Disk encryption

Related articles

- **dm-crypt**
- **TrueCrypt**
- **eCryptfs**
- **EncFS**
- **gocryptfs**
- **Tomb**
- **tcplay**
- **GnuPG**
- **Self-Encrypting Drives**

This article discusses **disk encryption** software, which on-the-fly encrypts / decrypts data written to / read from a **block device**, **disk partition** or directory. Examples for block devices are hard drives, flash drives and DVDs.

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

Contents

[hide]

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. An unauthorized person looking at the disk contents directly, will only find 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 are 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 are away.

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

You will still be vulnerable to:

- Attackers who can break into your system (e.g. over the Internet) while it is running and after you have already unlocked and mounted the encrypted parts of the disk.
- Attackers who are able to gain physical access to the computer while it is running (even if you use a screenlocker), or very shortly *after* it was running, if they 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 may be 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 authenticity checking and no plaintext boot partition) 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 cannot prevent all types of tampering (e.g. hardware keyloggers). The best remedy might be **hardware-based full disk encryption** and **Trusted Computing**.

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

System data encryption

While encrypting only the user data itself (often located within the home directory, or on removable media like a data DVD), is the simplest and least intrusive method, it has some significant drawbacks. In modern computer systems, there are many background processes that may cache and store information about user data or parts of the data itself in non-encrypted areas of the hard drive, like:

- swap partitions
  - (potential remedies: disable swapping, or use **encrypted swap** as well)
- `/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`)

The solution is to encrypt both system and user data, preventing unauthorized physical access to private data that may be cached by the system. This however comes with the disadvantage that unlocking of the encrypted parts of the disk has to happen at boot time. Another benefit of system data encryption is that complicates install malware like **keyloggers** or rootkits for someone with physical access.

Available methods

This article or section needs expansion.

Reason: **Ext4**, **ZFS** and possible other filesystems offer (native) encryption. (Discuss in **Talk:Disk encryption#**)

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 also 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** and **EncFS**.

Cloud-storage optimized

If you are deploying stacked filesystem encryption to achieve zero-knowledge synchronization with third-party-controlled locations such as cloud-storage services, you may want to consider alternatives to eCryptfs and EncFS, since these are not optimized for transmission of files over the Internet. There are some solutions designed for this purpose instead:

- **gocryptfs**
- `cryptomator`AUR (multi-platform)
- `cryfs`

Note that some cloud-storage services offer zero-knowledge encryption directly through their own **client applications**.

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 **loop 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

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. The management of dm-crypt is done with the `cryptsetup` userspace utility. It can be used for the following types of block-device encryption: *LUKS*(default), *plain*, and has limited features for *loopAES* and *Truecrypt* devices.

- LUKS, used by default, 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 and cryptographic security.

- plain dm-crypt mode, being the original kernel functionality, does not employ the convenience layer. It is more difficult to apply the same cryptographic strength with it. When doing so, longer keys (passphrases or keyfiles) are the result. It has, however, other advantages, described in the following **#Block device vs stacked filesystem encryption**.

TrueCrypt/VeraCrypt

A portable format, supporting encryption of whole disks/partitions or file containers, with compatibility across all major operating systems. **TrueCrypt** was discontinued by its developers in May 2014. The VeraCrypt fork was audited in 2016.

For practical implications of the chosen layer of operation, see the **#Block device vs stacked filesystem encryption** below, as well as the general write up for **eCryptfs**. See **Category:Encryption** for the available content of the methods compared below, as well as other tools not included in the table.

Block device vs stacked filesystem encryption

Aspect	Block device encryption	Stacked filesystem encryption
Encrypts	whole block devices	files
Container for encrypted data may be...	a disk or disk partition / a file acting as a virtual partition	a directory in an existing file system
Relation to filesystem	operates below filesystem layer: does not 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 are written/read
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 custom-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.	✗	✓
Allows offline file-based backups of encrypted files	✗	✓

Comparison table

This article or section needs expansion.

Reason: Fill in blanks. Add sources to checkmarks / crosses. What is *salt*, *key-slot diffusion* or *key scrubbing*? (Discuss in Talk:Disk encryption#)

This article or section is out of date.

Reason: The Windows compatibility row of the comparison table links discontinued programs. (Discuss in Talk:Disk encryption#Discontinued Windows software)

The column "dm-crypt +/- LUKS" denotes features of dm-crypt for both LUKS ("+") and plain ("-") encryption modes. If a specific feature requires using LUKS, this is indicated by "(with LUKS)". Likewise "(without LUKS)" indicates usage of LUKS is counter-productive to achieve the feature and plain mode should be used.

Summary	Loop-AES	dm-crypt +/- LUKS	TrueCrypt	VeraCrypt	eCryptfs	EncFS
Encryption type	block device	block device	block device	block device	stacked filesystem	stacked filesystem
Note	longest-existing one; possibly the fastest; works on legacy systems	de-facto standard for block device encryption on Linux; very flexible	very portable, well-polished but abandoned	maintained fork of TrueCrypt	slightly faster than EncFS; individual encrypted files portable between systems	easiest one to use; supports non-root administration
Availability in Arch Linux	requires manually compiled, custom kernel	*kernel modules:* already shipped with default kernel; *tools:* `device-mapper`, `cryptsetup`	`truecrypt`	`veracrypt`	*kernel module:* already shipped with default kernel; *tools:* `ecryptfs-utils`	`encfs`
License	GPL	GPL	TrueCrypt License 3.1	Apache License 2.0, parts subject to TrueCrypt License v3.0	GPL	GPL
Encryption implemented in...	kernelspace	kernelspace	kernelspace	kernelspace	kernelspace	userspace (using FUSE)
Cryptographic metadata	?	with LUKS: LUKS Header	begin/end of (decrypted)	begin/end of (decrypted)	header of each encrypted	control file at the top level

stored in...			device (**format**)[dead link 2018-07-15]	device (**format spec**)	file	of each EncFs container
Wrapped encryption key stored in...	?	with LUKS: LUKS header	begin/end of (decrypted) device (**format spec**)[dead link 2018-07-15]	begin/end of (decrypted) device (**format spec**)	key file that can be stored anywhere	key file that can be stored anywhere [1][2]
Usability features	**Loop-AES**	**dm-crypt +/- LUKS**	**TrueCrypt**	**VeraCrypt**	**eCryptfs**	**EncFs**
Non-root users can create/destroy containers for encrypted data	✗	✗	✗	✗	limited	✓
Provides a GUI	✗	✗	✓	✓	✗	✓ **optional**
Support for automounting on login	?	✓	✓ with **systemd and /etc/crypttab**	✓ with **systemd and /etc/crypttab**	✓	✓
Support for automatic unmounting in case of inactivity	?	?	?	?	?	✓
Security features	**Loop-AES**	**dm-crypt +/- LUKS**	**TrueCrypt**	**VeraCrypt**	**eCryptfs**	**EncFs**

Supported ciphers	AES	AES, Anubis, CAST5/6, Twofish, Serpent, Camellia, Blowfish,… (every cipher the kernel Crypto API offers)	AES, Twofish, Serpent	AES, Twofish, Serpernt, Camellia, Kuznyechik	AES, Blowfish, Twofish...	AES, Blowfish, Twofish, and any other ciphers available on the system
Support for salting	?	✓ (with LUKS)	✓	✓	✓	?
Support for cascading multiple ciphers	?	Not in one device, but blockdevices can be cascaded	✓ AES-Twofish, AES-Twofish-Serpent, Serpent-AES, Serpent-Twofish-AES, Twofish-Serpent	✓ AES-Twofish, AES-Twofish-Serpent, Serpent-AES, Serpent-Twofish-AES, Twofish-Serpent	?	✗
Support for key-slot diffusion	?	✓ (with LUKS)	?	?	?	?
Protection against key scrubbing	✓	✓ (without LUKS)	?	?	?	?
Support for multiple (independently revocable) keys for the same	?	✓ (with LUKS)	?	?	?	✗

encrypted data						
Performance features	**Loop-AES**	**dm-crypt +/- LUKS**	**TrueCrypt**	**VeraCrypt**	**eCryptfs**	**EncFs**
Multithreading support	?	✓ [3]	✓	✓	?	?
Hardware-accelerated encryption support	✓	✓	✓	✓	✓	✓ [4]
Block device encryption specific	**Loop-AES**	**dm-crypt +/- LUKS**	**TrueCrypt**	**VeraCrypt**		
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...	ext3, ext4, xfs (with caveats), jfs, nfs, cifs... [5]
Ability to encrypt filenames					✓	✓
Ability to *not* encrypt					✓	✓

filenames						
Optimized handling of sparse files					✗	✓
Compatibility & prevalence	**Loop-AES**	**dm-crypt +/- LUKS**	**TrueCrypt**	**VeraCrypt**	**eCryptfs**	**EncFs**
Supported Linux kernel versions	2.0 or newer	CBC-mode since 2.6.4, ESSIV 2.6.10, LRW 2.6.20, XTS 2.6.24	?	?	?	2.4 or newer
Encrypted data can also be accessed from Windows	✓ (with **CrossCrypt**, **LibreCrypt**)	? (with **FreeOTFE**, **LibreCrypt**)	✓	✓	?	? [6]
Encrypted data can also be accessed from Mac OS X	?	?	✓	✓	?	✓ [7]
Encrypted data can also be accessed from FreeBSD	?	?	✓ (with VeraCrypt)	✓	?	✓ [8]
Used by	?	Debian/Ubuntu installer (system encryption) Fedora installer	?	?	Ubuntu installer (home dir encryption) Chromium OS (encryption of cached user data [9])	?

1. well, a single file in those filesystems could be used as a container (virtual loop-back device!) but then one would not actually be using the filesystem (and the features it provides) anymore

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 snooping around your disk when 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 do you want to encrypt?

- only user data
- user data and system data
- 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 accommodated?

- Not at all
- Using a shared passphrase/key
- Independently 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)

Examples

In practice, it could turn out something like:

Example 1

Simple user data encryption (internal hard drive) using a virtual folder called `~/Private` in the user's home directory encrypted with **EncFS**

- encrypted versions of the files stored on-disk in `~/.Private`
- unlocked on demand with dedicated passphrase

Example 2

Partial system encryption with each user's home directory encrypted with **ECryptfs**

- unlocked on respective user login, using login passphrase
- `swap` and `/tmp` partitions encrypted with **Dm-crypt with LUKS**, using an automatically generated per-session throwaway key
- indexing/caching of contents of `/home` by *slocate* (and similar apps) disabled.

Example 3

System encryption - whole hard drive except `/boot` partition (however, `/boot` can be encrypted with **GRUB**) encrypted with **Dm-crypt with LUKS**

- unlocked during boot, using passphrases or USB stick with keyfiles
- Maybe different passphrases/keys per user - independently revocable
- Maybe encryption spanning multiple drives or partition layout flexibility with **LUKS on LVM**

Example 4

Hidden/plain system encryption - whole hard drive encrypted with **plain dm-crypt**

- USB-boot, using dedicated passphrase plus USB stick with keyfile
- data integrity checked before mounting

- `/boot` partition located on aforementioned USB stick

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

See **Security#Passwords**.

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 does not 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. In order to satisfy this goal, wiping the disk using high-quality random bytes is crucial.

The second goal only makes sense in combination with block device encryption, because in the case of stacked filesystem encryption the encrypted data can easily be located 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 (due to **disk fragmentation**). 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 length, 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:

```
                  ╔═══════╗
      sector 1    ║"???.."║
                  ╠═══════╣          ╭┈┈┈┈┈╮
      sector 2    ║"???.."║          ┆ key ┆
                  ╠═══════╣          ╰┈┬┈┈┈╯
                  :       :            │
                  ╠═══════╣            ▼            ┣┉┉┉┉┉┉┉┉┫
      sector n    ║"???.."║━━━━━━━━━━━(decryption)━━━━━━━━━▶┇"abc.."┇ sector n
                  ╠═══════╣                         ┣┉┉┉┉┉┉┉┉┫
                  :       :
                  ╚═══════╝

                  encrypted                         unencrypted
               blockdevice or                       data in RAM
                file on disk
```

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

In order to be able to de/encrypt data, the disk encryption system needs to know the unique secret "key" associated with it. Whenever the encrypted block device or folder in question is to be mounted, its corresponding key (called henceforth its "master key") must be supplied.

The entropy of the key is of utmost importance for the security of the encryption. A randomly generated byte string of a certain length, for example 32 bytes (256 bits), has desired properties but is not feasible to remember and apply manually during the mount.

For that reason two techniques are used as aides. The first is the application of cryptography to increase the entropic property of the master key, usually involving a separate human-friendly passphrase. For the different types of encryption the **#Comparison table** lists respective features. The second method is to create a keyfile with high entropy and store it on a medium separate from the data drive to be encrypted.

See also **Wikipedia:Authenticated encryption**.

Keys, keyfiles and passphrases

The following are examples how to store and cryptographically secure a master key with a keyfile:

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. See **#Cryptographic metadata** below for details.

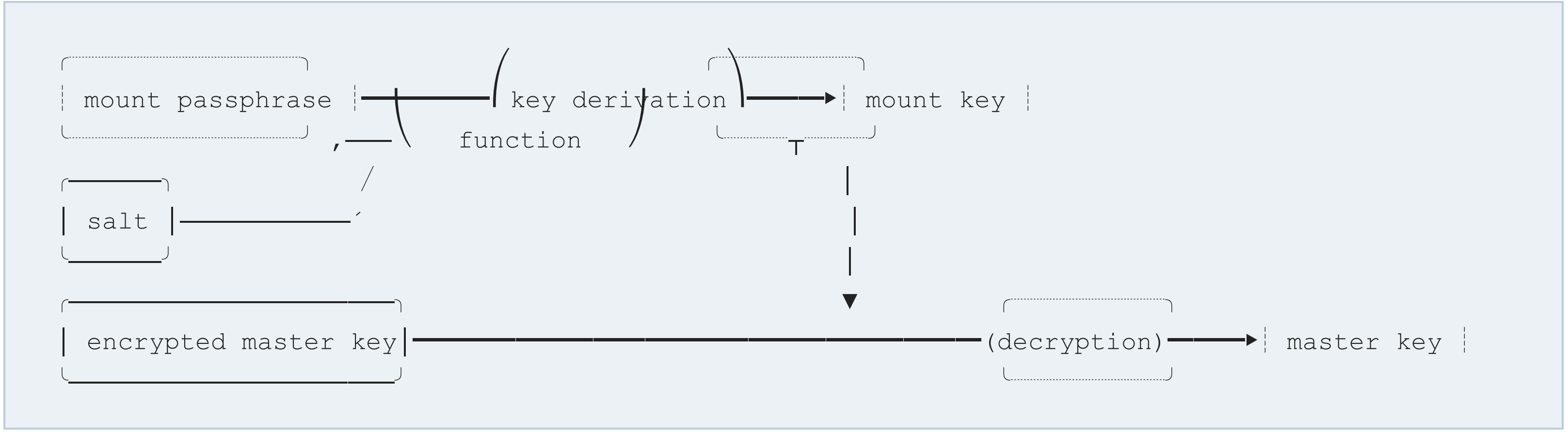
Randomly generated on-the-fly for each session

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.

Cryptographic metadata

Frequently the encryption techniques use cryptographic functions to enhance the security of the master key itself. On mount of the encrypted device the passphrase or keyfile is passed through these and only the result can unlock the master key to decrypt the data.

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.

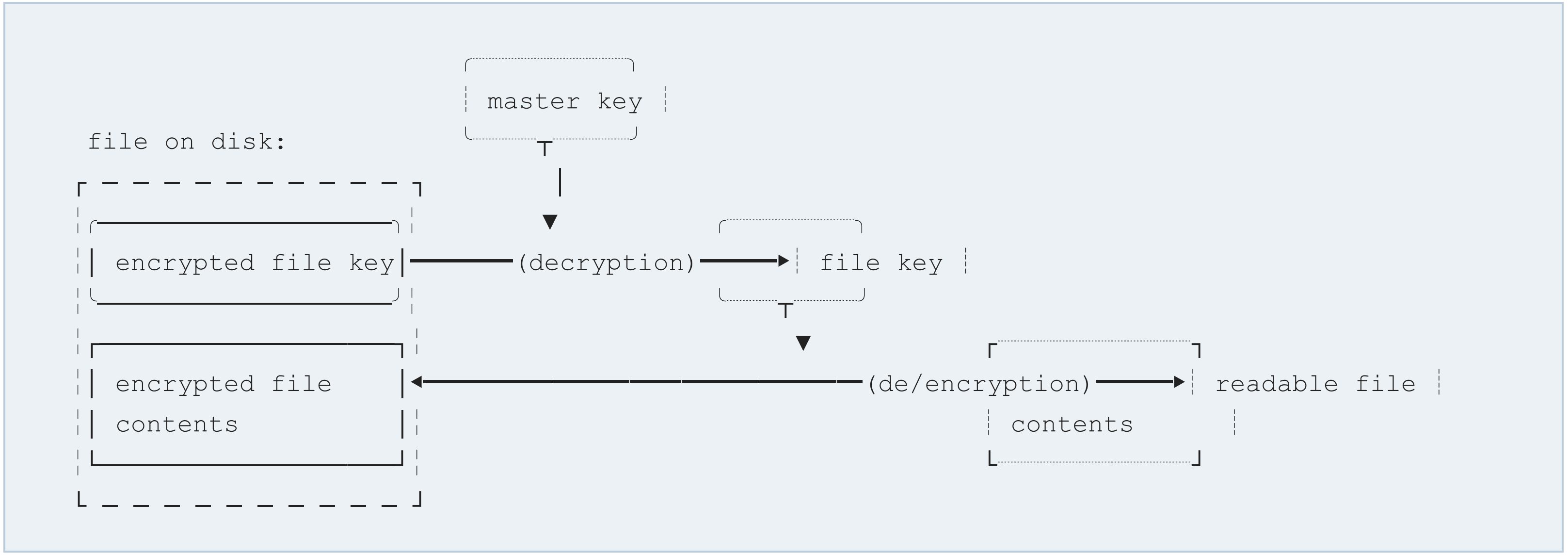
The **encrypted master key** can be stored on disk together with the encrypted data. This way, the confidentiality of the encrypted data depends completely on the secret passphrase.

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.

After it 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 one master key is used per device and, hence, all data. Some methods offer features to assign multiple passphrases/keyfiles for the same device and others not. Some use above mentioned functions to secure the master key and others give the control over the key security fully to the user. Two examples are explained by the cryptographic parameters used by **dm-crypt** in plain or LUKS modes.

When comparing the parameters used by both modes one notes that dm-crypt plain mode has parameters relating to how to locate the keyfile (e.g. `--keyfile-size`, `--keyfile-offset`). The dm-crypt LUKS mode does not need these, because each blockdevice contains a header with the cryptographic metadata at the beginning. The header includes the used cipher, the encrypted master-key itself and parameters required for its derivation for decryption. The latter parameters in turn result from options used during initial encryption of the master-key (e.g. `--iter-time`, `--use-random`).

For the dis-/advantages of the different techniques, please refer back to **#Comparison table** or browse the specific pages.

See also:

- 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**".

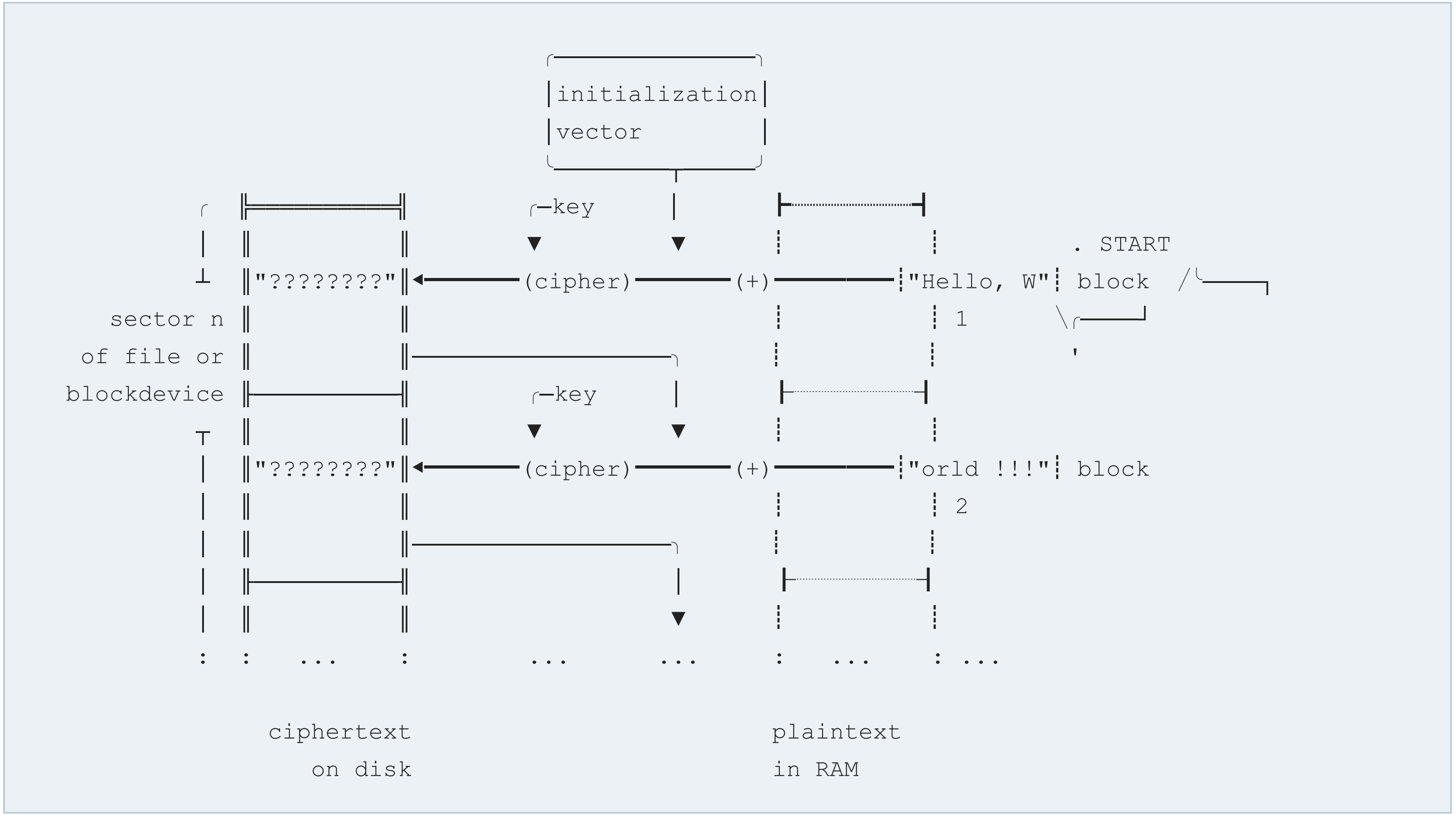
Disk encryption employs "block ciphers", which operate on fixed-length blocks of data, e.g. 16 bytes (128 bits). At the time of this writing, the predominantly used ones are:

	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
Serpent	128 bits	128, 192 or 256 bits	Considered the most secure of the five AES-competition finalists[10][11][12].

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**. 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 against such attacks (and hence do not require ESSIV). Some can also additionally guarantee authenticity of the encrypted data (i.e. confirm that it has not been modified/corrupted by someone who does not have access to the key).

See also:

- **Wikipedia:Disk encryption theory**
- **Wikipedia:Block cipher**
- **Wikipedia:Block cipher modes of operation**

Plausible deniability

See **Wikipedia:Plausible deniability**.

Category:

- Disk encryption