# Disk Encryption

From ArchWiki

This article discusses common techniques available in Arch Linux for cryptographically protecting a logical part of a storage disk (folder, partition, whole disk, ...), so that all data that is written to it is automatically encrypted, and decrypted on-the-fly when read again.

"Storage disks" in this context can be your computer's hard drive(s), external devices like USB sticks or DVD's, as well as *virtual* storage disks like loop-back devices or cloud storage *(as long as Arch Linux can address it as a block device or filesystem)*.

**Summary**

Transparent encryption/decryption software

**Related**

dm-crypt with LUKS

eCryptfs

TrueCrypt

EncFS

Mount encrypted volumes in parallel

## Contents



## Why use encryption?

Disk encryption ensures that files are always stored on disk in an encrypted form. The files only become available to the operating system and applications in readable form while the system is running and unlocked by a trusted user. Reading the encrypted sectors without permission will return garbled random-looking data instead of the actual files.

For example, this can prevent unauthorized viewing of the data when the computer or hard-disk is:

- located in a place to which non-trusted people might gain access while you're away
- lost or stolen, as with laptops, netbooks or external storage devices
- in the repair shop
- discarded after its end-of-life

In addition, disk encryption can also be used to add some security against unauthorized attempts to tamper with your operating system. For example, the installation of keyloggers or Trojan horses by attackers who can gain physical access to the system while you're away.

**Warning:** Disk encryption does **not** protect your data from all threats.

Including the following:

- Attackers who can break into your system (e.g. over the Internet) while it is running and after you've already unlocked and mounted the encrypted parts of the disk.
- Attackers who are able to gain physical access to the computer while (or very shortly after) it is running, and have the resources to perform a cold boot attack.
- A government entity, which not only has the resources to easily pull off the above attacks, but also may simply force you to give up your keys/passphrases using various techniques of coercion. In most non-democratic countries around the world, as well as in the USA and UK, it is legal for law enforcement agencies to do so if they have suspicions that you might be hiding something of interest.

A very strong disk encryption setup (e.g. full system encryption with no plaintext boot partition and authenticity checking) is required to stand a chance against professional attackers, who are able to tamper with your system before you use it. And even then it is doubtful whether it can really prevent all types of tampering (e.g. hardware keyloggers). The best remedy might be hardware-based full disk encryption (e.g. Trusted Computing).

**Warning:** Disk encryption also won't protect you against someone simply wiping your disk. Regular backups are recommended to keep your data safe.

### Data encryption vs system encryption

See the Wikipedia article on this subject for more information: **Disk encryption**

**Data encryption**, defined as encrypting only the user's data itself (often located within the `/home` directory, or on removable media like a data DVD), is the simplest and least intrusive use of disk encryption, but has some significant drawbacks.
In modern computing systems, there are many background processes that may cache/store information about user data or parts of the data itself in non-encrypted areas of the hard drive, like:

- swap partitions
  - *(potential remedy: disable swapping)*
- `/tmp` (temporary files created by user applications)
  - *(potential remedies: avoid such applications; mount `/tmp` inside a ramdisk)*
- `/var` (log files and databases and such; for example, mlocate stores an index of all file names in `/var/lib/mlocate/mlocate.db` )

In addition, mere data encryption will leave the system vulnerable to offline system tampering attacks *(see warnings above)*.

**System encryption**, defined as the encryption of the operating system *and* user data, helps to address some of the inadequacies of data encryption.

Benefits:

- Preventing unauthorized physical access to operating system files *(but see warning above)*
- Preventing unauthorized physical access to private data that may cached by the system.

Disadvantages:

- unlocking/locking of the encrypted parts of the disk can no longer coincide with user login/logout, because now the unlocking already needs to happen before or during boot

In practice, there's not always a clear line between data encryption and system encryption, and many different compromises and customized setups are possible.

In any case, disk encryption should only be viewed as an adjunct to the existing security mechanisms of the operating system - focused on securing offline physical access, while relying on *other* parts of the system to provide things like network security and user-based access control.

## Available methods

All disk encryption methods operate in such a way that even though the disk actually holds encrypted data, the operating system and applications "see" it as the corresponding normal readable data as long as the cryptographic container (i.e. the logical part of the disk that holds the encrypted data) has been "unlocked" and mounted.

For this to happen, some "secret information" (usually in the form of a keyfile and/or passphrase) needs to be supplied by the user, from which the actual encryption key can be derived (and stored in the kernel keyring for the duration of the session).

If you are completely unfamiliar with this sort of operation, please first read the #How the encryption works section below.

The available disk encryption methods can be separated into two types by their layer of operation:

### Stacked filesystem encryption

Stacked filesystem encryption solutions are implemented as a layer that stacks on top of an existing filesystem, causing all files written to an encryption-enabled folder to be encrypted on-the-fly before the underlying filesystem writes them to disk, and decrypted whenever the filesystem reads them from disk. This way, the files are stored in the host filesystem in encrypted form (meaning that their contents, and usually also their file/folder names, are replaced by random-looking data of roughly the same length), but other than that they still exist in that filesystem as they would without encryption, as normal files / symlinks / hardlinks / etc.

The way it is implemented, is that to unlock the folder storing the raw encrypted files in the host filesystem ("lower directory"), it is mounted (using a special stacked pseudo-filesystem) onto itself or optionally a different location ("upper directory"), where the same files then appear in readable form - until it is unmounted again, or the system is turned off.

Available solutions in this category are:

eCryptfs
: ...

EncFS
: ...

### Block device encryption

Block device encryption methods, on the other hand, operate *below* the filesystem layer and make sure that everything written to a certain block device (i.e. a whole disk, or a partition, or a file acting as a virtual loop-back device) is encrypted. This means that while the block device is offline, its whole content looks like a large blob of random data, with no way of determining what kind of filesystem and data it contains. Accessing the data happens, again, by mounting the protected container (in this case the block device) to an arbitrary location in a special way.

The following "block device encryption" solutions are available in Arch Linux:

loop-AES
: *loop-AES is a descendant of cryptoloop and is a secure and fast solution to system encryption.*
*However loop-AES is considered less user-friendly than other options as it requires non-standard kernel support.*

dm-crypt + LUKS
: *dm-crypt is the standard device-mapper encryption functionality provided by the Linux kernel. It can be used directly by those who like to have full control over all aspects of partition and key management.*
*LUKS is an additional convenience layer which stores all of the needed setup information for dm-crypt on the disk itself and abstracts partition and key management in an attempt to improve ease of use.*

TrueCrypt
: ...

For practical implications of the chosen layer of operation, see the comparison table below, as well as [1] (http://ecryptfs.sourceforge.net/ecryptfs-faq.html#compare) .

### Comparison table

| ***summary*** | **Loop-AES** | **dm-crypt + LUKS** | **Truecrypt** | **eCryptfs** | **EncFs** |
|---|---|---|---|---|---|
| type | block device encryption | | | stacked filesystem encryption | |
| main selling points | longest-exiting one; possibly the fastest; works on legacy systems | de-facto standard for block device encryption on Linux; very flexible | very portable, well-polished, self-contained solution | slightly faster than EncFS; individual encrypted files portable between systems | easiest one to use; supports non-root administration |
| availability in Arch Linux | must manually compile custom kernel | *kernel modules:* already shipped with default kernel; *tools:* `device-mapper` (https://www.archlinux.org/packages/?name=device-mapper) , `cryptsetup` (https://www.archlinux.org/packages/?name=cryptsetup) [core] | `truecrypt` (https://www.archlinux.org/packages/?name=truecrypt) [extra] | *kernel module:* already shipped with default kernel; *tools:* `ecryptfs-utils` (https://www.archlinux.org/packages/?name=ecryptfs-utils) [community] | `encfs` (https://www.archlinux.org/packages/?name=encfs) [community] |
| license | GPL | GPL | custom[1] | GPL | GPL |

| ***basic classification*** | **Loop-AES** | **dm-crypt + LUKS** | **Truecrypt** | **eCryptfs** | **EncFs** |
|---|---|---|---|---|---|
| encrypts... | whole block devices | | | files | |
| container for encrypted data may be... | ▪ a disk or disk partition<br>▪ a file acting as a virtual partition | | | ▪ a directory in an existing file system | |
| relation to filesystem | operates below the filesystem layer - doesn't care whether the content of the encrypted block device is a filesystem, a partition table, a LVM setup, or anything else | | | adds an additional layer to an existing filesystem, to automatically encrypt/decrypt files whenever they're written/read | |
| encryption implemented in... | kernelspace | kernelspace | kernelspace | kernelspace | userspace *(using FUSE)* |
| cryptographic metadata stored in... | ? | In LUKS Header | ? | header of each encrypted file | control file at the top level of each EncFs container |
| wrapped encryption key stored in... | ? | In LUKS Header | ? | key file that can be stored anywhere | control file at the top level of each EncFs container |

| ***practical implications*** | **Loop-AES** | **dm-crypt + LUKS** | **Truecrypt** | **eCryptfs** | **EncFs** |
|---|---|---|---|---|---|
| file metadata (number of files, dir structure, file sizes, permissions, mtimes, etc.) is encrypted | ✔ | | | ✖ *(file and dir names can be encrypted though)* | |
| can be used to encrypt whole hard drives (including partition tables) | ✔ | | | ✖ | |
| can be used to encrypt swap space | ✔ | | | ✖ | |
| can be used without pre-allocating a fixed amount of space for the encrypted data container | ✖ | | | ✔ | |
| can be used to protect existing filesystems without block device access, e.g. NFS or Samba shares, cloud storage, etc. | ✖[2] | | | ✔ | |
| allows offline file-based backups of encrypted files | ✖ | | | ✔ | |

| ***usability features*** | **Loop-AES** | **dm-crypt + LUKS** | **Truecrypt** | **eCryptfs** | **EncFs** |
|---|---|---|---|---|---|
| support for automounting on login | ? | ? | ? | ? | ✔ |
| support for automatic unmounting in case of inactivity | ? | ? | ? | ? | ✔ |
| non-root users can create/destroy containers for encrypted data | ✖ | ✖ | ✖ | ✖ | ✔ |
| provides a GUI | ✖ | ✖ | ✔ | ✖ | ✖ |

| *security features* | Loop-AES | dm-crypt + LUKS | Truecrypt | eCryptfs | EncFs |
|---|---|---|---|---|---|
| supported ciphers | AES | AES, Anubis, CAST5/6, Twofish, Serpent, Camellia, Blowfish, ... (every cipher the kernel Crypto API offers) | ? | AES, blowfish, twofish... | AES, Twofish |
| support for salting | ? | ✔ (with LUKS) | ✔ | ✔ | ? |
| support for cascading multiple ciphers | ? | ? | ✔ | ? | ✖ |
| support for key-slot diffusion | ? | ✔ (with LUKS) | ? | ? | ? |
| protection against key scrubbing | ✔ | ? | ? | ? | ? |
| support for multiple (independently revokable) keys for the same encrypted data | ? | ✔ (with LUKS) | ? | ? | ✖ |

| *performance features* | Loop-AES | dm-crypt + LUKS | Truecrypt | eCryptfs | EncFs |
|---|---|---|---|---|---|
| multithreading support | ? | ✔ [8] | ✔ | ? | ? |
| hardware-accelerated encryption support | ? | ✔ | ✔ | ✔ | ? |
| optimised handling of sparse files | ? | ? | ? | ✖ | ? |

| *block device encryption specific* | Loop-AES | dm-crypt + LUKS | Truecrypt |
|---|---|---|---|
| support for (manually) resizing the encrypted block device in-place | ? | ✔ | ✖ |

| *stacked filesystem encryption specific* | eCryptfs | EncFs |
|---|---|---|
| supported file systems | ext3, ext4, xfs (with caveats), jfs, nfs... | ? |
| ability to encrypt filenames | ✔ | ✔ |
| ability to *not* encrypt filenames | ✔ | ✔ |

| *compatibility & prevalence* | | Loop-AES | dm-crypt + LUKS | Truecrypt | eCryptfs | EncFs |
|---|---|---|---|---|---|---|
| supported Linux kernel versions | | 2.0 or newer | ? | ? | ? | 2.4 or newer |
| encrypted data can also be accessed from... | Windows | ✔ (with [3]) | ✔ (with [4]) | ✔ | ? | ? |
| | Mac OS X | ? | ? | ✔ | ? | ✔ [5] |
| | FreeBSD | ? | ? | ✖ | ? | ✔ [6] |
| used by | | ? | ▪ *Arch Linux installer* (system encryption)<br>▪ *Ubuntu alternate installer* (system encryption) | ? | ▪ *Ubuntu installer* (home dir encryption)<br>▪ *Chromium OS* (encryption of cached user data[7]) | ? |

## Preparation

### Choosing a setup

Which disk encryption setup is appropriate for you will depend on your goals (please read #Why_use_encryption? above) and system parameters.
Among other things, you will need to answer the following questions:

- What kind of "attacker" do you want to protect against?
  - Casual computer user trying to passively spy on your disk contents while your system is turned off / stolen / etc.
  - Professional cryptanalyst who can get repeated read/write access to your system before and after you use it
  - anything in between
- What encryption strategy shall be employed?
  - data encryption
  - system encryption
  - something in between
  - How should swap, `/tmp` , etc. be taken care of?

    - ignore, and hope no data is leaked
    - disable or mount as ramdisk
    - encrypt *(as part of full disk encryption, or separately)*

- How should encrypted parts of the disk be unlocked?
    - passphrase *(same as login password, or separate)*
    - keyfile *(e.g. on a USB stick, that you keep in a safe place or carry around with yourself)*
    - both

- *When* should encrypted parts of the disk be unlocked?
    - before boot
    - during boot
    - at login
    - manually on demand *(after login)*

- How should multiple users be accomodated?
    - not at all
    - using a shared passphrase/key
    - idependently issued and revocable passphrases/keys for the same encrypted part of the disk
    - separate encrypted parts of the disk for different users

Then you can go on to make the required technical choices (see #Available_methods above, and #How_the_encryption_works below), regarding:

- stacked filesystem encryption vs. blockdevice encryption
- key management
- cipher and mode of operation
- metadata storage
- location of the "lower directory" (in case of stacked filesystem encryption)
- ...

In practice, it could turn out something like:

***Example 1:*** simple data encryption (internal hard drive)

```
• a folder called "~/Private" in the user's home dir encrypted with EncFS
  ├──> encrypted versions of the files end up in ~/.Private
  └──> unlocked on demand with dedicated passphrase
```

***Example 2:*** simple data encryption (removable media)

```
• whole external USB drive encrypted with TrueCrypt
  └──> unlocked when attached to the computer
       (using dedicated passphrase + using ~/photos/2006-09-04a.jpg as covert keyfile)
```

***Example 3:*** partial system encryption

```
• each user's home directory encrypted with eCryptfs
  └──> unlocked on login, using login passphrase
• swap and /tmp partitions encrypted with dm-crypt+LUKS
  └──> using automatically generated per-session throwaway key
• indexing/caching of contents of /home by slocate (and similar apps) disabled
```

***Example 4:*** system encryption

```
• whole hard drive except /boot partition encrypted with dm-crypt+LUKS
  └──> unlocked during boot, using USB stick with keyfile (shared by all users)
```

***Example 5:*** paranoid system encryption

```
• whole hard drive encrypted with dm-crypt+LUKS
  └──> unlocked before boot, using dedicated passphrase + USB stick with keyfile
       (different one issued to each user - independently revocable)
• /boot partition located on aforementioned USB stick
```

Many many other combinations are of course possible. You should carefully plan what kind of setup will be appropriate for your system.

### Choosing a strong passphrase

When relying on a passphrase, it must be complex enough to not be easy to guess or break using brute-force attacks. The tenets of strong passphrases are based on *length* and *randomness*.
Refer to The passphrase FAQ (http://www.iusmentis.com/security/passphrasefaq/) for a detailed discussion, and especially consider the Diceware Passphrase (http://world.std.com/~reinhold/diceware.html) method.

Another aspect of the strength of the passphrase is that it must not be easily recoverable from other places. If you use the same passphrase for disk encryption as you use for your login password (useful e.g. to auto-mount the encrypted partition or folder on login), make sure that `/etc/shadow` either also ends up on an encrypted partition, or uses a strong hash algorithm (i.e. sha512/bcrypt, not md5) for the stored password hash (see SHA_password_hashes for more info).

### Preparing the disk

Before setting up disk encryption on a (part of a) disk, consider securely wiping it first. This consists of overwriting the entire drive or partition with a stream of zero bytes or random bytes, and is done for one or both of the following reasons:

- **prevent recovery of previously stored data**

  Disk encryption doesn't change the fact that individual sectors are only overwritten on demand, when the file system creates or modifies the data those particular sectors hold (see #How_the_encryption_works below). Sectors which the filesystem considers "not currently used" are not touched, and may still contain remnants of data from previous filesystems. The only way to make sure that all data which you previously stored on the drive can not be recovered, is to manually erase it.
  For this purpose it does not matter whether zero bytes or random bytes are used (although wiping with zero bytes will be much faster).

- **prevent disclosure of usage patterns on the encrypted drive**

  Ideally, the whole encrypted part of the disk should be indistinguishable from uniformly random data. This way, no unauthorized person can know which and how many sectors actually contain encrypted data - which may be a desirable goal in itself (as part of true confidentiality), and also serves as an additional barrier against attackers trying to break the encryption.
  For this purpose, wiping the disk using high-quality random data is crucial.

The second reason only makes sense in combination with block device encryptions, because in the case of stacked filesystem encryption the encrypted data is easily identifiable anyways (in the form of distinct encrypted files in the host filesystem). Also note that even if you only intend to encrypt a particular folder, you will have to erase the whole partition if you want to get rid of files that were previously stored in that folder in unencrypted form. If there are other folders on the same partition, you will have to back them up and move them back afterwards.

Once you have decided which kind of disk erasure you want to perform, refer to the Securely_wipe_disk article for technical instructions.

**Tip:** In deciding which method to use for secure erasure of a hard disk drive, remember that this will not need to be performed more than once for as long as the drive is used as an encrypted drive.

## How the encryption works

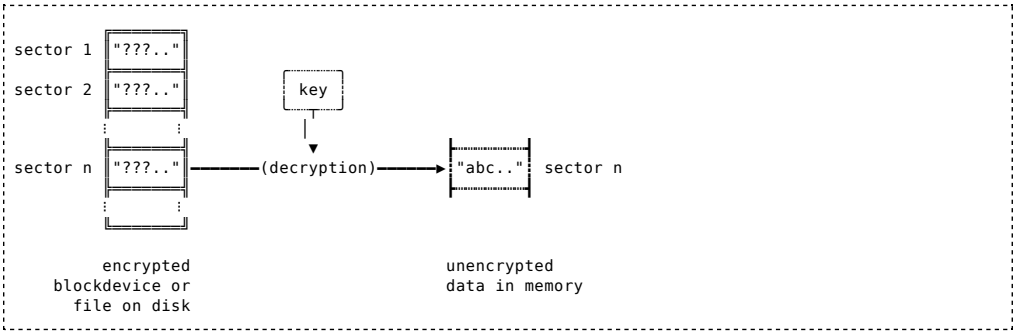
This section is intended as a high-level introduction to the concepts and processes which are at the heart of usual disk encryption setups.

It does not go into technical or mathematical details (consult the appropriate literature for that), but should provide a system administrator with a rough understanding of how different setup choices (especially regarding key management) can affect usability and security.

### Basic principle

For the purposes of disk encryption, each blockdevice (or individual file in the case of stacked filesystem encryption) is divided into **sectors** of equal lenght, for example 512 bytes (4,096 bits). The encryption/decryption then happens on a per-sector basis, so the n'th sector of the blockdevice/file on disk will store the encrypted version of the n'th sector of the original data.

Whenever the operating system or an application requests a certain fragment of data from the blockdevice/file, the whole sector (or sectors) that contains the data will be read from disk, decrypted on-the-fly, and temporarily stored in memory:




Similarly, on each write operation, all sectors that are affected must be re-encrypted complelety (while the rest of the sectors remain untouched).

### Keys, keyfiles and passphrases

In order to be able to de/encrypt data, the disk encryption system needs to know the unique secret "key" associated with it. This is a randomly generated byte string of a certain length, for example 32 bytes (256 bits).

Whenever the encrypted block device or folder in question is to be mounted, its corresponding key (called henceforth its "master key") must be retrieved - usually from one of the following locations:

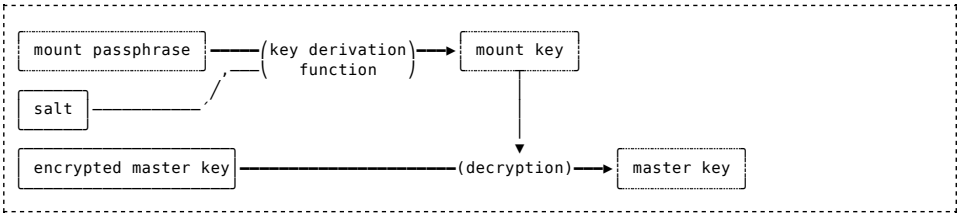
- ***stored in a plaintext keyfile***

  Simply storing the master key in a file (in readable form) is the simplest option. The file - called a "keyfile" - can be placed on a USB stick that you keep in a secure location and only connect to the computer when you want to mount the encrypted parts of the disk (e.g. during boot or login).

- ***stored in passphrase-protected form in a keyfile or on the disk itself***

  The master key (and thus the encrypted data) can be protected with a secret passphrase, which you will have to remember and enter each time you want to mount the encrypted block device or folder.

  A common setup is to apply so-called "key stretching" to the passphrase (via a "key derivation function"), and use the resulting enhanced passphrase as the mount key for decrypting the actual master key (which has been previously stored in encrypted form):

  The **key derivation function** (e.g. PBKDF2 or scrypt) is deliberately slow (it applies many iterations of a hash function, e.g. 1000 iterations of HMAC-SHA-512), so that brute-force attacks to find the passphrase are rendered infeasible. For the normal use-case of an authorized user, it will only need to be calculated once per session, so the small slowdown is not a problem.
  It also takes an additional blob of data, the so-called "**salt**", as an argument - this is randomly generated once during set-up of the disk encryption and stored unprotected as part of the cryptographic metadata. Because it will be a different value for each setup, this makes it infeasible for attackers to speed up brute-force attacks using precomputed tables for the key derivation function.

  The **encrypted master key** can be stored on disk together with the encrypted data. This way, the confidentiality of the encrypted data depends completely on the secret passphrase.

  Additional security can be attained by instead storing the encrypted master key in a keyfile on e.g. a USB stick. This provides **two-factor authentication**: Accessing the encrypted data now requires something only you *know* (the passphrase), and additionally something only you *have* (the keyfile).

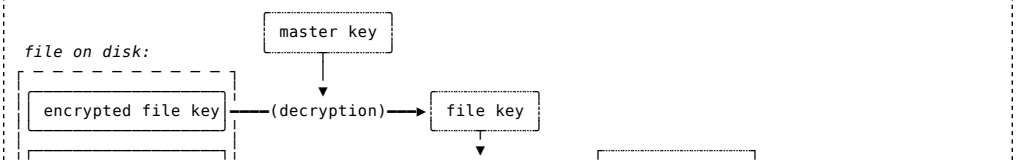
  Another way of achieving two-factor authentication is to augment the above key retrieval scheme to mathematically "combine" the passphrase with byte data read from one or more external files (located on a USB stick or similar), before passing it to the key derivation function.
  The files in question can be anything, e.g. normal JPEG images, which can be beneficial for #Plausible Deniability. They are still called "keyfiles" in this context, though.

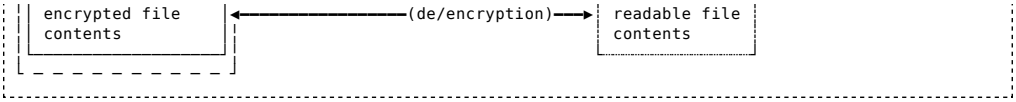
- ***randomly generated on-the-fly for each session***

  In some cases, e.g. when encrypting swap space or a `/tmp` partition, it is not necessary to keep a persistent master key at all. A new throwaway key can be randomly generated for each session, without requiring any user interaction. This means that once unmounted, all files written to the partition in question can never be decrypted again by *anyone* - which in those particular use-cases is perfectly fine.

After is has been derived, the master key is securely stored in memory (e.g. in a kernel keyring), for as long as the encrypted block device or folder is mounted.

It is usually not used for de/encrypting the disk data directly, though. For example, in the case of stacked filesystem encryption, each file can be automatically assigned its own encryption key. Whenever the file is to be read/modified, this file key first needs to be decrypted using the main key, before it can itself be used to de/encrypt the file contents:






In a similar manner, a separate key (e.g. one per folder) may be used for the encryption of file names in the case of stacked filesystem encryption.

In the case of block device encryption, ...

*Further reading:*

- Wikipedia:Passphrase
- Wikipedia:Key_(cryptography)
- Wikipedia:Key_management
- Wikipedia:Key_derivation_function

### Ciphers and modes of operation

The actual algorithm used for translating between pieces of unencrypted and encrypted data (so-called "plaintext" and "ciphertext") which correspond to each other with respect to a given encryption key, is called a "**cipher**".

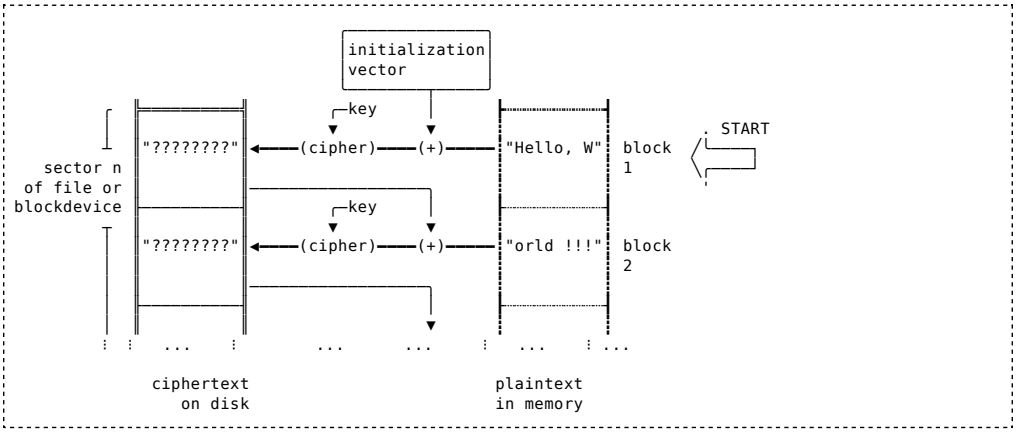
Disk encryption employs "block ciphers", which operate on fixed-length blocks of data, e.g. 16 bytes (128 bits). At the time of this writing, the predominantly used ones are:

| | **block size** | **key size** | **comment** |
|---|---|---|---|
| **AES** | 128 bits | 128, 192 or 256 bits | *approved by the NSA for protecting "SECRET" and "TOP SECRET" classified US-government information (when used with a key size of 192 or 256 bits)* |
| **Blowfish** | 64 bits | 32–448 bits | *one of the first patent-free secure ciphers that became publicly available, hence very well established on Linux* |
| **Twofish** | 128 bits | 128, 192 or 256 bits | *developed as successor of Blowfish, but has not attained as much widespread usage* |

Encrypting/decrypting a sector (see above) is achieved by dividing it into small blocks matching the cipher's block-size, and following a certain rule-set (a so-called "**mode of operation**") for how to consecutively apply the cipher to the individual blocks.

Simply applying it to each block separately without modification (dubbed the "*electronic codebook (ECB)*" mode) would not be secure, because if the same 16 bytes of plaintext always produce the same 16 bytes of ciphertext, an attacker could easily recognize patterns in the ciphertext that is stored on disk.

The most basic (and common) mode of operation used in practice is "*cipher-block chaining (CBC)*". When encrypting a sector with this mode, each block of plaintext data is combined in a mathematical way with the ciphertext of the previous block, before encrypting it using the cipher. For the first block, since it has no previous ciphertext to use, a special pre-generated data block stored with the sector's cryptographic metadata and called an "**initialization vector (IV)**" is used:




When decrypting, the procedure is reversed analogously.

One thing worth noting is the generation of the unique initialization vector for each sector. The simplest choice is to calculate it in a predictable fashion from a readily available value such as the sector number. However, this might allow an attacker with repeated access to the system to perform a so-called watermarking attack (http://en.wikipedia.org/wiki/Watermarking_attack) . To prevent that, a method called "Encrypted salt-sector initialization vector (**ESSIV**)" can be used to generate the initialization vectors in a way that makes them look completely random to a potential attacker.

There are also a number of other, more complicated modes of operation available for disk encryption, which already provide built-in security agains such attacks. Some can also additionally guarantee authenticity (see below) of the encrypted data.

*Further reading:*

- Wikipedia:Disk_encryption_theory
- Wikipedia:Block_cipher
- Wikipedia:Block_cipher_modes_of_operation

### Cryptographic metadata

### Data integrity/authenticity

*Further reading:*

- Wikipedia:Authenticated_encryption

### Plausible deniability

## Notes & References

1. ^ see http://www.truecrypt.org/legal/license
2. ^ well, a single file in those filesystems could be used as a container (virtual loop-back device!) but then one wouldn't actually be using the filesystem (and the features it provides) anymore
3. ^ CrossCrypt (http://www.scherrer.cc/crypt) - Open Source AES and TwoFish Linux compatible on the fly encryption for Windows XP and Windows 2000
4. ^ FreeOTFE (http://www.freeotfe.org) - supports Windows 2000 and later (for PC), and Windows Mobile 2003 and later (for PDA)
5. ^ see EncFs build instructions for Mac (http://www.arg0.net/encfs-mac-build)
6. ^ see http://www.freshports.org/sysutils/fusefs-encfs/
7. ^ see http://www.chromium.org/chromium-os/chromiumos-design-docs/protecting-cached-user-data
8. ^ http://kernelnewbies.org/Linux_2_6_38#head-49f5f735853f8cc7c4d89e5c266fe07316b49f4c