Ordo

2.0.0

Generated by Doxygen 1.8.4

Sat Jun 22 2013 17:00:55

Contents

Chapter 1

Main Page

Symmetric Cryptography Library

This is the github repository for Ordo, a minimalist cryptography library with an emphasis on symmetric cryptography, which strives to meet high performance, portability, and security standards, while remaining modular in design to facilitate adding new features and maintaining existing ones. The library is written in standard C, but some sections are assembly-optimized for performance. Note that while the library is technically usable at this point, it is still very much a work in progress and mustn't be deployed in security-sensitive applications.

Status

Ordo v2 is out! It's not completely finished but the library has just undergone a major overhaul. Enjoy.

Planned features:

- reformatting, some implementation rewrites and finishing the documentation

A basic test driver has been implemented, which reads a test vector file (see vectors file) by parsing a simple script. It allows to see if Ordo is correctly running at a glance - if you fail a test vector or get a segfault, you have a problem. Read the vectors file (in the test program) and the test driver's code to know more about this.

The design of the library is to have cryptographic primitives (be it cipher primitives, or hash primitives, etc...) be accessible from everywhere in the code, so that different modules are able to access them transparently (like encrypting, hashing, authenticating, encrypting+authenticating, etc...). Suggested library module names:

primitives -> contains declarations for all crypto primitives (block and stream ciphers, hashes, etc...) enc -> for encryption only (either using block ciphers with modes of operation (CBC, CTR, etc...) via enc_block, or with stream ciphers with enc_stream) hash -> for hashing/digest operations (MD5, Skein, etc...) auth -> for authentication-only modes of operation (HMAC, VMAC, etc...) encauth -> for encryption+authentication modes (GCM, CCM, etc...) random -> for pseudorandom number generation (using the OS-provided CSPRNG)

This way every part of the library is cleanly separated yet can share cryptographic code. It is not clear yet how much abstraction can be obtained from each individual section of the library - for block and stream cipher encryption the abstraction level is very high as block cipher modes of operation are quite modular and stream ciphers can literally be swapped in and out at will, but for "hash" for instance it will be much lower by the very nature of how hash functions are designed.

Feature Map

Essentially finished features are in **bold**, features currently in progress are in *italic*, and planned features are in standard font.

- **random**

- *primitives*
  - *block_ciphers*
    - **NullCipher**
    - **Threefish-256**
    - *AES*
  - *stream_ciphers*
    - **RC4**
  - *hash_functions*
    - **SHA-256**
    - **MD5**
    - *Skein-256*
  - misc
- **enc**
  - **enc_block**
    - **ECB**
    - **CBC**
    - **CTR**
    - **CFB**
    - **CFB**
  - **enc_stream**
- **hash**
- *auth*
  - **hmac**
- *kdf*
  - *PBKDF2 (w/ HMAC)*
- encauth
- *testing (test drivers)*
  - *vectors*
  - *performance*

This doesn't include every single feature but gives a high level overview:

Block Ciphers	**Stream Ciphers**	**Hash Functions**	**Modes**	**Authentication**	**Key Derivation**	**Misc**
AES	RC4	MD5	ECB	HMAC	PBKDF2	CSPRNG
Threefish-256	-	SHA-256	CBC	-	-	-
-	-	Skein-256	OFB	-	-	-
-	-	-	CFB	-	-	-
-	-	-	CTR	-	-	-

Documentation

Ordo is documented for Doxygen, and you can automatically generate all documentation via `make doc`. The HTML documentation will be generated in `doc/html` and the LaTeX documentation will be generated in `doc/latex` (note you need `pdflatex` and a working LaTeX environment for this to work). Symlinks will be automatically created in the `doc` directory for your convenience.

Alternatively, you can consult the online documentation at the `project page`, though it may not be completely up to date.

How To Build & Compatibility

As Ordo is somewhat environment-dependent (it needs to know, among others, the target operating system for some platform-specific API's such as memory locking, and the target processor's endianness and native word size for processor-specific optimizations) we use a custom makefile to facilitate the build process. The makefile *requires* the `gcc` compiler. The makefile is *not* set up for cross-compiling and you will need to set this up yourself if you wish to build for different operating systems. If you are building for the current operating system, then you may tweak the processor architecture and Ordo will optimize accordingly, but unless you know what you are doing you should just build for your current system.

In general, Ordo expects to be given the following information:

- Operating system, along with various system functions. This is provided by `gcc` and Ordo will automatically select the right codepath based on the operating system `gcc` is reportedly building on.
- Endianness. This is provided by the system libraries, or inferred from the operating system (e.g. Windows is always little-endian). Byte-swapping functions need not be available as Ordo has its own fallback functions, but are recommended for efficiency.
- Processor architecture. This is, again, provided by `gcc` based on compilation flags restricting or enabling instruction sets and other features.

The makefile is used as follows:

```
make extra=[arguments to gcc]
```

Where the `extra` argument is used to refine processor specification. For instance, if your processor supports the AES-NI instructions, you will want to pass `extra="-maes"`. If you want full optimization for your own system, you will want to provide `extra="-march=native"`. Those are passed directly to `gcc` so you can provide extra architecture information if you have more information on your target processor, in order to optimize the library further.

If your operating system is supported by Ordo, it *will run* as everything has a standard C code path. However, if specific optimizations are not available for your system and/or processor architecture, performance may not be ideal.

Finally, there are a few additional configuration options possible:

- `make strip=1` will strip symbols from the the built libraries using the `strip` tool, generally making them a bit smaller.
- `make debug=1` will enable the debug build functionality, which will disable all optimizations and assembly code paths, and enable `gdb` symbols. By default, debug mode is not enabled.
- `make shared=1` will build a shared library (`libordo.so`) instead of a static one (`libordo.a`) by default. Note that you will need to `make clean` if you want to change from a static to a shared library, as the object files are not compatible between both library types (shared libraries require position independent code whereas static ones don't).

To build and run the tests, use `make tests` and `make run_tests`. To build the samples, use `make samples`. The samples will be built into the `samples/bin` directory where you can try them out. Note the `shared`, `debug`, and `strip` arguments also apply to the tests and samples.
For most uses, the build process should go like this:

```
make
make doc
make tests
make run_tests # here, check it works properly
make samples
```

Finally, `make clean` will remove all generated files in the repository, leaving behind only original content.

Conclusion

Of course, do not use Ordo for anything other than testing or contributing for now! It can only be used once it has been completed and extensively checked (and even then, there may still be flaws and bugs, as in any other software).

Chapter 2

Todo List

File aes.S

Fix 32-bit "undefined reference" bug.

Implement AES key schedule with AES-NI, to get rid of the substitution box.

File random.h

Implement ordo_random for other platforms and add proper error handling for Windows.

File rc4.h

Better ABI translation for Windows assembler implementation(right now it's a brute-force push/pop/swap to explicitly translate parameter passing)

File skein256.h

Expand Skein-256 parameters (add possible extra blocks, such as personalization, hmac, nonce, etc...). This will probably require a rewrite of the UBI subsystem which is rather hardcoded and rigid at the moment.

Rewrite the UBI code properly.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 AES_PARAMS Struct Reference

```
#include <block_params.h>
```

Data Fields

- size_t rounds

5.1.1 Detailed Description

AES cipher parameters.

5.1.2 Field Documentation

5.1.2.1 size_t rounds

The number of rounds to use.

Remarks

> The defaults are 10 for a 128-bit key, 12 for a 192-bit key and 14 for a 256-bit key, and are standardized. It is strongly discouraged to lower the number of rounds below the defaults.

The documentation for this struct was generated from the following file:

- include/primitives/block_ciphers/block_params.h

5.2 CBC_PARAMS Struct Reference

CBC mode of operation parameters.

```
#include <mode_params.h>
```

Data Fields

- size_t padding

5.2.1 Detailed Description

CBC mode of operation parameters.

A parameter structure for CBC mode - this only contains whether padding should be enabled.

5.2.2 Field Documentation

5.2.2.1 size_t padding

Set the least significant bit to 0 to disable padding, 1 to enable it. All other bits are ignored. The default behaviour is 1.

The documentation for this struct was generated from the following file:

- include/enc/block_cipher_modes/mode_params.h

5.3 ECB_PARAMS Struct Reference

ECB mode of operation parameters.

`#include <mode_params.h>`

Data Fields

- size_t padding

5.3.1 Detailed Description

ECB mode of operation parameters.

A parameter structure for ECB mode - this only contains whether padding should be enabled.

5.3.2 Field Documentation

5.3.2.1 size_t padding

Set the least significant bit to 0 to disable padding, 1 to enable it. All other bits are ignored. The default behaviour is 1.

The documentation for this struct was generated from the following file:

- include/enc/block_cipher_modes/mode_params.h

5.4 RC4_PARAMS Struct Reference

`#include <stream_params.h>`

Data Fields

- size_t drop

5.4.1 Detailed Description

RC4 stream cipher parameters.

5.4.2 Field Documentation

5.4.2.1 size_t drop

The number of keystream bytes to drop prior to encryption.

The documentation for this struct was generated from the following file:

- include/primitives/stream_ciphers/stream_params.h

5.5 SKEIN256_PARAMS Struct Reference

Skein-256 hash parameters.

```
#include <hash_params.h>
```

Data Fields

- uint8_t schema [4]
- uint8_t version [2]
- uint8_t reserved [2]
- uint64_t outputLength
- uint8_t unused [16]

5.5.1 Detailed Description

Skein-256 hash parameters.

A parameter structure for Skein-256.

5.5.2 Field Documentation

5.5.2.1 uint8_t schema[4]

The schema identifier, on four bytes.

5.5.2.2 uint8_t version[2]

The version number, on two bytes.

5.5.2.3 uint8_t reserved[2]

Reserved - must be left zero.

5.5.2.4 uint64_t outputLength

Desired output length, in bits (note the actual output digest will be truncated to a byte boundary, so this should really always be a multiple of 8).

5.5.2.5 uint8_t unused[16]

Unused, must be left zero.

The documentation for this struct was generated from the following file:

- include/primitives/hash_functions/hash_params.h

5.6 THREEFISH256_PARAMS Struct Reference

```
#include <block_params.h>
```

Data Fields

- uint64_t tweak [2]

5.6.1 Detailed Description

Threefish-256 cipher parameters.

5.6.2 Field Documentation

5.6.2.1 uint64_t tweak[2]

The tweak word, on a pair of 64-bit words.

The documentation for this struct was generated from the following file:

- include/primitives/block_ciphers/block_params.h

Chapter 6

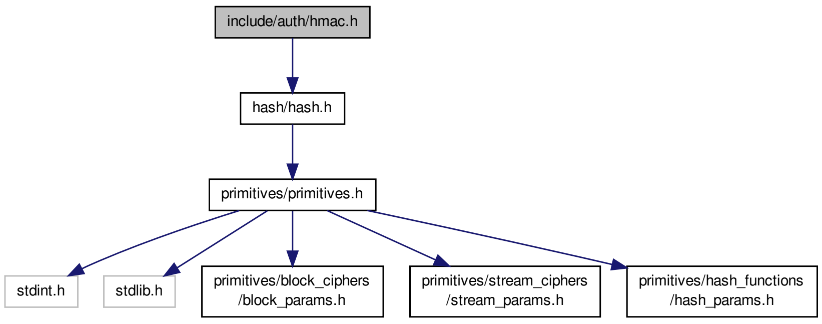
File Documentation

6.1 include/auth/hmac.h File Reference

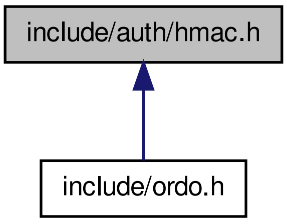
HMAC module.

```
#include <hash/hash.h>
```

Include dependency graph for hmac.h:


include/auth/hmac.h
hash/hash.h
primitives/primitives.h
stdint.h
stdlib.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

This graph shows which files directly or indirectly include this file:



Functions

- struct HMAC_CTX ∗ hmac_alloc (struct HASH_FUNCTION ∗hash)
- int hmac_init (struct HMAC_CTX ∗ctx, void ∗key, size_t key_size, void ∗hash_params)
- void hmac_update (struct HMAC_CTX ∗ctx, void ∗buffer, size_t size)
- int hmac_final (struct HMAC_CTX ∗ctx, void ∗digest)
- void hmac_free (struct HMAC_CTX ∗ctx)
- void hmac_copy (struct HMAC_CTX ∗dst, struct HMAC_CTX ∗src)

6.1.1 Detailed Description

HMAC module. Module for computing HMAC's (Hash-based Message Authentication Codes), which securely combine a hash function with a cryptographic key securely in order to provide both authentication and integrity, as per RFC 2104.

6.1.2 Function Documentation

6.1.2.1 struct HMAC_CTX∗ hmac_alloc (struct HASH_FUNCTION ∗ *hash*)

Allocates a new HMAC context.

Parameters

hash	The hash function to use.

Returns

Returns the allocated HMAC context, or nil if an error occurred.

Remarks

The PRF used for the HMAC will be the hash function as it behaves with default parameters. It is not possible to use hash function extensions (e.g. Skein in specialized HMAC mode) via this module.

6.1.2.2 int hmac_init (struct HMAC_CTX ∗ *ctx,* void ∗ *key,* size_t *key_size,* void ∗ *hash_params*)

Initializes an HMAC context, provided optional parameters.

Parameters

ctx	An allocated HMAC context.
key	A pointer to the key to use.
key_size	The size, in bytes, of the key.
hash_params	This points to specific hash function parameters, set to nil for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

Remarks

The hash parameters apply to the inner hash function only (the one used to hash the passed key with the inner mask).
Do not use hash parameters which modify the hash function's output length, or this function's behavior is undefined.

6.1.2.3 void hmac_update (struct HMAC_CTX ∗ *ctx,* void ∗ *buffer,* size_t *size*)

Updates an HMAC context, feeding more data into it.

Parameters

ctx	An allocated HMAC context.
buffer	A buffer containing the data.
size	The size, in bytes, of `buffer`.

Remarks

This function has the property that calling it in succession with buffers A and B is equivalent to calling it once by concatenating A and B together.

6.1.2.4 int hmac_final (struct HMAC_CTX * *ctx*, void * *digest*)

Finalizes a HMAC context, returning the final digest.

Parameters

ctx	An allocated HMAC context.
digest	A pointer to a buffer where the digest will be written.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

Remarks

The digest length is equal to the underlying hash function's digest length, which may be queried via `hash_-digest_length()`.

6.1.2.5 void hmac_free (struct HMAC_CTX * *ctx*)

Frees an HMAC context.

Parameters

ctx	An allocated HMAC context.

Remarks

Passing nil to this function is a no-op.

6.1.2.6 void hmac_copy (struct HMAC_CTX * *dst*, struct HMAC_CTX * *src*)

Deep-copies a context to another.

Parameters

dst	The destination context.
src	The source context.

Remarks

Both contexts need to have been allocated with the same hash function and (if initialized, which is likely) the same hash parameters, since parameters can affect the underlying hash function state's representation, unless the documentation indicates otherwise.

6.2 include/common/identification.h File Reference

Object ID manager.

6.2.1 Detailed Description

Object ID manager. This header contains definitions associating unique identifiers to block/stream ciphers, block cipher modes of operation, compression functions, hash functions, etc... This is important because the Ordo library's high level API's are based on objects (so to use the RC4 stream cipher, you would use the `RC4()` object) which are located in arrays to facilitate the implementation of functions such as `block_cipher_by_name()`, and are initialized via functions such as `loadPrimitives()` which should be called before using Ordo.

Each object has its own ID, for instance the NullCipher has the ID `BLOCK_CIPHER_NULLCIPHER` (which is defined as 0 since this is the most basic cipher, but this is arbitrary). This ID can then be used in functions such as `block_cipher_by_id()` which will return the correct block cipher object.

This also allows for a quick overview of what is implemented in Ordo so far.

6.3 include/common/ordo_errors.h File Reference

Error declarations.

Macros

- #define ORDO_SUCCESS
- #define ORDO_FAIL
- #define ORDO_LEFTOVER
- #define ORDO_KEY_SIZE
- #define ORDO_PADDING
- #define ORDO_ALLOC
- #define ORDO_ARG

6.3.1 Detailed Description

Error declarations. Contains error declarations.

6.3.2 Macro Definition Documentation

6.3.2.1 #define ORDO_SUCCESS

The function succeeded. This is defined as zero and is returned if a function encountered no error, unless specified otherwise.

6.3.2.2 #define ORDO_FAIL

The function failed due to an external error. This often indicates failure of an external component, such as the OS-provided pseudorandom number generator. Unless specified otherwise, Ordo is not responsible for this error.

6.3.2.3 #define ORDO_LEFTOVER

Unprocessed input was left over in the context. This applies to block cipher modes of operation for which padding has been disabled: if the input plaintext length is not a multiple of the cipher's block size, then the remaining incomplete block cannot be handled without padding, which is an error as it generally leads to inconsistent behavior on the part of the user.

6.3.2.4 #define ORDO_KEY_SIZE

The key size provided is invalid for this cryptographic primitive. This occurs if you give a primitive an incorrect key size, such as feeding a 128-bit key into a cipher which expects a 192-bit key. Primitives either have a range of possible key lengths (often characterized by a minimum and maximum key length, but this varies among algorithms) or only one specific key length. If you need to accept arbitrary length keys, you should consider hashing your key in some fashion before using for encryption.

6.3.2.5 #define ORDO_PADDING

The padding was not recognized and decryption could not be completed. This applies to block cipher modes for which padding is enabled: if the last block containing padding information is malformed, the latter will generally be unreadable and the correct message size cannot be retrieved, making correct decryption impossible. Note this may not occur all the time, as an incorrect last block generally has a 1/256 chance of being a valid padding block, and no error will occur (on the other hand, the returned plaintext will be incorrect). If you need to ensure the plaintext is decrypted intact, you probably want to use a MAC (Message Authentication Code) along with encryption.

6.3.2.6 #define ORDO_ALLOC

An attempt to allocate heap memory failed - this can be due to the system being low on memory or - more likely - the process to which the library is attached has reached its memory locking quota. If the former, there is not much to be done except get more memory. If the latter, either use less locked memory (which means avoiding using secure_alloc for large memory buffers) or increase your process memory locking quota by acquiring higher privileges, or simply changing the quota.

6.3.2.7 #define ORDO_ARG

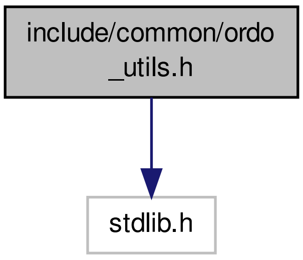
An invalid argument was passed to a function. Perhaps it was out of bounds.

6.4 include/common/ordo_utils.h File Reference

Various utility functions.

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

Include dependency graph for ordo_utils.h:



Functions

- int pad_check (unsigned char ∗buffer, unsigned char padding)
- void xor_buffer (unsigned char ∗dst, unsigned char ∗src, size_t len)
- void inc_buffer (unsigned char ∗buffer, size_t len)
- char ∗ error_msg (int code)

6.4.1 Detailed Description

Various utility functions. Contains error declarations.

6.4.2 Function Documentation

6.4.2.1 int pad_check (unsigned char ∗ *buffer,* unsigned char *padding*)

Checks whether a buffer conforms to PKCS padding.

Parameters

buffer	The buffer to check, which should point to the first padding byte.
padding	The padding byte value to check the buffer against.

Returns

Returns 1 if the buffer is valid, 0 otherwise.

6.4.2.2 void xor_buffer (unsigned char ∗ *dst,* unsigned char ∗ *src,* size_t *len*)

Performs a bitwise exclusive-or of one buffer onto another.

Parameters

dst	The destination buffer, where the result will be stored.
src	The source buffer, containing data to exclusive-or `dst` with.
len	The number of bytes to process in each buffer.

Remarks

This is conceptually equivalent to dst ^= src. Source and destination buffers may be the same (in which case the buffer will contain len zeroes).

6.4.2.3 void inc_buffer (unsigned char * *buffer,* size_t *len*)

Increments a buffer of arbitrary size as if it were a len-byte integer.

Parameters

buffer	Points to the buffer to increment.
len	The size, in bytes, of the buffer.

Remarks

Carry propagation is done left-to-right in memory storage order.

6.4.2.4 char* error_msg (int *code*)

Returns a readable error message from an error code.

Parameters

code	The error code to interpret.

Returns

A null-terminated string containing the message.

Remarks

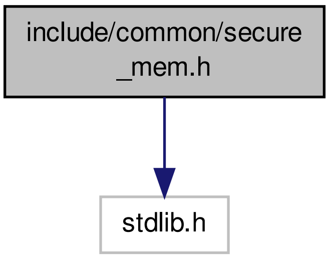
This is a placeholder convenience function used for testing only.

6.5 include/common/secure_mem.h File Reference

Secure memory API.

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

Include dependency graph for secure_mem.h:



Functions

- void * secure_alloc (size_t size)
- int secure_read_only (void *ptr, size_t size)
- void secure_erase (void *ptr, size_t size)
- void secure_free (void *ptr, size_t size)

6.5.1 Detailed Description

Secure memory API. Exposes the Secure Memory API, which is essentially a wrapper around malloc and free, taking care of locking and securely erasing memory for security-sensitive data. The library relies solely on this implementation to allocate cryptographic contexts.

6.5.2 Function Documentation

6.5.2.1 void* secure_alloc (size_t *size*)

This function returns a pointer that is locked in physical memory.

Parameters

size	The amount of memory to allocate, in bytes.

Returns

Returns the allocated pointer on success, or 0 if the function fails. The function can fail if allocation fails (if the system is out of memory) or if locking fails (if the process has reached its locked memory limit). Neither of these conditions should arise under normal operation.

Remarks

Sometimes, operating systems can decide to page out rarely-accessed memory to the hard drive. However, once the memory is needed and is paged back in, its footprint on the hard drive is not erased. Thus, if cryptographic material is paged out in this way, it can be compromised by hard drive analysis even months after the event occurred. This function prevents this by instructing the operating system not to page out the allocated memory.
Note that this is a hint to the operating system, nothing more. Consult your operating system's implementation of virtual memory locking to know more.
Memory may be left uninitialized upon allocation.

6.5.2.2 int secure_read_only (void ∗ *ptr,* size_t *size*)

This function sets memory as read-only. If this function succeeds, any attempt to write to the memory will incur an access violation, until the read-only restriction is lifted.

Parameters

ptr	The pointer to the memory to set as read-only.
size	The amount of memory, in bytes, to set as read-only.

Returns

Returns 0 on success, and anything else on failure.

6.5.2.3 void secure_erase (void ∗ *ptr,* size_t *size*)

This function wipes memory by overwriting it with zeroes.

Parameters

ptr	The pointer to the memory to wipe.
size	The amount of memory, in bytes, to wipe.

6.5.2.4 void secure_free (void ∗ *ptr,* size_t *size*)

This function frees a pointer, and securely erases the memory it points to.

Parameters

ptr	An allocated pointer to memory to erase and free.
size	The amount of memory, in bytes, pointed to by ptr.

Remarks

Passing zero to this function is valid and will do nothing.

6.6 include/common/version.h File Reference

Library version header.

Functions

- int ordo_version_major ()
- int ordo_version_minor ()
- int ordo_version_rev ()

6.6.1 Detailed Description

Library version header. This header allows code to access the library version.

6.6.2 Function Documentation

6.6.2.1 int ordo_version_major ( )

Returns the major version number of the library.

6.6.2.2 int ordo_version_minor ( )

Returns the minor version number of the library.

6.6.2.3 int ordo_version_rev ( )

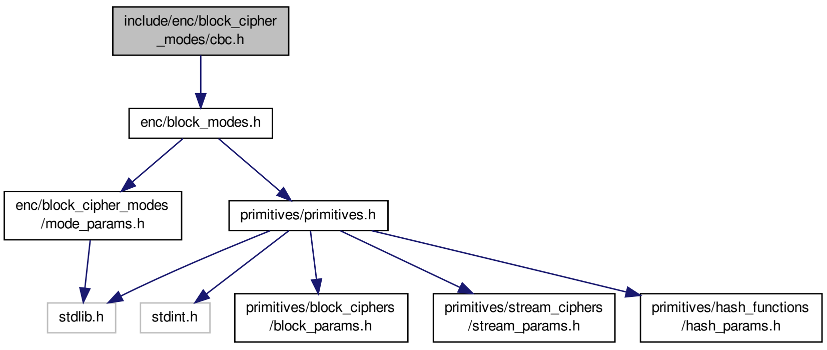
Returns the revision number of the library.

6.7 include/enc/block_cipher_modes/cbc.h File Reference

CBC block cipher mode of operation.

`#include <enc/block_modes.h>`

Include dependency graph for cbc.h:


include/enc/block_cipher_modes/cbc.h
enc/block_modes.h
enc/block_cipher_modes/mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers/block_params.h
primitives/stream_ciphers/stream_params.h
primitives/hash_functions/hash_params.h

6.7.1 Detailed Description

CBC block cipher mode of operation. The CBC mode divides the input message into blocks of the cipher's block size, and encrypts them in a sequential fashion, where each block depends on the previous one (and the first block depends on the initialization vector). If the input message's length is not a multiple of the cipher's block size, a padding mechanism is enabled by default which will pad the message to the correct length (and remove the extra data upon decryption). If padding is explicitly disabled through the mode of operation's parameters, the input's length must be a multiple of the cipher's block size.

If padding is enabled, `cbc_final()` requires a valid pointer to be passed in the `outlen` parameter and will always return a full blocksize of data, containing the last few ciphertext bytes containing the padding information.

If padding is disabled, `outlen` is also required, and will return the number of unprocessed plaintext bytes in the context. If this is any value other than zero, the function will also fail with `ORDO_LEFTOVER`.

See Also

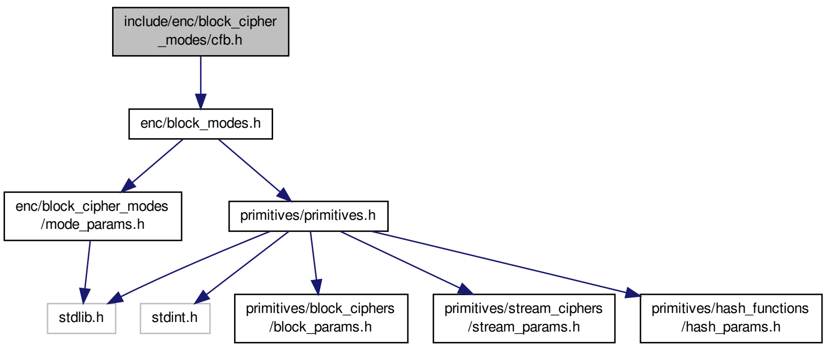
cbc.c

6.8 include/enc/block_cipher_modes/cfb.h File Reference

CFB block cipher mode of operation.

`#include <enc/block_modes.h>`

Include dependency graph for cfb.h:


include/enc/block_cipher_modes/cfb.h
enc/block_modes.h
enc/block_cipher_modes/mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers/block_params.h
primitives/stream_ciphers/stream_params.h
primitives/hash_functions/hash_params.h

6.8.1 Detailed Description

CFB block cipher mode of operation. The CFB mode generates a keystream by repeatedly encrypting an initialization vector and mixing in the plaintext, effectively turning a block cipher into a stream cipher. As such, CFB mode requires no padding, and the ciphertext size will always be equal to the plaintext size.

Note that the CFB keystream depends on the plaintext fed into it, as opposed to OFB mode. This also means the block cipher's inverse permutation is never used.

`cfb_final()` accepts 0 as an argument for `outlen`, since by design the CFB mode of operation does not produce any final data. However, if a valid pointer is passed, its value will be set to zero as expected.

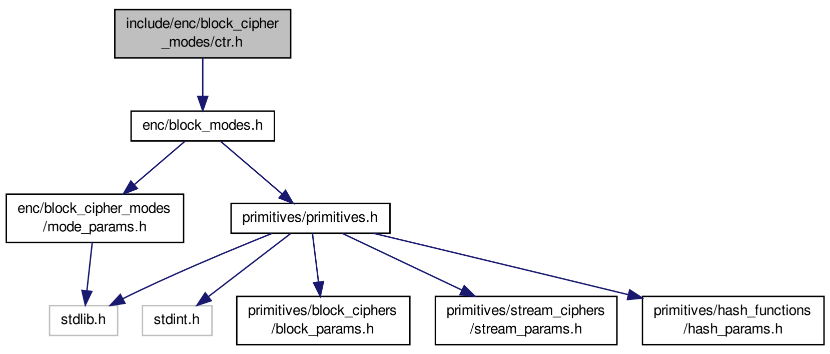
See Also

cfb.c

6.9 include/enc/block_cipher_modes/ctr.h File Reference

CTR block cipher mode of operation.

`#include <enc/block_modes.h>`
Include dependency graph for ctr.h:


include/enc/block_cipher _modes/ctr.h
enc/block_modes.h
enc/block_cipher_modes /mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

6.9.1 Detailed Description

CTR block cipher mode of operation. The CTR mode generates a keystream by repeatedly encrypting a counter starting from some initialization vector, effectively turning a block cipher into a stream cipher. As such, CTR mode requires no padding, and outlen will always be equal to inlen.

Note that the CTR keystream is independent of the plaintext, and is also spatially coherent (using a given initialization vector on a len-byte message will "use up" len bytes of the keystream) so care must be taken to avoid reusing the initialization vector in an insecure way. This also means the block cipher's inverse permutation is never used.

`ctr_final()` accepts 0 as an argument for `outlen`, since by design the CTR mode of operation does not produce any final data. However, if a valid pointer is passed, its value will be set to zero as expected.

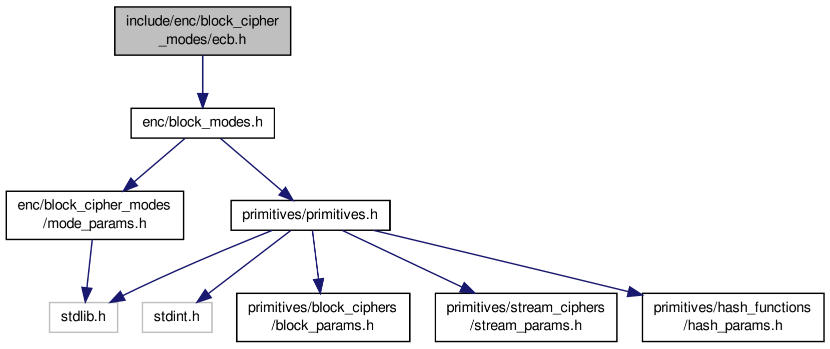
See Also

ctr.c

6.10 include/enc/block_cipher_modes/ecb.h File Reference

ECB block cipher mode of operation.

`#include <enc/block_modes.h>`
Include dependency graph for ecb.h:


include/enc/block_cipher_modes/ecb.h
enc/block_modes.h
enc/block_cipher_modes/mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers/block_params.h
primitives/stream_ciphers/stream_params.h
primitives/hash_functions/hash_params.h

6.10.1 Detailed Description

ECB block cipher mode of operation. The ECB mode divides the input message into blocks of the cipher's block size, and encrypts them individually. If the input message's length is not a multiple of the cipher's block size, a padding mechanism is enabled by default which will pad the message to the correct length (and remove the extra data upon decryption). If padding is explicitly disabled through the mode of operation's parameters, the input's length must be a multiple of the cipher's block size.

If padding is enabled, `ECB_Final()` requires a valid pointer to be passed in the `outlen` parameter and will always return a full blocksize of data, containing the last few ciphertext bytes containing the padding information.

If padding is disabled, `outlen` is also required, and will return the number of unprocessed plaintext bytes in the context. If this is any value other than zero, the function will also fail with `ORDO_LEFTOVER`.

The ECB mode does not require an initialization vector.

Note that the ECB mode is insecure in almost all situations and is not recommended for use.

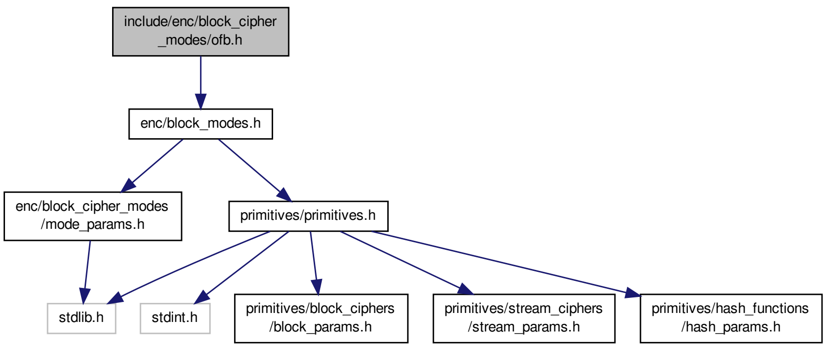
See Also

ecb.c

6.11 include/enc/block_cipher_modes/ofb.h File Reference

OFB block cipher mode of operation.

`#include <enc/block_modes.h>`
Include dependency graph for ofb.h:


include/enc/block_cipher _modes/ofb.h
enc/block_modes.h
enc/block_cipher_modes /mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

6.11.1 Detailed Description

OFB block cipher mode of operation. The OFB mode generates a keystream by repeatedly encrypting an initialization vector, effectively turning a block cipher into a stream cipher. As such, OFB mode requires no padding, and outlen will always be equal to inlen.

Note that the OFB keystream is independent of the plaintext, so a key/iv pair must never be used for more than one message. This also means the block cipher's inverse permutation is never used.

`ofb_final()` accepts 0 as an argument for `outlen`, since by design the OFB mode of operation does not produce any final data. However, if a valid pointer is passed, its value will be set to zero as expected.

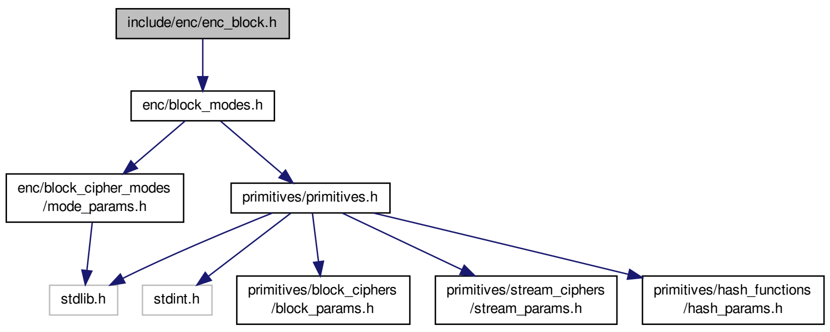
See Also

ofb.c

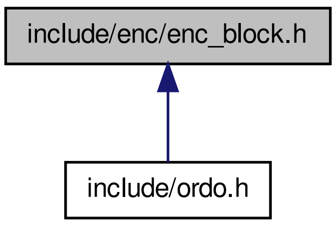
6.12 include/enc/enc_block.h File Reference

Block cipher symmetric encryption.

`#include <enc/block_modes.h>`
Include dependency graph for enc_block.h:


include/enc/enc_block.h
enc/block_modes.h
enc/block_cipher_modes /mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

This graph shows which files directly or indirectly include this file:



Functions

- struct ENC_BLOCK_CTX ∗ enc_block_alloc (struct BLOCK_CIPHER ∗cipher, struct BLOCK_MODE ∗mode)
- int enc_block_init (struct ENC_BLOCK_CTX ∗ctx, void ∗key, size_t keySize, void ∗iv, int dir, void ∗cipherParams, void ∗modeParams)
- void enc_block_update (struct ENC_BLOCK_CTX ∗ctx, void ∗in, size_t inlen, void ∗out, size_t ∗outlen)
- int enc_block_final (struct ENC_BLOCK_CTX ∗ctx, void ∗out, size_t ∗outlen)
- void enc_block_free (struct ENC_BLOCK_CTX ∗ctx)

6.12.1 Detailed Description

Block cipher symmetric encryption. Interface to encrypt plaintext and decrypt ciphertext with different block ciphers and modes of operation. Note it is always possible to skip this API and directly use the lower-level functions available in the individual mode of operation headers, but this interface abstracts away some of the more boilerplate details and so should be preferred.

If you wish to use the lower level API, you will need to manage your block cipher contexts yourself, which can give more flexibility in some particular cases but is often unnecessary.

The padding algorithm for modes of operation which use padding is PKCS7 (RFC 5652), which appends N bytes of value N, where N is the number of padding bytes required, in bytes (between 1 and the block cipher's block size).

See Also

enc_block.c

6.12.2 Function Documentation

6.12.2.1 struct ENC_BLOCK_CTX∗ enc_block_alloc (struct BLOCK_CIPHER ∗ *cipher,* struct BLOCK_MODE ∗ *mode*)

This function returns an allocated block cipher encryption context using a specific block cipher and mode of operation.

Parameters

cipher	The block cipher object to be used.
mode	The mode of operation object to be used.

Returns

Returns the allocated block cipher encryption context, or 0 if an error occurred.

6.12.2.2 int enc_block_init (struct ENC_BLOCK_CTX ∗ *ctx,* void ∗ *key,* size_t *keySize,* void ∗ *iv,* int *dir,* void ∗ *cipherParams,* void ∗ *modeParams*)

This function initializes a block cipher encryption context for encryption, provided a key, initialization vector, and cipher/mode-specific parameters.

Parameters

ctx	An allocated block cipher encryption context.
key	A buffer containing the key to use for encryption.
keySize	The size, in bytes, of the encryption key.
iv	This points to the initialization vector.
cipherParams	This points to specific cipher parameters, set to zero for default behavior.
modeParams	This points to specific mode of operation parameters, set to zero for default behavior.
dir	This represents the dir of encryption, set to 1 for encryption and 0 for decryption.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

Remarks

The initialization vector may be zero, if the mode of operation does not require one.

6.12.2.3 void enc_block_update (struct ENC_BLOCK_CTX ∗ *ctx*, void ∗ *in*, size_t *inlen*, void ∗ *out*, size_t ∗ *outlen*)

This function encrypts or decrypts a buffer of a given length using the provided block cipher encryption context.

Parameters

ctx	The block cipher encryption context to use. This context must have been allocated and initialized.
in	This points to a buffer containing plaintext (or ciphertext).
inlen	This contains the size of the `in` buffer, in bytes.
out	This points to a buffer which will contain the plaintext (or ciphertext).
outlen	This points to a variable which will contain the number of bytes written to `out`.

Remarks

See `blockEncryptModeUpdate()` for remarks about output buffer size.

6.12.2.4 int enc_block_final (struct ENC_BLOCK_CTX ∗ *ctx,* void ∗ *out,* size_t ∗ *outlen*)

This function finalizes a block cipher encryption context, and will process and return any leftover plaintext or ciphertext.

Parameters

ctx	The block cipher encryption context to use. This context must have been allocated and initialized.
out	This points to a buffer which will contain any remaining plaintext (or ciphertext).
outlen	This points to a variable which will contain the number of bytes written to `out`.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

Remarks

Once this function returns, the passed context can no longer be used for encryption or decryption.
See `blockEncryptModeFinal()` for remarks.

6.12.2.5 void enc_block_free (struct ENC_BLOCK_CTX ∗ *ctx*)

This function frees (deallocates) an initialized block cipher encryption context.

Parameters

ctx	The block cipher encryption context to be freed. This context needs to at least have been allocated.

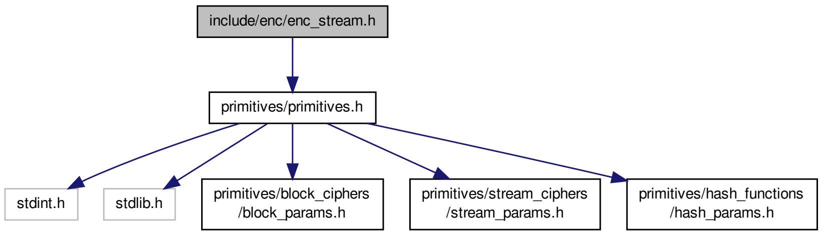
Remarks

Once this function returns, the passed context may no longer be used anywhere and sensitive information will be wiped. Do not call this function if `enc_block_alloc()` failed, as the latter correctly frees dangling context buffers in case of error.

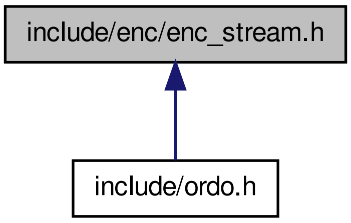
6.13 include/enc/enc_stream.h File Reference

Stream cipher symmetric encryption.

`#include <primitives/primitives.h>`
Include dependency graph for enc_stream.h:


include/enc/enc_stream.h
primitives/primitives.h
stdint.h
stdlib.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

This graph shows which files directly or indirectly include this file:



Functions

- struct ENC_STREAM_CTX ∗ enc_stream_alloc (struct STREAM_CIPHER ∗cipher)
- int enc_stream_init (struct ENC_STREAM_CTX ∗ctx, void ∗key, size_t keySize, void ∗cipherParams)
- void enc_stream_update (struct ENC_STREAM_CTX ∗ctx, void ∗inout, size_t len)
- void enc_stream_free (struct ENC_STREAM_CTX ∗ctx)

6.13.1 Detailed Description

Stream cipher symmetric encryption. Interface to encrypt plaintext and decrypt ciphertext with various stream ciphers.

See Also

enc_stream.c

6.13.2 Function Documentation

6.13.2.1 struct ENC_STREAM_CTX∗ enc_stream_alloc (struct STREAM_CIPHER ∗ *cipher*)

This function returns an allocated stream cipher encryption context using a specific stream cipher.

Parameters

cipher	The stream cipher object to be used.

Returns

Returns the allocated stream cipher encryption context, or 0 if an error occurred.

6.13.2.2 int enc_stream_init (struct ENC_STREAM_CTX ∗ *ctx,* void ∗ *key,* size_t *keySize,* void ∗ *cipherParams*)

This function initializes a stream cipher encryption context for encryption, provided a key and cipher parameters.

Parameters

ctx	An allocated stream cipher encryption context.
key	A buffer containing the key to use for encryption.
keySize	The size, in bytes, of the encryption key.
cipherParams	This points to specific stream cipher parameters, set to zero for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

6.13.2.3 void enc_stream_update (struct ENC_STREAM_CTX ∗ *ctx,* void ∗ *inout,* size_t *len*)

This function encrypts or decrypts a buffer of a given length using the provided stream cipher encryption context.

Parameters

ctx	The block cipher encryption context to use. This context must have been allocated and initialized.
inout	The plaintext or ciphertext buffer.
len	Number of bytes to read from the `inout` buffer.

Remarks

See `ordoEncryptStream()` for remarks about output buffer size.

6.13.2.4 void enc_stream_free (struct ENC_STREAM_CTX ∗ *ctx*)

This function frees (deallocates) an initialized stream cipher encryption context.

Parameters

ctx	The stream cipher encryption context to be freed. This context needs to at least have been allocated.

Remarks

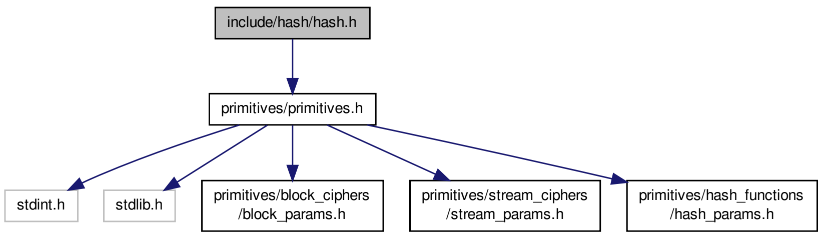
Once this function returns, the passed context may no longer be used anywhere and sensitive information will be wiped. Do not call this function if `enc_stream_alloc()` failed.

6.14 include/hash/hash.h File Reference

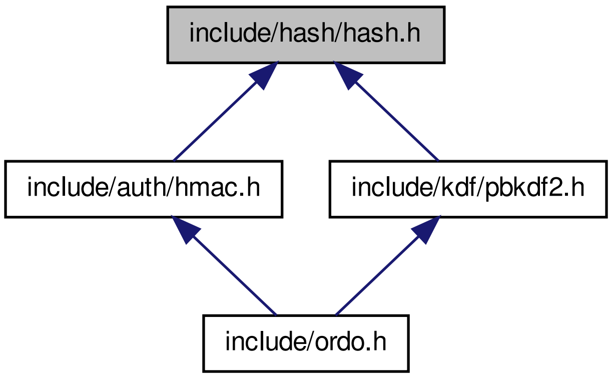
Hash function interface.

```
#include <primitives/primitives.h>
```

Include dependency graph for hash.h:


include/hash/hash.h
primitives/primitives.h
stdint.h
stdlib.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

This graph shows which files directly or indirectly include this file:



Functions

- struct HASH_CTX ∗ hash_alloc (struct HASH_FUNCTION ∗hash)
- int hash_init (struct HASH_CTX ∗ctx, void ∗hashParams)
- void hash_update (struct HASH_CTX ∗ctx, void ∗buffer, size_t size)
- void hash_final (struct HASH_CTX ∗ctx, void ∗digest)
- void hash_free (struct HASH_CTX ∗ctx)
- void hash_copy (struct HASH_CTX ∗dst, struct HASH_CTX ∗src)

6.14.1 Detailed Description

Hash function interface. Interface to compute cryptographic digests, using hash functions. This is a very thin generic wrapper around the low-level functions.

6.14.2 Function Documentation

6.14.2.1 struct HASH_CTX∗ hash_alloc (struct HASH_FUNCTION ∗ *hash*)

Returns an allocated hash function context using a given hash function.

Parameters

hash	The hash function to use.

Returns

Returns the allocated hash function context, or nil if an allocation error occurred.

6.14.2.2 int hash_init (struct HASH_CTX ∗ *ctx*, void ∗ *hashParams*)

Initializes a hash function context, provided optional parameters.

Parameters

ctx	An allocated hash function context.
hashParams	A pointer to specific hash function parameters, set to nil for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

Remarks

It is always valid to pass nil for `hashParams` if you do not wish to use more advanced features offered by a specific hash function.

6.14.2.3 void hash_update (struct HASH_CTX ∗ *ctx*, void ∗ *buffer*, size_t *size*)

Feeds data into a hash function context, updating the final digest.

Parameters

ctx	An allocated hash function context.
buffer	A buffer containing the data to hash.
size	The size, in bytes, of the data to read from `buffer`.

Remarks

This function has the property that Update(A) followed by Update(B) is equivalent to Update(A || B) where || denotes concatenation.

6.14.2.4 void hash_final (struct HASH_CTX ∗ *ctx*, void ∗ *digest*)

Finalizes a hash function context, returning the final digest.

Parameters

ctx	An allocated hash function context.
digest	A buffer into which the digest will be written.

Remarks

The `digest` buffer should be long enough to accomodate the digest. You can query the hash function's digest size in bytes via the `hashDigestSize()` macro.

6.14.2.5 void hash_free (struct HASH_CTX ∗ *ctx*)

Deallocates an initialized hash function context.

Parameters

ctx	The hash function context to be freed.

Remarks

The context need not have been initialized.
Passing nil to this function is a no-op.
Once this function returns, the passed context may no longer be used anywhere, and any sensitive information will be wiped.

6.14.2.6 void hash_copy (struct HASH_CTX ∗ *dst,* struct HASH_CTX ∗ *src*)

Performs a deep copy of one context into another.

Parameters

dst	The destination context.
src	The source context.

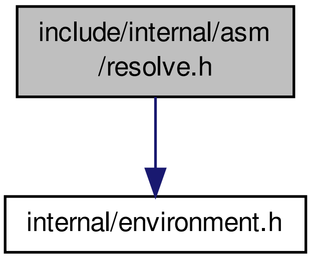
Remarks

Both contexts must have been allocated using the same hash function, else the function's behavior is undefined.

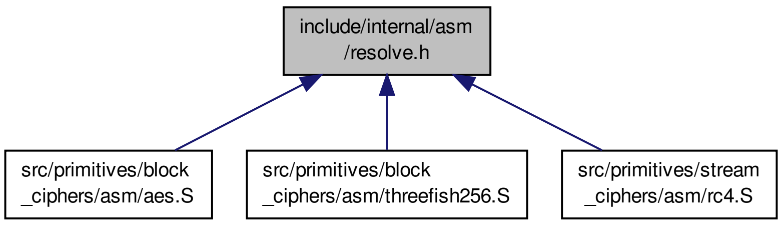
6.15 include/internal/asm/resolve.h File Reference

Assembly code path resolution.

```
#include <internal/environment.h>
```
Include dependency graph for resolve.h:



This graph shows which files directly or indirectly include this file:



6.15.1 Detailed Description

Assembly code path resolution. This header is designed to help library code switch between different code paths, e.g. x86_64 code versus standard C, and so on, using environment.h.

The following template should be followed, for consistency. Take RC4 as an example. It has two code paths: one for x86_64 processors, and once for all other hardware. But that assembly code path is also divided between a Linux and a Windows version, to account for ABI differences. So this header should declare one and **only** one of the following preprocessor tokens:

- `RC4_X86_64_LINUX`: use the x86_64 code path for Linux.
- `RC4_X86_64_WINDOWS`: use the x86_64 code path for Windows.
- `RC4_STANDARD`: use the standard C code path.

Also, if `ORDO_DEBUG` is defined (i.e. Ordo is being compiled in debug mode), the standard C code path **must** unconditionally be selected.

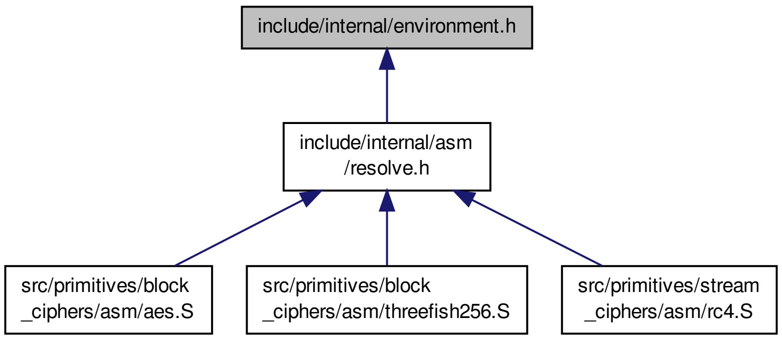
The relevant code (rc4.c and rc4.S) can then include/exclude accordingly, simplifying maintenance costs and improving overall readability.

This header is meant for internal use only.

6.16 include/internal/environment.h File Reference

Compile-time environment detection.

This graph shows which files directly or indirectly include this file:



6.16.1 Detailed Description

Compile-time environment detection. This header will provide definitions for the environment details under which Ordo is being built, trying to unify various compiler-specific details under a single interface.

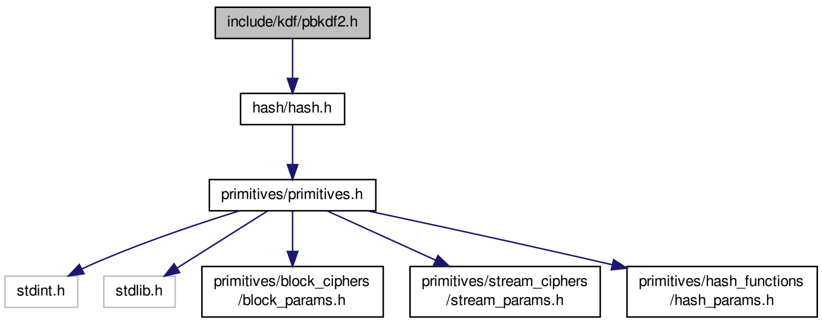
This file may only contain preprocessor macros as it is included in assembly files - it cannot contain declarations.

6.17 include/kdf/pbkdf2.h File Reference

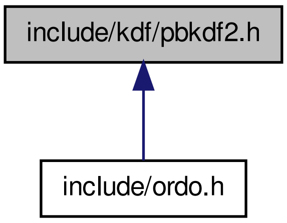
PBKDF2 module.

```
#include <hash/hash.h>
```

Include dependency graph for pbkdf2.h:



This graph shows which files directly or indirectly include this file:



Functions

- int pbkdf2 (struct HASH_FUNCTION ∗hash, void ∗password, size_t password_len, void ∗salt, size_t salt_len, void ∗output, size_t output_len, size_t iterations, void ∗hashParams)

6.17.1 Detailed Description

PBKDF2 module. Module for the PBKDF2 algorithm (Password-Based Key Derivation Function v2) which combines a keyed PRF (here HMAC) with a salt in order to generate secure cryptographic keys, as per RFC 2898. Also features a variable iteration count (work factor) to help thwart brute-force attacks.

Unlike most other cryptographic modules, the PBKDF2 API does not follow the traditional init/update/final pattern but is a context-free function as its inputs are almost always known in advance. As such this module does not benefit from the use of contexts.

6.17.2 Function Documentation

6.17.2.1 int pbkdf2 (struct HASH_FUNCTION ∗ *hash,* void ∗ *password,* size_t *password_len,* void ∗ *salt,* size_t *salt_len,* void ∗ *output,* size_t *output_len,* size_t *iterations,* void ∗ *hashParams*)

Derives a key using PBKDF2.

Parameters

hash	The hash function to use (the PRF used will be generic HMAC instantiated with this hash function)
password	A pointer to the password to use.
password_len	The length in bytes of the `password` buffer.
salt	A pointer to the salt to use.
salt_len	The length in bytes of the `salt` buffer.
output	A pointer to a buffer in which to write the derived key.
output_len	The desired length, in bytes, of the derived key. The `output` buffer should be at least `output_len` bytes long.

iterations	The number of PBKDF2 iterations to use.
hash_params	A pointer to hash-specific parameters, or nil if not used.

Returns

Returns `ORDO_SUCCESS` on success, and a negative value on error.

Remarks

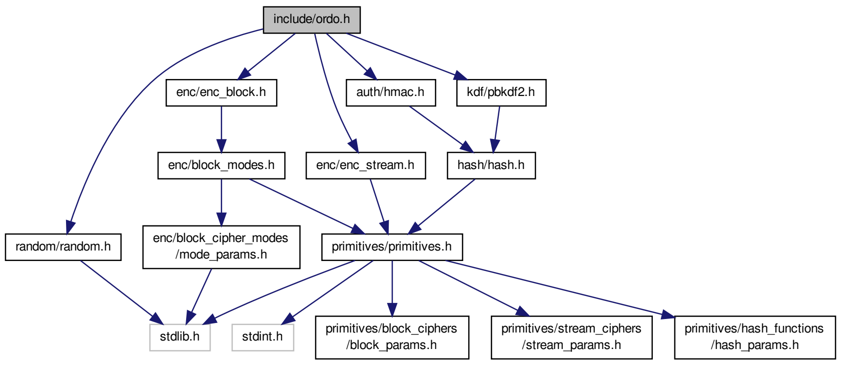
There is a maximum output length of 2^32 - 1 multiplied by the digest length of the chosen hash function, but it is unlikely to be reached as derived keys are generally no longer than a few hundred bits. Reaching the limit will result in an `ORDO_ARG` error code. This limit is mandated by the PBKDF2 specification.
Do not use hash parameters which modify the hash function's output length, or this function's behavior is undefined.

6.18 include/ordo.h File Reference

Ordo high-level API.

```
#include <enc/enc_stream.h>
#include <enc/enc_block.h>
#include <random/random.h>
#include <kdf/pbkdf2.h>
#include <auth/hmac.h>
```

Include dependency graph for ordo.h:



include/ordo.h
enc/enc_block.h
auth/hmac.h
kdf/pbkdf2.h
enc/block_modes.h
enc/enc_stream.h
hash/hash.h
random/random.h
enc/block_cipher_modes/mode_params.h
primitives/primitives.h
stdlib.h
stdint.h
primitives/block_ciphers/block_params.h
primitives/stream_ciphers/stream_params.h
primitives/hash_functions/hash_params.h

Functions

- void load_ordo ()
- int ordoEncrypt (void ∗in, size_t inlen, void ∗out, size_t ∗outlen, struct BLOCK_CIPHER ∗cipher, struct BLOCK_MODE ∗mode, void ∗key, size_t keySize, void ∗iv, void ∗cipherParams, void ∗modeParams)
- int ordoDecrypt (void ∗in, size_t inlen, void ∗out, size_t ∗outlen, struct BLOCK_CIPHER ∗cipher, struct BLOCK_MODE ∗mode, void ∗key, size_t keySize, void ∗iv, void ∗cipherParams, void ∗modeParams)
- int ordoEncryptStream (void ∗inout, size_t len, struct STREAM_CIPHER ∗cipher, void ∗key, size_t keySize, void ∗cipherParams)
- int ordoHash (void ∗in, size_t len, void ∗out, struct HASH_FUNCTION ∗hash, void ∗hashParams)
- int ordoHMAC (void ∗in, size_t len, void ∗key, size_t keySize, void ∗out, struct HASH_FUNCTION ∗hash, void ∗hashParams)

6.18.1 Detailed Description

Ordo high-level API. This is the highest-level API for Ordo, which forgoes the use of cryptographic contexts completely,resulting in more concise code at the cost of reduced flexibility.

See Also

ordo.c

6.18.2 Function Documentation

6.18.2.1 void load_ordo ( )

Loads Ordo - this calls all the load functions in the different interfaces (primitives, encrypt, etc...). After this function returns, all objects such as `RC4()`, `CBC()`, may be used.

6.18.2.2 int ordoEncrypt (void ∗ *in*, size_t *inlen*, void ∗ *out*, size_t ∗ *outlen*, struct BLOCK_CIPHER ∗ *cipher*, struct BLOCK_MODE ∗ *mode*, void ∗ *key*, size_t *keySize*, void ∗ *iv*, void ∗ *cipherParams*, void ∗ *modeParams*)

This function encrypts a buffer of a given length using a block cipher in a given mode of operation with the passed parameters.

Parameters

in	The plaintext buffer.
inlen	Number of bytes to read from the `in` buffer.
out	The ciphertext buffer, to which the ciphertext should be written.
outlen	This points to a variable which will contain the number of bytes written to `out`.
cipher	A block cipher object, describing the block cipher to use for encryption.
mode	The block cipher mode of operation to be used for encryption.
key	A buffer containing the key material to use for encryption.
keySize	The length, in bytes, of the `key` buffer.
iv	A buffer containing the initialization vector (this may be 0 if the mode of operation does not use an IV).
cipherParams	This points to specific block cipher parameters, set to zero for default behavior.
modeParams	This points to specific mode of operation parameters, set to zero for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, a negative error code on failure.

Remarks

One downside of this function is that it is not possible to encrypt data in chunks - the whole plaintext must be available before encryption can begin. If your requirements make this unacceptable, you should use the encryption interface, located one level of abstraction lower - see enc_block.h.
The out buffer should have enough space to contain the entire ciphertext, which may be larger than the plaintext if a mode which uses padding (with padding enabled) is used. See remarks about padding in enc_block.h.

6.18.2.3 int ordoDecrypt (void ∗ *in*, size_t *inlen*, void ∗ *out*, size_t ∗ *outlen*, struct BLOCK_CIPHER ∗ *cipher*, struct BLOCK_MODE ∗ *mode*, void ∗ *key*, size_t *keySize*, void ∗ *iv*, void ∗ *cipherParams*, void ∗ *modeParams*)

This function decrypts a buffer of a given length using a block cipher in a given mode of operation with the passed parameters.

Parameters

in	The ciphertext buffer.
inlen	Number of bytes to read from the `in` buffer.
out	The plaintext buffer, to which the plaintext should be written.
outlen	This points to a variable which will contain the number of bytes written to `out`.
cipher	A block cipher object, describing the block cipher to use for decryption.
mode	The block cipher mode of operation to be used for decryption.
key	A buffer containing the key material to use for decryption.
keySize	The length, in bytes, of the `key` buffer.
iv	A buffer containing the initialization vector (this may be 0 if the mode of operation does not use an IV).
cipherParams	This points to specific block cipher parameters, set to zero for default behavior.
modeParams	This points to specific mode of operation parameters, set to zero for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, a negative error code on failure.

Remarks

See ordoEncrypt for additional remarks.

6.18.2.4 int ordoEncryptStream (void ∗ *inout,* size_t *len,* struct STREAM_CIPHER ∗ *cipher,* void ∗ *key,* size_t *keySize,* void ∗ *cipherParams*)

This function encrypts or decrypts a buffer of a given length using a stream cipher.

Parameters

inout	The plaintext or ciphertext buffer.
len	Number of bytes to read from the `inout` buffer.
cipher	A stream cipher object, describing the stream cipher to use for encryption.
key	A buffer containing the key material to use for encryption.
keySize	The length, in bytes, of the `key` buffer.
cipherParams	This points to specific block cipher parameters, set to zero for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, a negative error code on failure.

Remarks

Stream ciphers are different from block ciphers in multiple ways:

- they do not require an IV because there is no standard way to add initialization vectors to a stream cipher.
- no mode of operation is required as stream ciphers work by generating a keystream and combining it with the plaintext or ciphertext (so they are a "mode" in themselves).
- there is no difference between encryption and decryption: encrypting the ciphertext again will produce the plaintext and vice versa.
- the encryption or decryption is done in-place directly in the `inout` buffer, since the ciphertext is always the same length as the plaintext. If you need two different buffers, make a copy of the plaintext before encrypting.

6.18.2.5 int ordoHash (void ∗ *in,* size_t *len,* void ∗ *out,* struct HASH_FUNCTION ∗ *hash,* void ∗ *hashParams*)

This function hashes a buffer of a given length into a digest using a hash function.

Parameters

in	The input buffer to hash.
len	Number of bytes to read from the `in` buffer.
out	The buffer in which to put the digest.
hash	A hash function object, describing the hash function to use.
hashParams	This points to specific hash function parameters, set to zero for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, a negative error code on failure.

6.18.2.6 int ordoHMAC (void ∗ *in,* size_t *len,* void ∗ *key,* size_t *keySize,* void ∗ *out,* struct HASH_FUNCTION ∗ *hash,* void ∗ *hashParams*)

This function returns the HMAC of a buffer using a key with any hash function.

Parameters

in	The input buffer to hash.
len	Number of bytes to read from the `in` buffer.
key	The key to use.
keySize	The length of the key to use, in bytes.
out	The buffer in which to put the digest.
hash	A hash function object, describing the hash function to use.
hashParams	This points to specific hash function parameters, set to zero for default behavior.

Returns

Returns `ORDO_SUCCESS` on success, a negative error code on failure.

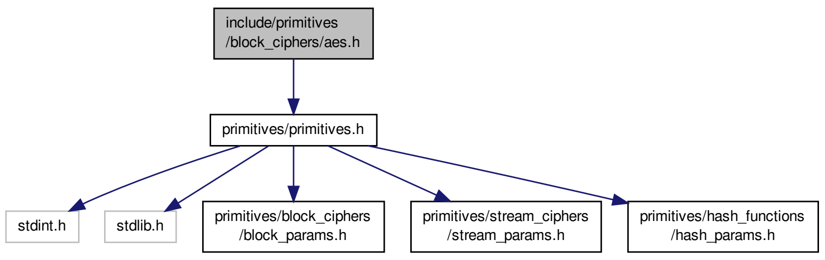
Remarks

Note the hash parameters only affect the inner hash (the one hashing the buffer), not the outer one or the potential key-processing one.

6.19 include/primitives/block_ciphers/aes.h File Reference

AES block cipher.

`#include <primitives/primitives.h>`
Include dependency graph for aes.h:



Functions

- struct AES_STATE ∗ aes_alloc ()
- int aes_init (struct AES_STATE ∗state, void ∗key, size_t keySize, struct AES_PARAMS ∗params)
- void aes_forward (struct AES_STATE ∗state, uint8_t ∗block)
- void aes_inverse (struct AES_STATE ∗state, uint8_t ∗block)
- void aes_free (struct AES_STATE ∗state)
- void aes_set_primitive (struct BLOCK_CIPHER ∗cipher)

6.19.1 Detailed Description

AES block cipher. AES (Advanced Encryption Standard) is a block cipher. It has a 128-bit block size and three possible key sizes, namely 128, 192 and 256 bits. It is based on the Rijndael cipher and was selected as the official encryption standard on November 2001 (FIPS 197).

6.19.2 Function Documentation

6.19.2.1 struct AES_STATE∗ aes_alloc ( )

Allocates and returns an uninitialized AES block cipher context.

Returns

The allocated context, or nil on allocation failure.

6.19.2.2 int aes_init (struct AES_STATE ∗ *state*, void ∗ *key*, size_t *keySize*, struct AES_PARAMS ∗ *params*)

Initializes an AES block cipher context.

Parameters

ctx	An allocated AES context.
key	A pointer to a buffer containing the encryption key.
keySize	The key size, in bytes, to be read from `key`.
params	A pointer to an AES parameter structure.

Returns

Returns `ORDO_SUCCESS` on success, `ORDO_KEY_SIZE` if the key size passed was invalid, `ORDO_ARG` if the round number provided in the parameters is invalid, or `ORDO_ALLOC` if an allocation error occurs.

Remarks

The `params` parameter may be nil if no parameters are required.

6.19.2.3 void aes_forward (struct AES_STATE ∗ *state*, uint8_t ∗ *block*)

Encrypts a 128-bit block (as an array of bytes).

Parameters

ctx	An initialized AES context.
block	A pointer to the block to encrypt.

Remarks

This function is deterministic, as are all of the block cipher `Forward` and `Inverse` functions, and will not modify the state of the provided context.

6.19.2.4 void aes_inverse (struct AES_STATE ∗ *state*, uint8_t ∗ *block*)

Decrypts a 128-bit block (as an array of bytes).

Parameters

ctx	An initialized AES context.
block	A pointer to the block to decrypt.

Remarks

See remarks for `aes_forward()`.

6.19.2.5 void aes_free (struct AES_STATE ∗ *state*)

Frees the memory associated with an AES cipher context and securely erases sensitive context information such as key material.

Parameters

ctx	An allocated AES context.

Remarks

The context need not have been initialized.
Passing nil to this function is a no-op.

6.19.2.6 void aes_set_primitive (struct BLOCK_CIPHER ∗ *cipher*)

This function populates a block cipher object with the AES functions and attributes, and is meant for internal use.

Parameters

cipher	A pointer to a block cipher object to populate.

Remarks

Once populated, the `BLOCK_CIPHER` struct can be freely used in the higher level `enc_block` interface.
If you have issued a call to `load_primitives()`, this function has already been called and you may use the `AES()` function to access the underlying AES block cipher object.

See Also

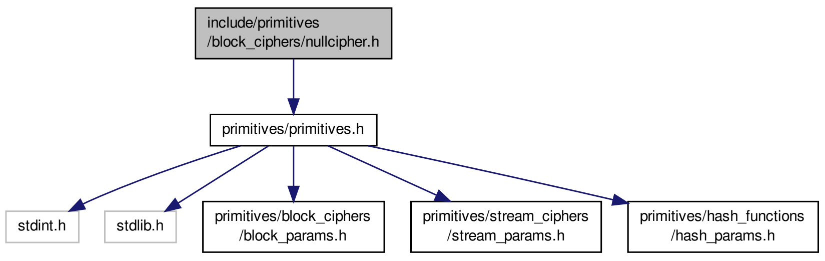
enc_block.h

6.20 include/primitives/block_ciphers/nullcipher.h File Reference

Null Cipher block cipher.

```
#include <primitives/primitives.h>
```

Include dependency graph for nullcipher.h:


include/primitives /block_ciphers/nullcipher.h
primitives/primitives.h
stdint.h
stdlib.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

Functions

- struct NULLCIPHER_STATE ∗ nullcipher_alloc ()
- int nullcipher_init (struct NULLCIPHER_STATE ∗state, void ∗key, size_t keySize, void ∗params)
- void nullcipher_forward (struct NULLCIPHER_STATE ∗state, void ∗block)
- void nullcipher_inverse (struct NULLCIPHER_STATE ∗state, void ∗block)
- void nullcipher_free (struct NULLCIPHER_STATE ∗state)
- void nullcipher_set_primitive (struct BLOCK_CIPHER ∗cipher)

6.20.1 Detailed Description

Null Cipher block cipher. This cipher is only used to debug the library and does absolutely nothing, in other words, it is the identity permutation. It accepts any key size but does not even attempt to read the key, and has no parameters. Its block size is 128 bits and is arbitrarily chosen.

6.20.2 Function Documentation

6.20.2.1 struct NULLCIPHER_STATE∗ nullcipher_alloc ( )

Allocates and returns an uninitialized AES block cipher context.

Returns

The allocated context, or nil on allocation failure.

6.20.2.2 int nullcipher_init (struct NULLCIPHER_STATE ∗ *state,* void ∗ *key,* size_t *keySize,* void ∗ *params*)

Initializes a NullCipher block cipher context.

Parameters

ctx	An allocated NullCipher context.
key	A pointer to a buffer containing the encryption key.
keySize	The key size, in bytes, to be read from `key`.
params	Ignored.

Returns

Cannot fail and returns `ORDO_SUCCESS`.

Remarks

This function does nothing.

6.20.2.3 void nullcipher_forward (struct NULLCIPHER_STATE ∗ *state*, void ∗ *block*)

Encrypts a 128-bit block.

Parameters

ctx	An initialized NullCipher context.
block	A pointer to the block to encrypt.

Remarks

This function does nothing.

6.20.2.4 void nullcipher_inverse (struct NULLCIPHER_STATE ∗ *state*, void ∗ *block*)

Decrypts a 128-bit block.

Parameters

ctx	An initialized NullCipher context.
block	A pointer to the block to decrypt.

Remarks

This function does nothing.

6.20.2.5 void nullcipher_free (struct NULLCIPHER_STATE ∗ *state*)

Frees the memory associated with a NullCipher cipher context.

Parameters

ctx	An allocated NullCipher context.

Remarks

Passing nil to this function is a no-op.

6.20.2.6 void nullcipher_set_primitive (struct BLOCK_CIPHER ∗ *cipher*)

This function populates a block cipher object with the NullCipher functions and attributes, and is meant for internal use.

Parameters

cipher	A pointer to a block cipher object to populate.

Remarks

Once populated, the `BLOCK_CIPHER` struct can be freely used in the higher level `enc_block` interface. If you have issued a call to `load_primitives()`, this function has already been called and you may use the `NullCipher()` function to access the underlying NullCipher block cipher object.

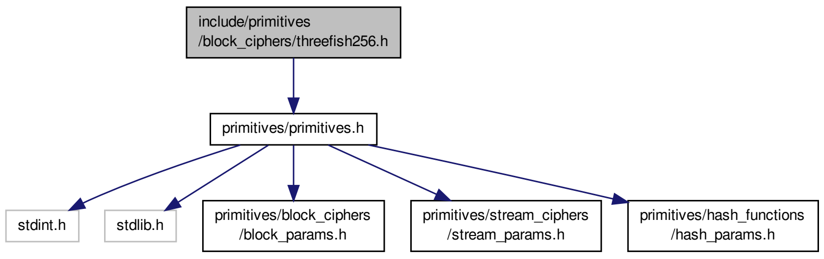
See Also

enc_block.h

6.21 include/primitives/block_ciphers/threefish256.h File Reference

Threefish-256 block cipher.

`#include <primitives/primitives.h>`
Include dependency graph for threefish256.h:



Functions

- struct THREEFISH256_STATE ∗ threefish256_alloc ()
- int threefish256_init (struct THREEFISH256_STATE ∗state, uint64_t ∗key, size_t keySize, struct THREEFISH256_PARAMS ∗params)
- void threefish256_forward (struct THREEFISH256_STATE ∗state, uint64_t ∗block)
- void threefish256_inverse (struct THREEFISH256_STATE ∗state, uint64_t ∗block)
- void threefish256_free (struct THREEFISH256_STATE ∗state)
- void threefish256_set_primitive (struct BLOCK_CIPHER ∗cipher)
- void threefish256_key_schedule (uint64_t key[4], uint64_t tweak[2], uint64_t subkeys[19][4])
- void threefish256_forward_raw (uint64_t block[4], uint64_t subkeys[19][4])
- void threefish256_inverse_raw (uint64_t block[4], uint64_t subkeys[19][4])

6.21.1 Detailed Description

Threefish-256 block cipher. Threefish-256 is a block cipher with a 256-bit block size and a 256-bit key size. It also has an optional 128-bit tweak, which can be set through the cipher parameters.

The Threefish ciphers were originally designed to be used as a building block for the Skein hash function family.

6.21.2 Function Documentation

6.21.2.1 struct THREEFISH256_STATE* threefish256_alloc ( )

Allocates and returns an uninitialized Threefish-256 block cipher context.

Returns

The allocated context, or nil on allocation failure.

6.21.2.2 int threefish256_init (struct THREEFISH256_STATE * *state,* uint64_t * *key,* size_t *keySize,* struct THREEFISH256_PARAMS * *params*)

Initializes a Threefish-256 block cipher context.

Parameters

ctx	An allocated Threefish-256 context.
key	A pointer to a 256-bit key, as a `uint64_t`[4] structure.
keySize	The key size, in bytes. Must be 32 (256 bits).
params	A pointer to a Threefish-256 parameter structure.

Returns

Returns `ORDO_SUCCESS` on success, or `ORDO_KEY_SIZE` if the key size passed was invalid.

Remarks

The `params` parameter may be nil if no parameters are required.

6.21.2.3 void threefish256_forward (struct THREEFISH256_STATE * *state,* uint64_t * *block*)

Encrypts a 256-bit block (as a `uint64_t`[4] structure).

Parameters

ctx	An initialized Threefish-256 context.
block	A pointer to the block to encrypt.

Remarks

This function is deterministic, as are all of the block cipher `Forward` and `Inverse` functions, and will not modify the state of the provided context.

6.21.2.4 void threefish256_inverse (struct THREEFISH256_STATE * *state,* uint64_t * *block*)

Decrypts a 256-bit block (as a `uint64_t`[4] structure).

Parameters

ctx	An initialized Threefish-256 context.
block	A pointer to the block to decrypt.

Remarks

See remarks for `threefish256_forward()`.

6.21.2.5 void threefish256_free (struct THREEFISH256_STATE ∗ *state*)

Frees the memory associated with a Threefish-256 cipher context and securely erases sensitive context information such as key material.

Parameters

ctx	An allocated Threefish-256 context.

Remarks

The context need not have been initialized.
Passing nil to this function is a no-op.

6.21.2.6 void threefish256_set_primitive (struct BLOCK_CIPHER ∗ *cipher*)

This function populates a block cipher object with the Threefish-256 functions and attributes, and is meant for internal use.

Parameters

cipher	A pointer to a block cipher object to populate.

Remarks

Once populated, the `BLOCK_CIPHER` struct can be freely used in the higher level `enc_block` interface.
If you have issued a call to `load_primitives()`, this function has already been called and you may use the `Threefish256()` function to access the underlying Threefish-256 block cipher object.

See Also

enc_block.h

6.21.2.7 void threefish256_key_schedule (uint64_t *key[4]*, uint64_t *tweak[2]*, uint64_t *subkeys[19][4]*)

This function is **stateless** and is meant to be used when a context-free access to the raw cryptographic block cipher is required (such as in the Skein hash function family which uses Threefish inside its compression function).

Remarks

As such, this function is for internal use only and may change with implementation. It is not recommended to use it in external code.
Performs the Threefish-256 key schedule.

6.21.2.8 void threefish256_forward_raw (uint64_t *block[4]*, uint64_t *subkeys[19][4]*)

See the `threefish256_key_schedule()` function.

Remarks

Computes the Threefish-256 forward permutation.

6.21.2.9 void threefish256_inverse_raw (uint64_t *block[4]*, uint64_t *subkeys[19][4]*)

See the `threefish256_key_schedule()` function.

Remarks

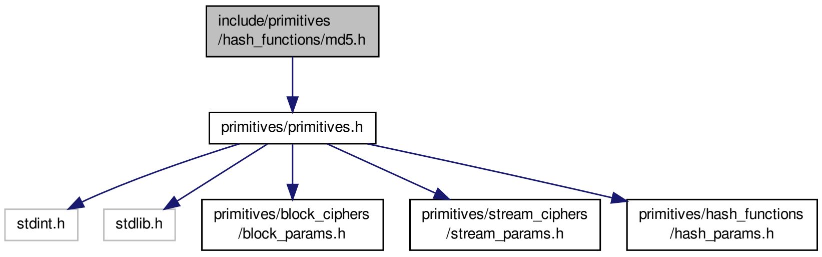
Computes the Threefish-256 inverse permutation.

6.22 include/primitives/hash_functions/md5.h File Reference

MD5 hash function.

```
#include <primitives/primitives.h>
```

Include dependency graph for md5.h:


include/primitives /hash_functions/md5.h
primitives/primitives.h
stdint.h
stdlib.h
primitives/block_ciphers /block_params.h
primitives/stream_ciphers /stream_params.h
primitives/hash_functions /hash_params.h

Functions

- struct MD5_STATE ∗ md5_alloc ()
- int md5_init (struct MD5_STATE ∗state, void ∗params)
- void md5_update (struct MD5_STATE ∗state, void ∗buffer, size_t size)
- void md5_final (struct MD5_STATE ∗state, void ∗digest)
- void md5_free (struct MD5_STATE ∗state)
- void md5_copy (struct MD5_STATE ∗dst, struct MD5_STATE ∗src)
- void md5_set_primitive (struct HASH_FUNCTION ∗hash)

6.22.1 Detailed Description

MD5 hash function. This is the MD5 hash function, which produces a 128-bit digest.

6.22.2 Function Documentation

6.22.2.1 struct MD5_STATE∗ md5_alloc ( )

Allocates and returns an uninitialized MD5 hash function context.

Returns

The allocated context, or nil on allocation failure.

6.22.2.2 int md5_init (struct MD5_STATE ∗ *state,* void ∗ *params*)

Initializes an MD5 hash function context.

Parameters

ctx	An allocated MD5 context.
params	Ignored.

Returns

Returns `ORDO_SUCCESS`.

6.22.2.3 void md5_update (struct MD5_STATE * *state,* void * *buffer,* size_t *size*)

Feeds a buffer into the MD5 context, contributing to the final digest.

Parameters

ctx	An initialized MD5 context.
buffer	A pointer to a buffer.
size	The amount of data, in bytes, to read from `buffer`.

Remarks

This function has the property that Update(A) followed by Update(B) is equivalent to Update(A $||$ B) where $||$ denotes concatenation.

6.22.2.4 void md5_final (struct MD5_STATE * *state,* void * *digest*)

Retrieves the final digest from the MD5 context.

Parameters

ctx	An initialized MD5 context.
digest	A buffer in which to write the digest.

Remarks

The buffer must be at least 16 bytes (128 bits) long.
If this function is immediately called after `MD5_Init()`, the result is the zero digest, that is, the digest corresponding to an input of length zero.

6.22.2.5 void md5_free (struct MD5_STATE * *state*)

Frees the memory associated with the MD5 context.

Parameters

ctx	An allocated MD5 context.

Remarks

The context need not have been initialized.
Passing nil to this function is a no-op.

6.22.2.6 void md5_copy (struct MD5_STATE * *dst,* struct MD5_STATE * *src*)

Performs a deep copy of a context into another.

Parameters

dst	The destination context.
src	The source context.

Remarks

Both contexts must have been allocated with `md5_alloc()`. If a generic interface working for any hash function is required, use `hashFunctionCopy()`.

6.22.2.7 void md5_set_primitive (struct HASH_FUNCTION ∗ *hash*)

Populates a stream cipher object with the MD5 functions and attributes, and is meant for internal use.

Parameters

hash	A pointer to a hash function object to populate.

Remarks

Once populated, the `HASH_FUNCTION` struct can be freely used in the higher level `hash` interface.
If you have issued a call to `load_primitives()`, this function has already been called and you may use the `MD5()` function to access the underlying MD5 hash function object.

See Also

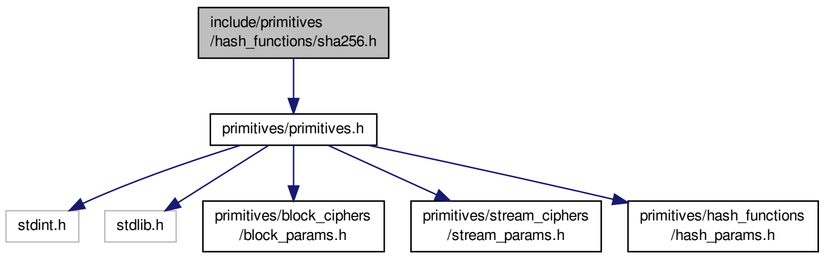
hash.h

6.23 include/primitives/hash_functions/sha256.h File Reference

SHA-256 hash function.

```
#include <primitives/primitives.h>
```

Include dependency graph for sha256.h:



Functions

- struct SHA256_STATE ∗ sha256_alloc ()
- int sha256_init (struct SHA256_STATE ∗state, void ∗params)
- void sha256_update (struct SHA256_STATE ∗state, void ∗buffer, size_t size)
- void sha256_final (struct SHA256_STATE ∗state, void ∗digest)
- void sha256_free (struct SHA256_STATE ∗state)
- void sha256_copy (struct SHA256_STATE ∗dst, struct SHA256_STATE ∗src)
- void sha256_set_primitive (struct HASH_FUNCTION ∗hash)

6.23.1 Detailed Description

SHA-256 hash function. This is the SHA-256 hash function, which produces a 256-bit digest.

6.23.2 Function Documentation

6.23.2.1 struct SHA256_STATE∗ sha256_alloc ( )

Allocates and returns an uninitialized SHA-256 hash function context.

Returns

The allocated context, or nil on allocation failure.

6.23.2.2 int sha256_init (struct SHA256_STATE ∗ *state,* void ∗ *params*)

Initializes an SHA-256 hash function context.

Parameters

ctx	An allocated SHA-256 context.
params	Ignored.

Returns

Returns `ORDO_SUCCESS`.

6.23.2.3 void sha256_update (struct SHA256_STATE ∗ *state,* void ∗ *buffer,* size_t *size*)

Feeds a buffer into the SHA-256 context, contributing to the final digest.

Parameters

ctx	An initialized SHA-256 context.
buffer	A pointer to a buffer.
size	The amount of data, in bytes, to read from `buffer`.

Remarks

This function has the property that Update(A) followed by Update(B) is equivalent to Update(A || B) where || denotes concatenation.

6.23.2.4 void sha256_final (struct SHA256_STATE ∗ *state,* void ∗ *digest*)

Retrieves the final digest from the SHA-256 context.

Parameters

ctx	An initialized SHA-256 context.
digest	A buffer in which to write the digest.

Remarks

The buffer must be at least 16 bytes (128 bits) long.
If this function is immediately called after `sha256_init()`, the result is the zero digest, that is, the digest corresponding to an input of length zero.

6.23.2.5 void sha256_free (struct SHA256_STATE $*$ *state*)

Frees the memory associated with the SHA-256 context.

Parameters

ctx	An allocated SHA-256 context.

Remarks

The context need not have been initialized.
Passing nil to this function is a no-op.

6.23.2.6 void sha256_copy (struct SHA256_STATE ∗ *dst,* struct SHA256_STATE ∗ *src*)

Performs a deep copy of a context into another.

Parameters

dst	The destination context.
src	The source context.

Remarks

Both contexts must have been allocated with `sha256_alloc()`. If a generic interface working for any hash function is required, use `hashFunctionCopy()`.

6.23.2.7 void sha256_set_primitive (struct HASH_FUNCTION ∗ *hash*)

Populates a stream cipher object with the SHA-256 functions and attributes, and is meant for internal use.

Parameters

hash	A pointer to a hash function object to populate.

Remarks

Once populated, the `HASH_FUNCTION` struct can be freely used in the higher level `hash` interface.
If you have issued a call to `load_primitives()`, this function has already been called and you may use the `SHA256()` function to access the underlying SHA-256 hash function object.

See Also

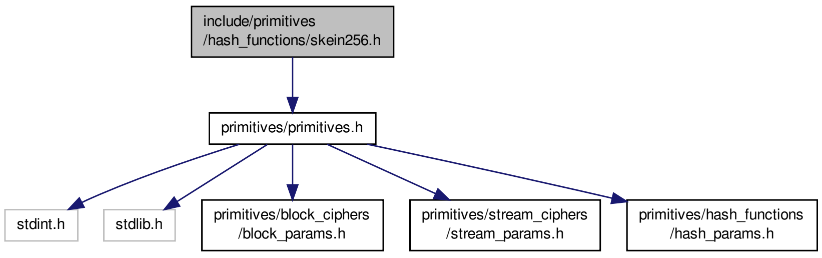
hash.h

6.24 include/primitives/hash_functions/skein256.h File Reference

Skein-256 hash function.

`#include <primitives/primitives.h>`

Include dependency graph for skein256.h:



6.24.1 Detailed Description

Skein-256 hash function. This is the Skein-256 hash function, which produces a 256-bit digest by default (but has parameters to output a longer digest) and has a 256-bit internal state. This implementation supports messages up to a length of 2^64 - 1 bytes instead of the 2^96 - 1 available, but we trust this will not be an issue. This is a rather flexible hash with lots of options. The following features are marked [x] if available, [ ] otherwise:

[x] Simple hashing (256-bit digest, any-length message)

[x] Variable-length output (any-length digest, any-length message, uses parameters)

[x] Semi-personalizable configuration block (everything is changeable, but generally you should only change the output length field if you want to remain compliant)

[ ] Personalization block

[ ] HMAC block

[ ] Other blocks

Todo Expand Skein-256 parameters (add possible extra blocks, such as personalization, hmac, nonce, etc...). This will probably require a rewrite of the UBI subsystem which is rather hardcoded and rigid at the moment.

Todo Rewrite the UBI code properly.

See Also

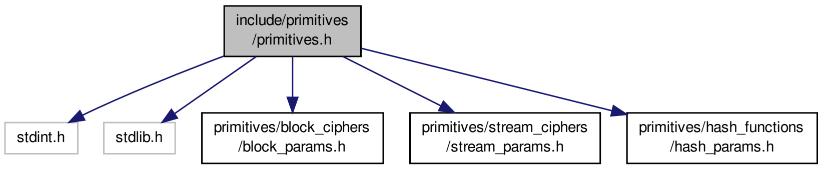
skein256.c

6.25 include/primitives/primitives.h File Reference

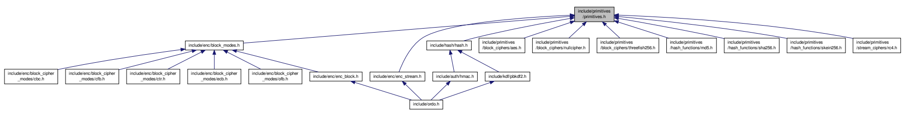
Cryptographic primitives.

```
#include <stdint.h>
#include <stdlib.h>
#include <primitives/block_ciphers/block_params.h>
#include <primitives/stream_ciphers/stream_params.h>
#include <primitives/hash_functions/hash_params.h>
```

Include dependency graph for primitives.h:



This graph shows which files directly or indirectly include this file:



Functions

- void load_primitives ()
- struct BLOCK_CIPHER ∗ NullCipher ()
- struct BLOCK_CIPHER ∗ Threefish256 ()
- struct BLOCK_CIPHER ∗ AES ()
- struct BLOCK_CIPHER ∗ block_cipher_by_name (char ∗name)
- struct BLOCK_CIPHER ∗ block_cipher_by_id (size_t id)
- struct STREAM_CIPHER ∗ RC4 ()
- struct STREAM_CIPHER ∗ stream_cipher_by_name (char ∗name)
- struct STREAM_CIPHER ∗ stream_cipher_by_id (size_t id)
- struct HASH_FUNCTION ∗ SHA256 ()
- struct HASH_FUNCTION ∗ MD5 ()
- struct HASH_FUNCTION ∗ Skein256 ()
- struct HASH_FUNCTION ∗ hash_function_by_name (char ∗name)
- struct HASH_FUNCTION ∗ hash_function_by_id (size_t id)

6.25.1 Detailed Description

Cryptographic primitives. This declares all the cryptographic primitives in the library, abstracting various types of primitives (block ciphers, hash functions, etc..) through higher level interfaces.

6.25.2 Function Documentation

6.25.2.1 void load_primitives ( )

Loads all primitives. This must be called before you may use `RC4()`, `NullCipher()`, etc... or the helper functions `block_cipher_by_name()` or `stream_cipher_by_id()`, and so on.

6.25.2.2 struct BLOCK_CIPHER∗ NullCipher ( )

The NullCipher block cipher.

6.25.2.3 struct BLOCK_CIPHER∗ Threefish256 ( )

The Threefish-256 block cipher.

6.25.2.4 struct BLOCK_CIPHER∗ AES ( )

The AES block cipher.

6.25.2.5 struct BLOCK_CIPHER∗ block_cipher_by_name (char ∗ *name*)

Returns a block cipher object from a name.

6.25.2.6 struct BLOCK_CIPHER∗ block_cipher_by_id (size_t *id*)

Returns a block cipher object from an ID.

6.25.2.7 struct STREAM_CIPHER∗ RC4 ( )

The RC4 stream cipher.

6.25.2.8 struct STREAM_CIPHER∗ stream_cipher_by_name (char ∗ *name*)

Returns a stream cipher object from a name.

6.25.2.9 struct STREAM_CIPHER∗ stream_cipher_by_id (size_t *id*)

Returns a stream cipher object from an ID.

6.25.2.10 struct HASH_FUNCTION∗ SHA256 ( )

The SHA256 hash function.

6.25.2.11 struct HASH_FUNCTION∗ MD5 ( )

The MD5 hash function.

6.25.2.12 struct HASH_FUNCTION∗ Skein256 ( )

The Skein-256 hash function.

6.25.2.13 struct HASH_FUNCTION∗ hash_function_by_name (char ∗ *name*)

Returns a hash function object from a name.

6.25.2.14 struct HASH_FUNCTION∗ hash_function_by_id (size_t *id*)

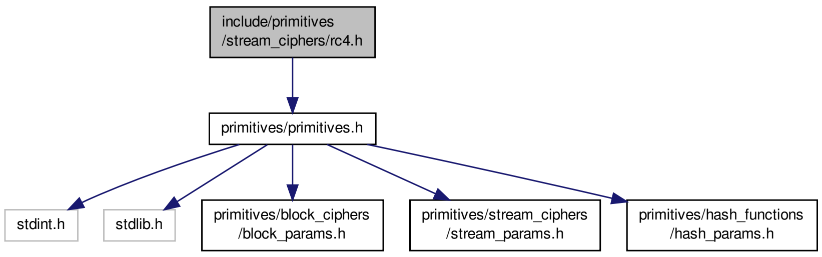
Returns a hash function object from an ID.

6.26 include/primitives/stream_ciphers/rc4.h File Reference

RC4 stream cipher.

```
#include <primitives/primitives.h>
```

Include dependency graph for rc4.h:


include/primitives/stream_ciphers/rc4.h
primitives/primitives.h
stdint.h
stdlib.h
primitives/block_ciphers/block_params.h
primitives/stream_ciphers/stream_params.h
primitives/hash_functions/hash_params.h

Functions

- struct RC4_STATE ∗ rc4_alloc ()
- int rc4_init (struct RC4_STATE ∗state, uint8_t ∗key, size_t keySize, struct RC4_PARAMS ∗params)
- void rc4_update (struct RC4_STATE ∗state, uint8_t ∗buffer, size_t len)
- void rc4_free (struct RC4_STATE ∗state)
- void rc4_set_primitive (struct STREAM_CIPHER ∗cipher)

6.26.1 Detailed Description

RC4 stream cipher. RC4 is a stream cipher, which accepts keys between 40 and 2048 bits (in multiples of 8 bits only). It accepts a parameter consisting of the number of initial keystream bytes to drop immediately after key schedule, effectively implementing RC4-drop[n]. If no drop parameter is passed, the implementation drops 2048 bytes by default.

Todo Better ABI translation for Windows assembler implementation(right now it's a brute-force push/pop/swap to explicitly translate parameter passing)

6.26.2 Function Documentation

6.26.2.1 struct RC4_STATE∗ rc4_alloc ( )

Allocates and returns an uninitialized RC4 stream cipher context.

Returns

The allocated context, or nil on allocation failure.

6.26.2.2 int rc4_init (struct RC4_STATE ∗ *state,* uint8_t ∗ *key,* size_t *keySize,* struct RC4_PARAMS ∗ *params*)

Initializes an RC4 stream cipher context.

Parameters

ctx	An allocated RC4 context.
key	A pointer to a buffer containing the encryption key.
keySize	The size, in bytes, of the key to read from `key`.
params	A pointer to an RC4 parameter structure.

Returns

Returns `ORDO_SUCCESS` on success, or `ORDO_KEY_SIZE` if the key size passed was invalid.

Remarks

The `params` parameter may be nil if no parameters are required.

6.26.2.3 void rc4_update (struct RC4_STATE $*$ *state,* uint8_t $*$ *buffer,* size_t *len*)

Encrypts or decrypts a buffer (as an array of bytes).

Parameters

ctx	An initialized RC4 context.
buffer	A pointer to the buffer to encrypt or decrypt.
len	The length of the buffer pointed to by `buffer`.

Remarks

This function will update the passed context, such that the keystream bytes generated to encrypt `buffer` will be discarded after use, and will not be produced again on subsequent calls.
Due to the nature of stream ciphers, encryption and decryption are identical, so this function serves both purposes.

6.26.2.4 void rc4_free (struct RC4_STATE $*$ *state*)

Frees the memory associated with an RC4 cipher context and securely erases sensitive context information such as state which may compromise the key.

Parameters

ctx	An allocated RC4 context.

Remarks

The context need not have been initialized.
Passing nil to this function is a no-op.

6.26.2.5 void rc4_set_primitive (struct STREAM_CIPHER $*$ *cipher*)

This function populates a stream cipher object with the RC4 functions and attributes, and is meant for internal use.

Parameters

cipher	A pointer to a stream cipher object to populate.

Remarks

Once populated, the `STREAM_CIPHER` struct can be freely used in the higher level `enc_stream` interface. If you have issued a call to `load_primitives()`, this function has already been called and you may use the `RC4()` function to access the underlying RC4 stream cipher object.

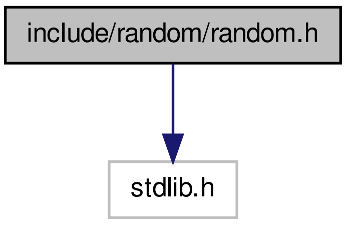
See Also

enc_stream.h

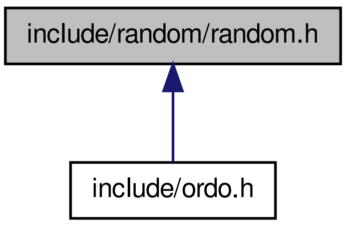
6.27 include/random/random.h File Reference

Cryptographically secure pseudorandom number generation.

```
#include <stdlib.h>
```
Include dependency graph for random.h:



This graph shows which files directly or indirectly include this file:



Functions

- int ordo_random (unsigned char ∗buffer, size_t size)

6.27.1 Detailed Description

Cryptographically secure pseudorandom number generation. Exposes the Ordo CSPRNG (Cryptographically Secure PseudoRandom Number Generator) interface, which is basically a cross-platform wrapper to the OS-provided entropy pool.

Linux: Reads from /dev/urandom

Windows: Acquires a CSP token and calls CryptGenRandom.

Todo Implement ordo_random for other platforms and add proper error handling for Windows.

See Also

random.c

6.27.2 Function Documentation

6.27.2.1 int ordo_random (unsigned char $*$ *buffer,* size_t *size*)

Generates cryptographic-grade pseudorandom numbers.

Parameters

buffer	Points to the buffer in which to write the pseudorandom stream.
size	The number of bytes to generate and to write to the buffer.

Returns

Returns `ORDO_SUCCESS` on success, and returns an error code on failure.

Remarks

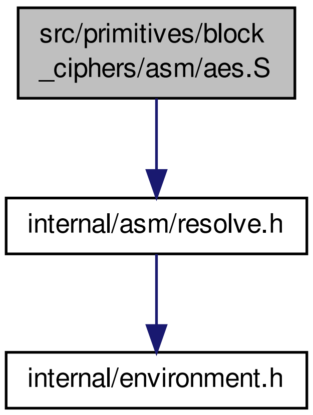
This function uses the underlying CSPRNG provided by your operating system.

6.28 src/primitives/block_ciphers/asm/aes.S File Reference

AES assembly code [ABI: Windows x64, Linux x64].

```
#include <internal/asm/resolve.h>
```

Include dependency graph for aes.S:



6.28.1 Detailed Description

AES assembly code [ABI: Windows x64, Linux x64]. This file contains an implementation of AES (Advanced Encryption Standard) for AES-NI-capable processors. This technically works for 32-bit processors too, but the x86 version doesn't seem to recognize the forward function declarations and complains about undefined references...

Todo Fix 32-bit "undefined reference" bug.

Implement AES key schedule with AES-NI, to get rid of the substitution box.

See Also

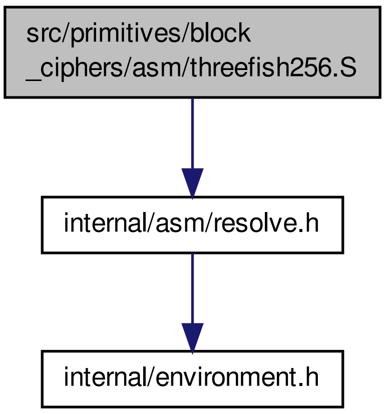
aes.c
aes.h

6.29 src/primitives/block_ciphers/asm/threefish256.S File Reference

Threefish-256 assembly code [ABI: Windows x64, Linux x64].

```
#include <internal/asm/resolve.h>
```

Include dependency graph for threefish256.S:



6.29.1 Detailed Description

Threefish-256 assembly code [ABI: Windows x64, Linux x64]. This file contains an optimized, hand-written assembler version of the Threefish-256 cipher primitive, for x64 processors. Only the forward and inverse permutations are included here, the key schedule and other functions remain in the standard implementation (threefish256.c). Note conditional compilation has been added here *and* in threefish.c to ensure that when this optimized version is enabled, the standard version in threefish.c is disabled and vice versa.

See Also

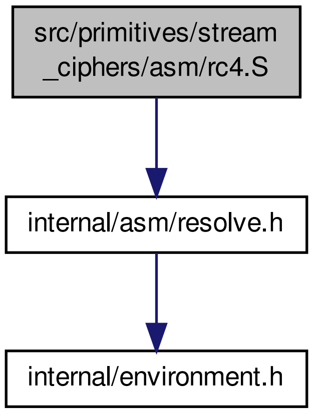
threefish256.c
threefish256.h

6.30 src/primitives/stream_ciphers/asm/rc4.S File Reference

RC4 assembly code [ABI: Windows x64, Linux x64].

```
#include <internal/asm/resolve.h>
```

Include dependency graph for rc4.S:



6.30.1 Detailed Description

RC4 assembly code [ABI: Windows x64, Linux x64]. This file contains an optimized, assembler version of RC4 for x86_64 processors (see original header for credits). It is considerably faster than the standard C version, and the exported rc4_update_ASM function implemented below takes the following parameters:

Param #1 : pointer to an RC4_STATE structure (the correct one, with the index pointers i, j first and 64-bit aligned)

Param #2 : 64-bit unsigned integer to the length of the buffer to encrypt with RC4

Param #3 : pointer to the input buffer

Param #4 : pointer to the output buffer (may be identical to the input buffer)

See Also

rc4.c
rc4.h

Index