ
The block size of SHA512: 128 bytes.

14.2 Types

cf_sha512_context
Incremental SHA512 hashing context.

cf_sha512_context.H

Intermediate values.

cf_sha512_context.partial

Unprocessed input.

cf_sha512_context.npartial

Number of bytes of unprocessed input.

cf_sha512_context.blocks

Number of full blocks processed.

14.3 Functions

void **cf_sha512_init** (*cf_sha512_context* *ctx)
Sets up *ctx* ready to hash a new message.

void **cf_sha512_update** (*cf_sha512_context* *ctx, const void *data, size_t nbytes)
Hashes *nbytes* at *data*. Copies the data if there isn't enough to make a full block.

void **cf_sha512_digest** (const *cf_sha512_context* *ctx, uint8_t *hash[CF_SHA512_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA512_HASHSZ* bytes to *hash*.

This leaves *ctx* unchanged.

void **cf_sha512_digest_final** (*cf_sha512_context* *ctx, uint8_t *hash[CF_SHA512_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA512_HASHSZ* bytes to *hash*.

This destroys *ctx*, but uses less stack than `cf_sha512_digest()`.

void **cf_sha384_init** (*cf_sha512_context* *ctx)
Sets up *ctx* ready to hash a new message.

nb. SHA384 uses SHA512's underlying types.

void **cf_sha384_update** (*cf_sha512_context* *ctx, const void *data, size_t nbytes)
Hashes *nbytes* at *data*. Copies the data if there isn't enough to make a full block.

void **cf_sha384_digest** (const *cf_sha512_context* *ctx, uint8_t *hash[CF_SHA384_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA384_HASHSZ* bytes to *hash*.

This leaves *ctx* unchanged.

void **cf_sha384_digest_final** (*cf_sha512_context* *ctx, uint8_t *hash[CF_SHA384_HASHSZ]*)
Finishes the hash operation, writing *CF_SHA384_HASHSZ* bytes to *hash*.

This destroys *ctx*, but uses less stack than `cf_sha384_digest()`.

14.4 Values

cf_sha384
Abstract interface to SHA384. See `cf_chash` for more information.

cf_sha512
Abstract interface to SHA512. See `cf_chash` for more information.

CHAPTER 15

SHA3/Keccak

This implementation is compatible with FIPS 202, rather than the original Keccak submission.

15.1 Macros

CF_SHA3_224_HASHSZ
The output size of SHA3-224: 28 bytes.

CF_SHA3_256_HASHSZ
The output size of SHA3-256: 32 bytes.

CF_SHA3_384_HASHSZ
The output size of SHA3-384: 48 bytes.

CF_SHA3_512_HASHSZ
The output size of SHA3-512: 64 bytes.

CF_SHA3_224_BLOCKSZ
The block size of SHA3-224.

CF_SHA3_256_BLOCKSZ
The block size of SHA3-256.

CF_SHA3_384_BLOCKSZ
The block size of SHA3-384.

CF_SHA3_512_BLOCKSZ
The block size of SHA3-512.

15.2 Types

cf_sha3_context
Incremental SHA3 hashing context.

cf_sha3_context.A

Intermediate state.

cf_sha3_context.partial

Unprocessed input.

cf_sha3_context.npartial

Number of bytes of unprocessed input.

cf_sha3_context.rate

Sponge absorption rate.

cf_sha3_context.rate

Sponge capacity.

15.3 Functions

void **cf_sha3_224_init**(*cf_sha3_context* *ctx)

void **cf_sha3_256_init**(*cf_sha3_context* *ctx)

void **cf_sha3_384_init**(*cf_sha3_context* *ctx)

void **cf_sha3_512_init**(*cf_sha3_context* *ctx)
Sets up *ctx* ready to hash a new message.

void **cf_sha3_224_update**(*cf_sha3_context* *ctx, const void *data, size_t *nbytes*)

void **cf_sha3_256_update**(*cf_sha3_context* *ctx, const void *data, size_t *nbytes*)

void **cf_sha3_384_update**(*cf_sha3_context* *ctx, const void *data, size_t *nbytes*)

void **cf_sha3_512_update**(*cf_sha3_context* *ctx, const void *data, size_t *nbytes*)
Hashes *nbytes* at *data*. Copies the data for processing later if there isn't enough to make a full block.

void **cf_sha3_224_digest**(const *cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_224_HASHSZ]*)

void **cf_sha3_256_digest**(const *cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_256_HASHSZ]*)

void **cf_sha3_384_digest**(const *cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_384_HASHSZ]*)

void **cf_sha3_512_digest**(const *cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_512_HASHSZ]*)
Finishes the hashing operation, writing result to *hash*.

This leaves *ctx* unchanged.

void **cf_sha3_224_digest_final**(*cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_224_HASHSZ]*)

void **cf_sha3_256_digest_final**(*cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_256_HASHSZ]*)

void **cf_sha3_384_digest_final**(*cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_384_HASHSZ]*)

void **cf_sha3_512_digest_final**(*cf_sha3_context* *ctx, uint8_t *hash[CF_SHA3_512_HASHSZ]*)
Finishes the hashing operation, writing result to *hash*.

This destroys the contents of *ctx*.

15.4 Values

cf_sha3_224

cf_sha3_256

cf_sha3_384

cf_sha3_512

Abstract interface to SHA3 functions. See *cf_chash* for more information.

CHAPTER 16

Hash_DRBG

This is Hash_DRBG from SP800-90A rev 1, with SHA256 as the underlying hash function.

This generator enforces a *reseed_interval* of 2^32-1: use `cf_hash_drbg_sha256_needs_reseed()` to check whether you need to reseed before use, and reseed using `cf_hash_drbg_sha256_reseed()`. If you try to use the generator when it thinks it needs reseeding, it will call *abort*.

Internally it enforces a *max_number_of_bits_per_request* of 2^19 bits. It sorts out chunking up multiple requests for you though, so feel free to ask for more than 2^16 bytes at a time. If you provide additional input when doing that, it is added only once, on the first subrequest.

It does not enforce any *max_length* or *max_personalization_string_length*.

16.1 Types

cf_hash_drbg_sha256
Hash_DRBG with SHA256 context.

cf_hash_drbg_sha256.V

Current internal state.

cf_hash_drbg_sha256.C

Current update offset.

cf_hash_drbg_sha256.reseed_counter

Current number of times entropy has been extracted from generator.

16.2 Functions

void **cf_hash_drbg_sha256_init**(*cf_hash_drbg_sha256* *ctx, const void *entropy, size_t *nentropy*, const void *nonce, size_t *nnonce*, const void *persn, size_t *npersn*)
Initialises the generator state *ctx*, using the provided *entropy*, *nonce* and personalisation string *persn*.

uint32_t **cf_hash_drbg_sha256_needs_reseed**(const *cf_hash_drbg_sha256* *ctx)

Returns non-zero if the generator needs reseeding. If this function returns non-zero, the next `cf_hash_drbg_sha256_gen()` or `cf_hash_drbg_sha256_gen_additional()` call will call *abort*.

void **cf_hash_drbg_sha256_reseed**(*cf_hash_drbg_sha256* *ctx, const void *entropy, size_t nentropy, const void *addnl, size_t naddnl)

Reseeds the generator with the given *entropy* and additional data *addnl*.

void **cf_hash_drbg_sha256_gen**(*cf_hash_drbg_sha256* *ctx, void *out, size_t nout)

Generates pseudo-random output, writing *nout* bytes at *out*. This function aborts if the generator needs seeding.

void **cf_hash_drbg_sha256_gen_additional**(*cf_hash_drbg_sha256* *ctx, const void *addnl, size_t naddnl, void *out, size_t nout)

Generates pseudo-random output, writing *nout* bytes at *out*. At the same time, *addnl* is input to the generator as further entropy. This function aborts if the generator needs seeding.

CHAPTER 17

HMAC_DRBG

This is HMAC_DRBG from SP800-90a r1 with any hash function.

This generator enforces a *reseed_interval* of 2^32-1: use *cf_hmac_drbg_needs_reseed()* to check whether you need to reseed before use, and reseed using *cf_hmac_drbg_reseed()*. If you try to use the generator when it thinks it needs reseeding, it will call *abort*.

Internally it enforces a *max_number_of_bits_per_request* of 2^19 bits. It sorts out chunking up multiple requests for you though, so feel free to ask for more than 2^16 bytes at a time. If you provide additional input when doing that, it is added only once, on the first subrequest.

It does not enforce any *max_length* or *max_personalization_string_length*.

17.1 Types

cf_hmac_drbg
HMAC_DRBG context.

cf_hmac_drbg.V

Current internal state.

cf_hmac_drbg.hmac

Current HMAC context, with key scheduled in it.

cf_hmac_drbg.reseed_counter

Current number of times entropy has been extracted from generator.

17.2 Functions

void **cf_hmac_drbg_init**(*cf_hmac_drbg* *ctx, const *cf_chash* *hash, const void *entropy, size_t nentropy, const void *nonce, size_t nnonce, const void *persn, size_t npersn)
Initialises the generator state *ctx*, using the provided *entropy*, *nonce* and personalisation string *persn*.

uint32_t **cf_hmac_drbg_needs_reseed**(const *cf_hmac_drbg* *ctx)

Returns non-zero if the generator needs reseeding. If this function returns non-zero, the next `cf_hmac_drbg_gen()` or `cf_hmac_drbg_gen_additional()` call will call *abort*.

void **cf_hmac_drbg_reseed**(*cf_hmac_drbg* *ctx, const void *entropy, size_t nentropy, const void *addnl, size_t naddnl)

Reseeds the generator with the given *entropy* and additional data *addnl*.

void **cf_hmac_drbg_gen**(*cf_hmac_drbg* *ctx, void *out, size_t nout)

Generates pseudo-random output, writing *nout* bytes at *out*. This function aborts if the generator needs seeding.

void **cf_hmac_drbg_gen_additional**(*cf_hmac_drbg* *ctx, const void *addnl, size_t naddnl, void *out, size_t nout)

Generates pseudo-random output, writing *nout* bytes at *out*. At the same time, *addnl* is input to the generator as further entropy. This function aborts if the generator needs seeding.

CHAPTER 18

Index

- genindex

Index