Filesystem-level encryption (fscrypt)

Introduction

fscrypt is a library which filesystems can hook into to support transparent encryption of files and directories.

Note: "fscrypt" in this document refers to the kernel-level portion, implemented in `fs/crypto/`, as opposed to the userspace tool fscrypt. This document only covers the kernel-level portion. For command-line examples of how to use encryption, see the documentation for the userspace tool fscrypt. Also, it is recommended to use the fscrypt userspace tool, or other existing userspace tools such as fscryptctl or Android's key management system, over using the kernel's API directly. Using existing tools reduces the chance of introducing your own security bugs. (Nevertheless, for completeness this documentation covers the kernel's API anyway.)

Unlike dm-crypt, fscrypt operates at the filesystem level rather than at the block device level. This allows it to encrypt different files with different keys and to have unencrypted files on the same filesystem. This is useful for multi-user systems where each user's data-at-rest needs to be cryptographically isolated from the others. However, except for filenames, fscrypt does not encrypt filesystem metadata.

Unlike eCryptfs, which is a stacked filesystem, fscrypt is integrated directly into supported filesystems --- currently ext4, F2FS, and UBIFS. This allows encrypted files to be read and written without caching both the decrypted and encrypted pages in the pagecache, thereby nearly halving the memory used and bringing it in line with unencrypted files. Similarly, half as many dentries and inodes are needed. eCryptfs also limits encrypted filenames to 143 bytes, causing application compatibility issues; fscrypt allows the full 255 bytes (NAME_MAX). Finally, unlike eCryptfs, the fscrypt API can be used by unprivileged users, with no need to mount anything.

fscrypt does not support encrypting files in-place. Instead, it supports marking an empty directory as encrypted. Then, after userspace provides the key, all regular files, directories, and symbolic links created in that directory tree are transparently encrypted.

Threat model

Offline attacks

Provided that userspace chooses a strong encryption key, fscrypt protects the confidentiality of file contents and filenames in the event of a single point-in-time permanent offline compromise of the block device content. fscrypt does not protect the confidentiality of non-filename metadata, e.g. file sizes, file permissions, file timestamps, and extended attributes. Also, the existence and location of holes (unallocated blocks which logically contain all zeroes) in files is not protected.

fscrypt is not guaranteed to protect confidentiality or authenticity if an attacker is able to manipulate the filesystem offline prior to an authorized user later accessing the filesystem.

Online attacks

fscrypt (and storage encryption in general) can only provide limited protection, if any at all, against online attacks. In detail:

Side-channel attacks

fscrypt is only resistant to side-channel attacks, such as timing or electromagnetic attacks, to the extent that the underlying Linux Cryptographic API algorithms or inline encryption hardware are. If a vulnerable algorithm is used, such as a table-based implementation of AES, it may be possible for an attacker to mount a side channel attack against the online system. Side channel attacks may also be mounted against applications consuming decrypted data.

Unauthorized file access

After an encryption key has been added, fscrypt does not hide the plaintext file contents or filenames from other users on the same system. Instead, existing access control mechanisms such as file mode bits, POSIX ACLs, LSMs, or namespaces should be used for this purpose.

(For the reasoning behind this, understand that while the key is added, the confidentiality of the data, from the perspective of the system itself, is *not* protected by the mathematical properties of encryption but rather only by the correctness of the kernel. Therefore, any encryption-specific access control checks would merely be enforced by kernel *code* and therefore would be largely redundant with the wide variety of access control mechanisms already available.)

Kernel memory compromise

An attacker who compromises the system enough to read from arbitrary memory, e.g. by mounting a physical attack or by exploiting a kernel security vulnerability, can compromise all encryption keys that are currently in use.

However, fscrypt allows encryption keys to be removed from the kernel, which may protect them from later compromise.

In more detail, the FS_IOC_REMOVE_ENCRYPTION_KEY ioctl (or the FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS ioctl) can wipe a master encryption key from kernel memory. If it does so, it will also try to evict all cached inodes which had been "unlocked" using the key, thereby wiping their per-file keys and making them once again appear "locked", i.e. in ciphertext or encrypted form.

However, these ioctls have some limitations:

- Per-file keys for in-use files will *not* be removed or wiped. Therefore, for maximum effect, userspace should close the relevant encrypted files and directories before removing a master key, as well as kill any processes whose working directory is in an affected encrypted directory.
- The kernel cannot magically wipe copies of the master key(s) that userspace might have as well. Therefore, userspace must wipe all copies of the master key(s) it makes as well; normally this should be done immediately after FS_IOC_ADD_ENCRYPTION_KEY, without waiting for FS_IOC_REMOVE_ENCRYPTION_KEY. Naturally, the same also applies to all higher levels in the key hierarchy. Userspace should also follow other security precautions such as mlock()ing memory containing keys to prevent it from being swapped out.
- In general, decrypted contents and filenames in the kernel VFS caches are freed but not wiped. Therefore, portions thereof may be recoverable from freed memory, even after the corresponding key(s) were wiped. To partially solve this, you can set CONFIG_PAGE_POISONING=y in your kernel config and add page_poison=1 to your kernel command line. However, this has a performance cost.
- Secret keys might still exist in CPU registers, in crypto accelerator hardware (if used by the crypto API to implement any of the algorithms), or in other places not explicitly considered here.

Limitations of v1 policies

v1 encryption policies have some weaknesses with respect to online attacks:

- There is no verification that the provided master key is correct. Therefore, a malicious user can temporarily associate the wrong key with another user's encrypted files to which they have read-only access. Because of filesystem caching, the wrong key will then be used by the other user's accesses to those files, even if the other user has the correct key in their own keyring. This violates the meaning of "read-only access".
- A compromise of a per-file key also compromises the master key from which it was derived.
- Non-root users cannot securely remove encryption keys.

All the above problems are fixed with v2 encryption policies. For this reason among others, it is recommended to use v2 encryption policies on all new encrypted directories.

Key hierarchy

Master Keys

Each encrypted directory tree is protected by a *master key*. Master keys can be up to 64 bytes long, and must be at least as long as the greater of the security strength of the contents and filenames encryption modes being used. For example, if any AES-256 mode is used, the master key must be at least 256 bits, i.e. 32 bytes. A stricter requirement applies if the key is used by a v1 encryption policy and AES-256-XTS is used; such keys must be 64 bytes.

To "unlock" an encrypted directory tree, userspace must provide the appropriate master key. There can be any number of master keys, each of which protects any number of directory trees on any number of filesystems.

Master keys must be real cryptographic keys, i.e. indistinguishable from random bytestrings of the same length. This implies that users **must not** directly use a password as a master key, zero-pad a shorter key, or repeat a shorter key. Security cannot be guaranteed if userspace makes any such error, as the cryptographic proofs and analysis would no longer apply.

Instead, users should generate master keys either using a cryptographically secure random number generator, or by using a KDF (Key Derivation Function). The kernel does not do any key stretching; therefore, if userspace derives the key from a low-entropy secret such as a passphrase, it is critical that a KDF designed for this purpose be used, such as scrypt, PBKDF2, or Argon2.

Key derivation function

With one exception, fscrypt never uses the master key(s) for encryption directly. Instead, they are only used as input to a KDF (Key Derivation Function) to derive the actual keys.

The KDF used for a particular master key differs depending on whether the key is used for v1 encryption policies or for v2 encryption policies. Users **must not** use the same key for both v1 and v2 encryption policies. (No real-world attack is currently known on this specific case of key reuse, but its security cannot be guaranteed since the cryptographic proofs and analysis would no longer apply.)

For v1 encryption policies, the KDF only supports deriving per-file encryption keys. It works by encrypting the master key with AES-128-ECB, using the file's 16-byte nonce as the AES key. The resulting ciphertext is used as the derived key. If the ciphertext is longer than needed, then it is truncated to the needed length.

For v2 encryption policies, the KDF is HKDF-SHA512. The master key is passed as the "input keying material", no salt is used, and a distinct "application-specific information string" is used for each distinct key to be derived. For example, when a per-file encryption key is derived, the application-specific information string is the file's nonce prefixed with "fscrypt\0" and a context byte. Different context bytes are used for other types of derived keys.

HKDF-SHA512 is preferred to the original AES-128-ECB based KDF because HKDF is more flexible, is nonreversible, and evenly distributes entropy from the master key. HKDF is also standardized and widely used by other software, whereas the AES-128-ECB based KDF is ad-hoc.

Per-file encryption keys

Since each master key can protect many files, it is necessary to "tweak" the encryption of each file so that the same plaintext in two files doesn't map to the same ciphertext, or vice versa. In most cases, fscrypt does this by deriving per-file keys. When a new encrypted inode (regular file, directory, or symlink) is created, fscrypt randomly generates a 16-byte nonce and stores it in the inode's encryption xattr. Then, it uses a KDF (as described in Key derivation function) to derive the file's key from the master key and nonce.

Key derivation was chosen over key wrapping because wrapped keys would require larger xattrs which would be less likely to fit in-line in the filesystem's inode table, and there didn't appear to be any significant advantages to key wrapping. In particular, currently there is no requirement to support unlocking a file with multiple alternative master keys or to support rotating master keys. Instead, the master keys may be wrapped in userspace, e.g. as is done by the fscrypt tool.

DIRECT_KEY policies

The Adiantum encryption mode (see Encryption modes and usage) is suitable for both contents and filenames encryption, and it accepts long IVs --- long enough to hold both an 8-byte logical block number and a 16-byte per-file nonce. Also, the overhead of each Adiantum key is greater than that of an AES-256-XTS key.

Therefore, to improve performance and save memory, for Adiantum a "direct key" configuration is supported. When the user has enabled this by setting FSCRYPT_POLICY_FLAG_DIRECT_KEY in the fscrypt policy, per-file encryption keys are not used. Instead, whenever any data (contents or filenames) is encrypted, the file's 16-byte nonce is included in the IV. Moreover:

- For v1 encryption policies, the encryption is done directly with the master key. Because of this, users **must not** use the same master key for any other purpose, even for other v1 policies.
- For v2 encryption policies, the encryption is done with a per-mode key derived using the KDF. Users may use the same master key for other v2 encryption policies.

IV_INO_LBLK_64 policies

When FSCRYPT_POLICY_FLAG_IV_INO_LBLK_64 is set in the fscrypt policy, the encryption keys are derived from the master key, encryption mode number, and filesystem UUID. This normally results in all files protected by the same master key sharing a single contents encryption key and a single filenames encryption key. To still encrypt different files' data differently, inode numbers are included in the IVs. Consequently, shrinking the filesystem may not be allowed.

This format is optimized for use with inline encryption hardware compliant with the UFS standard, which supports only 64 IV bits per I/O request and may have only a small number of keyslots.

IV_INO_LBLK_32 policies

IV_INO_LBLK_32 policies work like IV_INO_LBLK_64, except that for IV_INO_LBLK_32, the inode number is hashed with SipHash-2-4 (where the SipHash key is derived from the master key) and added to the file logical block number mod 2^32 to produce a 32-bit IV.

This format is optimized for use with inline encryption hardware compliant with the eMMC v5.2 standard, which supports only 32 IV bits per I/O request and may have only a small number of keyslots. This format results in some level of IV reuse, so it should only be used when necessary due to hardware limitations.

Key identifiers

For master keys used for v2 encryption policies, a unique 16-byte "key identifier" is also derived using the KDF. This value is stored in the clear, since it is needed to reliably identify the key itself.

Dirhash keys

For directories that are indexed using a secret-keyed dirhash over the plaintext filenames, the KDF is also used to derive a 128-bit SipHash-2-4 key per directory in order to hash filenames. This works just like deriving a per-file encryption key, except that a different KDF context is used. Currently, only casefolded ("case-insensitive") encrypted directories use this style of hashing.

Encryption modes and usage

fscrypt allows one encryption mode to be specified for file contents and one encryption mode to be specified for filenames. Different directory trees are permitted to use different encryption modes. Currently, the following pairs of encryption modes are supported:

- AES-256-XTS for contents and AES-256-CTS-CBC for filenames
- AES-128-CBC for contents and AES-128-CTS-CBC for filenames
- Adiantum for both contents and filenames

If unsure, you should use the (AES-256-XTS, AES-256-CTS-CBC) pair.

AES-128-CBC was added only for low-powered embedded devices with crypto accelerators such as CAAM or CESA that do not support XTS. To use AES-128-CBC, CONFIG_CRYPTO_ESSIV and CONFIG_CRYPTO_SHA256 (or another SHA-256 implementation) must be enabled so that ESSIV can be used.

Adiantum is a (primarily) stream cipher-based mode that is fast even on CPUs without dedicated crypto instructions. It's also a true wide-block mode, unlike XTS. It can also eliminate the need to derive per-file encryption keys. However, it depends on the security of two primitives, XChaCha12 and AES-256, rather than just one. See the paper "Adiantum: length-preserving encryption for entry-level processors" (https://eprint.iacr.org/2018/720.pdf) for more details. To use Adiantum, CONFIG_CRYPTO_ADIANTUM must be enabled. Also, fast implementations of ChaCha and NHPoly1305 should be enabled, e.g. CONFIG_CRYPTO_CHACHA20_NEON and CONFIG_CRYPTO_NHPOLY1305_NEON for ARM.

New encryption modes can be added relatively easily, without changes to individual filesystems. However, authenticated encryption (AE) modes are not currently supported because of the difficulty of dealing with ciphertext expansion.

Contents encryption

For file contents, each filesystem block is encrypted independently. Starting from Linux kernel 5.5, encryption of filesystems with block size less than system's page size is supported.

Each block's IV is set to the logical block number within the file as a little endian number, except that:

- With CBC mode encryption, ESSIV is also used. Specifically, each IV is encrypted with AES-256 where the AES-256 key is the SHA-256 hash of the file's data encryption key.
- With DIRECT_KEY policies, the file's nonce is appended to the IV. Currently this is only allowed with the Adiantum encryption mode.
- With IV_INO_LBLK_64 policies, the logical block number is limited to 32 bits and is placed in bits 0-31 of the IV. The inode number (which is also limited to 32 bits) is placed in bits 32-63.
- With IV_INO_LBLK_32 policies, the logical block number is limited to 32 bits and is placed in bits 0-31 of the IV. The inode number is then hashed and added mod 2^32.

Note that because file logical block numbers are included in the IVs, filesystems must enforce that blocks are never shifted around within encrypted files, e.g. via "collapse range" or "insert range".

Filenames encryption

For filenames, each full filename is encrypted at once. Because of the requirements to retain support for efficient directory lookups and filenames of up to 255 bytes, the same IV is used for every filename in a directory.

However, each encrypted directory still uses a unique key, or alternatively has the file's nonce (for DIRECT_KEY policies) or inode number (for IV_INO_LBLK_64 policies) included in the IVs. Thus, IV reuse is limited to within a single directory.

With CTS-CBC, the IV reuse means that when the plaintext filenames share a common prefix at least as long as the cipher block size (16 bytes for AES), the corresponding encrypted filenames will also share a common prefix. This is undesirable. Adiantum does not have this weakness, as it is a wide-block encryption mode.

All supported filenames encryption modes accept any plaintext length >= 16 bytes; cipher block alignment is not required. However, filenames shorter than 16 bytes are NUL-padded to 16 bytes before being encrypted. In addition, to reduce leakage of filename lengths via their ciphertexts, all filenames are NUL-padded to the next 4, 8, 16, or 32-byte boundary (configurable). 32 is recommended since this provides the best confidentiality, at the cost of making directory entries consume slightly more space. Note that since NUL (`\0`) is not otherwise a valid character in filenames, the padding will never produce duplicate plaintexts.

Symbolic link targets are considered a type of filename and are encrypted in the same way as filenames in directory entries, except that IV reuse is not a problem as each symlink has its own inode.

User API

Setting an encryption policy

FS_IOC_SET_ENCRYPTION_POLICY

The FS_IOC_SET_ENCRYPTION_POLICY ioctl sets an encryption policy on an empty directory or verifies that a directory or regular file already has the specified encryption policy. It takes in a pointer to struct fscrypt_policy_v1 or struct fscrypt_policy_v2, defined as follows:

```
#define FSCRYPT_POLICY_V1               0
#define FSCRYPT_KEY_DESCRIPTOR_SIZE     8
struct fscrypt_policy_v1 {
        __u8 version;
        __u8 contents_encryption_mode;
        __u8 filenames_encryption_mode;
        __u8 flags;
        __u8 master_key_descriptor[FSCRYPT_KEY_DESCRIPTOR_SIZE];
};
#define fscrypt_policy  fscrypt_policy_v1
```

```
#define FSCRYPT_POLICY_V2               2
#define FSCRYPT_KEY_IDENTIFIER_SIZE     16
struct fscrypt_policy_v2 {
        __u8 version;
        __u8 contents_encryption_mode;
        __u8 filenames_encryption_mode;
        __u8 flags;
        __u8 __reserved[4];
        __u8 master_key_identifier[FSCRYPT_KEY_IDENTIFIER_SIZE];
};
```

This structure must be initialized as follows:

- `version` must be FSCRYPT_POLICY_V1 (0) if struct fscrypt_policy_v1 is used or FSCRYPT_POLICY_V2 (2) if struct fscrypt_policy_v2 is used. (Note: we refer to the original policy version as "v1", though its version code is really 0.) For new encrypted directories, use v2 policies.

- `contents_encryption_mode` and `filenames_encryption_mode` must be set to constants from `<linux/fscrypt.h>` which identify the encryption modes to use. If unsure, use FSCRYPT_MODE_AES_256_XTS (1) for `contents_encryption_mode` and FSCRYPT_MODE_AES_256_CTS (4) for `filenames_encryption_mode`.

- `flags` contains optional flags from `<linux/fscrypt.h>`:

    - FSCRYPT_POLICY_FLAGS_PAD_*: The amount of NUL padding to use when encrypting filenames. If unsure, use FSCRYPT_POLICY_FLAGS_PAD_32 (0x3).
    - FSCRYPT_POLICY_FLAG_DIRECT_KEY: See DIRECT_KEY policies.
    - FSCRYPT_POLICY_FLAG_IV_INO_LBLK_64: See IV_INO_LBLK_64 policies.
    - FSCRYPT_POLICY_FLAG_IV_INO_LBLK_32: See IV_INO_LBLK_32 policies.

    v1 encryption policies only support the PAD_* and DIRECT_KEY flags. The other flags are only supported by v2 encryption policies.

    The DIRECT_KEY, IV_INO_LBLK_64, and IV_INO_LBLK_32 flags are mutually exclusive.

- For v2 encryption policies, `__reserved` must be zeroed.

- For v1 encryption policies, `master_key_descriptor` specifies how to find the master key in a keyring; see Adding keys. It is up to userspace to choose a unique `master_key_descriptor` for each master key. The e4crypt and fscrypt tools use the first 8 bytes of `SHA-512(SHA-512(master_key))`, but this particular scheme is not required. Also, the master key need not be in the keyring yet when FS_IOC_SET_ENCRYPTION_POLICY is executed. However, it must be added before any files can be created in the encrypted directory.

    For v2 encryption policies, `master_key_descriptor` has been replaced with `master_key_identifier`, which is longer and cannot be arbitrarily chosen. Instead, the key must first be added using FS_IOC_ADD_ENCRYPTION_KEY. Then, the `key_spec.u.identifier` the kernel returned in the struct fscrypt_add_key_arg must be used as the `master_key_identifier` in struct fscrypt_policy_v2.

If the file is not yet encrypted, then FS_IOC_SET_ENCRYPTION_POLICY verifies that the file is an empty directory. If so, the specified encryption policy is assigned to the directory, turning it into an encrypted directory. After that, and after providing the corresponding master key as described in Adding keys, all regular files, directories (recursively), and symlinks created in the directory will be encrypted, inheriting the same encryption policy. The filenames in the directory's entries will be encrypted as well.

Alternatively, if the file is already encrypted, then FS_IOC_SET_ENCRYPTION_POLICY validates that the specified encryption policy exactly matches the actual one. If they match, then the ioctl returns 0. Otherwise, it fails with EEXIST. This works on both regular files and directories, including nonempty directories.

When a v2 encryption policy is assigned to a directory, it is also required that either the specified key has been added by the current user or that the caller has CAP_FOWNER in the initial user namespace. (This is needed to prevent a user from encrypting their data with another user's key.) The key must remain added while FS_IOC_SET_ENCRYPTION_POLICY is executing. However, if the new encrypted directory does not need to be accessed immediately, then the key can be removed right away afterwards.

Note that the ext4 filesystem does not allow the root directory to be encrypted, even if it is empty. Users who want to encrypt an entire filesystem with one key should consider using dm-crypt instead.

FS_IOC_SET_ENCRYPTION_POLICY can fail with the following errors:

- `EACCES`: the file is not owned by the process's uid, nor does the process have the CAP_FOWNER capability in a namespace with the file owner's uid mapped
- `EEXIST`: the file is already encrypted with an encryption policy different from the one specified
- `EINVAL`: an invalid encryption policy was specified (invalid version, mode(s), or flags; or reserved bits were set); or a v1 encryption policy was specified but the directory has the casefold flag enabled (casefolding is incompatible with v1 policies).
- `ENOKEY`: a v2 encryption policy was specified, but the key with the specified `master_key_identifier` has not been added, nor does the process have the CAP_FOWNER capability in the initial user namespace
- `ENOTDIR`: the file is unencrypted and is a regular file, not a directory
- `ENOTEMPTY`: the file is unencrypted and is a nonempty directory
- `ENOTTY`: this type of filesystem does not implement encryption

- EOPNOTSUPP: the kernel was not configured with encryption support for filesystems, or the filesystem superblock has not had encryption enabled on it. (For example, to use encryption on an ext4 filesystem, CONFIG_FS_ENCRYPTION must be enabled in the kernel config, and the superblock must have had the "encrypt" feature flag enabled using `tune2fs -O encrypt` or `mkfs.ext4 -O encrypt`.)
- EPERM: this directory may not be encrypted, e.g. because it is the root directory of an ext4 filesystem
- EROFS: the filesystem is readonly

Getting an encryption policy

Two ioctls are available to get a file's encryption policy:

- FS_IOC_GET_ENCRYPTION_POLICY_EX
- FS_IOC_GET_ENCRYPTION_POLICY

The extended (_EX) version of the ioctl is more general and is recommended to use when possible. However, on older kernels only the original ioctl is available. Applications should try the extended version, and if it fails with ENOTTY fall back to the original version.

FS_IOC_GET_ENCRYPTION_POLICY_EX

The FS_IOC_GET_ENCRYPTION_POLICY_EX ioctl retrieves the encryption policy, if any, for a directory or regular file. No additional permissions are required beyond the ability to open the file. It takes in a pointer to struct fscrypt_get_policy_ex_arg, defined as follows:

```
struct fscrypt_get_policy_ex_arg {
        __u64 policy_size; /* input/output */
        union {
                __u8 version;
                struct fscrypt_policy_v1 v1;
                struct fscrypt_policy_v2 v2;
        } policy; /* output */
};
```

The caller must initialize `policy_size` to the size available for the policy struct, i.e. `sizeof(arg.policy)`.

On success, the policy struct is returned in `policy`, and its actual size is returned in `policy_size`. `policy.version` should be checked to determine the version of policy returned. Note that the version code for the "v1" policy is actually 0 (FSCRYPT_POLICY_V1).

FS_IOC_GET_ENCRYPTION_POLICY_EX can fail with the following errors:

- EINVAL: the file is encrypted, but it uses an unrecognized encryption policy version
- ENODATA: the file is not encrypted
- ENOTTY: this type of filesystem does not implement encryption, or this kernel is too old to support FS_IOC_GET_ENCRYPTION_POLICY_EX (try FS_IOC_GET_ENCRYPTION_POLICY instead)
- EOPNOTSUPP: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it
- EOVERFLOW: the file is encrypted and uses a recognized encryption policy version, but the policy struct does not fit into the provided buffer

Note: if you only need to know whether a file is encrypted or not, on most filesystems it is also possible to use the FS_IOC_GETFLAGS ioctl and check for FS_ENCRYPT_FL, or to use the statx() system call and check for STATX_ATTR_ENCRYPTED in stx_attributes.

FS_IOC_GET_ENCRYPTION_POLICY

The FS_IOC_GET_ENCRYPTION_POLICY ioctl can also retrieve the encryption policy, if any, for a directory or regular file. However, unlike FS_IOC_GET_ENCRYPTION_POLICY_EX, FS_IOC_GET_ENCRYPTION_POLICY only supports the original policy version. It takes in a pointer directly to struct fscrypt_policy_v1 rather than struct fscrypt_get_policy_ex_arg.

The error codes for FS_IOC_GET_ENCRYPTION_POLICY are the same as those for FS_IOC_GET_ENCRYPTION_POLICY_EX, except that FS_IOC_GET_ENCRYPTION_POLICY also returns `EINVAL` if the file is encrypted using a newer encryption policy version.

Getting the per-filesystem salt

Some filesystems, such as ext4 and F2FS, also support the deprecated ioctl FS_IOC_GET_ENCRYPTION_PWSALT. This ioctl retrieves a randomly generated 16-byte value stored in the filesystem superblock. This value is intended to used as a salt when deriving an encryption key from a passphrase or other low-entropy user credential.

FS_IOC_GET_ENCRYPTION_PWSALT is deprecated. Instead, prefer to generate and manage any needed salt(s) in userspace.

Getting a file's encryption nonce

Since Linux v5.7, the ioctl FS_IOC_GET_ENCRYPTION_NONCE is supported. On encrypted files and directories it gets the inode's 16-byte nonce. On unencrypted files and directories, it fails with ENODATA.

This ioctl can be useful for automated tests which verify that the encryption is being done correctly. It is not needed for normal use of fscrypt.

Adding keys

FS_IOC_ADD_ENCRYPTION_KEY

The FS_IOC_ADD_ENCRYPTION_KEY ioctl adds a master encryption key to the filesystem, making all files on the filesystem which were encrypted using that key appear "unlocked", i.e. in plaintext form. It can be executed on any file or directory on the target filesystem, but using the filesystem's root directory is recommended. It takes in a pointer to struct fscrypt_add_key_arg, defined as follows:

```
struct fscrypt_add_key_arg {
        struct fscrypt_key_specifier key_spec;
        __u32 raw_size;
        __u32 key_id;
        __u32 __reserved[8];
        __u8 raw[];
};

#define FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR        1
#define FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER        2

struct fscrypt_key_specifier {
        __u32 type;     /* one of FSCRYPT_KEY_SPEC_TYPE_* */
        __u32 __reserved;
        union {
                __u8 __reserved[32]; /* reserve some extra space */
                __u8 descriptor[FSCRYPT_KEY_DESCRIPTOR_SIZE];
                __u8 identifier[FSCRYPT_KEY_IDENTIFIER_SIZE];
        } u;
};

struct fscrypt_provisioning_key_payload {
        __u32 type;
        __u32 __reserved;
        __u8 raw[];
};
```

struct fscrypt_add_key_arg must be zeroed, then initialized as follows:

- If the key is being added for use by v1 encryption policies, then `key_spec.type` must contain FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR, and `key_spec.u.descriptor` must contain the descriptor of the key being added, corresponding to the value in the `master_key_descriptor` field of struct fscrypt_policy_v1. To add this type of key, the calling process must have the CAP_SYS_ADMIN capability in the initial user namespace.

  Alternatively, if the key is being added for use by v2 encryption policies, then `key_spec.type` must contain FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER, and `key_spec.u.identifier` is an *output* field which the kernel fills in with a cryptographic hash of the key. To add this type of key, the calling process does not need any privileges. However, the number of keys that can be added is limited by the user's quota for the keyrings service (see `Documentation/security/keys/core.rst`).

- `raw_size` must be the size of the `raw` key provided, in bytes. Alternatively, if `key_id` is nonzero, this field must be 0, since in that case the size is implied by the specified Linux keyring key.

- `key_id` is 0 if the raw key is given directly in the `raw` field. Otherwise `key_id` is the ID of a Linux keyring key of type "fscrypt-provisioning" whose payload is struct fscrypt_provisioning_key_payload whose `raw` field contains the raw key and whose `type` field matches `key_spec.type`. Since `raw` is variable-length, the total size of this key's payload must be `sizeof(struct fscrypt_provisioning_key_payload)` plus the raw key size. The process must have Search permission on this key.

  Most users should leave this 0 and specify the raw key directly. The support for specifying a Linux keyring key is intended mainly to allow re-adding keys after a filesystem is unmounted and re-mounted, without having to store the raw keys in userspace memory.

- `raw` is a variable-length field which must contain the actual key, `raw_size` bytes long. Alternatively, if `key_id` is nonzero, then this field is unused.

For v2 policy keys, the kernel keeps track of which user (identified by effective user ID) added the key, and only allows the key to be removed by that user --- or by "root", if they use FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS.

However, if another user has added the key, it may be desirable to prevent that other user from unexpectedly removing it. Therefore, FS_IOC_ADD_ENCRYPTION_KEY may also be used to add a v2 policy key *again*, even if it's already added by other user(s). In this case, FS_IOC_ADD_ENCRYPTION_KEY will just install a claim to the key for the current user, rather than actually add the key again (but the raw key must still be provided, as a proof of knowledge).

FS_IOC_ADD_ENCRYPTION_KEY returns 0 if either the key or a claim to the key was either added or already exists.

FS_IOC_ADD_ENCRYPTION_KEY can fail with the following errors:

- `EACCES`: FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR was specified, but the caller does not have the CAP_SYS_ADMIN capability in the initial user namespace; or the raw key was specified by Linux key ID but the process lacks Search permission on the key.
- `EDQUOT`: the key quota for this user would be exceeded by adding the key
- `EINVAL`: invalid key size or key specifier type, or reserved bits were set
- `EKEYREJECTED`: the raw key was specified by Linux key ID, but the key has the wrong type
- `ENOKEY`: the raw key was specified by Linux key ID, but no key exists with that ID
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it

Legacy method

For v1 encryption policies, a master encryption key can also be provided by adding it to a process-subscribed keyring, e.g. to a session keyring, or to a user keyring if the user keyring is linked into the session keyring.

This method is deprecated (and not supported for v2 encryption policies) for several reasons. First, it cannot be used in combination with FS_IOC_REMOVE_ENCRYPTION_KEY (see Removing keys), so for removing a key a workaround such as keyctl_unlink() in combination with `sync; echo 2 > /proc/sys/vm/drop_caches` would have to be used. Second, it doesn't match the fact that the locked/unlocked status of encrypted files (i.e. whether they appear to be in plaintext form or in ciphertext form) is global. This mismatch has caused much confusion as well as real problems when processes running under different UIDs, such as a `sudo` command, need to access encrypted files.

Nevertheless, to add a key to one of the process-subscribed keyrings, the add_key() system call can be used (see: `Documentation/security/keys/core.rst`). The key type must be "logon"; keys of this type are kept in kernel memory and cannot be read back by userspace. The key description must be "fscrypt:" followed by the 16-character lower case hex representation of the `master_key_descriptor` that was set in the encryption policy. The key payload must conform to the following structure:

```
#define FSCRYPT_MAX_KEY_SIZE            64

struct fscrypt_key {
        __u32 mode;
        __u8 raw[FSCRYPT_MAX_KEY_SIZE];
        __u32 size;
};
```

`mode` is ignored; just set it to 0. The actual key is provided in `raw` with `size` indicating its size in bytes. That is, the bytes `raw[0..size-1]` (inclusive) are the actual key.

The key description prefix "fscrypt:" may alternatively be replaced with a filesystem-specific prefix such as "ext4:". However, the filesystem-specific prefixes are deprecated and should not be used in new programs.

Removing keys

Two ioctls are available for removing a key that was added by FS_IOC_ADD_ENCRYPTION_KEY:

- FS_IOC_REMOVE_ENCRYPTION_KEY
- FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS

These two ioctls differ only in cases where v2 policy keys are added or removed by non-root users.

These ioctls don't work on keys that were added via the legacy process-subscribed keyrings mechanism.

Before using these ioctls, read the Kernel memory compromise section for a discussion of the security goals and limitations of these ioctls.

FS_IOC_REMOVE_ENCRYPTION_KEY

The FS_IOC_REMOVE_ENCRYPTION_KEY ioctl removes a claim to a master encryption key from the filesystem, and possibly removes the key itself. It can be executed on any file or directory on the target filesystem, but using the filesystem's root directory is recommended. It takes in a pointer to struct fscrypt_remove_key_arg, defined as follows:

```
struct fscrypt_remove_key_arg {
        struct fscrypt_key_specifier key_spec;
#define FSCRYPT_KEY_REMOVAL_STATUS_FLAG_FILES_BUSY      0x00000001
#define FSCRYPT_KEY_REMOVAL_STATUS_FLAG_OTHER_USERS     0x00000002
        __u32 removal_status_flags;     /* output */
        __u32 __reserved[5];
};
```

This structure must be zeroed, then initialized as follows:

- The key to remove is specified by `key_spec`:
    - To remove a key used by v1 encryption policies, set `key_spec.type` to FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR and fill in `key_spec.u.descriptor`. To remove this type of key, the calling process must have the CAP_SYS_ADMIN capability in the initial user namespace.
    - To remove a key used by v2 encryption policies, set `key_spec.type` to FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER and fill in `key_spec.u.identifier`.

For v2 policy keys, this ioctl is usable by non-root users. However, to make this possible, it actually just removes the current user's claim to the key, undoing a single call to FS_IOC_ADD_ENCRYPTION_KEY. Only after all claims are removed is the key really removed.

For example, if FS_IOC_ADD_ENCRYPTION_KEY was called with uid 1000, then the key will be "claimed" by uid 1000, and FS_IOC_REMOVE_ENCRYPTION_KEY will only succeed as uid 1000. Or, if both uids 1000 and 2000 added the key, then for each uid FS_IOC_REMOVE_ENCRYPTION_KEY will only remove their own claim. Only once *both* are removed is the key really removed. (Think of it like unlinking a file that may have hard links.)

If FS_IOC_REMOVE_ENCRYPTION_KEY really removes the key, it will also try to "lock" all files that had been unlocked with the key. It won't lock files that are still in-use, so this ioctl is expected to be used in cooperation with userspace ensuring that none of the files are still open. However, if necessary, this ioctl can be executed again later to retry locking any remaining files.

FS_IOC_REMOVE_ENCRYPTION_KEY returns 0 if either the key was removed (but may still have files remaining to be locked), the user's claim to the key was removed, or the key was already removed but had files remaining to be the locked so the ioctl retried locking them. In any of these cases, `removal_status_flags` is filled in with the following informational status flags:

- `FSCRYPT_KEY_REMOVAL_STATUS_FLAG_FILES_BUSY`: set if some file(s) are still in-use. Not guaranteed to be set in the case where only the user's claim to the key was removed.
- `FSCRYPT_KEY_REMOVAL_STATUS_FLAG_OTHER_USERS`: set if only the user's claim to the key was removed, not the key itself

FS_IOC_REMOVE_ENCRYPTION_KEY can fail with the following errors:

- `EACCES`: The FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR key specifier type was specified, but the caller does not have the CAP_SYS_ADMIN capability in the initial user namespace
- `EINVAL`: invalid key specifier type, or reserved bits were set
- `ENOKEY`: the key object was not found at all, i.e. it was never added in the first place or was already fully removed including all files locked; or, the user does not have a claim to the key (but someone else does).
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it

FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS

FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS is exactly the same as FS_IOC_REMOVE_ENCRYPTION_KEY, except that for v2 policy keys, the ALL_USERS version of the ioctl will remove all users' claims to the key, not just the current user's. I.e., the key itself will always be removed, no matter how many users have added it. This difference is only meaningful if non-root users are adding and removing keys.

Because of this, FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS also requires "root", namely the CAP_SYS_ADMIN capability in the initial user namespace. Otherwise it will fail with EACCES.

Getting key status

FS_IOC_GET_ENCRYPTION_KEY_STATUS

The FS_IOC_GET_ENCRYPTION_KEY_STATUS ioctl retrieves the status of a master encryption key. It can be executed on any file or directory on the target filesystem, but using the filesystem's root directory is recommended. It takes in a pointer to struct fscrypt_get_key_status_arg, defined as follows:

```
struct fscrypt_get_key_status_arg {
        /* input */
        struct fscrypt_key_specifier key_spec;
        __u32 __reserved[6];

        /* output */
#define FSCRYPT_KEY_STATUS_ABSENT               1
#define FSCRYPT_KEY_STATUS_PRESENT              2
#define FSCRYPT_KEY_STATUS_INCOMPLETELY_REMOVED 3
        __u32 status;
#define FSCRYPT_KEY_STATUS_FLAG_ADDED_BY_SELF   0x00000001
        __u32 status_flags;
        __u32 user_count;
        __u32 __out_reserved[13];
```

```
    };
```

The caller must zero all input fields, then fill in `key_spec`:

- To get the status of a key for v1 encryption policies, set `key_spec.type` to FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR and fill in `key_spec.u.descriptor`.
- To get the status of a key for v2 encryption policies, set `key_spec.type` to FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER and fill in `key_spec.u.identifier`.

On success, 0 is returned and the kernel fills in the output fields:

- `status` indicates whether the key is absent, present, or incompletely removed. Incompletely removed means that the master secret has been removed, but some files are still in use; i.e., FS_IOC_REMOVE_ENCRYPTION_KEY returned 0 but set the informational status flag FSCRYPT_KEY_REMOVAL_STATUS_FLAG_FILES_BUSY.
- `status_flags` can contain the following flags:
    - `FSCRYPT_KEY_STATUS_FLAG_ADDED_BY_SELF` indicates that the key has added by the current user. This is only set for keys identified by `identifier` rather than by `descriptor`.
- `user_count` specifies the number of users who have added the key. This is only set for keys identified by `identifier` rather than by `descriptor`.

FS_IOC_GET_ENCRYPTION_KEY_STATUS can fail with the following errors:

- `EINVAL`: invalid key specifier type, or reserved bits were set
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it

Among other use cases, FS_IOC_GET_ENCRYPTION_KEY_STATUS can be useful for determining whether the key for a given encrypted directory needs to be added before prompting the user for the passphrase needed to derive the key.

FS_IOC_GET_ENCRYPTION_KEY_STATUS can only get the status of keys in the filesystem-level keyring, i.e. the keyring managed by FS_IOC_ADD_ENCRYPTION_KEY and FS_IOC_REMOVE_ENCRYPTION_KEY. It cannot get the status of a key that has only been added for use by v1 encryption policies using the legacy mechanism involving process-subscribed keyrings.

Access semantics

With the key

With the encryption key, encrypted regular files, directories, and symlinks behave very similarly to their unencrypted counterparts --- after all, the encryption is intended to be transparent. However, astute users may notice some differences in behavior:

- Unencrypted files, or files encrypted with a different encryption policy (i.e. different key, modes, or flags), cannot be renamed or linked into an encrypted directory; see Encryption policy enforcement. Attempts to do so will fail with EXDEV. However, encrypted files can be renamed within an encrypted directory, or into an unencrypted directory.

  Note: "moving" an unencrypted file into an encrypted directory, e.g. with the *mv* program, is implemented in userspace by a copy followed by a delete. Be aware that the original unencrypted data may remain recoverable from free space on the disk; prefer to keep all files encrypted from the very beginning. The *shred* program may be used to overwrite the source files but isn't guaranteed to be effective on all filesystems and storage devices.
- Direct I/O is supported on encrypted files only under some circumstances. For details, see Direct I/O support.
- The fallocate operations FALLOC_FL_COLLAPSE_RANGE and FALLOC_FL_INSERT_RANGE are not supported on encrypted files and will fail with EOPNOTSUPP.
- Online defragmentation of encrypted files is not supported. The EXT4_IOC_MOVE_EXT and F2FS_IOC_MOVE_RANGE ioctls will fail with EOPNOTSUPP.
- The ext4 filesystem does not support data journaling with encrypted regular files. It will fall back to ordered data mode instead.
- DAX (Direct Access) is not supported on encrypted files.
- The maximum length of an encrypted symlink is 2 bytes shorter than the maximum length of an unencrypted symlink. For example, on an EXT4 filesystem with a 4K block size, unencrypted symlinks can be up to 4095 bytes long, while encrypted symlinks can only be up to 4093 bytes long (both lengths excluding the terminating null).

Note that mmap *is* supported. This is possible because the pagecache for an encrypted file contains the plaintext, not the ciphertext.

Without the key

Some filesystem operations may be performed on encrypted regular files, directories, and symlinks even before their encryption key has been added, or after their encryption key has been removed:

- File metadata may be read, e.g. using stat().
- Directories may be listed, in which case the filenames will be listed in an encoded form derived from their ciphertext. The current encoding algorithm is described in Filename hashing and encoding. The algorithm is subject to change, but it is guaranteed that the presented filenames will be no longer than NAME_MAX bytes, will not contain the `/` or `\0` characters, and will uniquely identify directory entries.

  The `.` and `..` directory entries are special. They are always present and are not encrypted or encoded.
- Files may be deleted. That is, nondirectory files may be deleted with unlink() as usual, and empty directories may be deleted with rmdir() as usual. Therefore, `rm` and `rm -r` will work as expected.
- Symlink targets may be read and followed, but they will be presented in encrypted form, similar to filenames in directories. Hence, they are unlikely to point to anywhere useful.

Without the key, regular files cannot be opened or truncated. Attempts to do so will fail with ENOKEY. This implies that any regular file operations that require a file descriptor, such as read(), write(), mmap(), fallocate(), and ioctl(), are also forbidden.

Also without the key, files of any type (including directories) cannot be created or linked into an encrypted directory, nor can a name in an encrypted directory be the source or target of a rename, nor can an O_TMPFILE temporary file be created in an encrypted directory. All such operations will fail with ENOKEY.

It is not currently possible to backup and restore encrypted files without the encryption key. This would require special APIs which have not yet been implemented.

Encryption policy enforcement

After an encryption policy has been set on a directory, all regular files, directories, and symbolic links created in that directory (recursively) will inherit that encryption policy. Special files --- that is, named pipes, device nodes, and UNIX domain sockets --- will not be encrypted.

Except for those special files, it is forbidden to have unencrypted files, or files encrypted with a different encryption policy, in an encrypted directory tree. Attempts to link or rename such a file into an encrypted directory will fail with EXDEV. This is also enforced during ->lookup() to provide limited protection against offline attacks that try to disable or downgrade encryption in known locations where applications may later write sensitive data. It is recommended that systems implementing a form of "verified boot" take advantage of this by validating all top-level encryption policies prior to access.

Inline encryption support

By default, fscrypt uses the kernel crypto API for all cryptographic operations (other than HKDF, which fscrypt partially implements itself). The kernel crypto API supports hardware crypto accelerators, but only ones that work in the traditional way where all inputs and outputs (e.g. plaintexts and ciphertexts) are in memory. fscrypt can take advantage of such hardware, but the traditional acceleration model isn't particularly efficient and fscrypt hasn't been optimized for it.

Instead, many newer systems (especially mobile SoCs) have *inline encryption hardware* that can encrypt/decrypt data while it is on its way to/from the storage device. Linux supports inline encryption through a set of extensions to the block layer called *blk-crypto*. blk-crypto allows filesystems to attach encryption contexts to bios (I/O requests) to specify how the data will be encrypted or decrypted in-line. For more information about blk-crypto, see :ref:`Documentation/block/inline-encryption.rst <inline_encryption>`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\(linux-master)(Documentation)(filesystems)fscrypt.rst`, line 1150); *backlink***
>
> Unknown interpreted text role "ref".

On supported filesystems (currently ext4 and f2fs), fscrypt can use blk-crypto instead of the kernel crypto API to encrypt/decrypt file contents. To enable this, set CONFIG_FS_ENCRYPTION_INLINE_CRYPT=y in the kernel configuration, and specify the "inlinecrypt" mount option when mounting the filesystem.

Note that the "inlinecrypt" mount option just specifies to use inline encryption when possible; it doesn't force its use. fscrypt will still fall back to using the kernel crypto API on files where the inline encryption hardware doesn't have the needed crypto capabilities (e.g. support for the needed encryption algorithm and data unit size) and where blk-crypto-fallback is unusable. (For blk-crypto-fallback to be usable, it must be enabled in the kernel configuration with CONFIG_BLK_INLINE_ENCRYPTION_FALLBACK=y.)

Currently fscrypt always uses the filesystem block size (which is usually 4096 bytes) as the data unit size. Therefore, it can only use inline encryption hardware that supports that data unit size.

Inline encryption doesn't affect the ciphertext or other aspects of the on-disk format, so users may freely switch back and forth between using "inlinecrypt" and not using "inlinecrypt".

Direct I/O support

For direct I/O on an encrypted file to work, the following conditions must be met (in addition to the conditions for direct I/O on an unencrypted file):

- The file must be using inline encryption. Usually this means that the filesystem must be mounted with `-o inlinecrypt` and inline encryption hardware must be present. However, a software fallback is also available. For details, see Inline encryption support.
- The I/O request must be fully aligned to the filesystem block size. This means that the file position the I/O is targeting, the lengths of all I/O segments, and the memory addresses of all I/O buffers must be multiples of this value. Note that the filesystem block size may be greater than the logical block size of the block device.

If either of the above conditions is not met, then direct I/O on the encrypted file will fall back to buffered I/O.

Implementation details

Encryption context

An encryption policy is represented on-disk by struct fscrypt_context_v1 or struct fscrypt_context_v2. It is up to individual filesystems to decide where to store it, but normally it would be stored in a hidden extended attribute. It should *not* be exposed by the xattr-related system calls such as getxattr() and setxattr() because of the special semantics of the encryption xattr. (In particular, there would be much confusion if an encryption policy were to be added to or removed from anything other than an empty directory.) These structs are defined as follows:

```
#define FSCRYPT_FILE_NONCE_SIZE 16

#define FSCRYPT_KEY_DESCRIPTOR_SIZE  8
struct fscrypt_context_v1 {
        u8 version;
        u8 contents_encryption_mode;
        u8 filenames_encryption_mode;
        u8 flags;
        u8 master_key_descriptor[FSCRYPT_KEY_DESCRIPTOR_SIZE];
        u8 nonce[FSCRYPT_FILE_NONCE_SIZE];
};

#define FSCRYPT_KEY_IDENTIFIER_SIZE  16
struct fscrypt_context_v2 {
        u8 version;
        u8 contents_encryption_mode;
        u8 filenames_encryption_mode;
        u8 flags;
        u8 __reserved[4];
        u8 master_key_identifier[FSCRYPT_KEY_IDENTIFIER_SIZE];
        u8 nonce[FSCRYPT_FILE_NONCE_SIZE];
};
```

The context structs contain the same information as the corresponding policy structs (see Setting an encryption policy), except that the context structs also contain a nonce. The nonce is randomly generated by the kernel and is used as KDF input or as a tweak to cause different files to be encrypted differently; see Per-file encryption keys and DIRECT_KEY policies.

Data path changes

When inline encryption is used, filesystems just need to associate encryption contexts with bios to specify how the block layer or the inline encryption hardware will encrypt/decrypt the file contents.

When inline encryption isn't used, filesystems must encrypt/decrypt the file contents themselves, as described below:

For the read path (->readpage()) of regular files, filesystems can read the ciphertext into the page cache and decrypt it in-place. The page lock must be held until decryption has finished, to prevent the page from becoming visible to userspace prematurely.

For the write path (->writepage()) of regular files, filesystems cannot encrypt data in-place in the page cache, since the cached plaintext must be preserved. Instead, filesystems must encrypt into a temporary buffer or "bounce page", then write out the temporary buffer. Some filesystems, such as UBIFS, already use temporary buffers regardless of encryption. Other filesystems, such as ext4 and F2FS, have to allocate bounce pages specially for encryption.

Filename hashing and encoding

Modern filesystems accelerate directory lookups by using indexed directories. An indexed directory is organized as a tree keyed by filename hashes. When a ->lookup() is requested, the filesystem normally hashes the filename being looked up so that it can quickly find the corresponding directory entry, if any.

With encryption, lookups must be supported and efficient both with and without the encryption key. Clearly, it would not work to hash the plaintext filenames, since the plaintext filenames are unavailable without the key. (Hashing the plaintext filenames would also make it impossible for the filesystem's fsck tool to optimize encrypted directories.) Instead, filesystems hash the ciphertext filenames, i.e. the bytes actually stored on-disk in the directory entries. When asked to do a ->lookup() with the key, the filesystem just encrypts the user-supplied name to get the ciphertext.

Lookups without the key are more complicated. The raw ciphertext may contain the `\0` and `/` characters, which are illegal in filenames. Therefore, readdir() must base64url-encode the ciphertext for presentation. For most filenames, this works fine; on ->lookup(), the filesystem just base64url-decodes the user-supplied name to get back to the raw ciphertext.

However, for very long filenames, base64url encoding would cause the filename length to exceed NAME_MAX. To prevent this, readdir() actually presents long filenames in an abbreviated form which encodes a strong "hash" of the ciphertext filename, along with the optional filesystem-specific hash(es) needed for directory lookups. This allows the filesystem to still, with a high degree of confidence, map the filename given in ->lookup() back to a particular directory entry that was previously listed by readdir(). See struct fscrypt_nokey_name in the source for more details.

Note that the precise way that filenames are presented to userspace without the key is subject to change in the future. It is only meant as a way to temporarily present valid filenames so that commands like `rm -r` work as expected on encrypted directories.

Tests

To test fscrypt, use xfstests, which is Linux's de facto standard filesystem test suite. First, run all the tests in the "encrypt" group on the relevant filesystem(s). One can also run the tests with the 'inlinecrypt' mount option to test the implementation for inline encryption support. For example, to test ext4 and f2fs encryption using kvm-xfstests:

```
kvm-xfstests -c ext4,f2fs -g encrypt
kvm-xfstests -c ext4,f2fs -g encrypt -m inlinecrypt
```

UBIFS encryption can also be tested this way, but it should be done in a separate command, and it takes some time for kvm-xfstests to set up emulated UBI volumes:

```
kvm-xfstests -c ubifs -g encrypt
```

No tests should fail. However, tests that use non-default encryption modes (e.g. generic/549 and generic/550) will be skipped if the needed algorithms were not built into the kernel's crypto API. Also, tests that access the raw block device (e.g. generic/399, generic/548, generic/549, generic/550) will be skipped on UBIFS.

Besides running the "encrypt" group tests, for ext4 and f2fs it's also possible to run most xfstests with the "test_dummy_encryption" mount option. This option causes all new files to be automatically encrypted with a dummy key, without having to make any API calls. This tests the encrypted I/O paths more thoroughly. To do this with kvm-xfstests, use the "encrypt" filesystem configuration:

```
kvm-xfstests -c ext4/encrypt,f2fs/encrypt -g auto
kvm-xfstests -c ext4/encrypt,f2fs/encrypt -g auto -m inlinecrypt
```

Because this runs many more tests than "-g encrypt" does, it takes much longer to run; so also consider using gce-xfstests instead of kvm-xfstests:

```
gce-xfstests -c ext4/encrypt,f2fs/encrypt -g auto
gce-xfstests -c ext4/encrypt,f2fs/encrypt -g auto -m inlinecrypt
```