LUKS1 On-Disk Format Specification
Version 1.2.3

Clemens Fruhwirth <clemens@endorphin.org>

January 20, 2018

Document History

Version	Date Author Changes
1.0	22.01.2005 Clemens Fruhwirth <clemens@endorphin.org> More clear distinction between raw data and string data by adding a byte[] data type for LUKS magic, salt and checksum data.
1.0.1	15.01.2006 Clemens Fruhwirth <clemens@endorphin.org> Corrected the hash-spec length in Figure 1 from 64 to 32 bytes as implied by offset calculation and all other assumptions in this document.
1.1	18.02.2006 Clemens Fruhwirth <clemens@endorphin.org> Added precise AFsplit specification. Removed lrw-plain mode spec as the LRW standardization process is not about to be finished any time soon; will be reintroduced when a normative documentation is released by SISWG. Extended introduction text. Thanks to Sarah Dean for providing valuable feedback with respect to the AFsplit specification.
1.1.1	08.12.2008 Clemens Fruhwirth <clemens@endorphin.org> Clarify IV reference point for decrypt/encrypt. Thanks to Michael Gorven for this suggestion.
1.2	11.04.2011 Milan Broz <mbroz@redhat.com> Fix hash block size/digest size AF comment. Clarify master key digest iteration count. Add XTS mode reference. Some minor typo fixes. Add reference to NIST SP 800-132.
1.2.1	16.10.2011 Milan Broz <mbroz@redhat.com> Mention detached header. Typo correction and some editing for punctuation, grammar and clarity. Thanks to Karl O. Pinc <kop@meme.com>.
1.2.2	4.5.2016 Milan Broz <mbroz@redhat.com> Clarification of fixed sector size and keyslots alignment.
1.2.3	20.1.2018 Milan Broz <gmazyland@gmail.com> Fixed links to inline document sources.

Introduction

LUKS[1] is short for "Linux Unified Key Setup". It has initially been developed to remedy the unpleasantness a user experienced that arise from deriving the encryption setup from changing user space, and forgotten command line arguments. The result of this changes are an unaccessible encryption storage. The reason for this to happen was, a unstandardised way to read, process and set up encryption keys, and if the user was unlucky, he upgraded to an incompatible version of user space tools that needed a good deal of knowledge to use with old encryption volumes.

LUKS has been invented to standardise key setup. But the project became bigger as anticipated, because standards creation involves decision making, which in turn demands for a justification of these decision. An overspring of this effort can be found as TKS1 [Fru04], a design model for secure key processing from entropy-weak sources[2]. LUKS is also treaded extensivly in Chapters 5 and 6 in "New Methods in Hard Disk Encryption", which provides a theoretic base for the security of PBKDF2 passwords and anti-forensic information splitting. See [Fru05b].

LUKS is the proof-of-concept implementation for TKS1. In LUKS 1.0, the implementation switched to TKS2, a varient of TKS1, introduced in [Fru05b]. Additionally to the security provided by the TKS1 model, LUKS gives the user the ability to associate more than one password with an encrypted partition. Any of these passwords can be changed or revoked in a secure manner.

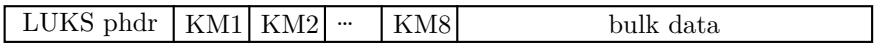
This document specifies the structure, syntax and semantic of the partition header and the key material. The LUKS design can be used with any cipher or cipher mode, but for compatibility reasons, LUKS standarises cipher names and cipher modes.

While the reference implementation is using dm-crypt, Linux' kernel facility for bulk data encryption, it is not tied to it in any particular way.

Next to the reference implementation which works on Linux, there is a Windows implementation named LibreCrypt (based on original FreeOTFE project by Sarah Dean).

1 Overview

A rough overall disk layout follows:



A LUKS partition starts with the LUKS partition header (phdr) and is followed by key material (labelled KM1, KM2 …KM8 in figure). After the key material, the bulk data is located, which is encrypted by the master key. The phdr contains information about the used cipher, cipher mode, the key length, a uuid and a master key checksum.

Also, the phdr contains information about the key slots. Every key slot is associated with a key material section after the phdr. When a key slot is active, the key slot stores an encrypted copy of the master key in its key

[1] This document describes version 1 of the LUKS on-disk format.

[2] such as a user password

material section. This encrypted copy is locked by a user password. Supplying this user password unlocks the decryption for the key material, which stores the master key. The master key in turn unlocks the bulk data. For a key slot, all parameters how to decrypt its key material with a given user password are stored in the phdr (f.e. salt, iteration depth).

A partition can have as many user passwords as there are key slots. To access a partition, the user has to supply only one of these passwords. If a password is changed, the old copy of the master key encrypted by the old password must be destroyed. Peter Gutmann has shown in [Gut96], how data destruction shall be done to maximise the chance, that no traces are left on the disk. Usually the master key comprises only 16 or 32 bytes. This small amount of data can easily be remapped as a whole to a reserved area. This action is taken by modern hard disk firmware, when a sector is likely to become unreadable due to mechanical wear. The original sectors become unaccessible and any traces of key data can't be purged if necessary.

To counter this problem, LUKS uses the anti-forensic information splitter to artificially inflate the volume of the key, as with a bigger data set the probability that the whole data set is remapped drops exponentially. The inflated encrypted master key is stored in the key material section. These sections are labelled as "KMx" in the figure above.

2 Prerequisites

2.1 Block encryption system

Instead of using cipher implementations like AES or Twofish internally, LUKS reuses the block encryption facility used for the bulk data. The following syntax is used in the pseudocode:

```
enc-data = encrypt(cipher-name, cipher-mode, key, original,
                   original-length)
original = decrypt(cipher-name, cipher-mode, key, enc-data,
                   original-length)
```

If the encryption primitive requires a certain block size, incomplete blocks are padded with zero. The zeros are stripped upon decryptions.[3]

2.2 Cryptographic hash

A cryptographic hash is necessary for the following two prerequisites. In PBKDF2 a pseudo-random function is needed, and for AFsplitting a diffusion function is needed. The pseudo-random function needs to be parameterisable, therefore the hash function is used in a HMAC setup [BCK97].

The following syntaxes may omit the *hash-spec* parameter, because the following pseudo code does not need a great variation of this parameter. The parameter can be obtained from the partition header and will not change, once initialised.

[3]These primitives are also used for key material en/decryption. The key material is always aligned to sector boundaries. If the block size of the underlaying encryption primitive is larger than one sector, the pseudocode of section 4.1 has to be changed respectively.

2.3 PBKDF2

LUKS needs to process password from entropy-weak sources like keyboard input. PKCS #5's password based key derive function 2 (PBKDF2) has been defined for the purpose to enhance the security properties of entropy-weak password, see [Kal97]. Therefore, LUKS depends on a working implementation of PBKDF2. LUKS uses SHA1 per default as the pseudorandom function (PRF) but any other hash function can be put in place by setting the *hash-spec* field. In the pseudo code, the following syntax is used:

```
result = PBKDF2(password,
                salt,
                iteration-count,
                derived-key-length)
```

Notice that the result of this function depends on the current setting of *hash-spec* but the parameter has been omitted. Think of *hash-spec* as sort of an environment variable.

2.4 AF-Splitter

LUKS uses anti-forensic information splitting as specified in [Fru05b]. The underlaying diffusion function shall be SHA1 for the reference implementation, but can be changed exactly as described in the remarks above. A C reference implementation using SHA1 is available from [Fru05a].

```
split-material = AFsplit(unsplit-material, length, stripes)
unsplit-material = AFmerge(split-material, length, stripes)
```

Notice that the result of AFsplit, *split-material*, is *stripes*-times as large as the original, that is $length * stripes$ bytes. Notice that the *length* parameter is the length of the original content and not the length of the *split-material* array.

When D is the unsplit material, H is a diffusion function, and n is the stripe number, AFsplit returns $s_1, s_2 \ldots s_n$ where $s_1 \ldots s_{n-1}$ are randomly chosen while s_n is computed according to:

$$d_0 = 0 \tag{1}$$

$$d_k = H(d_{k-1} \oplus s_k) \tag{2}$$

$$s_n = d_{n-1} \oplus D \tag{3}$$

To reverse the process, AFmerge computes d_{n-1} and recovers D from:

$$D = d_{n-1} \oplus s_n \tag{4}$$

2.4.1 H_1

H_1 is a hash function with an underlaying hash function P.[4] H_1 can operate on a variable amount of data, hence it is constructed for hash extension. The underlaying hash function is SHA1, we use it solely in LUKS. We use $|P|$ to denote the digest size of P, for SHA1 it is 160 bit.

[4] H_1's function definition stems from an implementation error that I'm responsible for. Do not try to analyse it, the structure given here is specified according to this implementation error and hence is a mistake itself. H_2 is the correct hash extension as originally envisioned.

The input to $H_1(d)$, namely d, is partitioned into individual data hunks. The partitioning repeatedly takes a data vector with the size $|P|$ as d_i with the final block (possibly shorter than $|P|$) d_n. The transformation happens as follows:

$$p_i = P(i \,||\, d_i) \tag{5}$$

The end of the last block p_n is cropped, so that its length is $|d_n|$. The integer i has to be delivered to the hash as an unsigned 32-bit integer in big-endien format.

2.4.2 H_2

All remarks for H_1 apply, except

$$p_i = P(i \,||\, d) \tag{6}$$

Notice the missing subscript of d in contrast to (5). This version will be used in future LUKS revisions.[5]

3 The partition header

3.1 Version 1

The LUKS partition header has the layout as described in Figure 1. It starts at sector 0 of the partition[6]. LUKS uses 3 primitive data types in its header,

- unsigned integer, 16 bit, stored in big endian
- unsigned integer, 32 bit, stored in big endian
- char[], a string stored as null terminated sequence of 8-bit characters[7]
- byte[], a sequence of bytes, treated as binary.

Further, there is an aggregated data type *key slot*, which elements are described in Figure 2.

A reference definition as C struct for phdr is available in the appendix.

3.2 Forward compatibility

LUKS' forward compatibility centers around the on-disk format. Future versions are required to be able to correctly interpret older phdr versions. Future versions are not required to be able to generate old versions of the phdr.

[5]The transition has not happend yet. It is likely that the transition will occour in conjunction with a version number bump to Version 2. Do not use H_2 until then.

[6]Since cryptsetup 1.4 the LUKS partition header can be detached and stored on another device. In this specific case the payload offset can be zero.

[7]Also known as C string.

start offset	field name	length	data type	description
0	magic	6	byte[]	magic for LUKS partition header, see LUKS_MAGIC
6	version	2	uint16_t	LUKS version
8	cipher-name	32	char[]	cipher name specification
40	cipher-mode	32	char[]	cipher mode specification
72	hash-spec	32	char[]	hash specification
104	payload-offset	4	uint32_t	start offset of the bulk data (in 512 bytes sectors)
108	key-bytes	4	uint32_t	number of key bytes
112	mk-digest	20	byte[]	master key checksum from PBKDF2
132	mk-digest-salt	32	byte[]	salt parameter for master key PBKDF2
164	mk-digest-iter	4	uint32_t	iterations parameter for master key PBKDF2
168	uuid	40	char[]	UUID of the partition
208	key-slot-1	48	key slot	key slot 1
256	key-slot-2	48	key slot	key slot 2
...	...	...	...	...
544	key-slot-8	48	key slot	key slot 8
592	total phdr size			

Figure 1: PHDR layout

offset	field name	length	data type	description
0	active	4	unit32_t	state of keyslot, enabled/disabled
4	iterations	4	uint32_t	iteration parameter for PBKDF2
8	salt	32	byte[]	salt parameter for PBKDF2
40	key-material-offset	4	uint32_t	start sector of key material
44	stripes	4	uint32_t	number of anti-forensic stripes

Figure 2: key slot layout

A LUKS implementation encountering a newer phdr version should not try to interpret it, and return an error. Of course, an error should be returned, if the phdr's magic is not present.

4 LUKS operations

4.1 Initialisation

The initialisation process takes a couple of parameters. First and most important, the master key. This key is used for the bulk data. This key must be created from an entropy strong (random) source, as the overcoming of entropy weak keys is one of LUKS' main objectives. For the following remarks, the pseudo code is available as Figure 3.

Further, the user specifices the cipher setup details that are stored in the *cipher-name* and *cipher-mode* fields. Although no LUKS operation manipulates these two strings, it is likely that the LUKS implementation will have to convert it into something suitable for the underlaying cipher system, as the interface is not likely to be as ideal as described in Section 2.1.

The overall disk layout depends on the length of the key material sections following the phdr. While the phdr is always constant in size, the key material section size depends on the length of the master key and the number of stripes used by the anti-forensic information splitter. The exact disk layout is generated by computing the size for the phdr and a key material section in sectors rounded up. Then the disk is filled sector-wise by phdr first, and following key material section 1 till key material section 8. After the eight key material section, the bulk data starts.

After determining the exact key layout and boundaries between phdr, key material and bulk data, the key material locations are written into the key slot entries in the phdr. The information about the bulk data start is written into the *payload*-offset field of the phdr. These values will not change during the lifetime of a LUKS partition and are simply cached for safety reasons as a miscalculation of these values can cause data corruption.[8]

The master key is checksummed, so a correct master key can be detected. To future-proof the checksumming, a hash is not only applied once but multiple times. In fact, the PBKDF2 primitive is reused. The master key is feed into the PBKDF2 process as if it were a user password. After the iterative hashing, the random chosen salt, the iteration count[9] and result are stored in the phdr.

Although everything is correctly initialised up to this point, the initialisation process should not stop here. Without an active key slot the partition is useless. At least one key slot should be activated from the master key still in memory.

[8] E.g. an incorrect bulk data offset can lead to overwritten key material and an incorrect key offset can result in overwritten encrypted data.

[9] Master key iteration count was set to 10 in previous revisions of LUKS. For new devices the iteration count should be determined by benchmarking with suggested minimum of 1000 iterations.

```
masterKeyLength = defined by user
masterKey = generate random vector, length: masterKeyLength

phdr.magic           = LUKS_MAGIC
phdr.version         = 1
phdr.cipher-name     = as supplied by user
phdr.cipher-mode     = as supplied by user
phdr.key-bytes       = masterKey
phdr.mk-digest-salt  = generate random vector,
                       length: LUKS_SALTSIZE

// benchmarked according to user input
// (in older versions fixed to 10)
phdr.mk-digest-iteration-count = as above

phdr.mk-digest = PBKDF2(masterKey,
                        phdr.mk-digest-salt,
                        phdr.mk-digest-iteration-count,
                        LUKS_DIGESTSIZE)
stripes = LUKS_STRIPES or user defined

// integer divisions, result rounded down
baseOffset = (size of phdr) / LUKS_SECTOR_SIZE + 1
keyMaterialSectors = (stripes * masterKeyLength) /
                     LUKS_SECTOR_SIZE + 1

for each keyslot in phdr as ks {
    // Align keyslot up to multiple of LUKS_ALIGN_KEYSLOTS
    baseOffset = round_up(baseOffset, LUKS_ALIGN_KEYSLOTS)
    ks.active = LUKS_KEY_DISABLED
    ks.stripes = stripes
    ks.key-material-offset = baseOffset
    baseOffset = baseOffset + keyMaterialSectors
}

phdr.payload-offset = baseOffset
phdr.uuid = generate uuid

write phdr to disk
```

Figure 3: Pseudo code for partition initialisation

4.2 Adding new passwords

To add a password to a LUKS partition one has to possess an unencrypted copy of the master key; either initialization must still be in progress or the master key must be recovered using a valid password to an existing key slot. The latter operation is sketched in Figure 4.

Assuming we have a good copy of the master key in memory the next steps are to fetch a salt from a random source and to choose a password iteration count[10]. This information is written into a free – that is disabled – key slot of the phdr.

The user password is entered and processed by PBKDF2. The master key is then split by the AFsplitter into a number of stripes. The number of stripes is determined by the *stripes* field already stored in the key slot. The split result is written into the key material section, but encrypted. The encryption uses the same cipher setup as the bulk data (cipher type, cipher mode, ...), but while for the bulk data the master key is used, the key material section is keyed by the result of the PBKDF2.

4.3 Master key recovery

To access the payload bulk data, the master key has to be recovered. Compare the pseudo code in Figure 5.

First, the user supplies a password. Then the password is processed by PBKDF2 for every active key slot individually and an attempt is made to recover the master key. The recovery is successful, when a master key candidate correctly checksums against the master key checksum stored in the phdr. Before this can happen, the master key candidate is read from storage, decrypted and after decryption processed by the anti-forensic information splitter in reverse gear, that is AFmerge.

When the checksumming of the master key succeeds for one key slot, the correct user key was given and the partition is successfully opened.

4.4 Password revocation

The key material section is wiped according to Peter Gutmann's data erasure principals [Gut96]. To wipe the sectors containing the key material, start from the sector as recorded in key slot's *key-material-offset* field, and proceed for *phdr.key-bytes * ks.stripes* bytes.

[10] The iteration count should be determined by benchmarking with suggested minimum of 1000 iterations.

```
masterKey = must be available, either because it is still in
            memory from initialisation or because it has been
            recovered by a correct password
masterKeyLength = phdr.key-bytes

emptyKeySlotIndex = find inactive key slot index in phdr by
                    scanning the keyslot.active field for
                    LUKS_KEY_DISABLED.

keyslot ks = phdr.keyslots[emptyKeySlotIndex]

PBKDF2-IterationsPerSecond = benchmark system
ks.iteration-count = PBKDF2-IterationsPerSecond *
                     intendedPasswordCheckingTime (in seconds)

ks.salt = generate random vector, length: LUKS_SALTSIZE

splitKey = AFsplit(masterKey,         // source
                   masterKeyLength,   // source length
                   ks.stripes)        // number of stripes

splitKeyLength = masterKeyLength * ks.stripes

pwd = read password from user input
pwd-PBKDF2ed = PBKDF2(password,
                      ks.salt,
                      ks.iteration-count
                      masterKeyLength) // key size is the same
                                       // as for the bulk data

encryptedKey = encrypt(phdr.cipher-name,  // cipher name
                       phdr.cipher-mode,  // cipher mode
                       pwd-PBKDF2ed,      // key
                       splitKey,          // content
                       splitKeyLength)    // content length

write to partition(encryptedKey,           // source
                   ks.key-material-offset, // sector number
                   splitKeyLength)         // length in bytes

ks.active = LUKS_KEY_ACTIVE     // mark key as active in phdr

update keyslot ks in phdr
```

Figure 4: Pseudo code for key creation

```
read phdr from disk
check for correct LUKS_MAGIC and compatible version number

masterKeyLength = phdr.key-bytes
pwd = read password from user input

for each active keyslot in phdr do as ks {
  pwd-PBKDF2ed = PBKDF2(pwd, ks.salt, ks.iteration-count
                        masterKeyLength)
  read from partition(encryptedKey,          // destination
              ks.key-material-offset,        // sector number
              masterKeyLength * ks.stripes) // number of bytes

  splitKey = decrypt(phdr.cipherSpec,  // cipher spec.
                     pwd-PBKDF2ed,     // key
                     encryptedKey,     // content
                     encrypted)        // content length

  masterKeyCandidate = AFmerge(splitKey, masterkeyLength,
                               ks.stripes)

  MKCandidate-PBKDF2ed = PBKDF2(masterKeyCandidate,
                                phdr.mk-digest-salt,
                                phdr.mk-digest-iter,
                                LUKS_DIGEST_SIZE)
  if equal(MKCandidate-PBKDF2ed, phdr.mk-digest) {
        break loop and return masterKeyCandidate as
        correct master key
  }
}
return error, password does not match any keyslot
```

Figure 5: Pseudo code for master key recovery

4.5 Password changing

The password changing is a synthetic operating of ”master key recovery”, ”new password adding”, and ”old password revocation”.

5 Constants

All strings and characters are to be encoded in ASCII.

Symbol	Value	Description
LUKS_MAGIC	{’L’,’U’,’K’,’S’, 0xBA,0xBE}	partition header starts with magic
LUKS_DIGESTSIZE	20	length of master key checksum
LUKS_SALTSIZE	32	length of the PBKDF2 salts
LUKS_NUMKEYS	8	number of key slots
LUKS_KEY_DISABLED	0x0000DEAD	magic for disabled key slot in key-block[i].active
LUKS_KEY_ENABLED	0x00AC71F3	magic for enabled key slot in key-block[i].active
LUKS_STRIPES	4000	number of stripes for AFsplit. See [Fru05b] for rationale.
LUKS_ALIGN_KEYSLOTS	4096	Default alignment for keyslot in bytes.[11]
LUKS_SECTOR_SIZE	512	LUKS version 1 always use sector of fixed size 512 bytes.

[11] Former version of this definition mentioned alignment to LUKS_SECTOR_SIZE only while reference implementation always used alignment to 4096 bytes. It is also required on modern drives with 4096 bytes sectors.

Bibliography

[BCK97] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. The HMAC papers. `https://cseweb.ucsd.edu/~mihir/papers/hmac.html`, 1996-1997.

[Fru04] Clemens Fruhwirth. TKS1 - An anti-forensic, two level, and iterated key setup scheme. `https://www.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/TKS1-draft.pdf`, 2004.

[Fru05a] Clemens Fruhwirth. Fruhwirth's Cryptography Website. `http://clemens.endorphin.org/cryptography`, 2005.

[Fru05b] Clemens Fruhwirth. New methods in hard disk encryption. `http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf`, 2005.

[Gut96] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. `https://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`, 1996.

[Kal97] Burt Kaliski. RFC 2898; PKCS #5: Password-Based Cryptography Specification Version 2.0. `http://www.faqs.org/rfcs/rfc2898.html`, 1996-1997.

[TBBC10] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for password-based key derivation, part 1: Storage applications. NIST SP 800-132, `https://csrc.nist.gov/publications/detail/sp/800-132/final`, 2010.

A PHDR as C struct

```
#define LUKS_MAGIC_L            6
#define LUKS_CIPHERNAME_L       32
#define LUKS_CIPHERMODE_L       32
#define LUKS_HASHSPEC_L         32
#define UUID_STRING_L           40

struct luks_phdr {
        char            magic[LUKS_MAGIC_L];
        uint16_t        version;
        char            cipherName[LUKS_CIPHERNAME_L];
        char            cipherMode[LUKS_CIPHERMODE_L];
        char            hashSpec[LUKS_HASHSPEC_L];
        uint32_t        payloadOffset;
        uint32_t        keyBytes;
        char            mkDigest[LUKS_DIGESTSIZE];
        char            mkDigestSalt[LUKS_SALTSIZE];
        uint32_t        mkDigestIterations;
        char            uuid[UUID_STRING_L];

        struct {
                uint32_t active;

                /* parameters for PBKDF2 processing */
                uint32_t passwordIterations;
                char     passwordSalt[LUKS_SALTSIZE];

                /* parameters for AF store/load */
                uint32_t keyMaterialOffset;
                uint32_t stripes;
        } keyblock[LUKS_NUMKEYS];
};
```

B Cipher and Hash specification registry

Even if the *cipher-name* and *cipher-mode* strings are not interpreted by any LUKS operation, they must have the same meaning for all implementations to achieve compatibility among different LUKS-based implementations. LUKS has to ensure that the underlaying cipher system can utilise the cipher name and cipher mode strings, and as these strings might not always be native to the cipher system, LUKS might need to map them into something appropriate.

Valid cipher names are listed in Table 1.

Valid cipher modes are listed in Table 2. By contract, cipher modes using IVs and tweaks must start from the all-zero IV/tweak. This applies for all calls to the encrypt/decrypt primitives especially when handling key material. Further, these IVs/tweaks cipher modes usually cut the cipher stream into independent blocks by reseeding tweaks/IVs at sector boundaries. The all-zero

cipher name	**normative document**
aes	Advanced Encryption Standard - FIPS PUB 197
twofish	Twofish: A 128-Bit Block Cipher - `https://www.schneier.com/paper-twofish-paper.html`
serpent	`https://www.cl.cam.ac.uk/~rja14/serpent.html`
cast5	RFC 2144
cast6	RFC 2612

Table 1: Valid cipher names

mode	**description**
ecb	The cipher output is used directly.
cbc-plain	The cipher is operated in CBC mode. The CBC chaining is cut every sector, and reinitialised with the sector number as initial vector (converted to 32-bit and to little-endian). This mode is specified in [Fru05b], Chapter 4.
cbc-essiv:*hash*	The cipher is operated in ESSIV mode using *hash* for generating the IV key for the original key. For instance, when using sha256 as hash, the cipher mode spec is "cbc-essiv:sha256". ESSIV is specified in [Fru05b], Chapter 4.
xts-plain64	`http://grouper.ieee.org/groups/1619/email/pdf00086.pdf`, *plain64* is 64-bit version of plain initial vector

Table 2: Valid cipher modes

hash-spec string	**normative document**
sha1	RFC 3174 - US Secure Hash Algorithm 1 (SHA1)
sha256	SHA variant according to FIPS 180-2
sha512	SHA variant according to FIPS 180-2
ripemd160	`http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html`

Table 3: Valid hash specifications

IV/tweak requirement for the first encrypted/decrypted block is equivalent to the requirement that the first block is defined to rest at sector 0.

Table 3 lists valid hash specs for *hash-spec* field. A compliant implementation does not have to support all cipher, cipher mode or hash specifications.