**ecryptfs\_init**

**ecryptfs\_init\_kmem\_caches()**

初始化ecryptfs\_cache\_infos数组，为数组成员的cache分配内存空间；

**register\_filesystem(&ecryptfs\_fs\_type)**

注册ecryptfs文件系统，Linux内核中的所有注册的文件系统形成一个单向链表， file\_systems为链表头；

**do\_sysfs\_registration()**

**ecryptfs\_init\_kthread()**

**ecryptfs\_init\_messaging()**

根据ecryptfs\_number\_of\_users的大小初始化ecryptfs\_daemon\_hash链表，每个用户在哈希表中的索引主要由该用户的euid决定；再根据ecryptfs\_message\_buf\_len的大小，也就是消息缓冲区的个数来初始化ecryptfs\_msg\_ctx\_arr消息传送队列，刚开始初始时都链入到ecryptfs\_msg\_ctx\_free\_list队列中；再将ecryptfs\_miscdev注册到misc设备中，Linux系统所有的misc设备的主设备号都为MISC\_MAJOR而ecryptfs文件系统所创建的misc设备次设备号由内核自动分配，自动分配是通过DYNAMIC\_MINORS来指定的。

**ecryptfs\_init\_crypto()**

初始化key\_tfm\_list双向链表；

**问题：**

1、ecryptfs文件系统extent的大小与page大小的关系？

2、Linux的slab内存分配机制？解决小块内存需求。Slab是通用分配机制，slob针对嵌入式系统，slub针对拥有大容量内存的大规模并行系统。

3、linux内核互斥锁mutex原理；

4、linux内核中的container\_of宏，根据结构中某成员的地址计算出该结构的首地址；

**ecryptfs\_mount：**

会先ecryptfs文件系统的超级块ecryptfs\_sb\_info

**ecryptfs\_parse\_options**

**ecryptfs\_opt\_sig,** **ecryptfs\_opt\_ecryptfs\_sig**

这两个参数是同一个意思，指定sig，命令行中是sig=，或者ecryptfs\_sig=

ecryptfs\_opt\_cipher, ecryptfs\_opt\_ecryptfs\_cipher

指定缺省的密码算法。

**ecryptfs文件系统内核模块可调节的参数：**

**ecryptfs\_verbosity**

**ecryptfs\_message\_buf\_len**

**ecryptfs\_message\_wait\_timeout**

**ecryptfs\_number\_of\_users**

|  |  |  |
| --- | --- | --- |
| ecryptfs\_opt\_sig | sig= |  |
| ecryptfs\_opt\_ecryptfs\_sig | ecryptfs\_sig= |  |
| ecryptfs\_opt\_cipher | cipher= |  |
| ecryptfs\_opt\_ecryptfs\_cipher | ecryptfs\_cipher= |  |
| ecryptfs\_opt\_ecryptfs\_key\_bytes |  |  |
| ecryptfs\_opt\_passthrough |  |  |
| ecryptfs\_opt\_xattr\_metadata | ecryptfs\_xattr\_metadata |  |
| ecryptfs\_opt\_encrypted\_view | ecryptfs\_encrypted\_view | ECRYPTFS\_XATTR\_METADATA\_ENABLED  ECRYPTFS\_ENCRYPTED\_VIEW\_ENABLED  会设置这两个标志位 |
| ecryptfs\_opt\_fnek\_sig |  |  |
| ecryptfs\_opt\_fn\_cipher |  |  |
| ecryptfs\_opt\_fn\_cipher\_key\_bytes |  |  |
| ecryptfs\_opt\_unlink\_sigs |  |  |
| ecryptfs\_opt\_mount\_auth\_tok\_only | ecryptfs\_mount\_auth\_tok\_only | ECRYPTFS\_GLOBAL\_MOUNT\_AUTH\_TOK\_ONLY设置了该标志，只能使用ecryptfs\_add\_global\_auth\_tok建立的token。 |
| ecryptfs\_opt\_check\_dev\_ruid |  |  |
| ecryptfs\_opt\_err |  |  |
|  |  |  |

**File.c**

**ecryptfs\_read\_update\_atime**

generic\_file\_read只更新上层inode的atime，并不更新下层磁盘文件系统inode的atime。此函数封装了generic\_file\_read，如果generic\_file\_read调用没有返回错误，则本函数对底层文件系统inode的atime进行更新。但本函数仅用于读文件的情况，读目录则通过ecryptfs\_read来完成。

**ecryptfs\_open(struct inode \*inode, struct file \*file)**

ecryptfs文件系统调用ecryptfs\_open来打开文件。

1、首先会根据输入参数file里的f\_path找到对应的dentry结构，再找到dentry结构对应的d\_sb超级块成员，进行ECRYPTFS\_ENCRYPTED\_VIEW\_ENABLED的判断。

2、分配一个新的ecryptfs\_file\_info结构，并挂接到file->private\_data；

3、通过前面的dentry，调用ecryptfs\_dentry\_to\_lower，找到底层磁盘文件系统的dentry，底层文件系统的dentry包含在ecryptfs\_dentry\_info结构中；

4、调用ecryptfs\_inode\_to\_private，找到inode对应的ecryptfs\_crypt\_stat结构，如果还没有置位，便置ECRYPTFS\_POLICY\_APPLIED标志位；

5、ecryptfs\_get\_lower\_file->ecryptfs\_init\_lower\_file->ecryptfs\_privileged\_open，ecryptfs\_privileged\_open中会调用VFS通用的dentry\_open，建立并分配底层文件系统的file结构。File结构中的f\_mapping和f\_op都从inode对应成员复制过来，并且调用底层文件系统即f\_op中的open操作。再建立一个ecryptfs\_open\_req结构，挂接到ecryptfs\_kthread\_ctl的req\_list链表尾部；

6、调用ecryptfs\_set\_file\_lower，将已经打开并分配好的底层文件系统的file指针保存在ecryptfs\_file\_info结构中的wfi\_file成员；

7、调用ecryptfs\_read\_metadata读取加密文件的元数据metadata，该函数会调用ecryptfs\_read\_lower 来读取文件头部的元数据，调用ecryptfs\_read\_headers\_virt来解析文件头部的元数据。如果不成功，则调用ecryptfs\_read\_xattr\_region读取元数据，之后仍然调用ecryptfs\_read\_headers\_virt来解析文件头部的元数据；

struct dentry{

…

void \*d\_fsdata; //VFS的dentry结构成员d\_fsdata对应了ecryptfs\_dentry\_info结构

…

};

struct ecryptfs\_dentry\_info {

struct path lower\_path;

struct ecryptfs\_crypt\_stat \*crypt\_stat;

};

struct ecryptfs\_inode\_info {

struct inode vfs\_inode; //应该是指向VFS的inode；

struct inode \*wii\_inode;

struct mutex lower\_file\_mutex;

atomic\_t lower\_file\_count;

struct file \*lower\_file;

struct ecryptfs\_crypt\_stat crypt\_stat;

};

lower\_file：底层文件系统的file结构指针，

struct file{

…

void \*private\_data; // ecryptfs文件系统对应ecryptfs\_file\_info结构

…

}

struct ecryptfs\_file\_info {

struct file \*wfi\_file;

struct ecryptfs\_crypt\_stat \*crypt\_stat;

};

wfi\_file：ecryptfs文件系统特有的ecryptfs\_file\_info结构指针是保存在VFS的file结构的private\_data中的，而wfi\_file则指向了已经被打开的底层文件系统的file结构；

static struct ecryptfs\_kthread\_ctl {

#define ECRYPTFS\_KTHREAD\_ZOMBIE 0x00000001

u32 flags;

struct mutex mux;

struct list\_head req\_list;

wait\_queue\_head\_t wait;

} ecryptfs\_kthread\_ctl;

struct ecryptfs\_open\_req {

#define ECRYPTFS\_REQ\_PROCESSED 0x00000001

#define ECRYPTFS\_REQ\_DROPPED 0x00000002

#define ECRYPTFS\_REQ\_ZOMBIE 0x00000004

u32 flags;

struct file \*\*lower\_file;

struct dentry \*lower\_dentry;

struct vfsmount \*lower\_mnt;

wait\_queue\_head\_t wait;

struct mutex mux;

struct list\_head kthread\_ctl\_list;

};

**kthread\_ctl\_list：**打开文件时，会产生一个ecryptfs\_open\_req请求，通过kthread\_ctl\_list链接到ecryptfs\_kthread\_ctl的req\_list链表尾部；

struct super\_block{

…

void \*s\_fs\_info; //保存ecryptfs文件系统特有的ecryptfs\_mount\_crypt\_stat结构；

…

}

**Crypto.c**

**ecryptfs\_read\_metadata**

**ecryptfs\_read\_headers\_virt**

　　1、调用ecryptfs\_validate\_marker验证ecryptfs文件的marker，marker=0x3c81b7f5，8字节；

2、调用ecryptfs\_i\_size\_init，如果ecryptfs\_crypt\_stat有ECRYPTFS\_METADATA\_IN\_XATTR标志，则表示文件尾部保存有扩展属性，则inode->i\_size要加上metadata\_size大小；

3、紧接着marker后，是处理标志位flags，4字节，其中高8bit是版本号。对metadata的标志位判断后，对ecryptfs\_crypt\_stat结构的flag相应位进行置位或者清除。标志位定义如下：

static struct ecryptfs\_flag\_map\_elem ecryptfs\_flag\_map[] = {

{0x00000001, ECRYPTFS\_ENABLE\_HMAC},

{0x00000002, ECRYPTFS\_ENCRYPTED},

{0x00000004, ECRYPTFS\_METADATA\_IN\_XATTR},

{0x00000008, ECRYPTFS\_ENCRYPT\_FILENAMES}

};

4、在第3步提取出来的版本号如果大于1，则调用parse\_header\_metadata解析元数据，该函数主要是确定元数据区的大小，头4个字节表示头部extent的大小，即一个extent有多少个字节，接下来两个字节表示extent的数量，这两个数相乘，即得到元数据区的总的字节数，记录在ecryptfs\_crypt\_stat的metadata\_size中；

5、最后调ecryptfs\_parse\_packet\_set进行解析，该函数定义在keystore.c中，主要对以下三种包进行判断，类型定义如下；

#define ECRYPTFS\_TAG\_3\_PACKET\_TYPE 0x8C

#define ECRYPTFS\_TAG\_1\_PACKET\_TYPE 0x01

#define ECRYPTFS\_TAG\_11\_PACKET\_TYPE 0xED

　　对于ECRYPTFS\_TAG\_1\_PACKET\_TYPE类型，调用parse\_tag\_1\_packet进行解析；

**crypto.c**

**void ecryptfs\_to\_hex(char \*dst, char \*src, size\_t src\_size)**

dst：

src：

src\_size：

**void ecryptfs\_from\_hex(char \*dst, char \*src, int dst\_size)**

**static int ecryptfs\_calculate\_md5(char \*dst,struct ecryptfs\_crypt\_stat \*crypt\_stat,char \*src, int len)**

**static int ecryptfs\_crypto\_api\_algify\_cipher\_name(char \*\*algified\_name,**

**char \*cipher\_name,char \*chaining\_modifier)**

algified\_name：将密码算法和加密模式合并成一个字符串保存在algified\_name；

cipher\_name：代表密码算法的字符串；

chaining\_modifier：代表加密模式的字符串，取值为ecb或者cbc；

**int ecryptfs\_derive\_iv(char \*iv, struct ecryptfs\_crypt\_stat \*crypt\_stat, loff\_t offset)**

iv：从crypt\_stat中的root\_iv和offset计算出md5值保存在iv；

crypt\_stat：用ecryptfs\_crypt\_stat结构中的root\_iv和iv\_bytes；

offset：对root\_iv+offset计算md5值；

**void ecryptfs\_init\_crypt\_stat(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

对crypt\_stat结构初始化，特别是ecryptfs\_crypt\_stat结构中的keysig\_list链表进行初始化；

**void ecryptfs\_destroy\_crypt\_stat(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

释放所有与crypt\_stat相关的内存，主要是释放ecryptfs\_crypt\_stat结构中的keysig\_list链表，该链表上的元素为ecryptfs\_key\_sig结构；

**void ecryptfs\_destroy\_mount\_crypt\_stat(struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat)**

释放所有与mount\_crypt\_stat相关的内存，主要是释放ecryptfs\_mount\_crypt\_stat结构中的global\_auth\_tok\_list链表，该链表上的元素为ecryptfs\_global\_auth\_tok结构；

**int virt\_to\_scatterlist(const void \*addr, int size, struct scatterlist \*sg,int sg\_size)**

**输入：**

addr：虚地址；

size：从虚地址开始的数据的大小，应为块大小的偶数倍；

sg：scatterlist数组的起始地址，如果设置为NULL则函数只返回需要的数组大小；

sg\_size：scatterlist数组的最大值；

输出：返回需要的scatterlist数组大小；

**static int encrypt\_scatterlist(struct ecryptfs\_crypt\_stat \*crypt\_stat,struct scatterlist \*dest\_sg,**

**struct scatterlist \*src\_sg, int size,unsigned char \*iv)**

crypt\_stat：

dest\_sg：存放加密后的数据；

scr\_sg：存放明文；

size：需要加密的数据长度；

iv：加密过程中用到的初始向量；

返回值：返回加密的数据长度，负数表示出错；

**static void ecryptfs\_lower\_offset\_for\_extent(loff\_t \*offset, loff\_t extent\_num,**

**struct ecryptfs\_crypt\_stat \*crypt\_stat)**

把ecryptfs文件系统的页索引（extent编号）转换成字节偏移量保存在offset。

offset：

extent\_num：

**static int ecryptfs\_encrypt\_extent(struct page \*enc\_extent\_page, struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**struct page \*page, unsigned long extent\_offset)**

对一个区（extent）的数据进行加密。成功返回0。一页可能包含几个extent。

enc\_extent\_page：加密后的数据地址；

crypt\_stat：

page：明文地址；

extent\_offset：需要加密的extent编号；

**int ecryptfs\_encrypt\_page(struct page \*page)**

分配一页内存作为临时保存加密数据的内存，根据一页内extent的大小，循环调用ecryptfs\_encrypt\_extent，再将extent编号通过ecryptfs\_lower\_offset\_for\_extent换算为字节偏移量，通过ecryptfs\_write\_lower写回底层文件系统，最后将临时缓冲区释放。

**static int ecryptfs\_decrypt\_extent(struct page \*page, struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**struct page \*enc\_extent\_page,unsigned long extent\_offset)**

**int ecryptfs\_decrypt\_page(struct page \*page)**

**static int decrypt\_scatterlist(struct ecryptfs\_crypt\_stat \*crypt\_stat, struct scatterlist \*dest\_sg,**

**struct scatterlist \*src\_sg, int size,unsigned char \*iv)**

**static int**

**ecryptfs\_encrypt\_page\_offset(struct ecryptfs\_crypt\_stat \*crypt\_stat, struct page \*dst\_page, int dst\_offset,**

**struct page \*src\_page, int src\_offset, int size,unsigned char \*iv)**

**static int**

**ecryptfs\_decrypt\_page\_offset(struct ecryptfs\_crypt\_stat \*crypt\_stat,struct page \*dst\_page, int dst\_offset,**

**struct page \*src\_page, int src\_offset, int size,unsigned char \*iv)**

**int ecryptfs\_init\_crypt\_ctx(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

对crypt\_stat结构进行初始化，主要是把ecryptfs\_crypt\_stat结构中的cipher密码算法字符串，与加密模式字符串合并后，在内核中查找对应名称的crypto\_blkcipher，把地址赋给tfm。

**static void set\_extent\_mask\_and\_shift(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

**void ecryptfs\_set\_default\_sizes(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

**int ecryptfs\_compute\_root\_iv(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

从key的md5值计算得来。

**static void ecryptfs\_generate\_new\_key(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

调用get\_random\_bytes随机数赋值给key。

**static void ecryptfs\_copy\_mount\_wide\_flags\_to\_inode\_flags(struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat)**

|  |  |
| --- | --- |
| **mount** | **inode** |
| ECRYPTFS\_XATTR\_METADATA\_ENABLED | ECRYPTFS\_METADATA\_IN\_XATTR |
| ECRYPTFS\_ENCRYPTED\_VIEW\_ENABLED | ECRYPTFS\_VIEW\_AS\_ENCRYPTED |
| ECRYPTFS\_GLOBAL\_ENCRYPT\_FILENAMES | ECRYPTFS\_ENCRYPT\_FILENAMES |
| ECRYPTFS\_GLOBAL\_ENCFN\_USE\_MOUNT\_FNEK | ECRYPTFS\_ENCFN\_USE\_MOUNT\_FNEK |
| ECRYPTFS\_GLOBAL\_ENCFN\_USE\_FEK | ECRYPTFS\_ENCFN\_USE\_FEK |

**static int ecryptfs\_copy\_mount\_wide\_sigs\_to\_inode\_sigs(struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat)**

根据mount\_crypt\_stat中的全局认证token链表global\_auth\_tok\_list上的ecryptfs\_global\_auth\_tok结构中的sig，创建ecryptfs\_key\_sig结构，并链接到ecryptfs\_crypt\_stat结构的keysig\_list链表。

**static void ecryptfs\_set\_default\_crypt\_stat\_vals(struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat)**

设置缺省的crypt\_stat值，标志位根据全局的mount\_crypt\_stat设置，缺省算法为AES，密钥长度16位。

**int ecryptfs\_new\_file\_context(struct inode \*ecryptfs\_inode)**

创建新inode即创建新文件时要调用该函数，创建该文件对应的ecryptfs\_crypt\_stat结构，该结构由i\_private保存地址。然后根据超级块中的ecryptfs\_mount\_crypt\_stat来建立文件相关的参数。ecryptfs\_mount\_crypt\_stat结构中的绝大部分参数是在安装ecryptfs文件系统时，由mount命令带的参数来决定的，有一些参数没有指定，则使用系统默认的参数。重要的参数有代表密码算法的字符串，密钥长度。还要调用随机数函数生成文件的密钥。另外要根据密码算法载入相应的内核密码处理模块，调用ecryptfs\_init\_crypt\_ctx，将相应的内核密码处理crypto\_blkcipher结构赋值给tfm。

**static int ecryptfs\_validate\_marker(char \*data)**

判断ecryptfs文件系统的标志：if ((m\_1 ^ MAGIC\_ECRYPTFS\_MARKER) == m\_2)

**static int ecryptfs\_process\_flags(struct ecryptfs\_crypt\_stat \*crypt\_stat, char \*page\_virt, int \*bytes\_read)**

处理pag\_virt地址开始的4字节的flags。

static struct ecryptfs\_flag\_map\_elem ecryptfs\_flag\_map[] = {

{0x00000001, ECRYPTFS\_ENABLE\_HMAC},

{0x00000002, ECRYPTFS\_ENCRYPTED},

{0x00000004, ECRYPTFS\_METADATA\_IN\_XATTR},

{0x00000008, ECRYPTFS\_ENCRYPT\_FILENAMES}

};

**static void write\_ecryptfs\_marker(char \*page\_virt, size\_t \*written)**

m\_2 = (m\_1 ^ MAGIC\_ECRYPTFS\_MARKER);

m\_1是取的随机数。

**void ecryptfs\_write\_crypt\_stat\_flags(char \*page\_virt, struct ecryptfs\_crypt\_stat \*crypt\_stat,size\_t \*written)**

回写4字节的flags标志到page\_virt。

**u8 ecryptfs\_code\_for\_cipher\_string(char \*cipher\_name, size\_t key\_bytes)**

根据密码算法cipher\_name以及密码长度key\_bytes，返回密码算法的编号标识。Aes密钥有16位24位和32位三种情况。

static struct ecryptfs\_cipher\_code\_str\_map\_elem

ecryptfs\_cipher\_code\_str\_map[] = {

{"aes",RFC2440\_CIPHER\_AES\_128 },

{"blowfish", RFC2440\_CIPHER\_BLOWFISH},

{"des3\_ede", RFC2440\_CIPHER\_DES3\_EDE},

{"cast5", RFC2440\_CIPHER\_CAST\_5},

{"twofish", RFC2440\_CIPHER\_TWOFISH},

{"cast6", RFC2440\_CIPHER\_CAST\_6},

{"aes", RFC2440\_CIPHER\_AES\_192},

{"aes", RFC2440\_CIPHER\_AES\_256}

};

**int ecryptfs\_cipher\_code\_to\_string(char \*str, u8 cipher\_code)**

将cipher\_code转换成密码算法的字符串。

**int ecryptfs\_read\_and\_validate\_header\_region(struct inode \*inode)**

调用ecryptfs\_read\_lower从底层文件系统读取文件长度和ecryptfs文件系统的marker并判断marker是否正确。如果正确则对inode的i\_size重新进行初始化，主要考虑的因素有：如果有ECRYPTFS\_METADATA\_IN\_XATTR属性，则要将文件大小加上metadata的大小。

**void ecryptfs\_write\_header\_metadata(char \*virt, struct ecryptfs\_crypt\_stat \*crypt\_stat, size\_t \*written)**

根据crypt\_stat中的extent\_size（字节数）和metadata\_size（字节数），计算出metadata占用的extent数，将extent\_size保存前virt的前4个字节，metadata的extent数保存在接下来2个字节。

**static int ecryptfs\_write\_headers\_virt(char \*page\_virt, size\_t max, size\_t \*size,**

**struct ecryptfs\_crypt\_stat \*crypt\_stat, struct dentry \*ecryptfs\_dentry)**

|  |  |  |
| --- | --- | --- |
| 0-7字节 | | 文件大小(big-endian) |
| 8-15字节 | | Ecryptfs文件系统的marker |
| 16-19字节 | 16字节 | 文件格式的版本号，0-255 |
| 17-18 | 保留 |
| 19 | 1bit保留；2bit：是否加密；3bit保留 |
| 20-23 | | 文件头部区的字节数(big-endian) |
| 24-25 | | 文件头部区的数量 |
| 26 | | RFC2440的认证token包 |
| Data Extent 0 | | Lower data (CBC encrypted) |
| Data Extent 1 | | Lower data (CBC encrypted) |
| …… | |  |

|  |
| --- |
| write\_ecryptfs\_marker |
| ecryptfs\_write\_crypt\_stat\_flags |
| ecryptfs\_write\_header\_metadata |
| ecryptfs\_generate\_key\_packet\_set |

**static int ecryptfs\_write\_metadata\_to\_contents(struct inode \*ecryptfs\_inode, char \*virt, size\_t virt\_len)**

调用ecryptfs\_write\_lower将metadata写到文件的头部。

**static int ecryptfs\_write\_metadata\_to\_xattr(struct dentry \*ecryptfs\_dentry,char \*page\_virt, size\_t size)**

调用ecryptfs\_setxattr将metadata写到文件的扩展属性中。

**static unsigned long ecryptfs\_get\_zeroed\_pages(gfp\_t gfp\_mask, unsigned int order)**

**int ecryptfs\_write\_metadata(struct dentry \*ecryptfs\_dentry,struct inode \*ecryptfs\_inode)**

|  |  |
| --- | --- |
| ecryptfs\_get\_zeroed\_pages |  |
| ecryptfs\_write\_headers\_virt |  |
| ecryptfs\_write\_metadata\_to\_xattr |  |
| ecryptfs\_write\_metadata\_to\_contents |  |

**static int parse\_header\_metadata(struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**char \*virt, int \*bytes\_read,int validate\_header\_size)**

crypt\_stat->metadata\_size = (((size\_t)num\_header\_extents\_at\_front \* (size\_t)header\_extent\_size));

**static void set\_default\_header\_data(struct ecryptfs\_crypt\_stat \*crypt\_stat)**

crypt\_stat->metadata\_size = ECRYPTFS\_MINIMUM\_HEADER\_EXTENT\_SIZE;

void ecryptfs\_i\_size\_init(const char \*page\_virt, struct inode \*inode)

**static int ecryptfs\_read\_headers\_virt(char \*page\_virt,struct ecryptfs\_crypt\_stat \*crypt\_stat,**

**struct dentry \*ecryptfs\_dentry,int validate\_header\_size)**

主要是解析page\_virt里的header信息，对crypt\_stat结构成员进行赋值。

|  |  |
| --- | --- |
| ecryptfs\_set\_default\_sizes |  |
| ecryptfs\_validate\_marker |  |
| ecryptfs\_i\_size\_init |  |
| ecryptfs\_process\_flags |  |
| parse\_header\_metadata |  |
| set\_default\_header\_data |  |
| ecryptfs\_parse\_packet\_set |  |

**int ecryptfs\_read\_xattr\_region(char \*page\_virt, struct inode \*ecryptfs\_inode)**

调用ecryptfs\_getxattr\_lower读取扩展属性到page\_virt。

**int ecryptfs\_read\_and\_validate\_xattr\_region(struct dentry \*dentry, struct inode \*inode)**

**int ecryptfs\_read\_metadata(struct dentry \*ecryptfs\_dentry)**

|  |  |
| --- | --- |
| ecryptfs\_copy\_mount\_wide\_flags\_to\_inode\_flags |  |
| ecryptfs\_read\_lower |  |
| ecryptfs\_read\_headers\_virt |  |
| ecryptfs\_read\_xattr\_region |  |
|  |  |
|  |  |
|  |  |

**static int ecryptfs\_encrypt\_filename(struct ecryptfs\_filename \*filename,**

**struct ecryptfs\_crypt\_stat \*crypt\_stat, struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat)**

加密文件名。要调两次ecryptfs\_write\_tag\_70\_packet，第一次根据filename\_size和包的格式计算出encrypted\_filename\_size；第二次才计算出encrypted\_filename。

**static int ecryptfs\_copy\_filename(char \*\*copied\_name, size\_t \*copied\_name\_size,**

**const char \*name, size\_t name\_size)**

**static int**

**ecryptfs\_process\_key\_cipher(struct crypto\_blkcipher \*\*key\_tfm,char \*cipher\_name, size\_t \*key\_size)**

|  |  |
| --- | --- |
| ecryptfs\_crypto\_api\_algify\_cipher\_name |  |
| **crypto\_alloc\_blkcipher** |  |
| **crypto\_blkcipher\_set\_flags** |  |
| get\_random\_bytes |  |
| **crypto\_blkcipher\_setkey** |  |

**int \_\_init ecryptfs\_init\_crypto(void)**

主要对key\_tfm\_list表头进行初始化。

**int ecryptfs\_destroy\_crypto(void)**

释放key\_tfm\_list链表上所有的ecryptfs\_key\_tfm。

**int ecryptfs\_add\_new\_key\_tfm(struct ecryptfs\_key\_tfm \*\*key\_tfm, char \*cipher\_name,size\_t key\_size)**

主要调用ecryptfs\_process\_key\_cipher对ecryptfs\_key\_tfm中的key\_tfm进行初始化。

**int ecryptfs\_tfm\_exists(char \*cipher\_name, struct ecryptfs\_key\_tfm \*\*key\_tfm)**

在key\_tfm\_list中查找对应cipher\_name的ecryptfs\_key\_tfm，若找到就把其地址保存在\*key\_tfm，成功返回1。

**int ecryptfs\_get\_tfm\_and\_mutex\_for\_cipher\_name(struct crypto\_blkcipher \*\*tfm,**

**struct mutex \*\*tfm\_mutex,char \*cipher\_name)**

|  |  |
| --- | --- |
| ecryptfs\_tfm\_exists |  |
| ecryptfs\_add\_new\_key\_tfm |  |

(\*tfm) = key\_tfm->key\_tfm;

(\*tfm\_mutex) = &key\_tfm->key\_tfm\_mutex;

**void ecryptfs\_encode\_for\_filename(unsigned char \*dst, size\_t \*dst\_size,unsigned char \*src, size\_t src\_size)**

portable\_filename\_chars

filename\_rev\_map

主要是把加密后的文件名变成文件系统可打印的字符串。

**static size\_t ecryptfs\_max\_decoded\_size(size\_t encoded\_size)**

**static void ecryptfs\_decode\_from\_filename(unsigned char \*dst, size\_t \*dst\_size,**

**const unsigned char \*src, size\_t src\_size)**

问题同上。

**int ecryptfs\_encrypt\_and\_encode\_filename(char \*\*encoded\_name,size\_t \*encoded\_name\_size,**

**struct ecryptfs\_crypt\_stat \*crypt\_stat,struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat,**

**const char \*name, size\_t name\_size)**

|  |  |
| --- | --- |
| ecryptfs\_encrypt\_filename |  |
| ecryptfs\_encode\_for\_filename |  |

先加密后进行编码转换。

**int ecryptfs\_decode\_and\_decrypt\_filename(char \*\*plaintext\_name,size\_t \*plaintext\_name\_size,**

**struct dentry \*ecryptfs\_dir\_dentry, const char \*name, size\_t name\_size)**

|  |  |
| --- | --- |
| ecryptfs\_decode\_from\_filename |  |
| ecryptfs\_parse\_tag\_70\_packet |  |
|  |  |
|  |  |

**int ecryptfs\_set\_f\_namelen(long \*namelen, long lower\_namelen,**

**struct ecryptfs\_mount\_crypt\_stat \*mount\_crypt\_stat)**