Linux Crypto Documentation

The kernel development community

Jan 15, 2023

CONTENTS

Author Stephan Mueller

Author Marek Vasut

This documentation outlines the Linux kernel crypto API with its concepts, details about developing cipher implementations, employment of the API for cryptographic use cases, as well as programming examples.

Table of contents

CHAPTER

ONE

KERNEL CRYPTO API INTERFACE SPECIFICATION

1.1 Introduction

The kernel crypto API offers a rich set of cryptographic ciphers as well as other data transformation mechanisms and methods to invoke these. This document contains a description of the API and provides example code.

To understand and properly use the kernel crypto API a brief explanation of its structure is given. Based on the architecture, the API can be separated into different components. Following the architecture specification, hints to developers of ciphers are provided. Pointers to the API function call documentation are given at the end.

The kernel crypto API refers to all algorithms as "transformations". Therefore, a cipher handle variable usually has the name "tfm". Besides cryptographic operations, the kernel crypto API also knows compression transformations and handles them the same way as ciphers.

The kernel crypto API serves the following entity types:

- consumers requesting cryptographic services
- data transformation implementations (typically ciphers) that can be called by consumers using the kernel crypto API

This specification is intended for consumers of the kernel crypto API as well as for developers implementing ciphers. This API specification, however, does not discuss all API calls available to data transformation implementations (i.e. implementations of ciphers and other transformations (such as CRC or even compression algorithms) that can register with the kernel crypto API).

Note: The terms "transformation" and cipher algorithm are used interchangeably.

1.2 Terminology

The transformation implementation is an actual code or interface to hardware which implements a certain transformation with precisely defined behavior.

The transformation object (TFM) is an instance of a transformation implementation. There can be multiple transformation objects associated with a single transformation implementation. Each of those transformation objects is held by a crypto API consumer or another transformation. Transformation object is allocated when a crypto API consumer requests a transformation implementation. The consumer is then provided with a structure, which contains a transformation object (TFM).

The structure that contains transformation objects may also be referred to as a “cipher handle”. Such a cipher handle is always subject to the following phases that are reflected in the API calls applicable to such a cipher handle:

1. Initialization of a cipher handle.
2. Execution of all intended cipher operations applicable for the handle where the cipher handle must be furnished to every API call.
3. Destruction of a cipher handle.

When using the initialization API calls, a cipher handle is created and returned to the consumer. Therefore, please refer to all initialization API calls that refer to the data structure type a consumer is expected to receive and subsequently to use. The initialization API calls have all the same naming conventions of crypto_alloc*.

The transformation context is private data associated with the transformation object.

CHAPTER

TWO

SCATTERLIST CRYPTOGRAPHIC API

2.1 Introduction

The Scatterlist Crypto API takes page vectors (scatterlists) as arguments, and works directly on pages. In some cases (e.g. ECB mode ciphers), this will allow for pages to be encrypted in-place with no copying.

One of the initial goals of this design was to readily support IPsec, so that processing can be applied to paged skb's without the need for linearization.

2.2 Details

At the lowest level are algorithms, which register dynamically with the API.

'Transforms' are user-instantiated objects, which maintain state, handle all of the implementation logic (e.g. manipulating page vectors) and provide an abstraction to the underlying algorithms. However, at the user level they are very simple.

Conceptually, the API layering looks like this:

```
[transform api]  (user interface)
[transform ops]  (per-type logic glue e.g. cipher.c, compress.c)
[algorithm api]  (for registering algorithms)
```

The idea is to make the user interface and algorithm registration API very simple, while hiding the core logic from both. Many good ideas from existing APIs such as Cryptoapi and Nettle have been adapted for this.

The API currently supports five main types of transforms: AEAD (Authenticated Encryption with Associated Data), Block Ciphers, Ciphers, Compressors and Hashes.

Please note that Block Ciphers is somewhat of a misnomer. It is in fact meant to support all ciphers including stream ciphers. The difference between Block Ciphers and Ciphers is that the latter operates on exactly one block while the former can operate on an arbitrary amount of data, subject to block size requirements (i.e., non-stream ciphers can only process multiples of blocks).

Here's an example of how to use the API:

```
#include <crypto/hash.h>
#include <linux/err.h>
```

```
#include <linux/scatterlist.h>

struct scatterlist sg[2];
char result[128];
struct crypto_ahash *tfm;
struct ahash_request *req;

tfm = crypto_alloc_ahash("md5", 0, CRYPTO_ALG_ASYNC);
if (IS_ERR(tfm))
        fail();

/* ... set up the scatterlists ... */

req = ahash_request_alloc(tfm, GFP_ATOMIC);
if (!req)
        fail();

ahash_request_set_callback(req, 0, NULL, NULL);
ahash_request_set_crypt(req, sg, result, 2);

if (crypto_ahash_digest(req))
        fail();

ahash_request_free(req);
crypto_free_ahash(tfm);
```

Many real examples are available in the regression test module (tcrypt.c).

2.3 Developer Notes

Transforms may only be allocated in user context, and cryptographic methods may only be called from softirq and user contexts. For transforms with a setkey method it too should only be called from user context.

When using the API for ciphers, performance will be optimal if each scatterlist contains data which is a multiple of the cipher's block size (typically 8 bytes). This prevents having to do any copying across non-aligned page fragment boundaries.

2.4 Adding New Algorithms

When submitting a new algorithm for inclusion, a mandatory requirement is that at least a few test vectors from known sources (preferably standards) be included.

Converting existing well known code is preferred, as it is more likely to have been reviewed and widely tested. If submitting code from LGPL sources, please consider changing the license to GPL (see section 3 of the LGPL).

Algorithms submitted must also be generally patent-free (e.g. IDEA will not be included in the mainline until around 2011), and be based on a recognized standard and/or have been subjected

to appropriate peer review.

Also check for any RFCs which may relate to the use of specific algorithms, as well as general application notes such as RFC2451 (“The ESP CBC-Mode Cipher Algorithms”).

It’s a good idea to avoid using lots of macros and use inlined functions instead, as gcc does a good job with inlining, while excessive use of macros can cause compilation problems on some platforms.

Also check the TODO list at the web site listed below to see what people might already be working on.

2.5 Bugs

Send bug reports to: linux-crypto@vger.kernel.org

Cc: Herbert Xu <herbert@gondor.apana.org.au>, David S. Miller <davem@redhat.com>

2.6 Further Information

For further patches and various updates, including the current TODO list, see: http://gondor.apana.org.au/~herbert/crypto/

2.7 Authors

- James Morris
- David S. Miller
- Herbert Xu

2.8 Credits

The following people provided invaluable feedback during the development of the API:

- Alexey Kuznetzov
- Rusty Russell
- Herbert Valerio Riedel
- Jeff Garzik
- Michael Richardson
- Andrew Morton
- Ingo Oeser
- Christoph Hellwig

Portions of this API were derived from the following projects:

Kerneli Cryptoapi (http://www.kerneli.org/)

- Alexander Kjeldaas
- Herbert Valerio Riedel
- Kyle McMartin
- Jean-Luc Cooke
- David Bryson
- Clemens Fruhwirth
- Tobias Ringstrom
- Harald Welte

and;

Nettle (https://www.lysator.liu.se/~nisse/nettle/)

- Niels Möller

Original developers of the crypto algorithms:

- Dana L. How (DES)
- Andrew Tridgell and Steve French (MD4)
- Colin Plumb (MD5)
- Steve Reid (SHA1)
- Jean-Luc Cooke (SHA256, SHA384, SHA512)
- Kazunori Miyazawa / USAGI (HMAC)
- Matthew Skala (Twofish)
- Dag Arne Osvik (Serpent)
- Brian Gladman (AES)
- Kartikey Mahendra Bhatt (CAST6)
- Jon Oberheide (ARC4)
- Jouni Malinen (Michael MIC)
- NTT(Nippon Telegraph and Telephone Corporation) (Camellia)

SHA1 algorithm contributors:

- Jean-Francois Dive

DES algorithm contributors:

- Raimar Falke
- Gisle Sælensminde
- Niels Möller

Blowfish algorithm contributors:

- Herbert Valerio Riedel
- Kyle McMartin

Twofish algorithm contributors:

- Werner Koch
- Marc Mutz

SHA256/384/512 algorithm contributors:

- Andrew McDonald
- Kyle McMartin
- Herbert Valerio Riedel

AES algorithm contributors:

- Alexander Kjeldaas
- Herbert Valerio Riedel
- Kyle McMartin
- Adam J. Richter
- Fruhwirth Clemens (i586)
- Linus Torvalds (i586)

CAST5 algorithm contributors:

- Kartikey Mahendra Bhatt (original developers unknown, FSF copyright).

TEA/XTEA algorithm contributors:

- Aaron Grothe
- Michael Ringe

Khazad algorithm contributors:

- Aaron Grothe

Whirlpool algorithm contributors:

- Aaron Grothe
- Jean-Luc Cooke

Anubis algorithm contributors:

- Aaron Grothe

Tiger algorithm contributors:

- Aaron Grothe

VIA PadLock contributors:

- Michal Ludvig

Camellia algorithm contributors:

- NTT(Nippon Telegraph and Telephone Corporation) (Camellia)

Generic scatterwalk code by Adam J. Richter <adam@yggdrasil.com>

Please send any credits updates or corrections to: Herbert Xu <herbert@gondor.apana.org.au>

CHAPTER

THREE

KERNEL CRYPTO API ARCHITECTURE

3.1 Cipher algorithm types

The kernel crypto API provides different API calls for the following cipher types:

- Symmetric ciphers
- AEAD ciphers
- Message digest, including keyed message digest
- Random number generation
- User space interface

3.2 Ciphers And Templates

The kernel crypto API provides implementations of single block ciphers and message digests. In addition, the kernel crypto API provides numerous “templates” that can be used in conjunction with the single block ciphers and message digests. Templates include all types of block chaining mode, the HMAC mechanism, etc.

Single block ciphers and message digests can either be directly used by a caller or invoked together with a template to form multi-block ciphers or keyed message digests.

A single block cipher may even be called with multiple templates. However, templates cannot be used without a single cipher.

See /proc/crypto and search for “name”. For example:

- aes
- ecb(aes)
- cmac(aes)
- ccm(aes)
- rfc4106(gcm(aes))
- sha1
- hmac(sha1)
- authenc(hmac(sha1),cbc(aes))

In these examples, “aes” and “sha1” are the ciphers and all others are the templates.

3.3 Synchronous And Asynchronous Operation

The kernel crypto API provides synchronous and asynchronous API operations.

When using the synchronous API operation, the caller invokes a cipher operation which is performed synchronously by the kernel crypto API. That means, the caller waits until the cipher operation completes. Therefore, the kernel crypto API calls work like regular function calls. For synchronous operation, the set of API calls is small and conceptually similar to any other crypto library.

Asynchronous operation is provided by the kernel crypto API which implies that the invocation of a cipher operation will complete almost instantly. That invocation triggers the cipher operation but it does not signal its completion. Before invoking a cipher operation, the caller must provide a callback function the kernel crypto API can invoke to signal the completion of the cipher operation. Furthermore, the caller must ensure it can handle such asynchronous events by applying appropriate locking around its data. The kernel crypto API does not perform any special serialization operation to protect the caller's data integrity.

3.4 Crypto API Cipher References And Priority

A cipher is referenced by the caller with a string. That string has the following semantics:

```
template(single block cipher)
```

where "template" and "single block cipher" is the aforementioned template and single block cipher, respectively. If applicable, additional templates may enclose other templates, such as

```
template1(template2(single block cipher))
```

The kernel crypto API may provide multiple implementations of a template or a single block cipher. For example, AES on newer Intel hardware has the following implementations: AES-NI, assembler implementation, or straight C. Now, when using the string "aes" with the kernel crypto API, which cipher implementation is used? The answer to that question is the priority number assigned to each cipher implementation by the kernel crypto API. When a caller uses the string to refer to a cipher during initialization of a cipher handle, the kernel crypto API looks up all implementations providing an implementation with that name and selects the implementation with the highest priority.

Now, a caller may have the need to refer to a specific cipher implementation and thus does not want to rely on the priority-based selection. To accommodate this scenario, the kernel crypto API allows the cipher implementation to register a unique name in addition to common names. When using that unique name, a caller is therefore always sure to refer to the intended cipher implementation.

The list of available ciphers is given in /proc/crypto. However, that list does not specify all possible permutations of templates and ciphers. Each block listed in /proc/crypto may contain the following information – if one of the components listed as follows are not applicable to a cipher, it is not displayed:

- name: the generic name of the cipher that is subject to the priority-based selection – this name can be used by the cipher allocation API calls (all names listed above are examples for such generic names)

- driver: the unique name of the cipher – this name can be used by the cipher allocation API calls
- module: the kernel module providing the cipher implementation (or “kernel” for statically linked ciphers)
- priority: the priority value of the cipher implementation
- refcnt: the reference count of the respective cipher (i.e. the number of current consumers of this cipher)
- selftest: specification whether the self test for the cipher passed
- type:
  - skcipher for symmetric key ciphers
  - cipher for single block ciphers that may be used with an additional template
  - shash for synchronous message digest
  - ahash for asynchronous message digest
  - aead for AEAD cipher type
  - compression for compression type transformations
  - rng for random number generator
  - kpp for a Key-agreement Protocol Primitive (KPP) cipher such as an ECDH or DH implementation
- blocksize: blocksize of cipher in bytes
- keysize: key size in bytes
- ivsize: IV size in bytes
- seedsize: required size of seed data for random number generator
- digestsize: output size of the message digest
- geniv: IV generator (obsolete)

3.5 Key Sizes

When allocating a cipher handle, the caller only specifies the cipher type. Symmetric ciphers, however, typically support multiple key sizes (e.g. AES-128 vs. AES-192 vs. AES-256). These key sizes are determined with the length of the provided key. Thus, the kernel crypto API does not provide a separate way to select the particular symmetric cipher key size.

3.6 Cipher Allocation Type And Masks

The different cipher handle allocation functions allow the specification of a type and mask flag. Both parameters have the following meaning (and are therefore not covered in the subsequent sections).

The type flag specifies the type of the cipher algorithm. The caller usually provides a 0 when the caller wants the default handling. Otherwise, the caller may provide the following selections which match the aforementioned cipher types:

- CRYPTO_ALG_TYPE_CIPHER Single block cipher
- CRYPTO_ALG_TYPE_COMPRESS Compression
- CRYPTO_ALG_TYPE_AEAD Authenticated Encryption with Associated Data (MAC)
- CRYPTO_ALG_TYPE_KPP Key-agreement Protocol Primitive (KPP) such as an ECDH or DH implementation
- CRYPTO_ALG_TYPE_HASH Raw message digest
- CRYPTO_ALG_TYPE_SHASH Synchronous multi-block hash
- CRYPTO_ALG_TYPE_AHASH Asynchronous multi-block hash
- CRYPTO_ALG_TYPE_RNG Random Number Generation
- CRYPTO_ALG_TYPE_AKCIPHER Asymmetric cipher
- CRYPTO_ALG_TYPE_PCOMPRESS Enhanced version of CRYPTO_ALG_TYPE_COMPRESS allowing for segmented compression / decompression instead of performing the operation on one segment only. CRYPTO_ALG_TYPE_PCOMPRESS is intended to replace CRYPTO_ALG_TYPE_COMPRESS once existing consumers are converted.

The mask flag restricts the type of cipher. The only allowed flag is CRYPTO_ALG_ASYNC to restrict the cipher lookup function to asynchronous ciphers. Usually, a caller provides a 0 for the mask flag.

When the caller provides a mask and type specification, the caller limits the search the kernel crypto API can perform for a suitable cipher implementation for the given cipher name. That means, even when a caller uses a cipher name that exists during its initialization call, the kernel crypto API may not select it due to the used type and mask field.

3.7 Internal Structure of Kernel Crypto API

The kernel crypto API has an internal structure where a cipher implementation may use many layers and indirections. This section shall help to clarify how the kernel crypto API uses various components to implement the complete cipher.

The following subsections explain the internal structure based on existing cipher implementations. The first section addresses the most complex scenario where all other scenarios form a logical subset.

3.7.1 Generic AEAD Cipher Structure

The following ASCII art decomposes the kernel crypto API layers when using the AEAD cipher with the automated IV generation. The shown example is used by the IPSEC layer.

For other use cases of AEAD ciphers, the ASCII art applies as well, but the caller may not use the AEAD cipher with a separate IV generator. In this case, the caller must generate the IV.

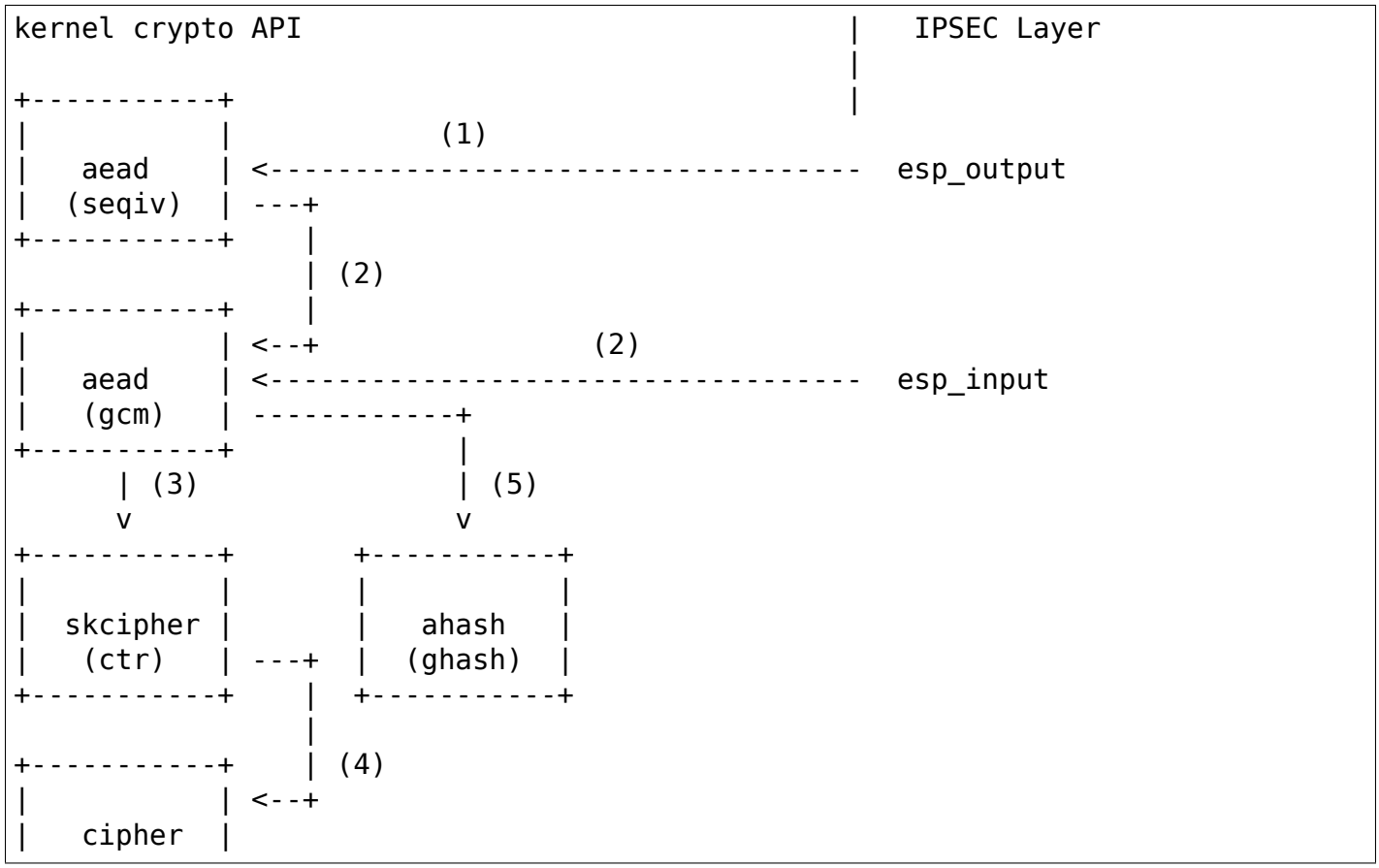
The depicted example decomposes the AEAD cipher of GCM(AES) based on the generic C implementations (gcm.c, aes-generic.c, ctr.c, ghash-generic.c, seqiv.c). The generic implementation serves as an example showing the complete logic of the kernel crypto API.

It is possible that some streamlined cipher implementations (like AES-NI) provide implementations merging aspects which in the view of the kernel crypto API cannot be decomposed into layers any more. In case of the AES-NI implementation, the CTR mode, the GHASH implementation and the AES cipher are all merged into one cipher implementation registered with the kernel crypto API. In this case, the concept described by the following ASCII art applies too. However, the decomposition of GCM into the individual sub-components by the kernel crypto API is not done any more.

Each block in the following ASCII art is an independent cipher instance obtained from the kernel crypto API. Each block is accessed by the caller or by other blocks using the API functions defined by the kernel crypto API for the cipher implementation type.

The blocks below indicate the cipher type as well as the specific logic implemented in the cipher.

The ASCII art picture also indicates the call structure, i.e. who calls which component. The arrows point to the invoked block where the caller uses the API applicable to the cipher type specified for the block.



```
|   (aes)   |
+-----------+
```

The following call sequence is applicable when the IPSEC layer triggers an encryption operation with the esp_output function. During configuration, the administrator set up the use of seqiv(rfc4106(gcm(aes))) as the cipher for ESP. The following call sequence is now depicted in the ASCII art above:

1. esp_output() invokes *crypto_aead_encrypt()* to trigger an encryption operation of the AEAD cipher with IV generator.

   The SEQIV generates the IV.

2. Now, SEQIV uses the AEAD API function calls to invoke the associated AEAD cipher. In our case, during the instantiation of SEQIV, the cipher handle for GCM is provided to SEQIV. This means that SEQIV invokes AEAD cipher operations with the GCM cipher handle.

   During instantiation of the GCM handle, the CTR(AES) and GHASH ciphers are instantiated. The cipher handles for CTR(AES) and GHASH are retained for later use.

   The GCM implementation is responsible to invoke the CTR mode AES and the GHASH cipher in the right manner to implement the GCM specification.

3. The GCM AEAD cipher type implementation now invokes the SKCIPHER API with the instantiated CTR(AES) cipher handle.

   During instantiation of the CTR(AES) cipher, the CIPHER type implementation of AES is instantiated. The cipher handle for AES is retained.

   That means that the SKCIPHER implementation of CTR(AES) only implements the CTR block chaining mode. After performing the block chaining operation, the CIPHER implementation of AES is invoked.

4. The SKCIPHER of CTR(AES) now invokes the CIPHER API with the AES cipher handle to encrypt one block.

5. The GCM AEAD implementation also invokes the GHASH cipher implementation via the AHASH API.

When the IPSEC layer triggers the esp_input() function, the same call sequence is followed with the only difference that the operation starts with step (2).

3.7.2 Generic Block Cipher Structure

Generic block ciphers follow the same concept as depicted with the ASCII art picture above.

For example, CBC(AES) is implemented with cbc.c, and aes-generic.c. The ASCII art picture above applies as well with the difference that only step (4) is used and the SKCIPHER block chaining mode is CBC.

3.7.3 Generic Keyed Message Digest Structure

Keyed message digest implementations again follow the same concept as depicted in the ASCII art picture above.

For example, HMAC(SHA256) is implemented with hmac.c and sha256_generic.c. The following ASCII art illustrates the implementation:

```
kernel crypto API             |       Caller
                              |
+-----------+         (1)     |
|           | <------------------  some_function
|   ahash   |
|   (hmac)  | ---+
+-----------+    |
                 | (2)
+-----------+    |
|           | <--+
|   shash   |
|  (sha256) |
+-----------+
```

The following call sequence is applicable when a caller triggers an HMAC operation:

1. The AHASH API functions are invoked by the caller. The HMAC implementation performs its operation as needed.

   During initialization of the HMAC cipher, the SHASH cipher type of SHA256 is instantiated. The cipher handle for the SHA256 instance is retained.

   At one time, the HMAC implementation requires a SHA256 operation where the SHA256 cipher handle is used.

2. The HMAC instance now invokes the SHASH API with the SHA256 cipher handle to calculate the message digest.

CHAPTER

FOUR

ASYNCHRONOUS TRANSFERS/TRANSFORMS API

4.1 1. Introduction

The async_tx API provides methods for describing a chain of asynchronous bulk memory transfers/transforms with support for inter-transactional dependencies. It is implemented as a dmaengine client that smooths over the details of different hardware offload engine implementations. Code that is written to the API can optimize for asynchronous operation and the API will fit the chain of operations to the available offload resources.

4.2 2.Genealogy

The API was initially designed to offload the memory copy and xor-parity-calculations of the md-raid5 driver using the offload engines present in the Intel(R) Xscale series of I/O processors. It also built on the 'dmaengine' layer developed for offloading memory copies in the network stack using Intel(R) I/OAT engines. The following design features surfaced as a result:

1. implicit synchronous path: users of the API do not need to know if the platform they are running on has offload capabilities. The operation will be offloaded when an engine is available and carried out in software otherwise.
2. cross channel dependency chains: the API allows a chain of dependent operations to be submitted, like xor->copy->xor in the raid5 case. The API automatically handles cases where the transition from one operation to another implies a hardware channel switch.
3. dmaengine extensions to support multiple clients and operation types beyond 'memcpy'

4.3 3. Usage

4.3.1 3.1 General format of the API

```
struct dma_async_tx_descriptor *
async_<operation>(<op specific parameters>, struct async_submit ctl *submit)
```

4.3.2 3.2 Supported operations

memcpy	memory copy between a source and a destination buffer
memset	fill a destination buffer with a byte value
xor	xor a series of source buffers and write the result to a destination buffer
xor_val	xor a series of source buffers and set a flag if the result is zero. The implementation attempts to prevent writes to memory
pq	generate the p+q (raid6 syndrome) from a series of source buffers
pq_val	validate that a p and or q buffer are in sync with a given series of sources
datap	(raid6_datap_recov) recover a raid6 data block and the p block from the given sources
2data	(raid6_2data_recov) recover 2 raid6 data blocks from the given sources

4.3.3 3.3 Descriptor management

The return value is non-NULL and points to a 'descriptor' when the operation has been queued to execute asynchronously. Descriptors are recycled resources, under control of the offload engine driver, to be reused as operations complete. When an application needs to submit a chain of operations it must guarantee that the descriptor is not automatically recycled before the dependency is submitted. This requires that all descriptors be acknowledged by the application before the offload engine driver is allowed to recycle (or free) the descriptor. A descriptor can be acked by one of the following methods:

1. setting the ASYNC_TX_ACK flag if no child operations are to be submitted
2. submitting an unacknowledged descriptor as a dependency to another async_tx call will implicitly set the acknowledged state.
3. calling async_tx_ack() on the descriptor.

4.3.4 3.4 When does the operation execute?

Operations do not immediately issue after return from the async_<operation> call. Offload engine drivers batch operations to improve performance by reducing the number of mmio cycles needed to manage the channel. Once a driver-specific threshold is met the driver automatically issues pending operations. An application can force this event by calling async_tx_issue_pending_all(). This operates on all channels since the application has no knowledge of channel to operation mapping.

4.3.5 3.5 When does the operation complete?

There are two methods for an application to learn about the completion of an operation.

1. Call dma_wait_for_async_tx(). This call causes the CPU to spin while it polls for the completion of the operation. It handles dependency chains and issuing pending operations.
2. Specify a completion callback. The callback routine runs in tasklet context if the offload engine driver supports interrupts, or it is called in application context if the operation is carried out synchronously in software. The callback can be set in the call to async_<operation>, or when the application needs to submit a chain of unknown length it

can use the async_trigger_callback() routine to set a completion interrupt/callback at the end of the chain.

4.3.6 3.6 Constraints

1. Calls to async_<operation> are not permitted in IRQ context. Other contexts are permitted provided constraint #2 is not violated.
2. Completion callback routines cannot submit new operations. This results in recursion in the synchronous case and spin_locks being acquired twice in the asynchronous case.

4.3.7 3.7 Example

Perform a xor->copy->xor operation where each operation depends on the result from the previous operation:

```
void callback(void *param)
{
        struct completion *cmp = param;

        complete(cmp);
}

void run_xor_copy_xor(struct page **xor_srcs,
                      int xor_src_cnt,
                      struct page *xor_dest,
                      size_t xor_len,
                      struct page *copy_src,
                      struct page *copy_dest,
                      size_t copy_len)
{
        struct dma_async_tx_descriptor *tx;
        addr_conv_t addr_conv[xor_src_cnt];
        struct async_submit_ctl submit;
        addr_conv_t addr_conv[NDISKS];
        struct completion cmp;

        init_async_submit(&submit, ASYNC_TX_XOR_DROP_DST, NULL, NULL, NULL,
                          addr_conv);
        tx = async_xor(xor_dest, xor_srcs, 0, xor_src_cnt, xor_len, &submit)

        submit->depend_tx = tx;
        tx = async_memcpy(copy_dest, copy_src, 0, 0, copy_len, &submit);

        init_completion(&cmp);
        init_async_submit(&submit, ASYNC_TX_XOR_DROP_DST | ASYNC_TX_ACK, tx,
                          callback, &cmp, addr_conv);
        tx = async_xor(xor_dest, xor_srcs, 0, xor_src_cnt, xor_len, &submit);

        async_tx_issue_pending_all();
```

```
        wait_for_completion(&cmp);
}
```

See include/linux/async_tx.h for more information on the flags. See the ops_run_* and ops_complete_* routines in drivers/md/raid5.c for more implementation examples.

4.4 4. Driver Development Notes

4.4.1 4.1 Conformance points

There are a few conformance points required in dmaengine drivers to accommodate assumptions made by applications using the async_tx API:

1. Completion callbacks are expected to happen in tasklet context
2. dma_async_tx_descriptor fields are never manipulated in IRQ context
3. Use async_tx_run_dependencies() in the descriptor clean up path to handle submission of dependent operations

4.4.2 4.2 “My application needs exclusive control of hardware channels”

Primarily this requirement arises from cases where a DMA engine driver is being used to support device-to-memory operations. A channel that is performing these operations cannot, for many platform specific reasons, be shared. For these cases the dma_request_channel() interface is provided.

The interface is:

```
struct dma_chan *dma_request_channel(dma_cap_mask_t mask,
                                     dma_filter_fn filter_fn,
                                     void *filter_param);
```

Where dma_filter_fn is defined as:

```
typedef bool (*dma_filter_fn)(struct dma_chan *chan, void *filter_param);
```

When the optional ‘filter_fn’ parameter is set to NULL dma_request_channel simply returns the first channel that satisfies the capability mask. Otherwise, when the mask parameter is insufficient for specifying the necessary channel, the filter_fn routine can be used to disposition the available channels in the system. The filter_fn routine is called once for each free channel in the system. Upon seeing a suitable channel filter_fn returns DMA_ACK which flags that channel to be the return value from dma_request_channel. A channel allocated via this interface is exclusive to the caller, until dma_release_channel() is called.

The DMA_PRIVATE capability flag is used to tag dma devices that should not be used by the general-purpose allocator. It can be set at initialization time if it is known that a channel will always be private. Alternatively, it is set when dma_request_channel() finds an unused “public” channel.

A couple caveats to note when implementing a driver and consumer:

1. Once a channel has been privately allocated it will no longer be considered by the general-purpose allocator even after a call to dma_release_channel().
2. Since capabilities are specified at the device level a dma_device with multiple channels will either have all channels public, or all channels private.

4.4.3 5. Source

include/linux/dmaengine.h: core header file for DMA drivers and api users

drivers/dma/dmaengine.c: offload engine channel management routines

drivers/dma/: location for offload engine drivers

include/linux/async_tx.h: core header file for the async_tx api

crypto/async_tx/async_tx.c: async_tx interface to dmaengine and common code

crypto/async_tx/async_memcpy.c: copy offload

crypto/async_tx/async_xor.c: xor and xor zero sum offload

CHAPTER

FIVE

ASYMMETRIC / PUBLIC-KEY CRYPTOGRAPHY KEY TYPE

5.1 Overview

The “asymmetric” key type is designed to be a container for the keys used in public-key cryptography, without imposing any particular restrictions on the form or mechanism of the cryptography or form of the key.

The asymmetric key is given a subtype that defines what sort of data is associated with the key and provides operations to describe and destroy it. However, no requirement is made that the key data actually be stored in the key.

A completely in-kernel key retention and operation subtype can be defined, but it would also be possible to provide access to cryptographic hardware (such as a TPM) that might be used to both retain the relevant key and perform operations using that key. In such a case, the asymmetric key would then merely be an interface to the TPM driver.

Also provided is the concept of a data parser. Data parsers are responsible for extracting information from the blobs of data passed to the instantiation function. The first data parser that recognises the blob gets to set the subtype of the key and define the operations that can be done on that key.

A data parser may interpret the data blob as containing the bits representing a key, or it may interpret it as a reference to a key held somewhere else in the system (for example, a TPM).

5.2 Key Identification

If a key is added with an empty name, the instantiation data parsers are given the opportunity to pre-parse a key and to determine the description the key should be given from the content of the key.

This can then be used to refer to the key, either by complete match or by partial match. The key type may also use other criteria to refer to a key.

The asymmetric key type’s match function can then perform a wider range of comparisons than just the straightforward comparison of the description with the criterion string:

1) If the criterion string is of the form “id:<hexdigits>” then the match function will examine a key’s fingerprint to see if the hex digits given after the “id:” match the tail. For instance:

```
keyctl search @s asymmetric id:5acc2142
```

   will match a key with fingerprint:

```
1A00 2040 7601 7889 DE11  882C 3823 04AD 5ACC 2142
```

2) If the criterion string is of the form "<subtype>:<hexdigits>" then the match will match the ID as in (1), but with the added restriction that only keys of the specified subtype (e.g. tpm) will be matched. For instance:

```
keyctl search @s asymmetric tpm:5acc2142
```

Looking in /proc/keys, the last 8 hex digits of the key fingerprint are displayed, along with the subtype:

```
1a39e171 I-----     1 perm 3f010000     0     0 asymmetric modsign.0: DSA␣
↪5acc2142 []
```

5.3 Accessing Asymmetric Keys

For general access to asymmetric keys from within the kernel, the following inclusion is required:

```
#include <crypto/public_key.h>
```

This gives access to functions for dealing with asymmetric / public keys. Three enums are defined there for representing public-key cryptography algorithms:

```
enum pkey_algo
```

digest algorithms used by those:

```
enum pkey_hash_algo
```

and key identifier representations:

```
enum pkey_id_type
```

Note that the key type representation types are required because key identifiers from different standards aren't necessarily compatible. For instance, PGP generates key identifiers by hashing the key data plus some PGP-specific metadata, whereas X.509 has arbitrary certificate identifiers.

The operations defined upon a key are:

1) Signature verification.

Other operations are possible (such as encryption) with the same key data required for verification, but not currently supported, and others (eg. decryption and signature generation) require extra key data.

5.3.1 Signature Verification

An operation is provided to perform cryptographic signature verification, using an asymmetric key to provide or to provide access to the public key:

```
int verify_signature(const struct key *key,
                     const struct public_key_signature *sig);
```

The caller must have already obtained the key from some source and can then use it to check the signature. The caller must have parsed the signature and transferred the relevant bits to the structure pointed to by sig:

```
struct public_key_signature {
        u8 *digest;
        u8 digest_size;
        enum pkey_hash_algo pkey_hash_algo : 8;
        u8 nr_mpi;
        union {
                MPI mpi[2];
                ...
        };
};
```

The algorithm used must be noted in sig->pkey_hash_algo, and all the MPIs that make up the actual signature must be stored in sig->mpi[] and the count of MPIs placed in sig->nr_mpi.

In addition, the data must have been digested by the caller and the resulting hash must be pointed to by sig->digest and the size of the hash be placed in sig->digest_size.

The function will return 0 upon success or -EKEYREJECTED if the signature doesn't match.

The function may also return -ENOTSUPP if an unsupported public-key algorithm or public-key/hash algorithm combination is specified or the key doesn't support the operation; -EBADMSG or -ERANGE if some of the parameters have weird data; or -ENOMEM if an allocation can't be performed. -EINVAL can be returned if the key argument is the wrong type or is incompletely set up.

5.4 Asymmetric Key Subtypes

Asymmetric keys have a subtype that defines the set of operations that can be performed on that key and that determines what data is attached as the key payload. The payload format is entirely at the whim of the subtype.

The subtype is selected by the key data parser and the parser must initialise the data required for it. The asymmetric key retains a reference on the subtype module.

The subtype definition structure can be found in:

```
#include <keys/asymmetric-subtype.h>
```

and looks like the following:

```
struct asymmetric_key_subtype {
        struct module           *owner;
        const char              *name;

        void (*describe)(const struct key *key, struct seq_file *m);
        void (*destroy)(void *payload);
        int (*query)(const struct kernel_pkey_params *params,
                     struct kernel_pkey_query *info);
        int (*eds_op)(struct kernel_pkey_params *params,
                      const void *in, void *out);
        int (*verify_signature)(const struct key *key,
                                const struct public_key_signature *sig);
};
```

Asymmetric keys point to this with their payload[asym_subtype] member.

The owner and name fields should be set to the owning module and the name of the subtype. Currently, the name is only used for print statements.

There are a number of operations defined by the subtype:

1) describe().

   Mandatory. This allows the subtype to display something in /proc/keys against the key. For instance the name of the public key algorithm type could be displayed. The key type will display the tail of the key identity string after this.

2) destroy().

   Mandatory. This should free the memory associated with the key. The asymmetric key will look after freeing the fingerprint and releasing the reference on the subtype module.

3) query().

   Mandatory. This is a function for querying the capabilities of a key.

4) eds_op().

   Optional. This is the entry point for the encryption, decryption and signature creation operations (which are distinguished by the operation ID in the parameter struct). The subtype may do anything it likes to implement an operation, including offloading to hardware.

5) verify_signature().

   Optional. This is the entry point for signature verification. The subtype may do anything it likes to implement an operation, including offloading to hardware.

5.5 Instantiation Data Parsers

The asymmetric key type doesn't generally want to store or to deal with a raw blob of data that holds the key data. It would have to parse it and error check it each time it wanted to use it. Further, the contents of the blob may have various checks that can be performed on it (eg. self-signatures, validity dates) and may contain useful data about the key (identifiers, capabilities).

Also, the blob may represent a pointer to some hardware containing the key rather than the key itself.

Examples of blob formats for which parsers could be implemented include:

- OpenPGP packet stream [RFC 4880].
- X.509 ASN.1 stream.
- Pointer to TPM key.
- Pointer to UEFI key.
- PKCS#8 private key [RFC 5208].
- PKCS#5 encrypted private key [RFC 2898].

During key instantiation each parser in the list is tried until one doesn't return -EBADMSG.

The parser definition structure can be found in:

```
#include <keys/asymmetric-parser.h>
```

and looks like the following:

```
struct asymmetric_key_parser {
        struct module   *owner;
        const char      *name;

        int (*parse)(struct key_preparsed_payload *prep);
};
```

The owner and name fields should be set to the owning module and the name of the parser.

There is currently only a single operation defined by the parser, and it is mandatory:

1) parse().

   This is called to preparse the key from the key creation and update paths. In particular, it is called during the key creation _before_ a key is allocated, and as such, is permitted to provide the key's description in the case that the caller declines to do so.

   The caller passes a pointer to the following struct with all of the fields cleared, except for data, datalen and quotalen [see Documentation/security/keys/core.rst]:

   ```
   struct key_preparsed_payload {
           char            *description;
           void            *payload[4];
           const void      *data;
           size_t          datalen;
   ```

```
        size_t          quotalen;
};
```

The instantiation data is in a blob pointed to by data and is datalen in size. The parse() function is not permitted to change these two values at all, and shouldn't change any of the other values _unless_ they are recognise the blob format and will not return -EBADMSG to indicate it is not theirs.

If the parser is happy with the blob, it should propose a description for the key and attach it to ->description, ->payload[asym_subtype] should be set to point to the subtype to be used, ->payload[asym_crypto] should be set to point to the initialised data for that subtype, ->payload[asym_key_ids] should point to one or more hex fingerprints and quotalen should be updated to indicate how much quota this key should account for.

When clearing up, the data attached to ->payload[asym_key_ids] and ->description will be kfree()'d and the data attached to ->payload[asm_crypto] will be passed to the subtype's ->destroy() method to be disposed of. A module reference for the subtype pointed to by ->payload[asym_subtype] will be put.

If the data format is not recognised, -EBADMSG should be returned. If it is recognised, but the key cannot for some reason be set up, some other negative error code should be returned. On success, 0 should be returned.

The key's fingerprint string may be partially matched upon. For a public-key algorithm such as RSA and DSA this will likely be a printable hex version of the key's fingerprint.

Functions are provided to register and unregister parsers:

```
int register_asymmetric_key_parser(struct asymmetric_key_parser *parser);
void unregister_asymmetric_key_parser(struct asymmetric_key_parser *subtype);
```

Parsers may not have the same name. The names are otherwise only used for displaying in debugging messages.

5.6 Keyring Link Restrictions

Keyrings created from userspace using add_key can be configured to check the signature of the key being linked. Keys without a valid signature are not allowed to link.

Several restriction methods are available:

1) Restrict using the kernel builtin trusted keyring

   - Option string used with KEYCTL_RESTRICT_KEYRING: - "builtin_trusted"

   The kernel builtin trusted keyring will be searched for the signing key. If the builtin trusted keyring is not configured, all links will be rejected. The ca_keys kernel parameter also affects which keys are used for signature verification.

2) Restrict using the kernel builtin and secondary trusted keyrings

   - Option string used with KEYCTL_RESTRICT_KEYRING: - "builtin_and_secondary_trusted"

   The kernel builtin and secondary trusted keyrings will be searched for the signing key. If the secondary trusted keyring is not configured, this restriction will behave like the

"builtin_trusted" option. The ca_keys kernel parameter also affects which keys are used for signature verification.

3) Restrict using a separate key or keyring

   - Option string used with KEYCTL_RESTRICT_KEYRING: - "key_or_keyring:<key or keyring serial number>[:chain]"

Whenever a key link is requested, the link will only succeed if the key being linked is signed by one of the designated keys. This key may be specified directly by providing a serial number for one asymmetric key, or a group of keys may be searched for the signing key by providing the serial number for a keyring.

When the "chain" option is provided at the end of the string, the keys within the destination keyring will also be searched for signing keys. This allows for verification of certificate chains by adding each certificate in order (starting closest to the root) to a keyring. For instance, one keyring can be populated with links to a set of root certificates, with a separate, restricted keyring set up for each certificate chain to be validated:

```
# Create and populate a keyring for root certificates
root_id=`keyctl add keyring root-certs "" @s`
keyctl padd asymmetric "" $root_id < root1.cert
keyctl padd asymmetric "" $root_id < root2.cert

# Create and restrict a keyring for the certificate chain
chain_id=`keyctl add keyring chain "" @s`
keyctl restrict_keyring $chain_id asymmetric key_or_keyring:$root_id:chain

# Attempt to add each certificate in the chain, starting with the
# certificate closest to the root.
keyctl padd asymmetric "" $chain_id < intermediateA.cert
keyctl padd asymmetric "" $chain_id < intermediateB.cert
keyctl padd asymmetric "" $chain_id < end-entity.cert
```

If the final end-entity certificate is successfully added to the "chain" keyring, we can be certain that it has a valid signing chain going back to one of the root certificates.

A single keyring can be used to verify a chain of signatures by restricting the keyring after linking the root certificate:

```
# Create a keyring for the certificate chain and add the root
chain2_id=`keyctl add keyring chain2 "" @s`
keyctl padd asymmetric "" $chain2_id < root1.cert

# Restrict the keyring that already has root1.cert linked.  The cert
# will remain linked by the keyring.
keyctl restrict_keyring $chain2_id asymmetric key_or_keyring:0:chain

# Attempt to add each certificate in the chain, starting with the
# certificate closest to the root.
keyctl padd asymmetric "" $chain2_id < intermediateA.cert
keyctl padd asymmetric "" $chain2_id < intermediateB.cert
keyctl padd asymmetric "" $chain2_id < end-entity.cert
```

If the final end-entity certificate is successfully added to the “chain2” keyring, we can be certain that there is a valid signing chain going back to the root certificate that was added before the keyring was restricted.

In all of these cases, if the signing key is found the signature of the key to be linked will be verified using the signing key. The requested key is added to the keyring only if the signature is successfully verified. -ENOKEY is returned if the parent certificate could not be found, or -EKEYREJECTED is returned if the signature check fails or the key is blacklisted. Other errors may be returned if the signature check could not be performed.

CHAPTER

SIX

DEVELOPING CIPHER ALGORITHMS

6.1 Registering And Unregistering Transformation

There are three distinct types of registration functions in the Crypto API. One is used to register a generic cryptographic transformation, while the other two are specific to HASH transformations and COMPRESSion. We will discuss the latter two in a separate chapter, here we will only look at the generic ones.

Before discussing the register functions, the data structure to be filled with each, *struct crypto_alg*, must be considered – see below for a description of this data structure.

The generic registration functions can be found in include/linux/crypto.h and their definition can be seen below. The former function registers a single transformation, while the latter works on an array of transformation descriptions. The latter is useful when registering transformations in bulk, for example when a driver implements multiple transformations.

```
int crypto_register_alg(struct crypto_alg *alg);
int crypto_register_algs(struct crypto_alg *algs, int count);
```

The counterparts to those functions are listed below.

```
void crypto_unregister_alg(struct crypto_alg *alg);
void crypto_unregister_algs(struct crypto_alg *algs, int count);
```

The registration functions return 0 on success, or a negative errno value on failure. crypto_register_algs() succeeds only if it successfully registered all the given algorithms; if it fails partway through, then any changes are rolled back.

The unregistration functions always succeed, so they don't have a return value. Don't try to unregister algorithms that aren't currently registered.

6.2 Single-Block Symmetric Ciphers [CIPHER]

Example of transformations: aes, serpent, ...

This section describes the simplest of all transformation implementations, that being the CIPHER type used for symmetric ciphers. The CIPHER type is used for transformations which operate on exactly one block at a time and there are no dependencies between blocks at all.

6.2.1 Registration specifics

The registration of [CIPHER] algorithm is specific in that *struct crypto_alg* field .cra_type is empty. The .cra_u.cipher has to be filled in with proper callbacks to implement this transformation.

See *struct cipher_alg* below.

6.2.2 Cipher Definition With struct cipher_alg

Struct cipher_alg defines a single block cipher.

Here are schematics of how these functions are called when operated from other part of the kernel. Note that the .cia_setkey() call might happen before or after any of these schematics happen, but must not happen during any of these are in-flight.

```
KEY ---.    PLAINTEXT ---.
       v                 v
 .cia_setkey() -> .cia_encrypt()
                          |
                          '-----> CIPHERTEXT
```

Please note that a pattern where .cia_setkey() is called multiple times is also valid:

```
KEY1 --.    PLAINTEXT1 --.         KEY2 --.    PLAINTEXT2 --.
       v                 v                v                v
 .cia_setkey() -> .cia_encrypt() -> .cia_setkey() -> .cia_encrypt()
                          |                                  |
                          '---> CIPHERTEXT1                  '---> CIPHERTEXT2
```

6.3 Multi-Block Ciphers

Example of transformations: cbc(aes), chacha20, ...

This section describes the multi-block cipher transformation implementations. The multi-block ciphers are used for transformations which operate on scatterlists of data supplied to the transformation functions. They output the result into a scatterlist of data as well.

6.3.1 Registration Specifics

The registration of multi-block cipher algorithms is one of the most standard procedures throughout the crypto API.

Note, if a cipher implementation requires a proper alignment of data, the caller should use the functions of crypto_skcipher_alignmask() to identify a memory alignment mask. The kernel crypto API is able to process requests that are unaligned. This implies, however, additional overhead as the kernel crypto API needs to perform the realignment of the data which may imply moving of data.

6.3.2 Cipher Definition With struct skcipher_alg

Struct skcipher_alg defines a multi-block cipher, or more generally, a length-preserving symmetric cipher algorithm.

6.3.3 Scatterlist handling

Some drivers will want to use the Generic ScatterWalk in case the hardware needs to be fed separate chunks of the scatterlist which contains the plaintext and will contain the ciphertext. Please refer to the ScatterWalk interface offered by the Linux kernel scatter / gather list implementation.

6.4 Hashing [HASH]

Example of transformations: crc32, md5, sha1, sha256,...

6.4.1 Registering And Unregistering The Transformation

There are multiple ways to register a HASH transformation, depending on whether the transformation is synchronous [SHASH] or asynchronous [AHASH] and the amount of HASH transformations we are registering. You can find the prototypes defined in include/crypto/internal/hash.h:

```
int crypto_register_ahash(struct ahash_alg *alg);

int crypto_register_shash(struct shash_alg *alg);
int crypto_register_shashes(struct shash_alg *algs, int count);
```

The respective counterparts for unregistering the HASH transformation are as follows:

```
void crypto_unregister_ahash(struct ahash_alg *alg);

void crypto_unregister_shash(struct shash_alg *alg);
void crypto_unregister_shashes(struct shash_alg *algs, int count);
```

6.4.2 Cipher Definition With struct shash_alg and ahash_alg

Here are schematics of how these functions are called when operated from other part of the kernel. Note that the .setkey() call might happen before or after any of these schematics happen, but must not happen during any of these are in-flight. Please note that calling .init() followed immediately by .finish() is also a perfectly valid transformation.

```
I)   DATA ------------.
                      v
      .init() -> .update() -> .final()      ! .update() might not be called
                  ^    |         |            at all in this scenario.
                  '----'         '---> HASH


```

```
II)  DATA -----------.-----------.
                     v           v
      .init() -> .update() -> .finup()      ! .update() may not be called
                  ^    |         |            at all in this scenario.
                  '----'         '---> HASH

III) DATA -----------.
                     v
                  .digest()                 ! The entire process is handled
                     |                        by the .digest() call.
                     '---------------> HASH
```

Here is a schematic of how the .export()/.import() functions are called when used from another part of the kernel.

```
KEY--.                 DATA--.
     v                       v                 ! .update() may not be called
 .setkey() -> .init() -> .update() -> .export()   at all in this scenario.
                          ^     |         |
                          '-----'         '--> PARTIAL_HASH

----------- other transformations happen here -----------

PARTIAL_HASH--.   DATA1--.
              v          v
          .import -> .update() -> .final()     ! .update() may not be called
                      ^    |         |           at all in this scenario.
                      '----'         '--> HASH1

PARTIAL_HASH--.   DATA2-.
              v         v
          .import -> .finup()
                       |
                       '---------------> HASH2
```

Note that it is perfectly legal to “abandon” a request object: - call .init() and then (as many times) .update() - _not_ call any of .final(), .finup() or .export() at any point in future

In other words implementations should mind the resource allocation and clean-up. No resources related to request objects should remain allocated after a call to .init() or .update(), since there might be no chance to free them.

6.4.3 Specifics Of Asynchronous HASH Transformation

Some of the drivers will want to use the Generic ScatterWalk in case the implementation needs to be fed separate chunks of the scatterlist which contains the input data. The buffer containing the resulting hash will always be properly aligned to .cra_alignmask so there is no need to worry about this.

CHAPTER

SEVEN

USER SPACE INTERFACE

7.1 Introduction

The concepts of the kernel crypto API visible to kernel space is fully applicable to the user space interface as well. Therefore, the kernel crypto API high level discussion for the in-kernel use cases applies here as well.

The major difference, however, is that user space can only act as a consumer and never as a provider of a transformation or cipher algorithm.

The following covers the user space interface exported by the kernel crypto API. A working example of this description is libkcapi that can be obtained from [1]. That library can be used by user space applications that require cryptographic services from the kernel.

Some details of the in-kernel kernel crypto API aspects do not apply to user space, however. This includes the difference between synchronous and asynchronous invocations. The user space API call is fully synchronous.

[1] https://www.chronox.de/libkcapi.html

7.2 User Space API General Remarks

The kernel crypto API is accessible from user space. Currently, the following ciphers are accessible:

- Message digest including keyed message digest (HMAC, CMAC)
- Symmetric ciphers
- AEAD ciphers
- Random Number Generators

The interface is provided via socket type using the type AF_ALG. In addition, the setsockopt option type is SOL_ALG. In case the user space header files do not export these flags yet, use the following macros:

```
#ifndef AF_ALG
#define AF_ALG 38
#endif
#ifndef SOL_ALG
#define SOL_ALG 279
#endif
```

A cipher is accessed with the same name as done for the in-kernel API calls. This includes the generic vs. unique naming schema for ciphers as well as the enforcement of priorities for generic names.

To interact with the kernel crypto API, a socket must be created by the user space application. User space invokes the cipher operation with the send()/write() system call family. The result of the cipher operation is obtained with the read()/recv() system call family.

The following API calls assume that the socket descriptor is already opened by the user space application and discusses only the kernel crypto API specific invocations.

To initialize the socket interface, the following sequence has to be performed by the consumer:

1. Create a socket of type AF_ALG with the struct sockaddr_alg parameter specified below for the different cipher types.
2. Invoke bind with the socket descriptor
3. Invoke accept with the socket descriptor. The accept system call returns a new file descriptor that is to be used to interact with the particular cipher instance. When invoking send/write or recv/read system calls to send data to the kernel or obtain data from the kernel, the file descriptor returned by accept must be used.

7.3 In-place Cipher operation

Just like the in-kernel operation of the kernel crypto API, the user space interface allows the cipher operation in-place. That means that the input buffer used for the send/write system call and the output buffer used by the read/recv system call may be one and the same. This is of particular interest for symmetric cipher operations where a copying of the output data to its final destination can be avoided.

If a consumer on the other hand wants to maintain the plaintext and the ciphertext in different memory locations, all a consumer needs to do is to provide different memory pointers for the encryption and decryption operation.

7.4 Message Digest API

The message digest type to be used for the cipher operation is selected when invoking the bind syscall. bind requires the caller to provide a filled struct sockaddr data structure. This data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "hash", /* this selects the hash logic in the kernel */
    .salg_name = "sha1" /* this is the cipher name */
};
```

The salg_type value “hash” applies to message digests and keyed message digests. Though, a keyed message digest is referenced by the appropriate salg_name. Please see below for the setsockopt interface that explains how the key can be set for a keyed message digest.

Using the send() system call, the application provides the data that should be processed with the message digest. The send system call allows the following flags to be specified:

- MSG_MORE: If this flag is set, the send system call acts like a message digest update function where the final hash is not yet calculated. If the flag is not set, the send system call calculates the final message digest immediately.

With the recv() system call, the application can read the message digest from the kernel crypto API. If the buffer is too small for the message digest, the flag MSG_TRUNC is set by the kernel.

In order to set a message digest key, the calling application must use the setsockopt() option of ALG_SET_KEY. If the key is not set the HMAC operation is performed without the initial HMAC state change caused by the key.

7.5 Symmetric Cipher API

The operation is very similar to the message digest discussion. During initialization, the struct sockaddr data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "skcipher", /* this selects the symmetric cipher */
    .salg_name = "cbc(aes)" /* this is the cipher name */
};
```

Before data can be sent to the kernel using the write/send system call family, the consumer must set the key. The key setting is described with the setsockopt invocation below.

Using the sendmsg() system call, the application provides the data that should be processed for encryption or decryption. In addition, the IV is specified with the data structure provided by the sendmsg() system call.

The sendmsg system call parameter of struct msghdr is embedded into the struct cmsghdr data structure. See recv(2) and cmsg(3) for more information on how the cmsghdr data structure is used together with the send/recv system call family. That cmsghdr data structure holds the following information specified with a separate header instances:

- specification of the cipher operation type with one of these flags:
  - ALG_OP_ENCRYPT - encryption of data
  - ALG_OP_DECRYPT - decryption of data
- specification of the IV information marked with the flag ALG_SET_IV

The send system call family allows the following flag to be specified:

- MSG_MORE: If this flag is set, the send system call acts like a cipher update function where more input data is expected with a subsequent invocation of the send system call.

Note: The kernel reports -EINVAL for any unexpected data. The caller must make sure that all data matches the constraints given in /proc/crypto for the selected cipher.

With the recv() system call, the application can read the result of the cipher operation from the kernel crypto API. The output buffer must be at least as large as to hold all blocks of the encrypted or decrypted data. If the output data size is smaller, only as many blocks are returned that fit into that output buffer size.

7.6 AEAD Cipher API

The operation is very similar to the symmetric cipher discussion. During initialization, the struct sockaddr data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "aead", /* this selects the symmetric cipher */
    .salg_name = "gcm(aes)" /* this is the cipher name */
};
```

Before data can be sent to the kernel using the write/send system call family, the consumer must set the key. The key setting is described with the setsockopt invocation below.

In addition, before data can be sent to the kernel using the write/send system call family, the consumer must set the authentication tag size. To set the authentication tag size, the caller must use the setsockopt invocation described below.

Using the sendmsg() system call, the application provides the data that should be processed for encryption or decryption. In addition, the IV is specified with the data structure provided by the sendmsg() system call.

The sendmsg system call parameter of struct msghdr is embedded into the struct cmsghdr data structure. See recv(2) and cmsg(3) for more information on how the cmsghdr data structure is used together with the send/recv system call family. That cmsghdr data structure holds the following information specified with a separate header instances:

- specification of the cipher operation type with one of these flags:
  - ALG_OP_ENCRYPT - encryption of data
  - ALG_OP_DECRYPT - decryption of data
- specification of the IV information marked with the flag ALG_SET_IV
- specification of the associated authentication data (AAD) with the flag ALG_SET_AEAD_ASSOCLEN. The AAD is sent to the kernel together with the plaintext / ciphertext. See below for the memory structure.

The send system call family allows the following flag to be specified:

- MSG_MORE: If this flag is set, the send system call acts like a cipher update function where more input data is expected with a subsequent invocation of the send system call.

Note: The kernel reports -EINVAL for any unexpected data. The caller must make sure that all data matches the constraints given in /proc/crypto for the selected cipher.

With the recv() system call, the application can read the result of the cipher operation from the kernel crypto API. The output buffer must be at least as large as defined with the memory structure below. If the output data size is smaller, the cipher operation is not performed.

The authenticated decryption operation may indicate an integrity error. Such breach in integrity is marked with the -EBADMSG error code.

7.6.1 AEAD Memory Structure

The AEAD cipher operates with the following information that is communicated between user and kernel space as one data stream:

- plaintext or ciphertext
- associated authentication data (AAD)
- authentication tag

The sizes of the AAD and the authentication tag are provided with the sendmsg and setsockopt calls (see there). As the kernel knows the size of the entire data stream, the kernel is now able to calculate the right offsets of the data components in the data stream.

The user space caller must arrange the aforementioned information in the following order:

- AEAD encryption input: AAD || plaintext
- AEAD decryption input: AAD || ciphertext || authentication tag

The output buffer the user space caller provides must be at least as large to hold the following data:

- AEAD encryption output: ciphertext || authentication tag
- AEAD decryption output: plaintext

7.7 Random Number Generator API

Again, the operation is very similar to the other APIs. During initialization, the struct sockaddr data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "rng", /* this selects the random number generator */
    .salg_name = "drbg_nopr_sha256" /* this is the RNG name */
};
```

Depending on the RNG type, the RNG must be seeded. The seed is provided using the setsockopt interface to set the key. For example, the ansi_cprng requires a seed. The DRBGs do not require a seed, but may be seeded. The seed is also known as a *Personalization String* in NIST SP 800-90A standard.

Using the read()/recvmsg() system calls, random numbers can be obtained. The kernel generates at most 128 bytes in one call. If user space requires more data, multiple calls to read()/recvmsg() must be made.

WARNING: The user space caller may invoke the initially mentioned accept system call multiple times. In this case, the returned file descriptors have the same state.

Following CAVP testing interfaces are enabled when kernel is built with CRYPTO_USER_API_RNG_CAVP option:

- the concatenation of *Entropy* and *Nonce* can be provided to the RNG via ALG_SET_DRBG_ENTROPY setsockopt interface. Setting the entropy requires CAP_SYS_ADMIN permission.

- *Additional Data* can be provided using the send()/sendmsg() system calls, but only after the entropy has been set.

7.8 Zero-Copy Interface

In addition to the send/write/read/recv system call family, the AF_ALG interface can be accessed with the zero-copy interface of splice/vmsplice. As the name indicates, the kernel tries to avoid a copy operation into kernel space.

The zero-copy operation requires data to be aligned at the page boundary. Non-aligned data can be used as well, but may require more operations of the kernel which would defeat the speed gains obtained from the zero-copy interface.

The system-inherent limit for the size of one zero-copy operation is 16 pages. If more data is to be sent to AF_ALG, user space must slice the input into segments with a maximum size of 16 pages.

Zero-copy can be used with the following code example (a complete working example is provided with libkcapi):

```
int pipes[2];

pipe(pipes);
/* input data in iov */
vmsplice(pipes[1], iov, iovlen, SPLICE_F_GIFT);
/* opfd is the file descriptor returned from accept() system call */
splice(pipes[0], NULL, opfd, NULL, ret, 0);
read(opfd, out, outlen);
```

7.9 Setsockopt Interface

In addition to the read/recv and send/write system call handling to send and retrieve data subject to the cipher operation, a consumer also needs to set the additional information for the cipher operation. This additional information is set using the setsockopt system call that must be invoked with the file descriptor of the open cipher (i.e. the file descriptor returned by the accept system call).

Each setsockopt invocation must use the level SOL_ALG.

The setsockopt interface allows setting the following data using the mentioned optname:

- ALG_SET_KEY – Setting the key. Key setting is applicable to:
  - the skcipher cipher type (symmetric ciphers)
  - the hash cipher type (keyed message digests)
  - the AEAD cipher type
  - the RNG cipher type to provide the seed
- ALG_SET_AEAD_AUTHSIZE – Setting the authentication tag size for AEAD ciphers. For a encryption operation, the authentication tag of the given size will be generated. For a

decryption operation, the provided ciphertext is assumed to contain an authentication tag of the given size (see section about AEAD memory layout below).

- ALG_SET_DRBG_ENTROPY – Setting the entropy of the random number generator. This option is applicable to RNG cipher type only.

7.10 User space API example

Please see [1] for libkcapi which provides an easy-to-use wrapper around the aforementioned Netlink kernel interface. [1] also contains a test application that invokes all libkcapi API calls.

[1] https://www.chronox.de/libkcapi.html

CHAPTER

EIGHT

CRYPTO ENGINE

8.1 Overview

The crypto engine (CE) API is a crypto queue manager.

8.2 Requirement

You must put, at the start of your transform context your_tfm_ctx, the structure crypto_engine:

```
struct your_tfm_ctx {
        struct crypto_engine engine;
        ...
};
```

The crypto engine only manages asynchronous requests in the form of crypto_async_request. It cannot know the underlying request type and thus only has access to the transform structure. It is not possible to access the context using container_of. In addition, the engine knows nothing about your structure "`struct your_tfm_ctx`". The engine assumes (requires) the placement of the known member `struct crypto_engine` at the beginning.

8.3 Order of operations

You are required to obtain a struct crypto_engine via `crypto_engine_alloc_init()`. Start it via `crypto_engine_start()`. When finished with your work, shut down the engine using `crypto_engine_stop()` and destroy the engine with `crypto_engine_exit()`.

Before transferring any request, you have to fill the context enginectx by providing functions for the following:

- `prepare_crypt_hardware`: Called once before any prepare functions are called.
- `unprepare_crypt_hardware`: Called once after all unprepare functions have been called.
- `prepare_cipher_request`/`prepare_hash_request`: Called before each corresponding request is performed. If some processing or other preparatory work is required, do it here.
- `unprepare_cipher_request`/`unprepare_hash_request`: Called after each request is handled. Clean up / undo what was done in the prepare function.

- `cipher_one_request/hash_one_request`: Handle the current request by performing the operation.

Note that these functions access the crypto_async_request structure associated with the received request. You are able to retrieve the original request by using:

```
container_of(areq, struct yourrequesttype_request, base);
```

When your driver receives a crypto_request, you must to transfer it to the crypto engine via one of:

- crypto_transfer_aead_request_to_engine()
- crypto_transfer_akcipher_request_to_engine()
- crypto_transfer_hash_request_to_engine()
- crypto_transfer_kpp_request_to_engine()
- crypto_transfer_skcipher_request_to_engine()

At the end of the request process, a call to one of the following functions is needed:

- crypto_finalize_aead_request()
- crypto_finalize_akcipher_request()
- crypto_finalize_hash_request()
- crypto_finalize_kpp_request()
- crypto_finalize_skcipher_request()

CHAPTER

NINE

PROGRAMMING INTERFACE

Table of contents

9.1 Block Cipher Algorithm Definitions

These data structures define modular crypto algorithm implementations, managed via crypto_register_alg() and crypto_unregister_alg().

struct **cipher_alg**
: single-block symmetric ciphers definition

Definition

```
struct cipher_alg {
  unsigned int cia_min_keysize;
  unsigned int cia_max_keysize;
  int (*cia_setkey)(struct crypto_tfm *tfm, const u8 *key, unsigned int␣
↪keylen);
  void (*cia_encrypt)(struct crypto_tfm *tfm, u8 *dst, const u8 *src);
  void (*cia_decrypt)(struct crypto_tfm *tfm, u8 *dst, const u8 *src);
};
```

Members

cia_min_keysize Minimum key size supported by the transformation. This is the smallest key length supported by this transformation algorithm. This must be set to one of the pre-defined values as this is not hardware specific. Possible values for this field can be found via git grep “_MIN_KEY_SIZE” include/crypto/

cia_max_keysize Maximum key size supported by the transformation. This is the largest key length supported by this transformation algorithm. This must be set to one of the pre-defined values as this is not hardware specific. Possible values for this field can be found via git grep “_MAX_KEY_SIZE” include/crypto/

cia_setkey Set key for the transformation. This function is used to either program a supplied key into the hardware or store the key in the transformation context for programming it later. Note that this function does modify the transformation context. This function can be called multiple times during the existence of the transformation object, so one must make sure the key is properly reprogrammed into the hardware. This function is also responsible for checking the key length for validity.

cia_encrypt Encrypt a single block. This function is used to encrypt a single block of data, which must be **cra_blocksize** big. This always operates on a full **cra_blocksize** and it is not possible to encrypt a block of smaller size. The supplied buffers must therefore also be at least of **cra_blocksize** size. Both the input and output buffers are always aligned to **cra_alignmask**. In case either of the input or output buffer supplied by user of the crypto API is not aligned to **cra_alignmask**, the crypto API will re-align the buffers. The re-alignment means that a new buffer will be allocated, the data will be copied into the new buffer, then the processing will happen on the new buffer, then the data will be copied back into the original buffer and finally the new buffer will be freed. In case a software fallback was put in place in the **cra_init** call, this function might need to use the fallback if the algorithm doesn't support all of the key sizes. In case the key was stored in transformation context, the key might need to be re-programmed into the hardware in this function. This function shall not modify the transformation context, as this function may be called in parallel with the same transformation object.

cia_decrypt Decrypt a single block. This is a reverse counterpart to **cia_encrypt**, and the conditions are exactly the same.

Description

All fields are mandatory and must be filled.

struct **compress_alg**
compression/decompression algorithm

Definition

```
struct compress_alg {
  int (*coa_compress)(struct crypto_tfm *tfm, const u8 *src, unsigned int slen,
 ↪ u8 *dst, unsigned int *dlen);
  int (*coa_decompress)(struct crypto_tfm *tfm, const u8 *src, unsigned int␣
 ↪slen, u8 *dst, unsigned int *dlen);
};
```

Members

coa_compress Compress a buffer of specified length, storing the resulting data in the specified buffer. Return the length of the compressed data in dlen.

coa_decompress Decompress the source buffer, storing the uncompressed data in the specified buffer. The length of the data is returned in dlen.

Description

All fields are mandatory.

struct **crypto_alg**
definition of a cryptograpic cipher algorithm

Definition

```
struct crypto_alg {
  struct list_head cra_list;
  struct list_head cra_users;
  u32 cra_flags;
  unsigned int cra_blocksize;
  unsigned int cra_ctxsize;
```

```
  unsigned int cra_alignmask;
  int cra_priority;
  refcount_t cra_refcnt;
  char cra_name[CRYPTO_MAX_ALG_NAME];
  char cra_driver_name[CRYPTO_MAX_ALG_NAME];
  const struct crypto_type *cra_type;
  union {
    struct cipher_alg cipher;
    struct compress_alg compress;
  } cra_u;
  int (*cra_init)(struct crypto_tfm *tfm);
  void (*cra_exit)(struct crypto_tfm *tfm);
  void (*cra_destroy)(struct crypto_alg *alg);
  struct module *cra_module;
#ifdef CONFIG_CRYPTO_STATS;
  union {
    struct crypto_istat_aead aead;
    struct crypto_istat_akcipher akcipher;
    struct crypto_istat_cipher cipher;
    struct crypto_istat_compress compress;
    struct crypto_istat_hash hash;
    struct crypto_istat_rng rng;
    struct crypto_istat_kpp kpp;
  } stats;
#endif ;
};
```

Members

cra_list internally used

cra_users internally used

cra_flags Flags describing this transformation. See include/linux/crypto.h CRYPTO_ALG_* flags for the flags which go in here. Those are used for fine-tuning the description of the transformation algorithm.

cra_blocksize Minimum block size of this transformation. The size in bytes of the smallest possible unit which can be transformed with this algorithm. The users must respect this value. In case of HASH transformation, it is possible for a smaller block than **cra_blocksize** to be passed to the crypto API for transformation, in case of any other transformation type, an error will be returned upon any attempt to transform smaller than **cra_blocksize** chunks.

cra_ctxsize Size of the operational context of the transformation. This value informs the kernel crypto API about the memory size needed to be allocated for the transformation context.

cra_alignmask Alignment mask for the input and output data buffer. The data buffer containing the input data for the algorithm must be aligned to this alignment mask. The data buffer for the output data must be aligned to this alignment mask. Note that the Crypto API will do the re-alignment in software, but only under special conditions and there is a performance hit. The re-alignment happens at these occasions for different **cra_u** types: cipher – For both input data and output data buffer; ahash – For output hash destination

buf; shash – For output hash destination buf. This is needed on hardware which is flawed by design and cannot pick data from arbitrary addresses.

cra_priority Priority of this transformation implementation. In case multiple transformations with same **cra_name** are available to the Crypto API, the kernel will use the one with highest **cra_priority**.

cra_refcnt internally used

cra_name Generic name (usable by multiple implementations) of the transformation algorithm. This is the name of the transformation itself. This field is used by the kernel when looking up the providers of particular transformation.

cra_driver_name Unique name of the transformation provider. This is the name of the provider of the transformation. This can be any arbitrary value, but in the usual case, this contains the name of the chip or provider and the name of the transformation algorithm.

cra_type Type of the cryptographic transformation. This is a pointer to struct crypto_type, which implements callbacks common for all transformation types. There are multiple options, such as `crypto_skcipher_type`, `crypto_ahash_type`, `crypto_rng_type`. This field might be empty. In that case, there are no common callbacks. This is the case for: cipher, compress, shash.

cra_u Callbacks implementing the transformation. This is a union of multiple structures. Depending on the type of transformation selected by **cra_type** and **cra_flags** above, the associated structure must be filled with callbacks. This field might be empty. This is the case for ahash, shash.

cra_u.cipher Union member which contains a single-block symmetric cipher definition. See **struct cipher_alg**.

cra_u.compress Union member which contains a (de)compression algorithm. See **struct compress_alg**.

cra_init Initialize the cryptographic transformation object. This function is used to initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

cra_exit Deinitialize the cryptographic transformation object. This is a counterpart to **cra_init**, used to remove various changes set in **cra_init**.

cra_destroy internally used

cra_module Owner of this transformation implementation. Set to THIS_MODULE

stats union of all possible crypto_istat_xxx structures

stats.aead statistics for AEAD algorithm

stats.akcipher statistics for akcipher algorithm

stats.cipher statistics for cipher algorithm

stats.compress statistics for compress algorithm

stats.hash statistics for hash algorithm

stats.rng statistics for rng algorithm

stats.kpp statistics for KPP algorithm

Description

The *struct crypto_alg* describes a generic Crypto API algorithm and is common for all of the transformations. Any variable not documented here shall not be used by a cipher implementation as it is internal to the Crypto API.

9.2 Symmetric Key Cipher API

Symmetric key cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_SKCIPHER (listed as type "skcipher" in /proc/crypto).

Asynchronous cipher operations imply that the function invocation for a cipher request returns immediately before the completion of the operation. The cipher request is scheduled as a separate kernel thread and therefore load-balanced on the different CPUs via the process scheduler. To allow the kernel crypto API to inform the caller about the completion of a cipher request, the caller must provide a callback function. That function is invoked with the cipher handle when the request completes.

To support the asynchronous operation, additional information than just the cipher handle must be supplied to the kernel crypto API. That additional information is given by filling in the skcipher_request data structure.

For the symmetric key cipher API, the state is maintained with the tfm cipher handle. A single tfm can be used across multiple calls and in parallel. For asynchronous block cipher calls, context data supplied and only used by the caller can be referenced the request data structure in addition to the IV used for the cipher request. The maintenance of such state information would be important for a crypto driver implementer to have, because when calling the callback function upon completion of the cipher operation, that callback function may need some information about which operation just finished if it invoked multiple in parallel. This state information is unused by the kernel crypto API.

struct crypto_skcipher ***crypto_alloc_skcipher**(const char *alg_name, u32 type, u32 mask)
allocate symmetric key cipher handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the skcipher cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for an skcipher. The returned struct crypto_skcipher is the cipher handle that is required for any subsequent API invocation for that skcipher.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void **crypto_free_skcipher**(struct crypto_skcipher *tfm)
zeroize and free cipher handle

Parameters

struct crypto_skcipher *tfm cipher handle to be freed

Description

If **tfm** is a NULL or error pointer, this function does nothing.

int **crypto_has_skcipher**(const char *alg_name, u32 type, u32 mask)
Search for the availability of an skcipher.

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the skcipher

u32 type specifies the type of the skcipher

u32 mask specifies the mask for the skcipher

Return

true when the skcipher is known to the kernel crypto API; false otherwise

unsigned int **crypto_skcipher_ivsize**(struct crypto_skcipher *tfm)
obtain IV size

Parameters

struct crypto_skcipher *tfm cipher handle

Description

The size of the IV for the skcipher referenced by the cipher handle is returned. This IV size may be zero if the cipher does not need an IV.

Return

IV size in bytes

unsigned int **crypto_skcipher_blocksize**(struct crypto_skcipher *tfm)
obtain block size of cipher

Parameters

struct crypto_skcipher *tfm cipher handle

Description

The block size for the skcipher referenced with the cipher handle is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **crypto_skcipher_setkey**(struct crypto_skcipher *tfm, const u8 *key, unsigned int keylen)
set key for cipher

Parameters

struct crypto_skcipher *tfm cipher handle

const u8 *key buffer holding the key

unsigned int keylen length of the key in bytes

Description

The caller provided key is set for the skcipher referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

struct crypto_skcipher ***crypto_skcipher_reqtfm**(struct skcipher_request *req)
obtain cipher handle from request

Parameters

struct skcipher_request *req skcipher_request out of which the cipher handle is to be obtained

Description

Return the crypto_skcipher handle when furnishing an skcipher_request data structure.

Return

crypto_skcipher handle

int **crypto_skcipher_encrypt**(struct skcipher_request *req)
encrypt plaintext

Parameters

struct skcipher_request *req reference to the skcipher_request handle that holds all information needed to perform the cipher operation

Description

Encrypt plaintext data using the skcipher_request handle. That data structure and how it is filled with data is discussed with the skcipher_request_* functions.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **crypto_skcipher_decrypt**(struct skcipher_request *req)
decrypt ciphertext

Parameters

struct skcipher_request *req reference to the skcipher_request handle that holds all information needed to perform the cipher operation

Description

Decrypt ciphertext data using the skcipher_request handle. That data structure and how it is filled with data is discussed with the skcipher_request_* functions.

Return

0 if the cipher operation was successful; < 0 if an error occurred

9.3 Symmetric Key Cipher Request Handle

The skcipher_request data structure contains all pointers to data required for the symmetric key cipher operation. This includes the cipher handle (which can be used by multiple skcipher_request instances), pointer to plaintext and ciphertext, asynchronous callback function, etc. It acts as a handle to the skcipher_request_* API calls in a similar way as skcipher handle to the crypto_skcipher_* API calls.

unsigned int **crypto_skcipher_reqsize**(struct crypto_skcipher *tfm)
obtain size of the request data structure

Parameters

struct crypto_skcipher *tfm cipher handle

Return

number of bytes

void **skcipher_request_set_tfm**(struct skcipher_request *req, struct crypto_skcipher *tfm)
update cipher handle reference in request

Parameters

struct skcipher_request *req request handle to be modified

struct crypto_skcipher *tfm cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing skcipher handle in the request data structure with a different one.

struct skcipher_request *__skcipher_request_alloc__(struct crypto_skcipher *tfm, gfp_t gfp)
allocate request data structure

Parameters

struct crypto_skcipher *tfm cipher handle to be registered with the request

gfp_t gfp memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the skcipher encrypt and decrypt API calls. During the allocation, the provided skcipher handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **skcipher_request_free**(struct skcipher_request *req)
zeroize and free request data structure

Parameters

struct skcipher_request *req request data structure cipher handle to be freed

void **skcipher_request_set_callback**(struct skcipher_request *req, u32 flags,
crypto_completion_t compl, void *data)
set asynchronous callback function

Parameters

struct skcipher_request *req request handle

u32 flags specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

crypto_completion_t compl callback function pointer to be registered with the request handle

void *data The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the "data" field in the crypto_async_request data structure provided to the callback function.

Description

This function allows setting the callback function that is triggered once the cipher operation completes.

The callback function is registered with the skcipher_request handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **skcipher_request_set_crypt**(struct skcipher_request *req, struct scatterlist *src, struct scatterlist *dst, unsigned int cryptlen, void *iv)

set data buffers

Parameters

struct skcipher_request *req request handle

struct scatterlist *src source scatter / gather list

struct scatterlist *dst destination scatter / gather list

unsigned int cryptlen number of bytes to process from **src**

void *iv IV for the cipher operation which must comply with the IV size defined by crypto_skcipher_ivsize

Description

This function allows setting of the source data and destination data scatter / gather lists.

For encryption, the source is treated as the plaintext and the destination is the ciphertext. For a decryption operation, the use is reversed - the source is the ciphertext and the destination is the plaintext.

9.4 Single Block Cipher API

The single block cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_CIPHER (listed as type “cipher” in /proc/crypto).

Using the single block cipher API calls, operations with the basic cipher primitive can be implemented. These cipher primitives exclude any block chaining operations including IV handling.

The purpose of this single block cipher API is to support the implementation of templates or other concepts that only need to perform the cipher operation on one block at a time. Templates invoke the underlying cipher primitive block-wise and process either the input or the output data of these cipher operations.

struct crypto_cipher *__crypto_alloc_cipher__(const char *alg_name, u32 type, u32 mask)
allocate single block cipher handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the single block cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for a single block cipher. The returned struct crypto_cipher is the cipher handle that is required for any subsequent API invocation for that single block cipher.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void __crypto_free_cipher__(struct crypto_cipher *tfm)
zeroize and free the single block cipher handle

Parameters

struct crypto_cipher *tfm cipher handle to be freed

int __crypto_has_cipher__(const char *alg_name, u32 type, u32 mask)
Search for the availability of a single block cipher

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the single block cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Return

true when the single block cipher is known to the kernel crypto API; false otherwise

unsigned int __crypto_cipher_blocksize__(struct crypto_cipher *tfm)
obtain block size for cipher

Parameters

struct crypto_cipher *tfm cipher handle

Description

The block size for the single block cipher referenced with the cipher handle tfm is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **crypto_cipher_setkey**(struct crypto_cipher *tfm, const u8 *key, unsigned int keylen)
set key for cipher

Parameters

struct crypto_cipher *tfm cipher handle

const u8 *key buffer holding the key

unsigned int keylen length of the key in bytes

Description

The caller provided key is set for the single block cipher referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

void **crypto_cipher_encrypt_one**(struct crypto_cipher *tfm, u8 *dst, const u8 *src)
encrypt one block of plaintext

Parameters

struct crypto_cipher *tfm cipher handle

u8 *dst points to the buffer that will be filled with the ciphertext

const u8 *src buffer holding the plaintext to be encrypted

Description

Invoke the encryption operation of one block. The caller must ensure that the plaintext and ciphertext buffers are at least one block in size.

void **crypto_cipher_decrypt_one**(struct crypto_cipher *tfm, u8 *dst, const u8 *src)
decrypt one block of ciphertext

Parameters

struct crypto_cipher *tfm cipher handle

u8 *dst points to the buffer that will be filled with the plaintext

const u8 *src buffer holding the ciphertext to be decrypted

Description

Invoke the decryption operation of one block. The caller must ensure that the plaintext and ciphertext buffers are at least one block in size.

9.5 Authenticated Encryption With Associated Data (AEAD) Algorithm Definitions

The AEAD cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_AEAD (listed as type "aead" in /proc/crypto)

The most prominent examples for this type of encryption is GCM and CCM. However, the kernel supports other types of AEAD ciphers which are defined with the following cipher string:

authenc(keyed message digest, block cipher)

For example: authenc(hmac(sha256), cbc(aes))

The example code provided for the symmetric key cipher operation applies here as well. Naturally all *skcipher* symbols must be exchanged the *aead* pendants discussed in the following. In addition, for the AEAD operation, the aead_request_set_ad function must be used to set the pointer to the associated data memory location before performing the encryption or decryption operation. In case of an encryption, the associated data memory is filled during the encryption operation. For decryption, the associated data memory must contain data that is used to verify the integrity of the decrypted data. Another deviation from the asynchronous block cipher operation is that the caller should explicitly check for -EBADMSG of the crypto_aead_decrypt. That error indicates an authentication error, i.e. a breach in the integrity of the message. In essence, that -EBADMSG error code is the key bonus an AEAD cipher has over "standard" block chaining modes.

Memory Structure:

The source scatterlist must contain the concatenation of associated data || plaintext or ciphertext.

The destination scatterlist has the same layout, except that the plaintext (resp. ciphertext) will grow (resp. shrink) by the authentication tag size during encryption (resp. decryption).

In-place encryption/decryption is enabled by using the same scatterlist pointer for both the source and destination.

Even in the out-of-place case, space must be reserved in the destination for the associated data, even though it won't be written to. This makes the in-place and out-of-place cases more consistent. It is permissible for the "destination" associated data to alias the "source" associated data.

As with the other scatterlist crypto APIs, zero-length scatterlist elements are not allowed in the used part of the scatterlist. Thus, if there is no associated data, the first element must point to the plaintext/ciphertext.

To meet the needs of IPsec, a special quirk applies to rfc4106, rfc4309, rfc4543, and rfc7539esp ciphers. For these ciphers, the final 'ivsize' bytes of the associated data buffer must contain a second copy of the IV. This is in addition to the copy passed to *aead_request_set_crypt()*. These two IV copies must not differ; different implementations of the same algorithm may behave differently in that case. Note that the algorithm might not actually treat the IV as associated data; nevertheless the length passed to *aead_request_set_ad()* must include it.

struct **aead_request**
AEAD request

Definition

```
struct aead_request {
  struct crypto_async_request base;
  unsigned int assoclen;
  unsigned int cryptlen;
  u8 *iv;
  struct scatterlist *src;
  struct scatterlist *dst;
  void *__ctx[] ;
};
```

Members

base Common attributes for async crypto requests

assoclen Length in bytes of associated data for authentication

cryptlen Length of data to be encrypted or decrypted

iv Initialisation vector

src Source data

dst Destination data

__ctx Start of private context data

struct **aead_alg**
AEAD cipher definition

Definition

```
struct aead_alg {
  int (*setkey)(struct crypto_aead *tfm, const u8 *key, unsigned int keylen);
  int (*setauthsize)(struct crypto_aead *tfm, unsigned int authsize);
  int (*encrypt)(struct aead_request *req);
  int (*decrypt)(struct aead_request *req);
  int (*init)(struct crypto_aead *tfm);
  void (*exit)(struct crypto_aead *tfm);
  unsigned int ivsize;
  unsigned int maxauthsize;
  unsigned int chunksize;
  struct crypto_alg base;
};
```

Members

setkey see struct skcipher_alg

setauthsize Set authentication size for the AEAD transformation. This function is used to specify the consumer requested size of the authentication tag to be either generated by the transformation during encryption or the size of the authentication tag to be supplied during the decryption operation. This function is also responsible for checking the authentication tag size for validity.

encrypt see struct skcipher_alg

decrypt see struct skcipher_alg

init Initialize the cryptographic transformation object. This function is used to initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

exit Deinitialize the cryptographic transformation object. This is a counterpart to **init**, used to remove various changes set in **init**.

ivsize see struct skcipher_alg

maxauthsize Set the maximum authentication tag size supported by the transformation. A transformation may support smaller tag sizes. As the authentication tag is a message digest to ensure the integrity of the encrypted data, a consumer typically wants the largest authentication tag possible as defined by this variable.

chunksize see struct skcipher_alg

base Definition of a generic crypto cipher algorithm.

Description

All fields except **ivsize** is mandatory and must be filled.

9.6 Authenticated Encryption With Associated Data (AEAD) Cipher API

struct crypto_aead *__crypto_alloc_aead__(const char *alg_name, u32 type, u32 mask)
allocate AEAD cipher handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the AEAD cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for an AEAD. The returned struct crypto_aead is the cipher handle that is required for any subsequent API invocation for that AEAD.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void __crypto_free_aead__(struct crypto_aead *tfm)
zeroize and free aead handle

Parameters

struct crypto_aead *tfm cipher handle to be freed

Description

If **tfm** is a NULL or error pointer, this function does nothing.

unsigned int **crypto_aead_ivsize**(struct crypto_aead *tfm)
obtain IV size

Parameters

struct crypto_aead *tfm cipher handle

Description

The size of the IV for the aead referenced by the cipher handle is returned. This IV size may be zero if the cipher does not need an IV.

Return

IV size in bytes

unsigned int **crypto_aead_authsize**(struct crypto_aead *tfm)
obtain maximum authentication data size

Parameters

struct crypto_aead *tfm cipher handle

Description

The maximum size of the authentication data for the AEAD cipher referenced by the AEAD cipher handle is returned. The authentication data size may be zero if the cipher implements a hard-coded maximum.

The authentication data may also be known as “tag value”.

Return

authentication data size / tag size in bytes

unsigned int **crypto_aead_blocksize**(struct crypto_aead *tfm)
obtain block size of cipher

Parameters

struct crypto_aead *tfm cipher handle

Description

The block size for the AEAD referenced with the cipher handle is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **crypto_aead_setkey**(struct crypto_aead *tfm, const u8 *key, unsigned int keylen)
set key for cipher

Parameters

struct crypto_aead *tfm cipher handle

const u8 *key buffer holding the key

unsigned int keylen length of the key in bytes

Description

The caller provided key is set for the AEAD referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_aead_setauthsize**(struct crypto_aead *tfm, unsigned int authsize)
set authentication data size

Parameters

struct crypto_aead *tfm cipher handle

unsigned int authsize size of the authentication data / tag in bytes

Description

Set the authentication data size / tag size. AEAD requires an authentication tag (or MAC) in addition to the associated data.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_aead_encrypt**(struct *aead_request* *req)
encrypt plaintext

Parameters

struct aead_request *req reference to the aead_request handle that holds all information needed to perform the cipher operation

Description

Encrypt plaintext data using the aead_request handle. That data structure and how it is filled with data is discussed with the aead_request_* functions.

IMPORTANT NOTE The encryption operation creates the authentication data / tag. That data is concatenated with the created ciphertext. The ciphertext memory size is therefore the given number of block cipher blocks + the size defined by the crypto_aead_setauthsize invocation. The caller must ensure that sufficient memory is available for the ciphertext and the authentication tag.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **crypto_aead_decrypt**(struct *aead_request* *req)
decrypt ciphertext

Parameters

struct aead_request *req reference to the aead_request handle that holds all information needed to perform the cipher operation

Description

Decrypt ciphertext data using the aead_request handle. That data structure and how it is filled with data is discussed with the aead_request_* functions.

IMPORTANT NOTE The caller must concatenate the ciphertext followed by the authentication data / tag. That authentication data / tag must have the size defined by the crypto_aead_setauthsize invocation.

Return

0 if the cipher operation was successful; -EBADMSG: The AEAD cipher operation performs the authentication of the data during the decryption operation. Therefore, the function returns this error if the authentication of the ciphertext was unsuccessful (i.e. the integrity of the ciphertext or the associated data was violated); < 0 if an error occurred.

9.7 Asynchronous AEAD Request Handle

The aead_request data structure contains all pointers to data required for the AEAD cipher operation. This includes the cipher handle (which can be used by multiple aead_request instances), pointer to plaintext and ciphertext, asynchronous callback function, etc. It acts as a handle to the aead_request_* API calls in a similar way as AEAD handle to the crypto_aead_* API calls.

unsigned int **crypto_aead_reqsize**(struct crypto_aead *tfm)
obtain size of the request data structure

Parameters

struct crypto_aead *tfm cipher handle

Return

number of bytes

void **aead_request_set_tfm**(struct *aead_request* *req, struct crypto_aead *tfm)
update cipher handle reference in request

Parameters

struct aead_request *req request handle to be modified

struct crypto_aead *tfm cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing aead handle in the request data structure with a different one.

struct *aead_request* ***aead_request_alloc**(struct crypto_aead *tfm, gfp_t gfp)
allocate request data structure

Parameters

struct crypto_aead *tfm cipher handle to be registered with the request

gfp_t gfp memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the AEAD encrypt and decrypt API calls. During the allocation, the provided aead handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **aead_request_free**(struct *aead_request* *req)
zeroize and free request data structure

Parameters

struct aead_request *req request data structure cipher handle to be freed

void **aead_request_set_callback**(struct *aead_request* *req, u32 flags, crypto_completion_t compl, void *data)
set asynchronous callback function

Parameters

struct aead_request *req request handle

u32 flags specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

crypto_completion_t compl callback function pointer to be registered with the request handle

void *data The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the “data” field in the crypto_async_request data structure provided to the callback function.

Description

Setting the callback function that is triggered once the cipher operation completes

The callback function is registered with the aead_request handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **aead_request_set_crypt**(struct *aead_request* *req, struct scatterlist *src, struct scatterlist *dst, unsigned int cryptlen, u8 *iv)
set data buffers

Parameters

struct aead_request *req request handle

struct scatterlist *src source scatter / gather list

struct scatterlist *dst destination scatter / gather list

unsigned int cryptlen number of bytes to process from **src**

u8 *iv IV for the cipher operation which must comply with the IV size defined by *crypto_aead_ivsize()*

Description

Setting the source data and destination data scatter / gather lists which hold the associated data concatenated with the plaintext or ciphertext. See below for the authentication tag.

For encryption, the source is treated as the plaintext and the destination is the ciphertext. For a decryption operation, the use is reversed - the source is the ciphertext and the destination is the plaintext.

The memory structure for cipher operation has the following structure:

- AEAD encryption input: assoc data || plaintext
- AEAD encryption output: assoc data || ciphertext || auth tag
- AEAD decryption input: assoc data || ciphertext || auth tag
- AEAD decryption output: assoc data || plaintext

Albeit the kernel requires the presence of the AAD buffer, however, the kernel does not fill the AAD buffer in the output case. If the caller wants to have that data buffer filled, the caller must either use an in-place cipher operation (i.e. same memory location for input/output memory location).

void **aead_request_set_ad**(struct *aead_request* *req, unsigned int assoclen)
set associated data information

Parameters

struct aead_request *req request handle

unsigned int assoclen number of bytes in associated data

Description

Setting the AD information. This function sets the length of the associated data.

9.8 Message Digest Algorithm Definitions

These data structures define modular message digest algorithm implementations, managed via crypto_register_ahash(), crypto_register_shash(), crypto_unregister_ahash() and crypto_unregister_shash().

struct **hash_alg_common**
define properties of message digest

Definition

```
struct hash_alg_common {
  unsigned int digestsize;
  unsigned int statesize;
  struct crypto_alg base;
};
```

Members

digestsize Size of the result of the transformation. A buffer of this size must be available to the **final** and **finup** calls, so they can store the resulting hash into it. For various predefined sizes, search include/crypto/ using git grep _DIGEST_SIZE include/crypto.

statesize Size of the block for partial state of the transformation. A buffer of this size must be passed to the **export** function as it will save the partial state of the transformation into it. On the other side, the **import** function will load the state from a buffer of this size as well.

base Start of data structure of cipher algorithm. The common data structure of crypto_alg contains information common to all ciphers. The hash_alg_common data structure now adds the hash-specific information.

struct **ahash_alg**
asynchronous message digest definition

Definition

```
struct ahash_alg {
  int (*init)(struct ahash_request *req);
  int (*update)(struct ahash_request *req);
  int (*final)(struct ahash_request *req);
  int (*finup)(struct ahash_request *req);
  int (*digest)(struct ahash_request *req);
  int (*export)(struct ahash_request *req, void *out);
  int (*import)(struct ahash_request *req, const void *in);
  int (*setkey)(struct crypto_ahash *tfm, const u8 *key, unsigned int keylen);
  int (*init_tfm)(struct crypto_ahash *tfm);
  void (*exit_tfm)(struct crypto_ahash *tfm);
  struct hash_alg_common halg;
};
```

Members

init [mandatory] Initialize the transformation context. Intended only to initialize the state of the HASH transformation at the beginning. This shall fill in the internal structures used during the entire duration of the whole transformation. No data processing happens at this point. Driver code implementation must not use req->result.

update [mandatory] Push a chunk of data into the driver for transformation. This function actually pushes blocks of data from upper layers into the driver, which then passes those to the hardware as seen fit. This function must not finalize the HASH transformation by calculating the final message digest as this only adds more data into the transformation. This function shall not modify the transformation context, as this function may be called in parallel with the same transformation object. Data processing can happen synchronously [SHASH] or asynchronously [AHASH] at this point. Driver must not use req->result.

final [mandatory] Retrieve result from the driver. This function finalizes the transformation and retrieves the resulting hash from the driver and pushes it back to upper layers. No data processing happens at this point unless hardware requires it to finish the transformation (then the data buffered by the device driver is processed).

finup [optional] Combination of **update** and **final**. This function is effectively a combination of **update** and **final** calls issued in sequence. As some hardware cannot do **update** and

final separately, this callback was added to allow such hardware to be used at least by IPsec. Data processing can happen synchronously [SHASH] or asynchronously [AHASH] at this point.

digest Combination of **init** and **update** and **final**. This function effectively behaves as the entire chain of operations, **init**, **update** and **final** issued in sequence. Just like **finup**, this was added for hardware which cannot do even the **finup**, but can only do the whole transformation in one run. Data processing can happen synchronously [SHASH] or asynchronously [AHASH] at this point.

export Export partial state of the transformation. This function dumps the entire state of the ongoing transformation into a provided block of data so it can be **import** 'ed back later on. This is useful in case you want to save partial result of the transformation after processing certain amount of data and reload this partial result multiple times later on for multiple re-use. No data processing happens at this point. Driver must not use req->result.

import Import partial state of the transformation. This function loads the entire state of the ongoing transformation from a provided block of data so the transformation can continue from this point onward. No data processing happens at this point. Driver must not use req->result.

setkey Set optional key used by the hashing algorithm. Intended to push optional key used by the hashing algorithm from upper layers into the driver. This function can store the key in the transformation context or can outright program it into the hardware. In the former case, one must be careful to program the key into the hardware at appropriate time and one must be careful that .setkey() can be called multiple times during the existence of the transformation object. Not all hashing algorithms do implement this function as it is only needed for keyed message digests. SHAx/MDx/CRCx do NOT implement this function. HMAC(MDx)/HMAC(SHAx)/CMAC(AES) do implement this function. This function must be called before any other of the **init**, **update**, **final**, **finup**, **digest** is called. No data processing happens at this point.

init_tfm Initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

exit_tfm Deinitialize the cryptographic transformation object. This is a counterpart to **init_tfm**, used to remove various changes set in **init_tfm**.

halg see *struct hash_alg_common*

struct **shash_alg**
synchronous message digest definition

Definition

```
struct shash_alg {
  int (*init)(struct shash_desc *desc);
  int (*update)(struct shash_desc *desc, const u8 *data, unsigned int len);
  int (*final)(struct shash_desc *desc, u8 *out);
  int (*finup)(struct shash_desc *desc, const u8 *data, unsigned int len, u8␣
↪*out);
  int (*digest)(struct shash_desc *desc, const u8 *data, unsigned int len, u8␣
↪*out);
```

```
  int (*export)(struct shash_desc *desc, void *out);
  int (*import)(struct shash_desc *desc, const void *in);
  int (*setkey)(struct crypto_shash *tfm, const u8 *key, unsigned int keylen);
  int (*init_tfm)(struct crypto_shash *tfm);
  void (*exit_tfm)(struct crypto_shash *tfm);
  unsigned int descsize;
  unsigned int digestsize ;
  unsigned int statesize;
  struct crypto_alg base;
};
```

Members

init see *struct ahash_alg*

update see *struct ahash_alg*

final see *struct ahash_alg*

finup see *struct ahash_alg*

digest see *struct ahash_alg*

export see *struct ahash_alg*

import see *struct ahash_alg*

setkey see *struct ahash_alg*

init_tfm Initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

exit_tfm Deinitialize the cryptographic transformation object. This is a counterpart to **init_tfm**, used to remove various changes set in **init_tfm**.

descsize Size of the operational state for the message digest. This state size is the memory size that needs to be allocated for shash_desc.__ctx

digestsize see *struct ahash_alg*

statesize see *struct ahash_alg*

base internally used

9.9 Asynchronous Message Digest API

The asynchronous message digest API is used with the ciphers of type CRYPTO_ALG_TYPE_AHASH (listed as type “ahash” in /proc/crypto)

The asynchronous cipher operation discussion provided for the CRYPTO_ALG_TYPE_SKCIPHER API applies here as well.

struct crypto_ahash *__crypto_alloc_ahash__(const char *alg_name, u32 type, u32 mask)
allocate ahash cipher handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the ahash cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for an ahash. The returned struct crypto_ahash is the cipher handle that is required for any subsequent API invocation for that ahash.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void **crypto_free_ahash**(struct crypto_ahash *tfm)
zeroize and free the ahash handle

Parameters

struct crypto_ahash *tfm cipher handle to be freed

Description

If **tfm** is a NULL or error pointer, this function does nothing.

unsigned int **crypto_ahash_digestsize**(struct crypto_ahash *tfm)
obtain message digest size

Parameters

struct crypto_ahash *tfm cipher handle

Description

The size for the message digest created by the message digest cipher referenced with the cipher handle is returned.

Return

message digest size of cipher

unsigned int **crypto_ahash_statesize**(struct crypto_ahash *tfm)
obtain size of the ahash state

Parameters

struct crypto_ahash *tfm cipher handle

Description

Return the size of the ahash state. With the *crypto_ahash_export()* function, the caller can export the state into a buffer whose size is defined with this function.

Return

size of the ahash state

struct crypto_ahash ***crypto_ahash_reqtfm**(struct ahash_request *req)
obtain cipher handle from request

Parameters

struct ahash_request *req asynchronous request handle that contains the reference to the ahash cipher handle

Description

Return the ahash cipher handle that is registered with the asynchronous request handle ahash_request.

Return

ahash cipher handle

unsigned int **crypto_ahash_reqsize**(struct crypto_ahash *tfm)
obtain size of the request data structure

Parameters

struct crypto_ahash *tfm cipher handle

Return

size of the request data

int **crypto_ahash_setkey**(struct crypto_ahash *tfm, const u8 *key, unsigned int keylen)
set key for cipher handle

Parameters

struct crypto_ahash *tfm cipher handle

const u8 *key buffer holding the key

unsigned int keylen length of the key in bytes

Description

The caller provided key is set for the ahash cipher. The cipher handle must point to a keyed hash in order for this function to succeed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_ahash_finup**(struct ahash_request *req)
update and finalize message digest

Parameters

struct ahash_request *req reference to the ahash_request handle that holds all information needed to perform the cipher operation

Description

This function is a “short-hand” for the function calls of crypto_ahash_update and crypto_ahash_final. The parameters have the same meaning as discussed for those separate functions.

Return

see *crypto_ahash_final()*

int **crypto_ahash_final**(struct ahash_request *req)
calculate message digest

Parameters

struct ahash_request *req reference to the ahash_request handle that holds all information needed to perform the cipher operation

Description

Finalize the message digest operation and create the message digest based on all data added to the cipher handle. The message digest is placed into the output buffer registered with the ahash_request handle.

Return

0 if the message digest was successfully calculated; -EINPROGRESS if data is fed into hardware (DMA) or queued for later; -EBUSY if queue is full and request should be resubmitted later; other < 0 if an error occurred

int **crypto_ahash_digest**(struct ahash_request *req)
calculate message digest for a buffer

Parameters

struct ahash_request *req reference to the ahash_request handle that holds all information needed to perform the cipher operation

Description

This function is a "short-hand" for the function calls of crypto_ahash_init, crypto_ahash_update and crypto_ahash_final. The parameters have the same meaning as discussed for those separate three functions.

Return

see *crypto_ahash_final()*

int **crypto_ahash_export**(struct ahash_request *req, void *out)
extract current message digest state

Parameters

struct ahash_request *req reference to the ahash_request handle whose state is exported

void *out output buffer of sufficient size that can hold the hash state

Description

This function exports the hash state of the ahash_request handle into the caller-allocated output buffer out which must have sufficient size (e.g. by calling *crypto_ahash_statesize()*).

Return

0 if the export was successful; < 0 if an error occurred

int **crypto_ahash_import**(struct ahash_request *req, const void *in)
import message digest state

Parameters

struct ahash_request *req reference to ahash_request handle the state is imported into

const void *in buffer holding the state

Description

This function imports the hash state into the ahash_request handle from the input buffer. That buffer should have been generated with the crypto_ahash_export function.

Return

0 if the import was successful; < 0 if an error occurred

int **crypto_ahash_init**(struct ahash_request *req)
(re)initialize message digest handle

Parameters

struct ahash_request *req ahash_request handle that already is initialized with all necessary data using the ahash_request_* API functions

Description

The call (re-)initializes the message digest referenced by the ahash_request handle. Any potentially existing state created by previous operations is discarded.

Return

see *crypto_ahash_final()*

9.10 Asynchronous Hash Request Handle

The ahash_request data structure contains all pointers to data required for the asynchronous cipher operation. This includes the cipher handle (which can be used by multiple ahash_request instances), pointer to plaintext and the message digest output buffer, asynchronous callback function, etc. It acts as a handle to the ahash_request_* API calls in a similar way as ahash handle to the crypto_ahash_* API calls.

void **ahash_request_set_tfm**(struct ahash_request *req, struct crypto_ahash *tfm)
update cipher handle reference in request

Parameters

struct ahash_request *req request handle to be modified

struct crypto_ahash *tfm cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing ahash handle in the request data structure with a different one.

struct ahash_request *__ahash_request_alloc__(struct crypto_ahash *tfm, gfp_t gfp)
allocate request data structure

Parameters

struct crypto_ahash *tfm cipher handle to be registered with the request

gfp_t gfp memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the ahash message digest API calls. During the allocation, the provided ahash handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **ahash_request_free**(struct ahash_request *req)
zeroize and free the request data structure

Parameters

struct ahash_request *req request data structure cipher handle to be freed

void **ahash_request_set_callback**(struct ahash_request *req, u32 flags,
crypto_completion_t compl, void *data)
set asynchronous callback function

Parameters

struct ahash_request *req request handle

u32 flags specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

crypto_completion_t compl callback function pointer to be registered with the request handle

void *data The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the "data" field in the `crypto_async_request` data structure provided to the callback function.

Description

This function allows setting the callback function that is triggered once the cipher operation completes.

The callback function is registered with the `ahash_request` handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **ahash_request_set_crypt**(struct ahash_request *req, struct scatterlist *src, u8 *result,
unsigned int nbytes)
set data buffers

Parameters

struct ahash_request *req ahash_request handle to be updated

struct scatterlist *src source scatter/gather list

u8 *result buffer that is filled with the message digest – the caller must ensure that the buffer has sufficient space by, for example, calling *crypto_ahash_digestsize()*

unsigned int nbytes number of bytes to process from the source scatter/gather list

Description

By using this call, the caller references the source scatter/gather list. The source scatter/gather list points to the data the message digest is to be calculated for.

9.11 Synchronous Message Digest API

The synchronous message digest API is used with the ciphers of type CRYPTO_ALG_TYPE_SHASH (listed as type “shash” in /proc/crypto)

The message digest API is able to maintain state information for the caller.

The synchronous message digest API can store user-related context in its shash_desc request data structure.

struct crypto_shash *__crypto_alloc_shash__(const char *alg_name, u32 type, u32 mask)
allocate message digest handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the message digest cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for a message digest. The returned `struct crypto_shash` is the cipher handle that is required for any subsequent API invocation for that message digest.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void __crypto_free_shash__(struct crypto_shash *tfm)
zeroize and free the message digest handle

Parameters

struct crypto_shash *tfm cipher handle to be freed

Description

If **tfm** is a NULL or error pointer, this function does nothing.

unsigned int __crypto_shash_blocksize__(struct crypto_shash *tfm)
obtain block size for cipher

Parameters

struct crypto_shash *tfm cipher handle

Description

The block size for the message digest cipher referenced with the cipher handle is returned.

Return

block size of cipher

unsigned int **crypto_shash_digestsize**(struct crypto_shash *tfm)
obtain message digest size

Parameters

struct crypto_shash *tfm cipher handle

Description

The size for the message digest created by the message digest cipher referenced with the cipher handle is returned.

Return

digest size of cipher

unsigned int **crypto_shash_descsize**(struct crypto_shash *tfm)
obtain the operational state size

Parameters

struct crypto_shash *tfm cipher handle

Description

The size of the operational state the cipher needs during operation is returned for the hash referenced with the cipher handle. This size is required to calculate the memory requirements to allow the caller allocating sufficient memory for operational state.

The operational state is defined with struct shash_desc where the size of that data structure is to be calculated as sizeof(struct shash_desc) + crypto_shash_descsize(alg)

Return

size of the operational state

int **crypto_shash_setkey**(struct crypto_shash *tfm, const u8 *key, unsigned int keylen)
set key for message digest

Parameters

struct crypto_shash *tfm cipher handle

const u8 *key buffer holding the key

unsigned int keylen length of the key in bytes

Description

The caller provided key is set for the keyed message digest cipher. The cipher handle must point to a keyed message digest cipher in order for this function to succeed.

Context

Any context.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_shash_digest**(struct shash_desc *desc, const u8 *data, unsigned int len, u8 *out)
calculate message digest for buffer

Parameters

struct shash_desc *desc see *crypto_shash_final()*

const u8 *data see *crypto_shash_update()*

unsigned int len see *crypto_shash_update()*

u8 *out see *crypto_shash_final()*

Description

This function is a "short-hand" for the function calls of crypto_shash_init, crypto_shash_update and crypto_shash_final. The parameters have the same meaning as discussed for those separate three functions.

Context

Any context.

Return

0 if the message digest creation was successful; < 0 if an error occurred

int **crypto_shash_export**(struct shash_desc *desc, void *out)
extract operational state for message digest

Parameters

struct shash_desc *desc reference to the operational state handle whose state is exported

void *out output buffer of sufficient size that can hold the hash state

Description

This function exports the hash state of the operational state handle into the caller-allocated output buffer out which must have sufficient size (e.g. by calling crypto_shash_descsize).

Context

Any context.

Return

0 if the export creation was successful; < 0 if an error occurred

int **crypto_shash_import**(struct shash_desc *desc, const void *in)
import operational state

Parameters

struct shash_desc *desc reference to the operational state handle the state imported into

const void *in buffer holding the state

Description

This function imports the hash state into the operational state handle from the input buffer. That buffer should have been generated with the crypto_ahash_export function.

Context

Any context.

Return

0 if the import was successful; < 0 if an error occurred

int **crypto_shash_init**(struct shash_desc *desc)
(re)initialize message digest

Parameters

struct shash_desc *desc operational state handle that is already filled

Description

The call (re-)initializes the message digest referenced by the operational state handle. Any potentially existing state created by previous operations is discarded.

Context

Any context.

Return

0 if the message digest initialization was successful; < 0 if an error occurred

int **crypto_shash_update**(struct shash_desc *desc, const u8 *data, unsigned int len)
add data to message digest for processing

Parameters

struct shash_desc *desc operational state handle that is already initialized

const u8 *data input data to be added to the message digest

unsigned int len length of the input data

Description

Updates the message digest state of the operational state handle.

Context

Any context.

Return

0 if the message digest update was successful; < 0 if an error occurred

int **crypto_shash_final**(struct shash_desc *desc, u8 *out)
calculate message digest

Parameters

struct shash_desc *desc operational state handle that is already filled with data

u8 *out output buffer filled with the message digest

Description

Finalize the message digest operation and create the message digest based on all data added to the cipher handle. The message digest is placed into the output buffer. The caller must ensure that the output buffer is large enough by using crypto_shash_digestsize.

Context

Any context.

Return

0 if the message digest creation was successful; < 0 if an error occurred

int **crypto_shash_finup**(struct shash_desc *desc, const u8 *data, unsigned int len, u8 *out)
calculate message digest of buffer

Parameters

struct shash_desc *desc see *crypto_shash_final()*

const u8 *data see *crypto_shash_update()*

unsigned int len see *crypto_shash_update()*

u8 *out see *crypto_shash_final()*

Description

This function is a "short-hand" for the function calls of crypto_shash_update and crypto_shash_final. The parameters have the same meaning as discussed for those separate functions.

Context

Any context.

Return

0 if the message digest creation was successful; < 0 if an error occurred

9.12 Random Number Algorithm Definitions

struct **rng_alg**
random number generator definition

Definition

```
struct rng_alg {
  int (*generate)(struct crypto_rng *tfm,const u8 *src, unsigned int slen, u8␣
↪*dst, unsigned int dlen);
  int (*seed)(struct crypto_rng *tfm, const u8 *seed, unsigned int slen);
  void (*set_ent)(struct crypto_rng *tfm, const u8 *data, unsigned int len);
  unsigned int seedsize;
  struct crypto_alg base;
};
```

Members

generate The function defined by this variable obtains a random number. The random number generator transform must generate the random number out of the context provided with this call, plus any additional data if provided to the call.

seed Seed or reseed the random number generator. With the invocation of this function call, the random number generator shall become ready for generation. If the random number generator requires a seed for setting up a new state, the seed must be provided by the consumer while invoking this function. The required size of the seed is defined with **seedsize** .

set_ent Set entropy that would otherwise be obtained from entropy source. Internal use only.

seedsize The seed size required for a random number generator initialization defined with this variable. Some random number generators does not require a seed as the seeding is implemented internally without the need of support by the consumer. In this case, the seed size is set to zero.

base Common crypto API algorithm data structure.

9.13 Crypto API Random Number API

The random number generator API is used with the ciphers of type CRYPTO_ALG_TYPE_RNG (listed as type “rng” in /proc/crypto)

struct crypto_rng *__crypto_alloc_rng__(const char *alg_name, u32 type, u32 mask)

- allocate RNG handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the message digest cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for a random number generator. The returned struct crypto_rng is the cipher handle that is required for any subsequent API invocation for that random number generator.

For all random number generators, this call creates a new private copy of the random number generator that does not share a state with other instances. The only exception is the “krng” random number generator which is a kernel crypto API use case for the get_random_bytes() function of the /dev/random driver.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

struct *rng_alg* *__crypto_rng_alg__(struct crypto_rng *tfm)
obtain name of RNG

Parameters

struct crypto_rng *tfm cipher handle

Description

Return the generic name (cra_name) of the initialized random number generator

Return

generic name string

void __crypto_free_rng__(struct crypto_rng *tfm)
zeroize and free RNG handle

Parameters

struct crypto_rng *tfm cipher handle to be freed

Description

If **tfm** is a NULL or error pointer, this function does nothing.

int **crypto_rng_generate**(struct crypto_rng *tfm, const u8 *src, unsigned int slen, u8 *dst, unsigned int dlen)
get random number

Parameters

struct crypto_rng *tfm cipher handle

const u8 *src Input buffer holding additional data, may be NULL

unsigned int slen Length of additional data

u8 *dst output buffer holding the random numbers

unsigned int dlen length of the output buffer

Description

This function fills the caller-allocated buffer with random numbers using the random number generator referenced by the cipher handle.

Return

0 function was successful; < 0 if an error occurred

int **crypto_rng_get_bytes**(struct crypto_rng *tfm, u8 *rdata, unsigned int dlen)
get random number

Parameters

struct crypto_rng *tfm cipher handle

u8 *rdata output buffer holding the random numbers

unsigned int dlen length of the output buffer

Description

This function fills the caller-allocated buffer with random numbers using the random number generator referenced by the cipher handle.

Return

0 function was successful; < 0 if an error occurred

int **crypto_rng_reset**(struct crypto_rng *tfm, const u8 *seed, unsigned int slen)
re-initialize the RNG

Parameters

struct crypto_rng *tfm cipher handle

const u8 *seed seed input data

unsigned int slen length of the seed input data

Description

The reset function completely re-initializes the random number generator referenced by the cipher handle by clearing the current state. The new state is initialized with the caller provided seed or automatically, depending on the random number generator type (the ANSI X9.31 RNG requires caller-provided seed, the SP800-90A DRBGs perform an automatic seeding). The seed is provided as a parameter to this function call. The provided seed should have the length of the seed size defined for the random number generator as defined by crypto_rng_seedsize.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_rng_seedsize**(struct crypto_rng *tfm)
obtain seed size of RNG

Parameters

struct crypto_rng *tfm cipher handle

Description

The function returns the seed size for the random number generator referenced by the cipher handle. This value may be zero if the random number generator does not implement or require a reseeding. For example, the SP800-90A DRBGs implement an automated reseeding after reaching a pre-defined threshold.

Return

seed size for the random number generator

9.14 Asymmetric Cipher Algorithm Definitions

struct **akcipher_request**
public key request

Definition

```
struct akcipher_request {
  struct crypto_async_request base;
  struct scatterlist *src;
  struct scatterlist *dst;
  unsigned int src_len;
  unsigned int dst_len;
  void *__ctx[] ;
};
```

Members

base Common attributes for async crypto requests

src Source data For verify op this is signature + digest, in that case total size of **src** is **src_len** + **dst_len**.

dst Destination data (Should be NULL for verify op)

src_len Size of the input buffer For verify op it's size of signature part of **src**, this part is supposed to be operated by cipher.

dst_len Size of **dst** buffer (for all ops except verify). It needs to be at least as big as the expected result depending on the operation. After operation it will be updated with the actual size of the result. In case of error where the dst sgl size was insufficient, it will be updated to the size required for the operation. For verify op this is size of digest part in **src**.

__ctx Start of private context data

struct **akcipher_alg**
generic public key algorithm

Definition

```
struct akcipher_alg {
  int (*sign)(struct akcipher_request *req);
  int (*verify)(struct akcipher_request *req);
  int (*encrypt)(struct akcipher_request *req);
  int (*decrypt)(struct akcipher_request *req);
  int (*set_pub_key)(struct crypto_akcipher *tfm, const void *key, unsigned␣
↪int keylen);
  int (*set_priv_key)(struct crypto_akcipher *tfm, const void *key, unsigned␣
↪int keylen);
  unsigned int (*max_size)(struct crypto_akcipher *tfm);
  int (*init)(struct crypto_akcipher *tfm);
  void (*exit)(struct crypto_akcipher *tfm);
  unsigned int reqsize;
  struct crypto_alg base;
};
```

Members

sign Function performs a sign operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

verify Function performs a complete verify operation as defined by public key algorithm, returning verification status. Requires digest value as input parameter.

encrypt Function performs an encrypt operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

decrypt Function performs a decrypt operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

set_pub_key Function invokes the algorithm specific set public key function, which knows how to decode and interpret the BER encoded public key and parameters

set_priv_key Function invokes the algorithm specific set private key function, which knows how to decode and interpret the BER encoded private key and parameters

max_size Function returns dest buffer size required for a given key.

init Initialize the cryptographic transformation object. This function is used to initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic

hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

exit Deinitialize the cryptographic transformation object. This is a counterpart to **init**, used to remove various changes set in **init**.

reqsize Request context size required by algorithm implementation

base Common crypto API algorithm data structure

9.15 Asymmetric Cipher API

The Public Key API is used with the algorithms of type CRYPTO_ALG_TYPE_AKCIPHER (listed as type "akcipher" in /proc/crypto)

struct crypto_akcipher *__crypto_alloc_akcipher__(const char *alg_name, u32 type, u32 mask)
allocate AKCIPHER tfm handle

Parameters

const char *alg_name is the cra_name / name or cra_driver_name / driver name of the public key algorithm e.g. "rsa"

u32 type specifies the type of the algorithm

u32 mask specifies the mask for the algorithm

Description

Allocate a handle for public key algorithm. The returned struct crypto_akcipher is the handle that is required for any subsequent API invocation for the public key operations.

Return

allocated handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void __crypto_free_akcipher__(struct crypto_akcipher *tfm)
free AKCIPHER tfm handle

Parameters

struct crypto_akcipher *tfm AKCIPHER tfm handle allocated with *crypto_alloc_akcipher()*

Description

If **tfm** is a NULL or error pointer, this function does nothing.

unsigned int __crypto_akcipher_maxsize__(struct crypto_akcipher *tfm)
Get len for output buffer

Parameters

struct crypto_akcipher *tfm AKCIPHER tfm handle allocated with *crypto_alloc_akcipher()*

Description

Function returns the dest buffer size required for a given key. Function assumes that the key is already set in the transformation. If this function is called without a setkey or with a failed setkey, you will end up in a NULL dereference.

int **crypto_akcipher_encrypt**(struct *akcipher_request* *req)
Invoke public key encrypt operation

Parameters

struct akcipher_request *req asymmetric key request

Description

Function invokes the specific public key encrypt operation for a given public key algorithm

Return

zero on success; error code in case of error

int **crypto_akcipher_decrypt**(struct *akcipher_request* *req)
Invoke public key decrypt operation

Parameters

struct akcipher_request *req asymmetric key request

Description

Function invokes the specific public key decrypt operation for a given public key algorithm

Return

zero on success; error code in case of error

int **crypto_akcipher_sign**(struct *akcipher_request* *req)
Invoke public key sign operation

Parameters

struct akcipher_request *req asymmetric key request

Description

Function invokes the specific public key sign operation for a given public key algorithm

Return

zero on success; error code in case of error

int **crypto_akcipher_verify**(struct *akcipher_request* *req)
Invoke public key signature verification

Parameters

struct akcipher_request *req asymmetric key request

Description

Function invokes the specific public key signature verification operation for a given public key algorithm.

Note

req->dst should be NULL, req->src should point to SG of size (req->src_size + req->dst_size), containing signature (of req->src_size length) with appended digest (of req->dst_size length).

Return

zero on verification success; error code in case of error.

int **crypto_akcipher_set_pub_key**(struct crypto_akcipher *tfm, const void *key, unsigned int keylen)
Invoke set public key operation

Parameters

struct crypto_akcipher *tfm tfm handle

const void *key BER encoded public key, algo OID, paramlen, BER encoded parameters

unsigned int keylen length of the key (not including other data)

Description

Function invokes the algorithm specific set key function, which knows how to decode and interpret the encoded key and parameters

Return

zero on success; error code in case of error

int **crypto_akcipher_set_priv_key**(struct crypto_akcipher *tfm, const void *key, unsigned int keylen)
Invoke set private key operation

Parameters

struct crypto_akcipher *tfm tfm handle

const void *key BER encoded private key, algo OID, paramlen, BER encoded parameters

unsigned int keylen length of the key (not including other data)

Description

Function invokes the algorithm specific set key function, which knows how to decode and interpret the encoded key and parameters

Return

zero on success; error code in case of error

9.16 Asymmetric Cipher Request Handle

struct *akcipher_request* ***akcipher_request_alloc**(struct crypto_akcipher *tfm, gfp_t gfp)
allocates public key request

Parameters

struct crypto_akcipher *tfm AKCIPHER tfm handle allocated with *crypto_alloc_akcipher()*

gfp_t gfp allocation flags

Return

allocated handle in case of success or NULL in case of an error.

void **akcipher_request_free**(struct *akcipher_request* *req)
zeroize and free public key request

Parameters

struct akcipher_request *req request to free

void **akcipher_request_set_callback**(struct *akcipher_request* *req, u32 flgs, crypto_completion_t cmpl, void *data)
Sets an asynchronous callback.

Parameters

struct akcipher_request *req request that the callback will be set for

u32 flgs specify for instance if the operation may backlog

crypto_completion_t cmpl callback which will be called

void *data private data used by the caller

Description

Callback will be called when an asynchronous operation on a given request is finished.

void **akcipher_request_set_crypt**(struct *akcipher_request* *req, struct scatterlist *src, struct scatterlist *dst, unsigned int src_len, unsigned int dst_len)
Sets request parameters

Parameters

struct akcipher_request *req public key request

struct scatterlist *src ptr to input scatter list

struct scatterlist *dst ptr to output scatter list or NULL for verify op

unsigned int src_len size of the src input scatter list to be processed

unsigned int dst_len size of the dst output scatter list or size of signature portion in **src** for verify op

Description

Sets parameters required by crypto operation

9.17 Key-agreement Protocol Primitives (KPP) Cipher Algorithm Definitions

struct **kpp_request**

Definition

```
struct kpp_request {
  struct crypto_async_request base;
  struct scatterlist *src;
  struct scatterlist *dst;
  unsigned int src_len;
```

```
  unsigned int dst_len;
  void *__ctx[] ;
};
```

Members

base Common attributes for async crypto requests

src Source data

dst Destination data

src_len Size of the input buffer

dst_len Size of the output buffer. It needs to be at least as big as the expected result depending on the operation After operation it will be updated with the actual size of the result. In case of error where the dst sgl size was insufficient, it will be updated to the size required for the operation.

__ctx Start of private context data

struct **crypto_kpp**
user-instantiated object which encapsulate algorithms and core processing logic

Definition

```
struct crypto_kpp {
  struct crypto_tfm base;
};
```

Members

base Common crypto API algorithm data structure

struct **kpp_alg**
generic key-agreement protocol primitives

Definition

```
struct kpp_alg {
  int (*set_secret)(struct crypto_kpp *tfm, const void *buffer, unsigned int␣
↪len);
  int (*generate_public_key)(struct kpp_request *req);
  int (*compute_shared_secret)(struct kpp_request *req);
  unsigned int (*max_size)(struct crypto_kpp *tfm);
  int (*init)(struct crypto_kpp *tfm);
  void (*exit)(struct crypto_kpp *tfm);
  unsigned int reqsize;
  struct crypto_alg base;
};
```

Members

set_secret Function invokes the protocol specific function to store the secret private key along with parameters. The implementation knows how to decode the buffer

generate_public_key Function generate the public key to be sent to the counterpart. In case of error, where output is not big enough req->dst_len will be updated to the size required

compute_shared_secret Function compute the shared secret as defined by the algorithm. The result is given back to the user. In case of error, where output is not big enough, req->dst_len will be updated to the size required

max_size Function returns the size of the output buffer

init Initialize the object. This is called only once at instantiation time. In case the cryptographic hardware needs to be initialized. Software fallback should be put in place here.

exit Undo everything **init** did.

reqsize Request context size required by algorithm implementation

base Common crypto API algorithm data structure

struct **kpp_secret**
small header for packing secret buffer

Definition

```
struct kpp_secret {
  unsigned short type;
  unsigned short len;
};
```

Members

type define type of secret. Each kpp type will define its own

len specify the len of the secret, include the header, that follows the struct

9.18 Key-agreement Protocol Primitives (KPP) Cipher API

The KPP API is used with the algorithm type CRYPTO_ALG_TYPE_KPP (listed as type "kpp" in /proc/crypto)

struct *crypto_kpp* ***crypto_alloc_kpp**(const char *alg_name, u32 type, u32 mask)
allocate KPP tfm handle

Parameters

const char *alg_name is the name of the kpp algorithm (e.g. "dh", "ecdh")

u32 type specifies the type of the algorithm

u32 mask specifies the mask for the algorithm

Description

Allocate a handle for kpp algorithm. The returned *struct crypto_kpp* is required for any following API invocation

Return

allocated handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void **crypto_free_kpp**(struct *crypto_kpp* *tfm)
free KPP tfm handle

Parameters

struct crypto_kpp *tfm KPP tfm handle allocated with *crypto_alloc_kpp()*

Description

If **tfm** is a NULL or error pointer, this function does nothing.

int **crypto_kpp_set_secret**(struct *crypto_kpp* *tfm, const void *buffer, unsigned int len)
Invoke kpp operation

Parameters

struct crypto_kpp *tfm tfm handle

const void *buffer Buffer holding the packet representation of the private key. The structure of the packet key depends on the particular KPP implementation. Packing and unpacking helpers are provided for ECDH and DH (see the respective header files for those implementations).

unsigned int len Length of the packet private key buffer.

Description

Function invokes the specific kpp operation for a given alg.

Return

zero on success; error code in case of error

int **crypto_kpp_generate_public_key**(struct *kpp_request* *req)
Invoke kpp operation

Parameters

struct kpp_request *req kpp key request

Description

Function invokes the specific kpp operation for generating the public part for a given kpp algorithm.

To generate a private key, the caller should use a random number generator. The output of the requested length serves as the private key.

Return

zero on success; error code in case of error

int **crypto_kpp_compute_shared_secret**(struct *kpp_request* *req)
Invoke kpp operation

Parameters

struct kpp_request *req kpp key request

Description

Function invokes the specific kpp operation for computing the shared secret for a given kpp algorithm.

Return

zero on success; error code in case of error

unsigned int **crypto_kpp_maxsize**(struct *crypto_kpp* *tfm)
Get len for output buffer

Parameters

struct crypto_kpp *tfm KPP tfm handle allocated with *crypto_alloc_kpp()*

Description

Function returns the output buffer size required for a given key. Function assumes that the key is already set in the transformation. If this function is called without a setkey or with a failed setkey, you will end up in a NULL dereference.

9.19 Key-agreement Protocol Primitives (KPP) Cipher Request Handle

struct *kpp_request* ***kpp_request_alloc**(struct *crypto_kpp* *tfm, gfp_t gfp)
allocates kpp request

Parameters

struct crypto_kpp *tfm KPP tfm handle allocated with *crypto_alloc_kpp()*

gfp_t gfp allocation flags

Return

allocated handle in case of success or NULL in case of an error.

void **kpp_request_free**(struct *kpp_request* *req)
zeroize and free kpp request

Parameters

struct kpp_request *req request to free

void **kpp_request_set_callback**(struct *kpp_request* *req, u32 flgs, crypto_completion_t cmpl, void *data)
Sets an asynchronous callback.

Parameters

struct kpp_request *req request that the callback will be set for

u32 flgs specify for instance if the operation may backlog

crypto_completion_t cmpl callback which will be called

void *data private data used by the caller

Description

Callback will be called when an asynchronous operation on a given request is finished.

void **kpp_request_set_input**(struct *kpp_request* *req, struct scatterlist *input, unsigned int input_len)
Sets input buffer

Parameters

struct kpp_request *req kpp request

struct scatterlist *input ptr to input scatter list

unsigned int input_len size of the input scatter list

Description

Sets parameters required by generate_public_key

void **kpp_request_set_output**(struct *kpp_request* *req, struct scatterlist *output, unsigned int output_len)
Sets output buffer

Parameters

struct kpp_request *req kpp request

struct scatterlist *output ptr to output scatter list

unsigned int output_len size of the output scatter list

Description

Sets parameters required by kpp operation

9.20 ECDH Helper Functions

To use ECDH with the KPP cipher API, the following data structure and functions should be used.

The ECC curves known to the ECDH implementation are specified in this header file.

To use ECDH with KPP, the following functions should be used to operate on an ECDH private key. The packet private key that can be set with the KPP API function call of crypto_kpp_set_secret.

struct **ecdh**
define an ECDH private key

Definition

```
struct ecdh {
  char *key;
  unsigned short key_size;
};
```

Members

key Private ECDH key

key_size Size of the private ECDH key

unsigned int **crypto_ecdh_key_len**(const struct *ecdh* *params)
Obtain the size of the private ECDH key

Parameters

const struct ecdh *params private ECDH key

Description

This function returns the packet ECDH key size. A caller can use that with the provided ECDH private key reference to obtain the required memory size to hold a packet key.

Return

size of the key in bytes

int **crypto_ecdh_encode_key**(char *buf, unsigned int len, const struct *ecdh* *p)
encode the private key

Parameters

char *buf Buffer allocated by the caller to hold the packet ECDH private key. The buffer should be at least crypto_ecdh_key_len bytes in size.

unsigned int len Length of the packet private key buffer

const struct ecdh *p Buffer with the caller-specified private key

Description

The ECDH implementations operate on a packet representation of the private key.

Return

-EINVAL if buffer has insufficient size, 0 on success

int **crypto_ecdh_decode_key**(const char *buf, unsigned int len, struct *ecdh* *p)
decode a private key

Parameters

const char *buf Buffer holding a packet key that should be decoded

unsigned int len Length of the packet private key buffer

struct ecdh *p Buffer allocated by the caller that is filled with the unpacked ECDH private key.

Description

The unpacking obtains the private key by pointing **p** to the correct location in **buf**. Thus, both pointers refer to the same memory.

Return

-EINVAL if buffer has insufficient size, 0 on success

9.21 DH Helper Functions

To use DH with the KPP cipher API, the following data structure and functions should be used.

To use DH with KPP, the following functions should be used to operate on a DH private key. The packet private key that can be set with the KPP API function call of crypto_kpp_set_secret.

struct **dh**
define a DH private key

Definition

```
struct dh {
  const void *key;
  const void *p;
  const void *g;
  unsigned int key_size;
  unsigned int p_size;
  unsigned int g_size;
};
```

Members

key Private DH key

p Diffie-Hellman parameter P

g Diffie-Hellman generator G

key_size Size of the private DH key

p_size Size of DH parameter P

g_size Size of DH generator G

unsigned int **crypto_dh_key_len**(const struct *dh* *params)
Obtain the size of the private DH key

Parameters

const struct dh *params private DH key

Description

This function returns the packet DH key size. A caller can use that with the provided DH private key reference to obtain the required memory size to hold a packet key.

Return

size of the key in bytes

int **crypto_dh_encode_key**(char *buf, unsigned int len, const struct *dh* *params)
encode the private key

Parameters

char *buf Buffer allocated by the caller to hold the packet DH private key. The buffer should be at least crypto_dh_key_len bytes in size.

unsigned int len Length of the packet private key buffer

const struct dh *params Buffer with the caller-specified private key

Description

The DH implementations operate on a packet representation of the private key.

Return

-EINVAL if buffer has insufficient size, 0 on success

int **crypto_dh_decode_key**(const char *buf, unsigned int len, struct *dh* *params)
decode a private key

Parameters

const char *buf Buffer holding a packet key that should be decoded

unsigned int len Length of the packet private key buffer

struct dh *params Buffer allocated by the caller that is filled with the unpacked DH private key.

Description

The unpacking obtains the private key by pointing **p** to the correct location in **buf**. Thus, both pointers refer to the same memory.

Return

-EINVAL if buffer has insufficient size, 0 on success

CHAPTER

TEN

CODE EXAMPLES

10.1 Code Example For Symmetric Key Cipher Operation

This code encrypts some data with AES-256-XTS. For sake of example, all inputs are random bytes, the encryption is done in-place, and it's assumed the code is running in a context where it can sleep.

```
static int test_skcipher(void)
{
        struct crypto_skcipher *tfm = NULL;
        struct skcipher_request *req = NULL;
        u8 *data = NULL;
        const size_t datasize = 512; /* data size in bytes */
        struct scatterlist sg;
        DECLARE_CRYPTO_WAIT(wait);
        u8 iv[16];  /* AES-256-XTS takes a 16-byte IV */
        u8 key[64]; /* AES-256-XTS takes a 64-byte key */
        int err;

        /*
         * Allocate a tfm (a transformation object) and set the key.
         *
         * In real-world use, a tfm and key are typically used for many
         * encryption/decryption operations.  But in this example, we'll just␣
↪do a
         * single encryption operation with it (which is not very efficient).
         */

        tfm = crypto_alloc_skcipher("xts(aes)", 0, 0);
        if (IS_ERR(tfm)) {
                pr_err("Error allocating xts(aes) handle: %ld\n", PTR_
↪ERR(tfm));
                return PTR_ERR(tfm);
        }

        get_random_bytes(key, sizeof(key));
        err = crypto_skcipher_setkey(tfm, key, sizeof(key));
        if (err) {
                pr_err("Error setting key: %d\n", err);
                goto out;
```

```
        }

        /* Allocate a request object */
        req = skcipher_request_alloc(tfm, GFP_KERNEL);
        if (!req) {
                err = -ENOMEM;
                goto out;
        }

        /* Prepare the input data */
        data = kmalloc(datasize, GFP_KERNEL);
        if (!data) {
                err = -ENOMEM;
                goto out;
        }
        get_random_bytes(data, datasize);

        /* Initialize the IV */
        get_random_bytes(iv, sizeof(iv));

        /*
         * Encrypt the data in-place.
         *
         * For simplicity, in this example we wait for the request to complete
         * before proceeding, even if the underlying implementation is␣
↪asynchronous.
         *
         * To decrypt instead of encrypt, just change crypto_skcipher_
↪encrypt() to
         * crypto_skcipher_decrypt().
         */
        sg_init_one(&sg, data, datasize);
        skcipher_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG |
                                           CRYPTO_TFM_REQ_MAY_SLEEP,
                                      crypto_req_done, &wait);
        skcipher_request_set_crypt(req, &sg, &sg, datasize, iv);
        err = crypto_wait_req(crypto_skcipher_encrypt(req), &wait);
        if (err) {
                pr_err("Error encrypting data: %d\n", err);
                goto out;
        }

        pr_debug("Encryption was successful\n");
out:
        crypto_free_skcipher(tfm);
        skcipher_request_free(req);
        kfree(data);
        return err;
}
```

10.2 Code Example For Use of Operational State Memory With SHASH

```
struct sdesc {
    struct shash_desc shash;
    char ctx[];
};

static struct sdesc *init_sdesc(struct crypto_shash *alg)
{
    struct sdesc *sdesc;
    int size;

    size = sizeof(struct shash_desc) + crypto_shash_descsize(alg);
    sdesc = kmalloc(size, GFP_KERNEL);
    if (!sdesc)
        return ERR_PTR(-ENOMEM);
    sdesc->shash.tfm = alg;
    return sdesc;
}

static int calc_hash(struct crypto_shash *alg,
             const unsigned char *data, unsigned int datalen,
             unsigned char *digest)
{
    struct sdesc *sdesc;
    int ret;

    sdesc = init_sdesc(alg);
    if (IS_ERR(sdesc)) {
        pr_info("can't alloc sdesc\n");
        return PTR_ERR(sdesc);
    }

    ret = crypto_shash_digest(&sdesc->shash, data, datalen, digest);
    kfree(sdesc);
    return ret;
}

static int test_hash(const unsigned char *data, unsigned int datalen,
             unsigned char *digest)
{
    struct crypto_shash *alg;
    char *hash_alg_name = "sha1-padlock-nano";
    int ret;

    alg = crypto_alloc_shash(hash_alg_name, 0, 0);
    if (IS_ERR(alg)) {
            pr_info("can't alloc alg %s\n", hash_alg_name);
            return PTR_ERR(alg);
```

```
    }
    ret = calc_hash(alg, data, datalen, digest);
    crypto_free_shash(alg);
    return ret;
}
```

10.3 Code Example For Random Number Generator Usage

```
static int get_random_numbers(u8 *buf, unsigned int len)
{
    struct crypto_rng *rng = NULL;
    char *drbg = "drbg_nopr_sha256"; /* Hash DRBG with SHA-256, no PR */
    int ret;

    if (!buf || !len) {
        pr_debug("No output buffer provided\n");
        return -EINVAL;
    }

    rng = crypto_alloc_rng(drbg, 0, 0);
    if (IS_ERR(rng)) {
        pr_debug("could not allocate RNG handle for %s\n", drbg);
        return PTR_ERR(rng);
    }

    ret = crypto_rng_get_bytes(rng, buf, len);
    if (ret < 0)
        pr_debug("generation of random numbers failed\n");
    else if (ret == 0)
        pr_debug("RNG returned no data");
    else
        pr_debug("RNG returned %d bytes of data\n", ret);

out:
    crypto_free_rng(rng);
    return ret;
}
```

CHAPTER

ELEVEN

FAST & PORTABLE DES ENCRYPTION & DECRYPTION

Note: Below is the original README file from the descore.shar package, converted to ReST format.

---

des - fast & portable DES encryption & decryption.

Copyright © 1992 Dana L. How

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Author's address: how@isl.stanford.edu

==>> To compile after untarring/unsharring, just `make` <<==

This package was designed with the following goals:

1. Highest possible encryption/decryption PERFORMANCE.
2. PORTABILITY to any byte-addressable host with a 32bit unsigned C type
3. Plug-compatible replacement for KERBEROS's low-level routines.

This second release includes a number of performance enhancements for register-starved machines. My discussions with Richard Outerbridge, 71755.204@compuserve.com, sparked a number of these enhancements.

To more rapidly understand the code in this package, inspect desSmallFips.i (created by typing `make`) BEFORE you tackle desCode.h. The latter is set up in a parameterized fashion so it can easily be modified by speed-daemon hackers in pursuit of that last microsecond. You will find it more illuminating to inspect one specific implementation, and then move on to the common abstract skeleton with this one in mind.

performance comparison to other available des code which i could compile on a SPARCStation 1 (cc -O4, gcc -O2):

this code (byte-order independent):

- 30us per encryption (options: 64k tables, no IP/FP)
- 33us per encryption (options: 64k tables, FIPS standard bit ordering)
- 45us per encryption (options: 2k tables, no IP/FP)
- 48us per encryption (options: 2k tables, FIPS standard bit ordering)
- 275us to set a new key (uses 1k of key tables)

  this has the quickest encryption/decryption routines i've seen. since i was interested in fast des filters rather than crypt(3) and password cracking, i haven't really bothered yet to speed up the key setting routine. also, i have no interest in re-implementing all the other junk in the mit kerberos des library, so i've just provided my routines with little stub interfaces so they can be used as drop-in replacements with mit's code or any of the mit- compatible packages below. (note that the first two timings above are highly variable because of cache effects).

kerberos des replacement from australia (version 1.95):

- 53us per encryption (uses 2k of tables)
- 96us to set a new key (uses 2.25k of key tables)

  so despite the author's inclusion of some of the performance improvements i had suggested to him, this package's encryption/decryption is still slower on the sparc and 68000. more specifically, 19-40% slower on the 68020 and 11-35% slower on the sparc, depending on the compiler; in full gory detail (ALT_ECB is a libdes variant):

compiler	machine	desCore libdes	ALT_ECB slower by
gcc 2.1 -O2	Sun 3/110	304 uS 369.5uS	461.8uS 22%
cc -O1	Sun 3/110	336 uS 436.6uS	399.3uS 19%
cc -O2	Sun 3/110	360 uS 532.4uS	505.1uS 40%
cc -O4	Sun 3/110	365 uS 532.3uS	505.3uS 38%
gcc 2.1 -O2	Sun 4/50	48 uS 53.4uS	57.5uS 11%
cc -O2	Sun 4/50	48 uS 64.6uS	64.7uS 35%
cc -O4	Sun 4/50	48 uS 64.7uS	64.9uS 35%

  (my time measurements are not as accurate as his).

  the comments in my first release of desCore on version 1.92:

  - 68us per encryption (uses 2k of tables)
  - 96us to set a new key (uses 2.25k of key tables)

    this is a very nice package which implements the most important of the optimizations which i did in my encryption routines. it's a bit weak on common low-level optimizations which is why it's 39%-106% slower. because he was interested in fast crypt(3) and password-cracking applications, he also used the same ideas to speed up the key-setting routines with impressive results. (at some point i may do the same in my package). he also implements the rest of the mit des library.

(code from eay@psych.psy.uq.oz.au via comp.sources.misc)

fast crypt(3) package from denmark:

> the des routine here is buried inside a loop to do the crypt function and i didn't feel like ripping it out and measuring performance. his code takes 26 sparc instructions to compute one des iteration; above, Quick (64k) takes 21 and Small (2k) takes 37. he claims to use 280k of tables but the iteration calculation seems to use only 128k. his tables and code are machine independent.
>
> (code from glad@daimi.aau.dk via alt.sources or comp.sources.misc)

swedish reimplementation of Kerberos des library

- 108us per encryption (uses 34k worth of tables)
- 134us to set a new key (uses 32k of key tables to get this speed!)

> the tables used seem to be machine-independent; he seems to have included a lot of special case code so that, e.g., `long` loads can be used instead of 4 `char` loads when the machine's architecture allows it.
>
> (code obtained from chalmers.se:pub/des)

crack 3.3c package from england:

> as in crypt above, the des routine is buried in a loop. it's also very modified for crypt. his iteration code uses 16k of tables and appears to be slow.
>
> (code obtained from aem@aber.ac.uk via alt.sources or comp.sources.misc)

`highly optimized` and tweaked Kerberos/Athena code (byte-order dependent):

- 165us per encryption (uses 6k worth of tables)
- 478us to set a new key (uses <1k of key tables)

> so despite the comments in this code, it was possible to get faster code AND smaller tables, as well as making the tables machine-independent. (code obtained from prep.ai.mit.edu)

UC Berkeley code (depends on machine-endedness):

- 226us per encryption
- 10848us to set a new key

> table sizes are unclear, but they don't look very small (code obtained from wuarchive.wustl.edu)

11.1 motivation and history

a while ago i wanted some des routines and the routines documented on sun's man pages either didn't exist or dumped core. i had heard of kerberos, and knew that it used des, so i figured i'd use its routines. but once i got it and looked at the code, it really set off a lot of pet peeves - it was too convoluted, the code had been written without taking advantage of the regular structure of operations such as IP, E, and FP (i.e. the author didn't sit down and think before coding), it was excessively slow, the author had attempted to clarify the code by adding MORE statements to make the data movement more `consistent` instead of simplifying his implementation and

cutting down on all data movement (in particular, his use of L1, R1, L2, R2), and it was full of idiotic `tweaks` for particular machines which failed to deliver significant speedups but which did obfuscate everything. so i took the test data from his verification program and rewrote everything else.

a while later i ran across the great crypt(3) package mentioned above. the fact that this guy was computing 2 sboxes per table lookup rather than one (and using a MUCH larger table in the process) emboldened me to do the same - it was a trivial change from which i had been scared away by the larger table size. in his case he didn't realize you don't need to keep the working data in TWO forms, one for easy use of half the sboxes in indexing, the other for easy use of the other half; instead you can keep it in the form for the first half and use a simple rotate to get the other half. this means i have (almost) half the data manipulation and half the table size. in fairness though he might be encoding something particular to crypt(3) in his tables - i didn't check.

i'm glad that i implemented it the way i did, because this C version is portable (the ifdef's are performance enhancements) and it is faster than versions hand-written in assembly for the sparc!

11.2 porting notes

one thing i did not want to do was write an enormous mess which depended on endedness and other machine quirks, and which necessarily produced different code and different lookup tables for different machines. see the kerberos code for an example of what i didn't want to do; all their endedness-specific `optimizations` obfuscate the code and in the end were slower than a simpler machine independent approach. however, there are always some portability considerations of some kind, and i have included some options for varying numbers of register variables. perhaps some will still regard the result as a mess!

1) i assume everything is byte addressable, although i don't actually depend on the byte order, and that bytes are 8 bits. i assume word pointers can be freely cast to and from char pointers. note that 99% of C programs make these assumptions. i always use unsigned char's if the high bit could be set.

2) the typedef `word` means a 32 bit unsigned integral type. if `unsigned long` is not 32 bits, change the typedef in desCore.h. i assume sizeof(word) == 4 EVERYWHERE.

the (worst-case) cost of my NOT doing endedness-specific optimizations in the data loading and storing code surrounding the key iterations is less than 12%. also, there is the added benefit that the input and output work areas do not need to be word-aligned.

11.3 OPTIONAL performance optimizations

1) you should define one of `i386, vax, mc68000,` or `sparc,` whichever one is closest to the capabilities of your machine. see the start of desCode.h to see exactly what this selection implies. note that if you select the wrong one, the des code will still work; these are just performance tweaks.

2) for those with functional `asm` keywords: you should change the ROR and ROL macros to use machine rotate instructions if you have them. this will save 2 instructions and a temporary per use, or about 32 to 40 instructions per en/decryption.

note that gcc is smart enough to translate the ROL/R macros into machine rotates!

these optimizations are all rather persnickety, yet with them you should be able to get performance equal to assembly-coding, except that:

1) with the lack of a bit rotate operator in C, rotates have to be synthesized from shifts. so access to `asm` will speed things up if your machine has rotates, as explained above in (3) (not necessary if you use gcc).

2) if your machine has less than 12 32-bit registers i doubt your compiler will generate good code.

   `i386` tries to configure the code for a 386 by only declaring 3 registers (it appears that gcc can use ebx, esi and edi to hold register variables). however, if you like assembly coding, the 386 does have 7 32-bit registers, and if you use ALL of them, use `scaled by 8` address modes with displacement and other tricks, you can get reasonable routines for DesQuickCore... with about 250 instructions apiece. For DesSmall... it will help to rearrange des_keymap, i.e., now the sbox # is the high part of the index and the 6 bits of data is the low part; it helps to exchange these.

   since i have no way to conveniently test it i have not provided my shoehorned 386 version. note that with this release of desCore, gcc is able to put everything in registers(!), and generate about 370 instructions apiece for the DesQuickCore... routines!

11.4 coding notes

the en/decryption routines each use 6 necessary register variables, with 4 being actively used at once during the inner iterations. if you don't have 4 register variables get a new machine. up to 8 more registers are used to hold constants in some configurations.

i assume that the use of a constant is more expensive than using a register:

a) additionally, i have tried to put the larger constants in registers. registering priority was by the following:

   - anything more than 12 bits (bad for RISC and CISC)
   - greater than 127 in value (can't use movq or byte immediate on CISC)
   - 9-127 (may not be able to use CISC shift immediate or add/sub quick),
   - 1-8 were never registered, being the cheapest constants.

b) the compiler may be too stupid to realize table and table+256 should be assigned to different constant registers and instead repetitively do the arithmetic, so i assign these to explicit `m` register variables when possible and helpful.

i assume that indexing is cheaper or equivalent to auto increment/decrement, where the index is 7 bits unsigned or smaller. this assumption is reversed for 68k and vax.

i assume that addresses can be cheaply formed from two registers, or from a register and a small constant. for the 68000, the `two registers and small offset` form is used sparingly. all index scaling is done explicitly - no hidden shifts by log2(sizeof).

the code is written so that even a dumb compiler should never need more than one hidden temporary, increasing the chance that everything will fit in the registers. KEEP THIS MORE SUBTLE POINT IN MIND IF YOU REWRITE ANYTHING.

(actually, there are some code fragments now which do require two temps, but fixing it would either break the structure of the macros or require declaring another temporary).

11.5 special efficient data format

bits are manipulated in this arrangement most of the time (S7 S5 S3 S1):

```
003130292827xxxx242322212019xxxx161514131211xxxx080706050403xxxx
```

(the x bits are still there, i'm just emphasizing where the S boxes are). bits are rotated left 4 when computing S6 S4 S2 S0:

```
282726252423xxxx201918171615xxxx121110090807xxxx040302010031xxxx
```

the rightmost two bits are usually cleared so the lower byte can be used as an index into an sbox mapping table. the next two x'd bits are set to various values to access different parts of the tables.

how to use the routines

datatypes: pointer to 8 byte area of type DesData used to hold keys and input/output blocks to des.

pointer to 128 byte area of type DesKeys used to hold full 768-bit key. must be long-aligned.

DesQuickInit() call this before using any other routine with `Quick` in its name. it generates the special 64k table these routines need.

DesQuickDone() frees this table

DesMethod(m, k) m points to a 128byte block, k points to an 8 byte des key which must have odd parity (or -1 is returned) and which must not be a (semi-)weak key (or -2 is returned). normally DesMethod() returns 0.

m is filled in from k so that when one of the routines below is called with m, the routine will act like standard des en/decryption with the key k. if you use DesMethod, you supply a standard 56bit key; however, if you fill in m yourself, you will get a 768bit key - but then it won't be standard. it's 768bits not 1024 because the least significant two bits of each byte are not used. note that these two bits will be set to magic constants which speed up the encryption/decryption on some machines. and yes, each byte controls a specific sbox during a specific iteration.

you really shouldn't use the 768bit format directly; i should provide a routine that converts 128 6-bit bytes (specified in S-box mapping order or something) into the right format for you. this would entail some byte concatenation and rotation.

Des{Small|Quick}{Fips|Core}{Encrypt|Decrypt}(d, m, s) performs des on the 8 bytes at s into the 8 bytes at `d. (d,s: char *)`.

uses m as a 768bit key as explained above.

the Encrypt|Decrypt choice is obvious.

Fips|Core determines whether a completely standard FIPS initial and final permutation is done; if not, then the data is loaded and stored in a nonstandard bit order (FIPS w/o IP/FP).

Fips slows down Quick by 10%, Small by 9%.

Small|Quick determines whether you use the normal routine or the crazy quick one which gobbles up 64k more of memory. Small is 50% slower then Quick, but Quick needs 32 times as much memory. Quick is included for programs that do nothing but DES, e.g., encryption filters, etc.

11.6 Getting it to compile on your machine

there are no machine-dependencies in the code (see porting), except perhaps the `now()` macro in desTest.c. ALL generated tables are machine independent. you should edit the Makefile with the appropriate optimization flags for your compiler (MAX optimization).

11.7 Speeding up kerberos (and/or its des library)

note that i have included a kerberos-compatible interface in desUtil.c through the functions des_key_sched() and des_ecb_encrypt(). to use these with kerberos or kerberos-compatible code put desCore.a ahead of the kerberos-compatible library on your linker's command line. you should not need to #include desCore.h; just include the header file provided with the kerberos library.

11.8 Other uses

the macros in desCode.h would be very useful for putting inline des functions in more complicated encryption routines.

INDEX

D

E

H

K

R

S