Linux内核支持很多加密算法，包括对称加密算法，如AES；摘要算法，如sha1,md5；压缩算法，如deflate。不过内核好像不支持非对称加密算法。这些算法作为加密函数框架的最底层，提供加密和解密的实际操作。这些函数可以在内核crypto文件夹下，相应的文件中找到。不过内核模块不能直接调用这些函数，因为它们并没有export。内核提供一个统一的框架，来管理这些算法。加密算法通过crypto\_register\_alg()和crypto\_unregister\_alg()注册。内核将加密算法分为三类，1）cipher，2）compress，3）digest。加密函数框架中有相应的API封装，提供给模块调用。

对于使用这些加密函数，首先通过crypto\_alloc\_tfm()来分配一个加密函数对象的实例。初始化这些实例，然后就可以通过框架提供的API对数据进行加密和解密。完成以后，必须通过crypto\_free\_tfm()撤销实例。

算法模版

1. 模版的基本概念

算法模版是加密框架的第一个重要概念。内核中有很多算法是动态生成的，例如cbc(des)算法。内核并不存在这样的算法，它事实上是cbc和des的组合，但是内核加密框架从统一抽像管理的角度。将cbc(des)看做一个算法，在实际使用时动态分配并向内核注册该算法。这样，可以将cbc抽像为一个模版，它可以同任意的加密算法进行组合。算法模版使用结构crypto\_template来描述，其结构原型：

struct crypto\_template {

struct list\_head list; //模版链表成员，用于注册

struct hlist\_head instances; //算法实例链表首部

struct module \*module; //模块指针

struct crypto\_instance \*(\*alloc)(struct rtattr \*\*tb); //算法实例分配

void (\*free)(struct crypto\_instance \*inst); //算法实例释放

char name[CRYPTO\_MAX\_ALG\_NAME]; //模版名称

};

例如，一个名为cbc的算法模版，可以用它来动态分配cbc(des),cbc(twofish)……诸如此类。  
  
crypto/algapi.c下包含了模版的一些常用操作。最为常见的就是模版的注册与注销，其实质是对以crypto\_template\_list为首的链表的操作过程：

static LIST\_HEAD(crypto\_template\_list);

int crypto\_register\_template(struct crypto\_template \*tmpl)

{

        struct crypto\_template \*q;

        int err = -EEXIST;

        down\_write(&crypto\_alg\_sem);

        //遍历crypto\_template\_list，看当前模板是否被注册

        list\_for\_each\_entry(q, &crypto\_template\_list, list) {

                if (q == tmpl)

                        goto out;

        }

        //注册之

        list\_add(&tmpl->list, &crypto\_template\_list);

        //事件通告

        crypto\_notify(CRYPTO\_MSG\_TMPL\_REGISTER, tmpl);

        err = 0;

out:

        up\_write(&crypto\_alg\_sem);

        return err;

}

EXPORT\_SYMBOL\_GPL(crypto\_register\_template);

注销算法模版，除了模版本身，还有一个重要的内容是处理算法模版产生的算法实例，关于算法实例，后文详述。

void crypto\_unregister\_template(struct crypto\_template \*tmpl)

{

        struct crypto\_instance \*inst;

        struct hlist\_node \*p, \*n;

        struct hlist\_head \*list;

        LIST\_HEAD(users);

        down\_write(&crypto\_alg\_sem);

        BUG\_ON(list\_empty(&tmpl->list));

        //注销算法模版，并重新初始化模版的list成员

        list\_del\_init(&tmpl->list);

        //首先移除模版上的所有算法实例

        list = &tmpl->instances;

        hlist\_for\_each\_entry(inst, p, list, list) {

                int err = crypto\_remove\_alg(&inst->alg, &users);

                BUG\_ON(err);

        }

        crypto\_notify(CRYPTO\_MSG\_TMPL\_UNREGISTER, tmpl);

        up\_write(&crypto\_alg\_sem);

        //释放模版的所有算法实例分配的内存

        hlist\_for\_each\_entry\_safe(inst, p, n, list, list) {

                BUG\_ON(atomic\_read(&inst->alg.cra\_refcnt) != 1);

                tmpl->free(inst);

        }

        crypto\_remove\_final(&users);

}

EXPORT\_SYMBOL\_GPL(crypto\_unregister\_template);

2.        算法模版的查找

crypto\_lookup\_template函数根据名称，查找相应的模版：

struct crypto\_template \*crypto\_lookup\_template(const char \*name)

{

        return try\_then\_request\_module(\_\_crypto\_lookup\_template(name), name);

}

\_\_crypto\_lookup\_template完成实质的模版模找工作，而try\_then\_request\_module则尝试动态插入相应的内核模块，如果需要的话：

static struct crypto\_template \*\_\_crypto\_lookup\_template(const char \*name)

{

        struct crypto\_template \*q, \*tmpl = NULL;

        down\_read(&crypto\_alg\_sem);

        //遍历crypto\_template\_list链，匹备模版名称

        list\_for\_each\_entry(q, &crypto\_template\_list, list) {

                if (strcmp(q->name, name))

                        continue;

//查找命中，需要对其增加引用，防止其正在使用时，模块被卸载。完成该操作后返回查找到的模版

                if (unlikely(!crypto\_tmpl\_get(q)))

                        continue;

                tmpl = q;

                break;

        }

        up\_read(&crypto\_alg\_sem);

        return tmpl;

}

3.        模版的算法实例分配时机  
模版可以看做一个静态的概念，其只有被动态创建后才具有生命力，本文将模版通过alloc分配创建的算法（对像）称为“实例(instance)”。  
算法模版的核心作用是，上层调用者构造一个完整合法的算法名称，如hmac(md5)，触发模版的alloc动作，为该名称分配一个算法实例，类似于为类实例化一个对像，最终的目的还是使用算法本身。对于xfrm来说，一个典型的算法模版的实例分配触发流程如下所述:  
xfrm包裹了一层加密框架支持，参后文“ xfrm加密框架”一节，其算法查找函数为xfrm\_find\_algo，它调用crypto\_has\_alg函数进行算法的查找，以验证自己支持的算法是否被内核支持，如xfrm支持cbc(des)，但此时并不知道内核是否有这个算法(如果该算法首次被使用，则还没有分配算法实例)。crypto\_has\_alg会调用crypto\_alg\_mod\_lookup完成查找工作，crypto\_alg\_mod\_lookup函数查找不命中，会调用crypto\_probing\_notify函数进行请求探测：

struct crypto\_alg \*crypto\_alg\_mod\_lookup(const char \*name, u32 type, u32 mask)

{

        ……

        ok = crypto\_probing\_notify(CRYPTO\_MSG\_ALG\_REQUEST, larval);

        ……

}

请求是通过通知链表来通告的：

int crypto\_probing\_notify(unsigned long val, void \*v)

{

        int ok;

        ok = blocking\_notifier\_call\_chain(&crypto\_chain, val, v);

        if (ok == NOTIFY\_DONE) {

                request\_module("cryptomgr");

                ok = blocking\_notifier\_call\_chain(&crypto\_chain, val, v);

        }

        return ok;

}

在algboss.c中注册了一个名为cryptomgr\_notifier的通告块结构，其通告处理函数为cryptomgr\_notify：

static struct notifier\_block cryptomgr\_notifier = {

        .notifier\_call = cryptomgr\_notify,

};

static int \_\_init cryptomgr\_init(void)

{

        return crypto\_register\_notifier(&cryptomgr\_notifier);

}

static void \_\_exit cryptomgr\_exit(void)

{

        int err = crypto\_unregister\_notifier(&cryptomgr\_notifier);

        BUG\_ON(err);

}

这样，当有算法被使用的时候，会调用通告块的处理函数cryptomgr\_notify，因为此时的消息是CRYPTO\_MSG\_ALG\_REQUEST，所以cryptomgr\_schedule\_probe进行算法的探测：

static int cryptomgr\_notify(struct notifier\_block \*this, unsigned long msg,  void \*data)

{

        switch (msg) {

        case CRYPTO\_MSG\_ALG\_REQUEST:

                return cryptomgr\_schedule\_probe(data);

……

        return NOTIFY\_DONE;

}

cryptomgr\_schedule\_probe启动一个名为cryptomgr\_probe的内核线程来进行算法模版的探测：

static int cryptomgr\_schedule\_probe(struct crypto\_larval \*larval)

{

        ……

        //构造param，以供后面使用

        ……

        thread = kthread\_run(cryptomgr\_probe, param, "cryptomgr\_probe");

        ……

}

cryptomgr\_probe完成具体的算法探测过程：

static int cryptomgr\_probe(void \*data)

{

        struct cryptomgr\_param \*param = data;

        struct crypto\_template \*tmpl;

        struct crypto\_instance \*inst;

        int err;

        //查找算法模版

        tmpl = crypto\_lookup\_template(param->template);

        if (!tmpl)

                goto err;

        //循环调用模版的alloc函数分配算法实列，并将模版注册之

        //这里值得注意的是循环的条件，当返回码为-EAGAIN时，会循环再次尝试

        //这样使用的一个场景后面会分析到

        do {

                inst = tmpl->alloc(param->tb);

                if (IS\_ERR(inst))

                        err = PTR\_ERR(inst);

                else if ((err = crypto\_register\_instance(tmpl, inst)))

                        tmpl->free(inst);

        } while (err == -EAGAIN && !signal\_pending(current));

        //查找中会增加引用，这里已经用完了释放之

        crypto\_tmpl\_put(tmpl);

        if (err)

                goto err;

out:

        kfree(param);

        module\_put\_and\_exit(0);

err:

        crypto\_larval\_error(param->larval, param->otype, param->omask);

        goto out;

}

理解了算法的注册与查找后，再来理解这个函数就非常容易了，其核心在do{}while循环中，包含了算法实例的分配和注册动作。针对每一种算法模版，其alloc动作不尽一致。后文会对xfrm使用的算法模版一一阐述。  
  
为什么不把“算法实例”直接称之为“算法”，这是因为实例包含了更多的内容，其由结构struct crypto\_instance可以看出：

struct crypto\_instance {

        struct crypto\_alg alg;        //对应的算法名称

        struct crypto\_template \*tmpl;        //所属的算法模版

        struct hlist\_node list;                        //链表成员

        void \*\_\_ctx[] CRYPTO\_MINALIGN\_ATTR;  //上下文信息指针

};

内核使用struct crypto\_alg描述一个算法（该结构在后文使用时再来分析），可见一个算法实例除了包含其对应的算法，还包含更多的内容。  
  
当分配成功后，cryptomgr\_probe会调用crypto\_register\_instance将其注册，以期将来可以顺利地找到并使用它：

int crypto\_register\_instance(struct crypto\_template \*tmpl,

                             struct crypto\_instance \*inst)

{

        struct crypto\_larval \*larval;

        int err;

        //对算法进行合法性检查，并构造完整的驱动名称

        err = crypto\_check\_alg(&inst->alg);

        if (err)

                goto err;

        //设置算法内核模块指针指向所属模版

        inst->alg.cra\_module = tmpl->module;

        down\_write(&crypto\_alg\_sem);

        //注册算法实例对应的算法

        larval = \_\_crypto\_register\_alg(&inst->alg);

        if (IS\_ERR(larval))

                goto unlock;

        //成功后，将算法再注册到所属的模版上面

        hlist\_add\_head(&inst->list, &tmpl->instances);

        //设置模版指针

        inst->tmpl = tmpl;

unlock:

        up\_write(&crypto\_alg\_sem);

        err = PTR\_ERR(larval);

        if (IS\_ERR(larval))

                goto err;

        crypto\_wait\_for\_test(larval);

        err = 0;

err:

        return err;

}

注册的一个重要工作，就是调用\_\_crypto\_register\_alg将实例所对应的算法注册到加密框架子系统中。算法注册成功后，上层调用者就可以调用crypto\_alg\_mod\_lookup等函数进行查找，并使用该算法了。

三、        HMAC  
MAC（消息认证码）与hash函数非常相似，只是生成固定长度的消息摘要时需要秘密的密钥而已。  
HAMC是密钥相关的哈希运算消息认证码（keyed-Hash Message Authentication Code）,HMAC运算利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出。具体的算法描述详见：<http://baike.baidu.com/view/1136366.htm?fr=ala0_1>。  
根据HMAC的特点(可以和类似md5、sha等hash算法组合，构造出hmac(md5)这样的算法)，Linux 加密框架将其抽像为一个算法模版。本章将假设上层调用者使用了名为hmac(md5)的算法，展示这一算法是如何被构造、初始化及调用以实现数据验证的。  
  
1.        算法模版的注册与注销

1. static struct crypto\_template hmac\_tmpl = {
2. .name = "hmac",
3. .alloc = hmac\_alloc,
4. .free = hmac\_free,
5. .module = THIS\_MODULE,
6. };
7. static int \_\_init hmac\_module\_init(void)
8. {
9. return crypto\_register\_template(&hmac\_tmpl);
10. }
11. static void \_\_exit hmac\_module\_exit(void)
12. {
13. crypto\_unregister\_template(&hmac\_tmpl);
14. }

模版的注册与注销前文已经描述过了。  
  
2.        算法实例的分配  
当一个算法需要被使用却查找不到的时候，会尝试调用其模版对应分配相应的算法实列，这也适用于hmac，其alloc函数指针指向hmac\_alloc：

1. static struct crypto\_instance \* hmac\_alloc (struct rtattr \*\*tb)
2. {
3. struct crypto\_instance \*inst;
4. struct crypto\_alg \*alg;
5. int err;
6. int ds;
7. //类型检查，所属算法必需为hash类型
8. err = crypto\_check\_attr\_type(tb, CRYPTO\_ALG\_TYPE\_HASH);
9. if (err)
10. return ERR\_PTR(err);
11. //根据参数名称，查找相应的子算法，如md5,shax等
12. alg = crypto\_get\_attr\_alg(tb, CRYPTO\_ALG\_TYPE\_HASH,
13. CRYPTO\_ALG\_TYPE\_HASH\_MASK);
14. //查找失败
15. if (IS\_ERR(alg))
16. return ERR\_CAST(alg);
17. //初始化算法实例
18. inst = ERR\_PTR(-EINVAL);
20. //计算算法实列的消息摘要大小(输出大小)
21. ds = alg->cra\_type == &crypto\_hash\_type ?
22. alg->cra\_hash.digestsize :
23. alg->cra\_type ?
24. \_\_crypto\_shash\_alg(alg)->digestsize :
25. alg->cra\_digest.dia\_digestsize;
26. if (ds > alg->cra\_blocksize)
27. goto out\_put\_alg;
28. //分配一个算法实列，这样，一个新的算法，如hmac(md5)就横空出世了
29. inst = crypto\_alloc\_instance("hmac", alg);
30. //分配失败
31. if (IS\_ERR(inst))
32. goto out\_put\_alg;
33. //初始化算法实例，其相应的成员等于其子算法中的对应成员
34. //类型
35. inst->alg.cra\_flags = CRYPTO\_ALG\_TYPE\_HASH;
36. //优先级
37. inst->alg.cra\_priority = alg->cra\_priority;
38. //计算消息摘要的块长度(输入大小)
39. inst->alg.cra\_blocksize = alg->cra\_blocksize;
40. //对齐掩码
41. inst->alg.cra\_alignmask = alg->cra\_alignmask;
42. //类型指针指向crypto\_hash\_type
43. inst->alg.cra\_type = &crypto\_hash\_type;
44. //消息摘要大小
45. inst->alg.cra\_hash.digestsize = ds;
46. //计算算法所需的上下文空间大小
47. inst->alg.cra\_ctxsize = sizeof(struct hmac\_ctx) +
48. ALIGN(inst->alg.cra\_blocksize \* 2 + ds,
49. sizeof(void \*));
50. //初始化和退出函数
51. inst->alg.cra\_init = hmac\_init\_tfm;
52. inst->alg.cra\_exit = hmac\_exit\_tfm;
53. //置相应hash算法的操作函数，包含hash函数标准的init/update/final和digest/setkey
54. inst->alg.cra\_hash.init = hmac\_init;
55. inst->alg.cra\_hash.update = hmac\_update;
56. inst->alg.cra\_hash.final = hmac\_final;
57. //消息摘要函数
58. inst->alg.cra\_hash.digest = hmac\_digest;
59. //setkey(密钥设置函数)
60. inst->alg.cra\_hash.setkey = hmac\_setkey;
61. out\_put\_alg:
62. crypto\_mod\_put(alg);
63. return inst;
64. }

每个模版的alloc动作虽不同，但是它们基本上遵循一些共性的操作：  
1、        合法性检验，如类型检查；  
2、        取得其子算法（即被模版所包裹的算法，如hmac(md5)中，就是md5）的算法指针;   
3、        调用crypto\_alloc\_instance分配一个相应的算法实列；  
4、        对分配成功的算法实例进行实始化，这也是理解该算法实例最核心的部份，因为它初始化算法运行所需的一些必要参数和虚函数指针；  
  
crypto\_alloc\_instance(algapi.c) 函数用于分配一个算法实例，这个函数有两个重要功能，一个是分配内存空间，另一个是初始化spawn。

1. //name: 模版名称
2. //alg:模版的子算法
3. struct crypto\_instance \*crypto\_alloc\_instance(const char \*name,
4. struct crypto\_alg \*alg)
5. {
6. struct crypto\_instance \*inst;
7. struct crypto\_spawn \*spawn;
8. int err;
9. //分配一个算法实例，crypto\_instance结构的最后一个成员ctx是一个指针变量，所以，在分配空间的时候，在其尾部追加相应的空间，可以使用ctx访问之。
10. //另一个重要的概念是，算法实例中包含了算法，这个分配，同时也完成了算法实例对应的算法的分配工作。
11. inst = kzalloc(sizeof(\*inst) + sizeof(\*spawn), GFP\_KERNEL);
12. if (!inst)
13. return ERR\_PTR(-ENOMEM);
14. err = -ENAMETOOLONG;
15. //构造完成的算法名称
16. if (snprintf(inst->alg.cra\_name, CRYPTO\_MAX\_ALG\_NAME, "%s(%s)", name,
17. alg->cra\_name) >= CRYPTO\_MAX\_ALG\_NAME)
18. goto err\_free\_inst;
19. //构造完整的算法驱动名称
20. if (snprintf(inst->alg.cra\_driver\_name, CRYPTO\_MAX\_ALG\_NAME, "%s(%s)",
21. name, alg->cra\_driver\_name) >= CRYPTO\_MAX\_ALG\_NAME)
22. goto err\_free\_inst;
23. //spawn指向算法实例的上下文成员，可以这样做是因为\_\_ctx是一个可变长的成员，在分配实例的时候，
24. //在尾部增加了一个spawn的空间
25. spawn = crypto\_instance\_ctx(inst);
26. //初始化spawn
27. err = crypto\_init\_spawn(spawn, alg, inst,
28. CRYPTO\_ALG\_TYPE\_MASK | CRYPTO\_ALG\_ASYNC);
29. if (err)
30. goto err\_free\_inst;
31. return inst;
32. err\_free\_inst:
33. kfree(inst);
34. return ERR\_PTR(err);
35. }

crypto\_instance\_ctx取出算法实例的ctx指针，返回值是void \*，这意味着可以根具不同的需要，将其转换为所需的类型：

1. static inline void \*crypto\_instance\_ctx(struct crypto\_instance \*inst)
2. {
3. return inst->\_\_ctx;
4. }

一个算法实例被分配成员后，其会被注册至加密子系统，这样，一个算法，例如，hmac(md5)就可以直接被使用了。  
  
3.        待孵化的卵  
        已经看到了从模版到算法实例的第一层抽像，每个算法在每一次被使用时，它们的运行环境不尽相同，例如，可能会拥有不同的密钥。将算法看成一个类，则在每一次运行调用时，需要为它产生一个“对像”，这在内核中被称为transform，简称为tfm。后文会详细看到分配一个tfm的过程，现在引入这一概念，主要是为了分析spawn。  
加密或认证算法，在调用时，都需要分配其算法对应的tfm，在分配算法实例的同时，并没有为之分配相应的tfm结构，这是因为真正的算法还没有被调用，这并不是进行tfm结构分配的最佳地点。在初始化算法实例的时候，加密框架使用了XXX\_spawn\_XXX函数簇来解决这一问题。这样的算法对像，被称为spawn(卵)。也就是说，在算法实例分配的时候，只是下了一个蛋（设置好spawn），等到合适的时候来对其进行孵化，这个“合适的时候”，通常指为调用算法实际使用的时候。  
  
在crypto\_alloc\_instance分配算法实例的时候，就顺便分配了spawn，然后调用crypto\_init\_spawn对其进行初始化：

1. int crypto\_init\_spawn(struct crypto\_spawn \*spawn, struct crypto\_alg \*alg,
2. struct crypto\_instance \*inst, u32 mask)
3. {
4. int err = -EAGAIN;
5. //初始化其成员
6. spawn->inst = inst;
7. spawn->mask = mask;
8. down\_write(&crypto\_alg\_sem);
9. if (!crypto\_is\_moribund(alg)) {
10. //加入链表，每个spawn，都被加入到算法的cra\_users链，即算做算法的一个用户
11. list\_add(&spawn->list, &alg->cra\_users);
12. //spawn的alg成员指针指向当前成员，这就方便引用了
13. spawn->alg = alg;
14. err = 0;
15. }
16. up\_write(&crypto\_alg\_sem);
17. return err;
18. }

所以，所谓算法的spawn的初始化，就是初始化crypto\_spawn结构，核心的操作是设置其对应的算法实例、算法，以及一个加入算法的链表的过程。

4.        算法的初始化  
有了算法实例，仅表示内核拥有这一种“算法”——加引号的意思是说，它可能并不以类似md5.c这样的源代码形式存现，而是通过模版动态创建的。实际要使用该算法，需要为算法分配“运行的对像”，即tfm。  
  
4.1        tfm  
内核加密框架中，使用结构crypto\_alg来描述一个算法，每一个算法(实例)相当于一个类，在实际的使用环境中，需要为它分配一个对像，在内核加密框架中，这个“对像”被称为transform（简称tfm）。transform意味“变换”，可能译为“蜕变”更为合适。作者对它的注释是：  
/\*  
\* Transforms: user-instantiated objects which encapsulate algorithms  
\* and core processing logic.  Managed via crypto\_alloc\_\*() and  
\* crypto\_free\_\*(), as well as the various helpers below.  
……  
\*/  
  
tfm是加密框架中一个极为重要的概念，它由结构crypto\_tfm描述：

1. struct crypto\_tfm {
2. u32 crt\_flags;
4. union {
5. struct ablkcipher\_tfm ablkcipher;
6. struct aead\_tfm aead;
7. struct blkcipher\_tfm blkcipher;
8. struct cipher\_tfm cipher;
9. struct hash\_tfm hash;
10. struct ahash\_tfm ahash;
11. struct compress\_tfm compress;
12. struct rng\_tfm rng;
13. } crt\_u;
14. void (\*exit)(struct crypto\_tfm \*tfm);
16. struct crypto\_alg \*\_\_crt\_alg;
17. void \*\_\_crt\_ctx[] CRYPTO\_MINALIGN\_ATTR;
18. };

这些成员的作用，将在后面一一看到，值得注意的是，针对每种算法不同，结构定义了一个名为crt\_u的联合体，以对应每种算法的tfm的具体操作，例如加密/解密，求hash，压缩/解压等，加密框架引入了一组名为xxx\_tfm的结构封装，xxx表示算法类型，也就是crt\_u成员。其定义如下：

1. struct ablkcipher\_tfm {
2. int (\*setkey)(struct crypto\_ablkcipher \*tfm, const u8 \*key,
3. unsigned int keylen);
4. int (\*encrypt)(struct ablkcipher\_request \*req);
5. int (\*decrypt)(struct ablkcipher\_request \*req);
6. int (\*givencrypt)(struct skcipher\_givcrypt\_request \*req);
7. int (\*givdecrypt)(struct skcipher\_givcrypt\_request \*req);
8. struct crypto\_ablkcipher \*base;
9. unsigned int ivsize;
10. unsigned int reqsize;
11. };
12. struct aead\_tfm {
13. int (\*setkey)(struct crypto\_aead \*tfm, const u8 \*key,
14. unsigned int keylen);
15. int (\*encrypt)(struct aead\_request \*req);
16. int (\*decrypt)(struct aead\_request \*req);
17. int (\*givencrypt)(struct aead\_givcrypt\_request \*req);
18. int (\*givdecrypt)(struct aead\_givcrypt\_request \*req);
19. struct crypto\_aead \*base;
20. unsigned int ivsize;
21. unsigned int authsize;
22. unsigned int reqsize;
23. };
24. struct blkcipher\_tfm {
25. void \*iv;
26. int (\*setkey)(struct crypto\_tfm \*tfm, const u8 \*key,
27. unsigned int keylen);
28. int (\*encrypt)(struct blkcipher\_desc \*desc, struct scatterlist \*dst,
29. struct scatterlist \*src, unsigned int nbytes);
30. int (\*decrypt)(struct blkcipher\_desc \*desc, struct scatterlist \*dst,
31. struct scatterlist \*src, unsigned int nbytes);
32. };
33. struct cipher\_tfm {
34. int (\*cit\_setkey)(struct crypto\_tfm \*tfm,
35. const u8 \*key, unsigned int keylen);
36. void (\*cit\_encrypt\_one)(struct crypto\_tfm \*tfm, u8 \*dst, const u8 \*src);
37. void (\*cit\_decrypt\_one)(struct crypto\_tfm \*tfm, u8 \*dst, const u8 \*src);
38. };
39. struct hash\_tfm {
40. int (\*init)(struct hash\_desc \*desc);
41. int (\*update)(struct hash\_desc \*desc,
42. struct scatterlist \*sg, unsigned int nsg);
43. int (\*final)(struct hash\_desc \*desc, u8 \*out);
44. int (\*digest)(struct hash\_desc \*desc, struct scatterlist \*sg,
45. unsigned int nsg, u8 \*out);
46. int (\*setkey)(struct crypto\_hash \*tfm, const u8 \*key,
47. unsigned int keylen);
48. unsigned int digestsize;
49. };
50. struct ahash\_tfm {
51. int (\*init)(struct ahash\_request \*req);
52. int (\*update)(struct ahash\_request \*req);
53. int (\*final)(struct ahash\_request \*req);
54. int (\*digest)(struct ahash\_request \*req);
55. int (\*setkey)(struct crypto\_ahash \*tfm, const u8 \*key,
56. unsigned int keylen);
57. unsigned int digestsize;
58. unsigned int reqsize;
59. };
60. struct compress\_tfm {
61. int (\*cot\_compress)(struct crypto\_tfm \*tfm,
62. const u8 \*src, unsigned int slen,
63. u8 \*dst, unsigned int \*dlen);
64. int (\*cot\_decompress)(struct crypto\_tfm \*tfm,
65. const u8 \*src, unsigned int slen,
66. u8 \*dst, unsigned int \*dlen);
67. };
68. struct rng\_tfm {
69. int (\*rng\_gen\_random)(struct crypto\_rng \*tfm, u8 \*rdata,
70. unsigned int dlen);
71. int (\*rng\_reset)(struct crypto\_rng \*tfm, u8 \*seed, unsigned int slen);
72. };

为了直接访问这些成员，定义了如下宏：

1. #define crt\_ablkcipher        crt\_u.ablkcipher
2. #define crt\_aead        crt\_u.aead
3. #define crt\_blkcipher        crt\_u.blkcipher
4. #define crt\_cipher        crt\_u.cipher
5. #define crt\_hash        crt\_u.hash
6. #define crt\_ahash        crt\_u.ahash
7. #define crt\_compress        crt\_u.compress
8. #define crt\_rng                crt\_u.rng

这样，要访问hash算法的hash成员，就可以直接使用crt\_hash，而不是crt\_u.hash。  
  
每种算法访问tfm都使用了二次封装，例如：

1. struct crypto\_ablkcipher {
2. struct crypto\_tfm base;
3. };
4. struct crypto\_aead {
5. struct crypto\_tfm base;
6. };
7. struct crypto\_blkcipher {
8. struct crypto\_tfm base;
9. };
10. struct crypto\_cipher {
11. struct crypto\_tfm base;
12. };
13. struct crypto\_comp {
14. struct crypto\_tfm base;
15. };
16. struct crypto\_hash {
17. struct crypto\_tfm base;
18. };
19. struct crypto\_rng {
20. struct crypto\_tfm base;
21. };

其base成员就是相应算法的tfm。因为它们拥有相应的起始地址，可以很方便地强制类型转换来操作，内核为此专门定义了一组函数，以hash为例，完成这一工作的是crypto\_hash\_cast：

1. static inline struct crypto\_hash \*\_\_crypto\_hash\_cast(struct crypto\_tfm \*tfm)
2. {
3. return (struct crypto\_hash \*)tfm;
4. }
5. static inline struct crypto\_hash \*crypto\_hash\_cast(struct crypto\_tfm \*tfm)
6. {
7. BUG\_ON((crypto\_tfm\_alg\_type(tfm) ^ CRYPTO\_ALG\_TYPE\_HASH) &
8. CRYPTO\_ALG\_TYPE\_HASH\_MASK);
9. return \_\_crypto\_hash\_cast(tfm);
10. }

当然，针对各种不同的算法，还有许多不同的XXX\_cast函数。这些cast函数，将tfm强制转换为其所属的算法类型的封装结构。  
  
4.2 tfm的分配  
对于算法的实始化，其核心功能就是分配一个tfm，并设置其上下文环境，例如密钥等参数，然后初始化上述struct xxx\_tfm结构。对于hash类的算法来讲，分配tfm是由crypto\_alloc\_hash(crypt.h) 这个API来完成的，以AH为例，在其初始化过程中有：

1. static int ah\_init\_state(struct xfrm\_state \*x)
2. {
3. struct crypto\_hash \*tfm;
4. ……
5. tfm = crypto\_alloc\_hash(x->aalg->alg\_name, 0, CRYPTO\_ALG\_ASYNC);
6. if (IS\_ERR(tfm))
7. goto error;
8. ……
9. }

AH调用crypto\_alloc\_hash为SA中指定的算法（如hmac(md5)）分配一个tfm，第二个参数为0，第三个参数指明了AH使用异步模式。

1. static inline struct crypto\_hash \*crypto\_alloc\_hash(const char \*alg\_name,
2. u32 type, u32 mask)
3. {
4. //初始化相应的类型的掩码
5. type &= ~CRYPTO\_ALG\_TYPE\_MASK;                //清除类型的CRYPTO\_ALG\_TYPE\_MASK位
6. mask &= ~CRYPTO\_ALG\_TYPE\_MASK;                //清除掩码的CRYPTO\_ALG\_TYPE\_MASK位
7. type |= CRYPTO\_ALG\_TYPE\_HASH;                //置类型CRYPTO\_ALG\_TYPE\_HASH位
8. mask |= CRYPTO\_ALG\_TYPE\_HASH\_MASK;        //置掩码CRYPTO\_ALG\_TYPE\_HASH\_MASK位
9. //最终的分配函数是crypto\_alloc\_base，它分配一个base(每个算法的tfm)，再将其强制类型转换为所需要结构类型
10. return \_\_crypto\_hash\_cast(crypto\_alloc\_base(alg\_name, type, mask));
11. }

crypto\_alloc\_base首先检查相应的算法是否存在，对于hmac(md5)这个例子，xfrm在SA的增加中，会触发相应的算法查找，最终会调用hmac模版的alloc分配算法实例（当然也包括算法本身），然后向内核注册算法及算法实例，所以，查找会命中。接下来的工作，是调用tfm的核心分配函数\_\_crypto\_alloc\_tfm进行分配，其实现如下：

1. struct crypto\_tfm \*crypto\_alloc\_base(const char \*alg\_name, u32 type, u32 mask)
2. {
3. struct crypto\_tfm \*tfm;
4. int err;
5. for (;;) {
6. struct crypto\_alg \*alg;
7. //根据算法名称，查找相应的算法,它会首先尝试已经加载的算法，如果失败，也会尝试
8. //动态插入内核模块
9. alg = crypto\_alg\_mod\_lookup(alg\_name, type, mask);
10. //查找失败，返回退出循环
11. if (IS\_ERR(alg)) {
12. err = PTR\_ERR(alg);
13. goto err;
14. }
15. //查找成功，为算法分配tfm
16. tfm = \_\_crypto\_alloc\_tfm(alg, type, mask);
17. //分配成功，返回之
18. if (!IS\_ERR(tfm))
19. return tfm;
20. //释放引用计算，因为查找会增加引用
21. crypto\_mod\_put(alg);
22. //获取返回错误值，根据其值，决定是否要继续尝试
23. err = PTR\_ERR(tfm);
24. err:
25. if (err != -EAGAIN)
26. break;
27. if (signal\_pending(current)) {
28. err = -EINTR;
29. break;
30. }
31. }
32. return ERR\_PTR(err);
33. }

\_\_crypto\_alloc\_tfm是内核加密框架中又一重要的函数，它完成了对算法tfm的分配和初始化的工作：

struct crypto\_tfm \*\_\_crypto\_alloc\_tfm(struct crypto\_alg \*alg, u32 type,

                                      u32 mask)

{

        struct crypto\_tfm \*tfm = NULL;

        unsigned int tfm\_size;

        int err = -ENOMEM;

        //计算tfm所需的空间大小，它包括了tfm结构本身和算法上下文大小

        tfm\_size = sizeof(\*tfm) + crypto\_ctxsize(alg, type, mask);

        //分配tfm

        tfm = kzalloc(tfm\_size, GFP\_KERNEL);

        if (tfm == NULL)

                goto out\_err;

        //\_\_crt\_alg成员指向其所属的算法，对于hmac而言，它就是hmac(xxx)，例如hmac(md5)

        tfm->\_\_crt\_alg = alg;

        //初始化tfm选项

        err = crypto\_init\_ops(tfm, type, mask);

        if (err)

                goto out\_free\_tfm;

        //调用算法的初始化函数，初始化tfm，这有个先决条件是tfm本身没有exit函数的实现

        if (!tfm->exit && alg->cra\_init && (err = alg->cra\_init(tfm)))

                goto cra\_init\_failed;

        goto out;

cra\_init\_failed:

        crypto\_exit\_ops(tfm);

out\_free\_tfm:

        if (err == -EAGAIN)

                crypto\_shoot\_alg(alg);

        kfree(tfm);

out\_err:

        tfm = ERR\_PTR(err);

out:

        return tfm;

}

crypto\_init\_ops负责初始化tfm的选项，对于一个真正的算法（例如md5、dst）和一个伪算法（我说的“伪”，是指由模版动态分配的，如hmac(xxx), authenc(xxx,xxx)），因为并不存在这样的算法，只是内核的一个抽像，故称为"伪"，它们的初始化过程是截然不同的。一个伪算法，它都设置了其所属的类型cra\_type，例如，对于hmac(xxx)而言，它指向了crypto\_hash\_type。这样，初始化时，实质上调用的是其所属类型的init函数：

static int crypto\_init\_ops(struct crypto\_tfm \*tfm, u32 type, u32 mask)

{

        //获取tfm所属算法的所属类型

        const struct crypto\_type \*type\_obj = tfm->\_\_crt\_alg->cra\_type;

        //如果设置了类型，调用类型的init

        if (type\_obj)

                return type\_obj->init(tfm, type, mask);

        //否则，判断算法的类型，调用相应的初始化函数，这些在不同的算法实现中分析

        switch (crypto\_tfm\_alg\_type(tfm)) {

        case CRYPTO\_ALG\_TYPE\_CIPHER:

                return crypto\_init\_cipher\_ops(tfm);

        case CRYPTO\_ALG\_TYPE\_DIGEST:

                if ((mask & CRYPTO\_ALG\_TYPE\_HASH\_MASK) !=

                    CRYPTO\_ALG\_TYPE\_HASH\_MASK)

                        return crypto\_init\_digest\_ops\_async(tfm);

                else

                        return crypto\_init\_digest\_ops(tfm);

        case CRYPTO\_ALG\_TYPE\_COMPRESS:

                return crypto\_init\_compress\_ops(tfm);

        default:

                break;

        }

        BUG();

        return -EINVAL;

}

算法类型的概念很好理解，因为若干个hmac(xxx)都拥有一此相同的类型属性（其它伪算法同样如此），所以可以将它们抽像管理。  
对于hash类型的算法而言，它们拥有一个共同的类型crypto\_hash\_type，其定义在hash.c中：

const struct crypto\_type crypto\_hash\_type = {

        .ctxsize = crypto\_hash\_ctxsize,

        .init = crypto\_init\_hash\_ops,

#ifdef CONFIG\_PROC\_FS

        .show = crypto\_hash\_show,

#endif

};

它的init函数指针指向crypto\_init\_hash\_ops：

static int crypto\_init\_hash\_ops(struct crypto\_tfm \*tfm, u32 type, u32 mask)

{

        struct hash\_alg \*alg = &tfm->\_\_crt\_alg->cra\_hash;

        //其消息摘要大小不同超过1/8个页面

        if (alg->digestsize > PAGE\_SIZE / 8)

                return -EINVAL;

        //根据掩码位，判断是同步初始化还是异步，对于crypto\_alloc\_hash调用下来的而言，它

        //设置了CRYPTO\_ALG\_TYPE\_HASH\_MASK位，所以是同步初始化

        if ((mask & CRYPTO\_ALG\_TYPE\_HASH\_MASK) != CRYPTO\_ALG\_TYPE\_HASH\_MASK)

                return crypto\_init\_hash\_ops\_async (tfm);

        else

                return crypto\_init\_hash\_ops\_sync(tfm);

}

在我们AH的例子中，AH使用了异步模式，所以crypto\_init\_hash\_ops\_async会被调用。  
  
前述hash\_tfm结构封装了hash类型的算法的通用的操作：

struct hash\_tfm {

        int (\*init)(struct hash\_desc \*desc);

        int (\*update)(struct hash\_desc \*desc,

                      struct scatterlist \*sg, unsigned int nsg);

        int (\*final)(struct hash\_desc \*desc, u8 \*out);

        int (\*digest)(struct hash\_desc \*desc, struct scatterlist \*sg,

                      unsigned int nsg, u8 \*out);

        int (\*setkey)(struct crypto\_hash \*tfm, const u8 \*key,

                      unsigned int keylen);

        unsigned int digestsize;

};

先来看同步模式的初始化操作，crypto\_init\_hash\_ops\_sync函数负责初始化这一结构：

static int crypto\_init\_hash\_ops\_sync(struct crypto\_tfm \*tfm)

{

        struct hash\_tfm \*crt = &tfm->crt\_hash;

        struct hash\_alg \*alg = &tfm->\_\_crt\_alg->cra\_hash;

        //置tfm相应操作为算法本身的对应操作，

        //对于hmac(xxx)算法而言，这些东东在hmac\_alloc中已经初始化过了，也就是hmac\_init等函数

        crt->init       = alg->init;

        crt->update     = alg->update;

        crt->final      = alg->final;

        crt->digest     = alg->digest;

        crt->setkey     = hash\_setkey;

        crt->digestsize = alg->digestsize;

        return 0;

}

异步模式则稍有不同，它使用了hash类型算法的通用函数：

static int crypto\_init\_hash\_ops\_async(struct crypto\_tfm \*tfm)

{

        struct ahash\_tfm \*crt = &tfm->crt\_ahash;

        struct hash\_alg  \*alg = &tfm->\_\_crt\_alg->cra\_hash;

        crt->init       = hash\_async\_init;

        crt->update     = hash\_async\_update;

        crt->final      = hash\_async\_final;

        crt->digest     = hash\_async\_digest;

        crt->setkey     = hash\_async\_setkey;

        crt->digestsize = alg->digestsize;

        return 0;

}

不论是同步还是异步，算法的tfm都得到的相应的初始化。回到\_\_crypto\_alloc\_tfm中来，\_\_crypto\_alloc\_tfm函数的最后一步是调用算法的cra\_init函数(如果它存在的话)，对于hmac(xxx)而言，它在分配的时候指向hmac\_init\_tfm。hmac\_init\_tfm的主要工作就是对hmac(xxx)的spawn进行孵化操作。还记得“待孵化的卵”吗？前面讲了只是初始化它，现在到了孵化的时候了。

static int hmac\_init\_tfm(struct crypto\_tfm \*tfm)

{

        struct crypto\_hash \*hash;

        //因为算法实例的第一个成员就是alg，在注册算法时，就是注册的它，所以可以很方便地通过tfm的\_\_crt\_alg强制类型转换得到对应的算法实例

        struct crypto\_instance \*inst = (void \*)tfm->\_\_crt\_alg;

        //取得算法实例的\_\_ctx域，也就是spawn

        struct crypto\_spawn \*spawn = crypto\_instance\_ctx(inst);

        //取得tfm的上下文指针

        struct hmac\_ctx \*ctx = hmac\_ctx(\_\_crypto\_hash\_cast(tfm));

        //对hmac(xxx)进行孵化，以hmac(md5)为例，这将得到一个md5算法的tfm，当然，通过强制类型转换，它被封装在结构crypto\_hash中

        hash = crypto\_spawn\_hash(spawn);

        if (IS\_ERR(hash))

                return PTR\_ERR(hash);

        //设置子算法指向孵化的tfm

        ctx->child = hash;

        return 0;

}

crypto\_spawn\_hash展示了如何对hash算法簇进行spawn的孵化操作：

static inline struct crypto\_hash \*crypto\_spawn\_hash(struct crypto\_spawn \*spawn)

{

        //初始化孵化所需的类型和掩码

        u32 type = CRYPTO\_ALG\_TYPE\_HASH;

        u32 mask = CRYPTO\_ALG\_TYPE\_HASH\_MASK;

        //调用crypto\_spawn\_tfm孵化一个tfm，并强制类型转换

        return \_\_crypto\_hash\_cast(crypto\_spawn\_tfm(spawn, type, mask));

}

最后的任务交给了crypto\_spawn\_tfm函数，它为算法孵化一个tfm，因为spawn的alg成员指向了所要孵化的算法，使得这一操作很容易实现：

struct crypto\_tfm \*crypto\_spawn\_tfm(struct crypto\_spawn \*spawn, u32 type,

                                    u32 mask)

{

        struct crypto\_alg \*alg;

        struct crypto\_alg \*alg2;

        struct crypto\_tfm \*tfm;

        down\_read(&crypto\_alg\_sem);

        //要孵化的spawn所属的算法

        alg = spawn->alg;

        alg2 = alg;

        //查找算法所属模块

        if (alg2)

                alg2 = crypto\_mod\_get(alg2);

        up\_read(&crypto\_alg\_sem);

        //如果其所属模块没了，则标注算法为DYING，出错退回

        if (!alg2) {

                if (alg)

                        crypto\_shoot\_alg(alg);

                return ERR\_PTR(-EAGAIN);

        }

        //初始化tfm

        tfm = ERR\_PTR(-EINVAL);

        //验证掩码标志位

        if (unlikely((alg->cra\_flags ^ type) & mask))

                goto out\_put\_alg;

        //为算法分配相应的tfm，这样，一个算法的spawn就孵化完成了

        tfm = \_\_crypto\_alloc\_tfm(alg, type, mask);

        if (IS\_ERR(tfm))

                goto out\_put\_alg;

        return tfm;

out\_put\_alg:

        crypto\_mod\_put(alg);

        return tfm;

}

又绕回了\_\_crypto\_alloc\_tfm函数，其实现之前已经分析过了，对于一个普通的算法(非模版产生的算法，如md5)，其初始化工作略有不同，在了解其初始化工作之前，需要对一个实际的算法作了解。  
  
顺例说一句，内核的这种抽像管理方式，功能异常地强大，可以想像，它可以抽像更多层的嵌套。所以hmac(xxx)中，xxx不一定就是一个md5之类，可能还是一层形如xxx(xxx)的抽像，理论上，它可以像变形金刚一样。  
  
4.3 小结一下  
本节分析了一个算法的tfm是如何生成的，因为算法可以是多层的组装，在生成上层算法的同时，它也要为其所包含的算法分配tfm，这一过程称之为spawn。

5.        MD5的tfm分配  
5.1 md5结构  
内核加密框架中，所有的hash算法用结构shash\_alg描述，它封装了crypto\_alg和hash函数所需的一些特别的元素，其定义如下：

struct shash\_alg {

        //hash函数的各种操作

        int (\*init)(struct shash\_desc \*desc);

        int (\*reinit)(struct shash\_desc \*desc);

        int (\*update)(struct shash\_desc \*desc, const u8 \*data,

                      unsigned int len);

        int (\*final)(struct shash\_desc \*desc, u8 \*out);

        int (\*finup)(struct shash\_desc \*desc, const u8 \*data,

                     unsigned int len, u8 \*out);

        int (\*digest)(struct shash\_desc \*desc, const u8 \*data,

                      unsigned int len, u8 \*out);

        int (\*setkey)(struct crypto\_shash \*tfm, const u8 \*key,

                      unsigned int keylen);

        //算法所需的私有空间大小

        unsigned int descsize;

        //摘要大小

        unsigned int digestsize;

        //封装的算法结构

        struct crypto\_alg base;

};

md5算法结构定义在md5.c中：

static struct shash\_alg alg = {

        .digestsize        =        MD5\_DIGEST\_SIZE,                //md5算法输出16字节定长的摘要信息

        .init                =        md5\_init,                                        //算法初始化函数

        .update                =        md5\_update,                        //算法核心函数

        .final                =        md5\_final,                        //结束处理

        .descsize        =        sizeof(struct md5\_ctx),                //私有的上下文结构大小

        .base                =        {                                                                //base成员，crypto\_alg结构

                .cra\_name        =        "md5",                                                //名称

                .cra\_flags        =        CRYPTO\_ALG\_TYPE\_SHASH,                //类型标志

                .cra\_blocksize        =        MD5\_HMAC\_BLOCK\_SIZE,        //HMAC块大小

                .cra\_module        =        THIS\_MODULE,                                //所属模块

        }

};

5.2 算法的注册与注销  
//初始化注册之

static int \_\_init md5\_mod\_init(void)

{

        return crypto\_register\_shash(&alg);

}

//注销函数

static void \_\_exit md5\_mod\_fini(void)

{

        crypto\_unregister\_shash(&alg);

}

crypto\_register\_shash实质上还是调用统一接口crypto\_register\_alg注册，在注册之前，针对算法的特性，做了一些检查之类的动作：

int crypto\_register\_shash(struct shash\_alg \*alg)

{

        //内核使用crypto\_alg结构低层抽像描述一个算法，hash算法使用结构struct shash\_alg来进行高层封装，

        //所以后者包含了前者，这是通过其base成员实现的。base成员的部份子成员在初始化时已经赋值。

        struct crypto\_alg \*base = &alg->base;

        //合法性验证，要求digestsize和descsize大小不能超过8分之一的页面

        if (alg->digestsize > PAGE\_SIZE / 8 ||

            alg->descsize > PAGE\_SIZE / 8)

                return -EINVAL;

        //当前算法指向一个上层的名称crypo\_shash\_type，它是所有hash算法的类型的上层统一类型抽像，

        //与hmac(xxx)不同的是，后者是crypto\_hash\_type

        base->cra\_type = &crypto\_shash\_type;

        //置算法的标志位

        base->cra\_flags &= ~CRYPTO\_ALG\_TYPE\_MASK;

        base->cra\_flags |= CRYPTO\_ALG\_TYPE\_SHASH;

        //注册算法

        return crypto\_register\_alg(base);

}

通用的算法注册接口是crypto\_register\_alg，这个函数在算法实例的注册中已经看到过，只是当时并不是展开分析它的一个好时机：

int crypto\_register\_alg(struct crypto\_alg \*alg)

{

        struct crypto\_larval \*larval;

        int err;

        //合法性检查，如果合法，构造算法的驱动完整名称，它以“-generic”为后缀

        err = crypto\_check\_alg(alg);

        if (err)

                return err;

        down\_write(&crypto\_alg\_sem);

        larval = \_\_crypto\_register\_alg(alg);

        up\_write(&crypto\_alg\_sem);

        if (IS\_ERR(larval))

                return PTR\_ERR(larval);

        crypto\_wait\_for\_test(larval);

        return 0;

}

所有的算法都注册到一个全局算法链表crypto\_alg\_list，crypto\_alg有一个链表成员cra\_list，用于注册。注册的实质，就是一个链表插入操作。  
  
特别地，对于每一个算法，都有一个与之对应的larval(幼虫？？？)，larval本身也是一个算法，只是其标志位标注为CRYPTO\_ALG\_LARVAL，它同样在注册算法的同时，也被注册到crypto\_alg\_list。每个larval用结构crypto\_larval描述：

struct crypto\_larval {

        struct crypto\_alg alg;                        //算法结构

        struct crypto\_alg \*adult;                        //成年人^o^，指向其所属的算法

        struct completion completion;        //???

        u32 mask;                                        //掩码位

};

crypto\_larval\_alloc函数完成算法的larval的分配：

struct crypto\_larval \*crypto\_larval\_alloc(const char \*name, u32 type, u32 mask)

{

        struct crypto\_larval \*larval;

        //分配larval

        larval = kzalloc(sizeof(\*larval), GFP\_KERNEL);

        if (!larval)

                return ERR\_PTR(-ENOMEM);

        //置掩码位与对应的adult算法一致

        larval->mask = mask;

        //算法标志位置CRYPTO\_ALG\_LARVAL

        larval->alg.cra\_flags = CRYPTO\_ALG\_LARVAL | type;

        //算法优先级衡为1

        larval->alg.cra\_priority = -1;

        //置destroy函数

        larval->alg.cra\_destroy = crypto\_larval\_destroy;

        //拷贝算法名称

        strlcpy(larval->alg.cra\_name, name, CRYPTO\_MAX\_ALG\_NAME);

        //初始化???

        init\_completion(&larval->completion);

        return larval;

}

\_\_crypto\_register\_alg函数完成算法及其larval的注册：

static struct crypto\_larval \*\_\_crypto\_register\_alg(struct crypto\_alg \*alg)

{

        struct crypto\_alg \*q;

        struct crypto\_larval \*larval;

        int ret = -EAGAIN;

        //判断算法是否处于DEAD状态，CRYPTO\_ALG\_DEAD

        if (crypto\_is\_dead(alg))

                goto err;

        //初始化链表，准备注册

        INIT\_LIST\_HEAD(&alg->cra\_users);

        /\* No cheating! \*/

        //清除CRYPTO\_ALG\_TESTED标志位

        alg->cra\_flags &= ~CRYPTO\_ALG\_TESTED;

        //初始化返回值：“已经存在”

        ret = -EEXIST;

        //置引用计算器

        atomic\_set(&alg->cra\_refcnt, 1);

        //遍历算法链表，查找算法是否存在

        list\_for\_each\_entry(q, &crypto\_alg\_list, cra\_list) {

                //已经存在

                if (q == alg)

                        goto err;

                //如果遍历到的算法处于垂死状态（CRYPTO\_ALG\_DEAD | CRYPTO\_ALG\_DYING），忽略之

                if (crypto\_is\_moribund(q))

                        continue;

                //如果遍历到的算法是否是一个larval，若是，且刚好与待注册算法名称匹备，出错返回，否则忽略之

                if (crypto\_is\_larval(q)) {

                        if (!strcmp(alg->cra\_driver\_name, q->cra\_driver\_name))

                                goto err;

                        continue;

                }

                //判断是否已经注册

                if (!strcmp(q->cra\_driver\_name, alg->cra\_name) ||

                    !strcmp(q->cra\_name, alg->cra\_driver\_name))

                        goto err;

        }

        //分配larval

        larval = crypto\_larval\_alloc(alg->cra\_name,

                                     alg->cra\_flags | CRYPTO\_ALG\_TESTED, 0);

        if (IS\_ERR(larval))

                goto out;

        ret = -ENOENT;

        //adult指针指向所属算法

        larval->adult = crypto\_mod\_get(alg);

        if (!larval->adult)

                goto free\_larval;

        //初始化引用计数器

        atomic\_set(&larval->alg.cra\_refcnt, 1);

        //拷贝驱动名称

        memcpy(larval->alg.cra\_driver\_name, alg->cra\_driver\_name,

               CRYPTO\_MAX\_ALG\_NAME);

        //置优先级

        larval->alg.cra\_priority = alg->cra\_priority;

        //注册算法与算法的larval算法

        list\_add(&alg->cra\_list, &crypto\_alg\_list);

        list\_add(&larval->alg.cra\_list, &crypto\_alg\_list);

out:

        return larval;

free\_larval:

        kfree(larval);

err:

        larval = ERR\_PTR(ret);

        goto out;

}

有注册就有注销，crypto\_unregister\_shash函数完成相应的注销操作，类似地，它是crypto\_unregister\_alg的包裹：

int crypto\_unregister\_shash(struct shash\_alg \*alg)

{

        return crypto\_unregister\_alg(&alg->base);

}

int crypto\_unregister\_alg(struct crypto\_alg \*alg)

{

        int ret;

        LIST\_HEAD(list);

        //crypto\_remove\_alg将算法从全局链中删除，并清空算法所有的spawn

        down\_write(&crypto\_alg\_sem);

        ret = crypto\_remove\_alg(alg, &list);

        up\_write(&crypto\_alg\_sem);

        //都没有看到返回大于0的地方

        if (ret)

                return ret;

        //验证引用计数器

        BUG\_ON(atomic\_read(&alg->cra\_refcnt) != 1);

        //如果有destroy，调用之

        if (alg->cra\_destroy)

                alg->cra\_destroy(alg);

        crypto\_remove\_final(&list);

        return 0;

}

5.3 分配tfm  
在hmac(md5)中调用crypto\_spawn\_tfm进行算法的孵化，这将调用\_\_crypto\_alloc\_tfm为真实hash算法分配tfm。因为\_\_crypto\_alloc\_tfm函数已经分析过了，对于md5算法而言，重点需要关注的就是其选项初始化和cra\_init函数。  
  
在注册md5算法（或其它hash算法时），设置了算法相应的类型：

base->cra\_type = &crypto\_shash\_type;

crypto\_shash\_type定义在crypto/shash.c中，其定义如下：

static const struct crypto\_type crypto\_shash\_type = {

        .ctxsize = crypto\_shash\_ctxsize,

        .extsize = crypto\_shash\_extsize,

        .init = crypto\_init\_shash\_ops,

        .init\_tfm = crypto\_shash\_init\_tfm,

#ifdef CONFIG\_PROC\_FS

        .show = crypto\_shash\_show,

#endif

        .maskclear = ~CRYPTO\_ALG\_TYPE\_MASK,

        .maskset = CRYPTO\_ALG\_TYPE\_MASK,

        .type = CRYPTO\_ALG\_TYPE\_SHASH,

        .tfmsize = offsetof(struct crypto\_shash, base),

};

这里关注的是它的init函数，同hmac类似，选项初始化将触发crypto\_shash\_type的crypto\_init\_shash\_ops函数：

static int crypto\_init\_shash\_ops(struct crypto\_tfm \*tfm, u32 type, u32 mask)

{

        switch (mask & CRYPTO\_ALG\_TYPE\_MASK) {

        case CRYPTO\_ALG\_TYPE\_HASH\_MASK:

                return crypto\_init\_shash\_ops\_compat(tfm);

        case CRYPTO\_ALG\_TYPE\_AHASH\_MASK:

                return crypto\_init\_shash\_ops\_async(tfm);

        }

        return -EINVAL;

}

crypto\_init\_shash\_ops函数根据掩码位判断是否使用异步初始化，在上层算法孵化过程中，crypto\_spawn\_hash函数设置了相应的类型和掩码标志：

        u32 type = CRYPTO\_ALG\_TYPE\_HASH;

        u32 mask = CRYPTO\_ALG\_TYPE\_HASH\_MASK;

所以，如果是由crypto\_spawn\_hash触发的spawn动作，这crypto\_init\_shash\_ops\_compat  
函数将被调用:

static int crypto\_init\_shash\_ops\_compat(struct crypto\_tfm \*tfm)

{

        struct hash\_tfm \*crt = &tfm->crt\_hash;    //事实上是crt\_u.hash

        struct crypto\_alg \*calg = tfm->\_\_crt\_alg;  //tfm对应的算法,这里也就是md5算法对应的crypto\_alg结构

        struct shash\_alg \*alg = \_\_crypto\_shash\_alg(calg);    //上层封装的hash算法,hash算法结构shash\_alg中通过base字段封装了crypto\_alg

        struct shash\_desc \*desc = crypto\_tfm\_ctx(tfm);   //取得tfm上下文成员，即\_\_crt\_ctx

        struct crypto\_shash \*shash;                                                        //crypto\_hash是hash算法的tfm上层封装

        //尝试插入算法模块，如果是以模块编译存在的话

        if (!crypto\_mod\_get(calg))

                return -EAGAIN;

        //再为算法分配一个tfm，用于设置tfm的\_\_crt\_ctx成员

        shash = crypto\_create\_tfm(calg, &crypto\_shash\_type);

        if (IS\_ERR(shash)) {

                crypto\_mod\_put(calg);

                return PTR\_ERR(shash);

        }

        //desc实质上指向的是tfm的上下文成员，它的空间在分配tfm时候已经分配，所以这里可以直接地使用。

        //\_\_crt\_ctx是一个void \*类型，这里被强制转换为名为desc的struct shash\_desc，其第一个成员又是tfm(结构crypto\_shash)

        //这里指向分配到的shash

        desc->tfm = shash;

        //注册tfm的exit动作

        tfm->exit = crypto\_exit\_shash\_ops\_compat;

        //设置tfm的hash操作的各种动作

        crt->init = shash\_compat\_init;

        crt->update = shash\_compat\_update;

        crt->final  = shash\_compat\_final;

        crt->digest = shash\_compat\_digest;

        crt->setkey = shash\_compat\_setkey;

        crt->digestsize = alg->digestsize;

        return 0;

}

与hmac(xxx)初始化选项相比有两个重要的不同：  
1、tfm对应的算法已经没有spawn，它调用crypto\_create\_tfm为tfm再次分配了一个tfm，可以通过其上下文成员，强制类型转换为  
shash\_desc结构后来访问；  
2、tfm登记注册了exit动作，指向crypto\_exit\_shash\_ops\_compat；  
  
简单地来看一下crypto\_create\_tfm函数，这个函数创建一个tfm，名为create，而非alloc，它有两个参数，一个是所属的算法，一个是所属的类型(称为前端-frontend)：

void \*crypto\_create\_tfm(struct crypto\_alg \*alg,

                        const struct crypto\_type \*frontend)

{

        char \*mem;

        struct crypto\_tfm \*tfm = NULL;

        unsigned int tfmsize;

        unsigned int total;

        int err = -ENOMEM;

        //计算需要的总大小:前端类型的大小，加自身结构的大小，加一个扩展空间,扩展大小根据具体算法而定

        tfmsize = frontend->tfmsize;

        total = tfmsize + sizeof(\*tfm) + frontend->extsize(alg, frontend);

        //分配之

        mem = kzalloc(total, GFP\_KERNEL);

        if (mem == NULL)

                goto out\_err;

        //跳过前端大小

        tfm = (struct crypto\_tfm \*)(mem + tfmsize);

        //关联实际所属的算法

        tfm->\_\_crt\_alg = alg;

        //调用前端类型的实始化tfm函数,对于crypto\_shash\_type来讲，它是crypto\_shash\_init\_tfm

        err = frontend->init\_tfm(tfm, frontend);

        if (err)

                goto out\_free\_tfm;

        //初始化tfm，如果算法有初始化函数的话，则调用初始化函数初始tfm。

        if (!tfm->exit && alg->cra\_init && (err = alg->cra\_init(tfm)))

                goto cra\_init\_failed;

        goto out;

cra\_init\_failed:

        crypto\_exit\_ops(tfm);

out\_free\_tfm:

        if (err == -EAGAIN)

                crypto\_shoot\_alg(alg);

        kfree(mem);

out\_err:

        mem = ERR\_PTR(err);

out:

        return mem;

}

对于新的tfm来讲，它的初始化选项函数是crypto\_shash\_init\_tfm函数：

static int crypto\_shash\_init\_tfm(struct crypto\_tfm \*tfm,

                                 const struct crypto\_type \*frontend)

{

        return 0;

}

shash并没有实现这一函数。  
  
另外，算法的cra\_init函数将被调用，它会对新的tfm进行初始化操作。在md5初始化的时候，没有设置其base成员的cra\_init。  
  
回到最上层的tfm的分配过程中，在初始化选项完成后，因为它的exit已经登记注册为crypto\_exit\_shash\_ops\_compat，所以它的cra\_init初始化操作也不会被触发。  
  
6.        密钥设置  
经过以上步骤，算法hmac(md5)的tfm已经分配完成了，下一步就是为它设置密钥及其它参数，为数据到来做好准备。以ah协议为例：

        tfm = crypto\_alloc\_hash(x->aalg->alg\_name, 0, CRYPTO\_ALG\_ASYNC);

        if (IS\_ERR(tfm))

                goto error;

        ahp->tfm = tfm;

        if (crypto\_hash\_setkey(tfm, x->aalg->alg\_key,

                               (x->aalg->alg\_key\_len + 7) / 8))

                goto error;

对于hash算法簇，crypto\_hash\_setkey在最上层封装了密钥设置操作：

static inline int crypto\_hash\_setkey(struct crypto\_hash \*hash,

                                     const u8 \*key, unsigned int keylen)

{

        return crypto\_hash\_crt(hash)->setkey(hash, key, keylen);

}

上层调用者调用crypto\_hash\_setkey 进行密钥设置，它包含了三个参数：  
  
        算法的tfmλ  
        密钥λ  
        密钥长度λ  
  
这将会最终调用hmac(md5)算法的tfm的crt\_hash(它是hash\_tfm结构，封装了算法的具体操作)setkey函数，  
对于异步模式而言，setkey函数指针在crypto\_init\_hash\_ops\_async中被指向了hash\_async\_setkey函数：

static int hash\_async\_setkey(struct crypto\_ahash \*tfm\_async, const u8 \*key,

                        unsigned int keylen)

{

        //取得算法的tfm

struct crypto\_tfm  \*tfm      = crypto\_ahash\_tfm(tfm\_async);

//强制类型转换为hash算法簇的tfm

        struct crypto\_hash \*tfm\_hash = \_\_crypto\_hash\_cast(tfm);

        //取得对应的算法

        struct hash\_alg    \*alg      = &tfm->\_\_crt\_alg->cra\_hash;

        //调用算法的setkey

        return alg->setkey(tfm\_hash, key, keylen);

}

对于hmac(md5)而言，算法的setkey函数指针是在模版分配算法实例时初始化的：  
inst->alg.cra\_hash.setkey = hmac\_setkey;  
  
hmac\_setkey函数完成算法的密钥设置工作：

static int hmac\_setkey(struct crypto\_hash \*parent,

                       const u8 \*inkey, unsigned int keylen)

{

        int bs = crypto\_hash\_blocksize(parent);                        //取得bs

        int ds = crypto\_hash\_digestsize(parent);                //取得ds

        char \*ipad = crypto\_hash\_ctx\_aligned(parent);        //取得ctx空间，ipad指向这段区域

        char \*opad = ipad + bs;                                                        //opad跳过bs

        char \*digest = opad + bs;                                                //digest再跳过一个bs

        struct hmac\_ctx \*ctx = align\_ptr(digest + ds, sizeof(void \*));

        struct crypto\_hash \*tfm = ctx->child;

        unsigned int i;

        //hmac算法的密钥长度不能超过bs，并且，一般建议其长度不小于ds。这里处理keylen大于bs的情况，需要

        //重新计算一个密钥和密钥长度，计算的过程就是对密钥求hash签名，把签名值做为密钥。

        if (keylen > bs) {

                struct hash\_desc desc;

                struct scatterlist tmp;

                int tmplen;

                int err;

                desc.tfm = tfm;

                desc.flags = crypto\_hash\_get\_flags(parent);

                desc.flags &= CRYPTO\_TFM\_REQ\_MAY\_SLEEP;

                err = crypto\_hash\_init(&desc);

                if (err)

                        return err;

                tmplen = bs \* 2 + ds;

                sg\_init\_one(&tmp, ipad, tmplen);

                for (; keylen > tmplen; inkey += tmplen, keylen -= tmplen) {

                        memcpy(ipad, inkey, tmplen);

                        err = crypto\_hash\_update(&desc, &tmp, tmplen);

                        if (err)

                                return err;

                }

                if (keylen) {

                        memcpy(ipad, inkey, keylen);

                        err = crypto\_hash\_update(&desc, &tmp, keylen);

                        if (err)

                                return err;

                }

                err = crypto\_hash\_final(&desc, digest);

                if (err)

                        return err;

                inkey = digest;

                keylen = ds;

        }

        //构造算法密钥，先拷贝输入的密钥

        memcpy(ipad, inkey, keylen);

        //然后补齐bs -keylen，在尾部填充0

        memset(ipad + keylen, 0, bs - keylen);

        //拷贝ipad至opad

        memcpy(opad, ipad, bs);

        //hmac算法本身需要两个参数ipad和opad，分别表示0x36和0x5c重复bs次，

        //然后它将密钥补齐0后，与ipad和opad求异或，这里作者实现是直接将密钥拷贝

        //到ipad/opad后，直接求异或，减少了一个变量的使用。

        for (i = 0; i < bs; i++) {

                ipad[i] ^= 0x36;

                opad[i] ^= 0x5c;

        }

        return 0;

}

7.        数据验证  
7.1 验证的通用步骤  
通过之前的一系列操作，算法已经做好准备，等待数据的到来。在做好了准备工作后，就可以开始数据验证了。上层调用者主要调用以下三个API进行数据的验证处理：  
        crypto\_hash\_initλ  
        crypto\_hash\_updateλ  
        crypto\_hash\_finalλ  
以AH协议为例，它调用ah\_mac\_digest封装了这三个函数：

static inline int ah\_mac\_digest(struct ah\_data \*ahp, struct sk\_buff \*skb,

                                u8 \*auth\_data)

{

        struct hash\_desc desc;

        int err;

        desc.tfm = ahp->tfm;

        desc.flags = 0;

        memset(auth\_data, 0, ahp->icv\_trunc\_len);

        err = crypto\_hash\_init(&desc);

        if (unlikely(err))

                goto out;

        err = skb\_icv\_walk(skb, &desc, 0, skb->len, crypto\_hash\_update);

        if (unlikely(err))

                goto out;

        err = crypto\_hash\_final(&desc, ahp->work\_icv);

out:

        return err;

}

关于AH的更多细节，参九贱《Linux IPSec设计与实现》。  
  
加密框架使用结构struct XXX\_desc封装了算法使用时的一些描述信息，XXX表示算法类型。主要包括了算法tfm和一些标志信息等，例如：

struct hash\_desc {

        struct crypto\_hash \*tfm;

        u32 flags;

};

与此类似的还有：

struct blkcipher\_desc {

        struct crypto\_blkcipher \*tfm;

        void \*info;

        u32 flags;

};

struct cipher\_desc {

        struct crypto\_tfm \*tfm;

        void (\*crfn)(struct crypto\_tfm \*tfm, u8 \*dst, const u8 \*src);

        unsigned int (\*prfn)(const struct cipher\_desc \*desc, u8 \*dst,

                             const u8 \*src, unsigned int nbytes);

        void \*info;

};

上述算法的三个调用函数中，都需要这个结构做为参数。  
  
7.2 分散/聚合支持  
skb\_icv\_walk完成对skb的分散/聚合的支持，这与skb的特性有关，如果不需要了解这些细节，可以跳过本节。  
针对skb的特性，验证过程支持了skb 的分散/聚合，内核使用scatterlist描述一个分散/聚合的数据结构，它针对硬件体系的不同而不同，以x86为例：

struct scatterlist {

#ifdef CONFIG\_DEBUG\_SG

        unsigned long        sg\_magic;

#endif

        unsigned long        page\_link;

        unsigned int        offset;

        unsigned int        length;

        dma\_addr\_t        dma\_address;

        unsigned int        dma\_length;

};

针对分散/聚合结构的操作，一个重要的函数是sg\_init\_one，它初始化一个sg，然后将它指向一个指定的地址：

/\*\*

\* sg\_init\_one - Initialize a single entry sg list

\* @sg:                 SG entry

\* @buf:         Virtual address for IO

\* @buflen:         IO length

\*

\*\*/

void sg\_init\_one(struct scatterlist \*sg, const void \*buf, unsigned int buflen)

{

        sg\_init\_table(sg, 1);

        sg\_set\_buf(sg, buf, buflen);

}

sg\_init\_table初始化sg表：

/\*\*

\* sg\_init\_table - Initialize SG table

\* @sgl:           The SG table

\* @nents:           Number of entries in table

\*

\* Notes:

\*   If this is part of a chained sg table, sg\_mark\_end() should be

\*   used only on the last table part.

\*

\*\*/

void sg\_init\_table(struct scatterlist \*sgl, unsigned int nents)

{

        //为什么叫sg表，从memset来看，若干个sg是连续的，所以也可以很容易地编写sg\_init\_two,sg\_init\_three

        memset(sgl, 0, sizeof(\*sgl) \* nents);

#ifdef CONFIG\_DEBUG\_SG

        //若支持debug，则置sg\_magic

        {

                unsigned int i;

                for (i = 0; i < nents; i++)

                        sgl[i].sg\_magic = SG\_MAGIC;

        }

#endif

        sg\_mark\_end(&sgl[nents - 1]);

}

sg\_mark\_end标志sg表的结束，它将表的最后一个sg传递过来，设置其page\_link成员。

/\*\*

\* sg\_mark\_end - Mark the end of the scatterlist

\* @sg:                 SG entryScatterlist

\*

\* Description:

\*   Marks the passed in sg entry as the termination point for the sg

\*   table. A call to sg\_next() on this entry will return NULL.

\*

\*\*/

static inline void sg\_mark\_end(struct scatterlist \*sg)

{

#ifdef CONFIG\_DEBUG\_SG

        BUG\_ON(sg->sg\_magic != SG\_MAGIC);

#endif

        /\*

         \* Set termination bit, clear potential chain bit

         \*/

        sg->page\_link |= 0x02;

        sg->page\_link &= ~0x01;

}

sg\_set\_buf将sg指针指向给定的内存缓冲区：

/\*\*

\* sg\_set\_buf - Set sg entry to point at given data

\* @sg:                 SG entry

\* @buf:         Data

\* @buflen:         Data length

\*

\*\*/

static inline void sg\_set\_buf(struct scatterlist \*sg, const void \*buf,

                              unsigned int buflen)

{

        sg\_set\_page(sg, virt\_to\_page(buf), buflen, offset\_in\_page(buf));

}

sg\_set\_page主要就是设置page\_link标志和offset及length成员：

/\*\*

\* sg\_set\_page - Set sg entry to point at given page

\* @sg:                 SG entry

\* @page:         The page

\* @len:         Length of data

\* @offset:         Offset into page

\*

\* Description:

\*   Use this function to set an sg entry pointing at a page, never assign

\*   the page directly. We encode sg table information in the lower bits

\*   of the page pointer. See sg\_page() for looking up the page belonging

\*   to an sg entry.

\*

\*\*/

static inline void sg\_set\_page(struct scatterlist \*sg, struct page \*page,

                               unsigned int len, unsigned int offset)

{

        sg\_assign\_page(sg, page);

        sg->offset = offset;

        sg->length = len;

}

/\*\*

\* sg\_assign\_page - Assign a given page to an SG entry

\* @sg:                    SG entry

\* @page:            The page

\*

\* Description:

\*   Assign page to sg entry. Also see sg\_set\_page(), the most commonly used

\*   variant.

\*

\*\*/

static inline void sg\_assign\_page(struct scatterlist \*sg, struct page \*page)

{

        unsigned long page\_link = sg->page\_link & 0x3;

        /\*

         \* In order for the low bit stealing approach to work, pages

         \* must be aligned at a 32-bit boundary as a minimum.

         \*/

        BUG\_ON((unsigned long) page & 0x03);

#ifdef CONFIG\_DEBUG\_SG

        BUG\_ON(sg->sg\_magic != SG\_MAGIC);

        BUG\_ON(sg\_is\_chain(sg));

#endif

        sg->page\_link = page\_link | (unsigned long) page;

}

7.3 init  
crypto\_hash\_init完成数据验证的初始化工作：

static inline int crypto\_hash\_init(struct hash\_desc \*desc)

{

        return crypto\_hash\_crt(desc->tfm)->init(desc);

}

static inline struct hash\_tfm \*crypto\_hash\_crt(struct crypto\_hash \*tfm)

{

        return &crypto\_hash\_tfm(tfm)->crt\_hash;

}

所以算法的tfm的crt\_hash的init函数将被调用，对于hmac(md5)来说，在初始化tfm的时候将它指向了算法的init函数，也是，hmac\_alloc中设置的hmac\_init：  
inst->alg.cra\_hash.init = hmac\_init;

static int hmac\_init(struct hash\_desc \*pdesc)

{

        struct crypto\_hash \*parent = pdesc->tfm;

        int bs = crypto\_hash\_blocksize(parent);

        int ds = crypto\_hash\_digestsize(parent);

        char \*ipad = crypto\_hash\_ctx\_aligned(parent);

        struct hmac\_ctx \*ctx = align\_ptr(ipad + bs \* 2 + ds, sizeof(void \*));

        struct hash\_desc desc;

        struct scatterlist tmp;

        int err;

        //取得子算法的tfm，这个child指针是在hmac\_init\_tfm中初始化的

        desc.tfm = ctx->child;

        //设置标志位

        desc.flags = pdesc->flags & CRYPTO\_TFM\_REQ\_MAY\_SLEEP;

        //使用ipad初始化一个临时的sg

        sg\_init\_one(&tmp, ipad, bs);

        //初始化子算法的tfm

        err = crypto\_hash\_init(&desc);

        if (unlikely(err))

                return err;

        //hmac算法要求将数据流接在经过异或运算后的bs长的字符串之后(ipad)，然后一起使用hash算法，求hash值。参<IPSEC vpn的安全实施> page.113 算法第3/4步,

        //作者这里是分开求的，先在初始化init中对ipad求hash，然后在update中对数据流求hash

        return crypto\_hash\_update(&desc, &tmp, bs);

}

crypto\_hash\_init函数根据算法的tfm进行初始化操作：

static inline int crypto\_hash\_init(struct hash\_desc \*desc)

{

        return crypto\_hash\_crt(desc->tfm)->init(desc);

}

以md5算法为例，它将调用md5的tfm的init函数。如前所述，在crypto\_init\_shash\_ops\_compat函数，它被指向了shash\_compat\_init函数：

static int shash\_compat\_init(struct hash\_desc \*hdesc)

{

        struct shash\_desc \*desc = crypto\_hash\_ctx(hdesc->tfm);

        desc->flags = hdesc->flags;

        return crypto\_shash\_init(desc);

}

static inline int crypto\_shash\_init(struct shash\_desc \*desc)

{

        return crypto\_shash\_alg(desc->tfm)->init(desc);

}

static inline struct shash\_alg \*crypto\_shash\_alg(struct crypto\_shash \*tfm)

{

        return \_\_crypto\_shash\_alg(crypto\_shash\_tfm(tfm)->\_\_crt\_alg);

}

所以，最终tfm所属算法的init函数会被调用，以md5为例，最终调用的将是md5\_init，  
它完成算法所需的上下文的一些信息的初始化，参下一节<md5算法实现>。  
  
7.4 update  
update是hash的一个核心处理过程，它的工作就是对一段数据求hash签名：

static inline int crypto\_hash\_update(struct hash\_desc \*desc,

                                     struct scatterlist \*sg,

                                     unsigned int nbytes)

{

        return crypto\_hash\_crt(desc->tfm)->update(desc, sg, nbytes);

}

在异步模式下，tfm的相应操作指向了hash\_async\_update  
crt->update     = hash\_async\_update;

static int hash\_async\_update(struct ahash\_request \*req)

{

        struct crypto\_tfm \*tfm = req->base.tfm;

        struct hash\_alg   \*alg = &tfm->\_\_crt\_alg->cra\_hash;

        struct hash\_desc  desc = {

                .tfm = \_\_crypto\_hash\_cast(tfm),

                .flags = req->base.flags,

        };

        return alg->update(&desc, req->src, req->nbytes);

}

这样算法的update将被调用。在hmac\_alloc中，算法的update被设置为hmac\_update  
inst->alg.cra\_hash.update = hmac\_update;  
  
hmac\_update函数直接调用子算法的tfm的update:

static int hmac\_update(struct hash\_desc \*pdesc,

                       struct scatterlist \*sg, unsigned int nbytes)

{

        struct hmac\_ctx \*ctx = hmac\_ctx(pdesc->tfm);

        struct hash\_desc desc;

        desc.tfm = ctx->child;

        desc.flags = pdesc->flags & CRYPTO\_TFM\_REQ\_MAY\_SLEEP;

        return crypto\_hash\_update(&desc, sg, nbytes);

}

同crypto\_hash\_init类似，子算法的tfm的update指向了shash\_compat\_update函数：

static int shash\_compat\_update(struct hash\_desc \*hdesc, struct scatterlist \*sg,

                               unsigned int len)

{

        struct shash\_desc \*desc = crypto\_hash\_ctx(hdesc->tfm);

        struct crypto\_hash\_walk walk;

        int nbytes;

        for (nbytes = crypto\_hash\_walk\_first\_compat(hdesc, &walk, sg, len);

             nbytes > 0; nbytes = crypto\_hash\_walk\_done(&walk, nbytes))

                nbytes = crypto\_shash\_update(desc, walk.data, nbytes);

        return nbytes;

}

int crypto\_shash\_update(struct shash\_desc \*desc, const u8 \*data,

                        unsigned int len)

{

        struct crypto\_shash \*tfm = desc->tfm;

        struct shash\_alg \*shash = crypto\_shash\_alg(tfm);

        unsigned long alignmask = crypto\_shash\_alignmask(tfm);

        if ((unsigned long)data & alignmask)

                return shash\_update\_unaligned(desc, data, len);

        return shash->update(desc, data, len);

}

这样，最终算法的update函数会被调用，对md5而言，就是md5\_update。  
  
但是前述ah\_mac\_digest函数并没有直接调用crypto\_hash\_update，为了支持分散/聚合，其使用了skb\_icv\_walk:  
  
err = skb\_icv\_walk(skb, &desc, 0, skb->len, crypto\_hash\_update);  
  
@skb:要进行hash签名的报文;  
@desc:算法描述结构；  
@offset:要验证的数据的偏移；  
@len:报文长度；  
@icv\_update:数据验证的update函数指针，上层调用传递过来的是crypto\_hash\_update  
          
通过上述分析，已经知道crypto\_hash\_update最终调用的是md5\_update。这里重点分析skb\_icv\_walk对分散/聚合的支持。  
关于分散/聚合在skb的操作，可以参考<深入Linux网络内幕>page 489。

/\* Move to common area: it is shared with AH. \*/

int skb\_icv\_walk(const struct sk\_buff \*skb, struct hash\_desc \*desc,

                 int offset, int len, icv\_update\_fn\_t icv\_update)

{

        //分段缓冲区中，主要缓冲区的数据量，不包含各个frags和frag\_list

        int start = skb\_headlen(skb);

        //copy为需要处理的主缓冲区的数据总长度，上层调用者可以传递一个偏移值，以忽略一部份数据

        int i, copy = start - offset;

        struct sk\_buff \*frag\_iter;

        struct scatterlist sg;

        int err;

        /\* Checksum header. \*/

        //验证主缓冲区，如果有数据的话

        if (copy > 0) {

                //校正copy值，它不应该大于要验证的总长度

                if (copy > len)

                        copy = len;

                //初始化一个sg，它指向要处理的数据跳过偏移的区域

                sg\_init\_one(&sg, skb->data + offset, copy);

                //对数据进行hash签名计算

                err = icv\_update(desc, &sg, copy);

                if (unlikely(err))

                        return err;

                //要处理的总长度减去已处理的长度，如果其为0，则表示数据已经处理完了，不需要处理其它数据片段

                if ((len -= copy) == 0)

                        return 0;

                //否则，计算偏移值，为处理各个数据片段做准备

                offset += copy;

        }

        //遍历各个数据片段

        for (i = 0; i < skb\_shinfo(skb)->nr\_frags; i++) {

                int end;

                WARN\_ON(start > offset + len);

                //end不停地累计主缓冲区和遍历过到的各个数据片段的大小的总和‘

                end = start + skb\_shinfo(skb)->frags[i].size;

                //计算当前要处理的数据的长度，即总大小减去偏移值（偏移值也会不停地递增）

                if ((copy = end - offset) > 0) {

                        //取得当前数据片段页面指针

                        skb\_frag\_t \*frag = &skb\_shinfo(skb)->frags[i];

                        //校正copy值

                        if (copy > len)

                                copy = len;

                        //用当前片段初始化一个sg

                        sg\_init\_table(&sg, 1);

                        sg\_set\_page(&sg, frag->page, copy,

                                    frag->page\_offset + offset-start);

                        //对当前片段进行hash签名计算

                        err = icv\_update(desc, &sg, copy);

                        if (unlikely(err))

                                return err;

                        //同样，计算剩余长度后判断是否终止处理

                        if (!(len -= copy))

                                return 0;

                        //重新校正偏移值

                        offset += copy;

                }

                //重新计算start值，配置之前的end = start + ……，所以end是一个总和值

                start = end;

        }

        //处理frag\_list，遍历frag\_list中的每个成员，处理流程同上

        skb\_walk\_frags(skb, frag\_iter) {

                int end;

                WARN\_ON(start > offset + len);

                end = start + frag\_iter->len;

                if ((copy = end - offset) > 0) {

                        if (copy > len)

                                copy = len;

                        err = skb\_icv\_walk(frag\_iter, desc, offset-start,

                                           copy, icv\_update);

                        if (unlikely(err))

                                return err;

                        if ((len -= copy) == 0)

                                return 0;

                        offset += copy;

                }

                start = end;

        }

        BUG\_ON(len);

        return 0;

}

7.5 final  
hash签名计算的最后一个步骤就是调用crypto\_hash\_final，它主要完成一些收尾的工作处理，并将计算好的签名值拷贝出来。

static inline int crypto\_hash\_final(struct hash\_desc \*desc, u8 \*out)

{

        return crypto\_hash\_crt(desc->tfm)->final(desc, out);

}

同样地，hmac(md5)算法的tfm是指向算法本身的final函数，也就是hmac\_final：

static int hmac\_final(struct hash\_desc \*pdesc, u8 \*out)

{

        struct crypto\_hash \*parent = pdesc->tfm;

        int bs = crypto\_hash\_blocksize(parent);

        int ds = crypto\_hash\_digestsize(parent);

        char \*opad = crypto\_hash\_ctx\_aligned(parent) + bs;

        char \*digest = opad + bs;

        struct hmac\_ctx \*ctx = align\_ptr(digest + ds, sizeof(void \*));

        struct hash\_desc desc;

        struct scatterlist tmp;

        int err;

        desc.tfm = ctx->child;

        desc.flags = pdesc->flags & CRYPTO\_TFM\_REQ\_MAY\_SLEEP;

        //使用opad初始化一个sg

        sg\_init\_one(&tmp, opad, bs + ds);

        //这次因为desc是子算法的tfm，所以md5的final函数将会被调用，

        //它处理一些收尾工作，并将计算好的hash值拷贝出来放到digest中

        err = crypto\_hash\_final(&desc, digest);

        if (unlikely(err))

                return err;

        //hmac算法要求，将经过之前计算好的消息摘要连接在经过异或的字符串(opad)之后，

        //放在opad之后，是通过digest = opad + bs来实现的。然后调用hash算法再求消息摘要值。

        //计算出最终要求的值。

        return crypto\_hash\_digest(&desc, &tmp, bs + ds, out);

}

static inline int crypto\_hash\_digest(struct hash\_desc \*desc,

                                     struct scatterlist \*sg,

                                     unsigned int nbytes, u8 \*out)

{

        return crypto\_hash\_crt(desc->tfm)->digest(desc, sg, nbytes, out);

}

对于md5算法来讲，digest在初始化时指向了shash\_compat\_digest：

int crypto\_shash\_digest(struct shash\_desc \*desc, const u8 \*data,

                        unsigned int len, u8 \*out)

{

        struct crypto\_shash \*tfm = desc->tfm;

        struct shash\_alg \*shash = crypto\_shash\_alg(tfm);

        unsigned long alignmask = crypto\_shash\_alignmask(tfm);

        if (((unsigned long)data | (unsigned long)out) & alignmask ||

            !shash->digest)

                return shash\_digest\_unaligned(desc, data, len, out);

        return shash->digest(desc, data, len, out);

}

md5算法并没有定义digest函数，所以shash\_digest\_unaligned将会被调用，这样，init/update/final将会被再执行一次：

static int shash\_digest\_unaligned(struct shash\_desc \*desc, const u8 \*data,

                                  unsigned int len, u8 \*out)

{

        return crypto\_shash\_init(desc) ?:

               crypto\_shash\_update(desc, data, len) ?:

               crypto\_shash\_final(desc, out);

}

最终计算好的消息摘要值，就通过out返回给最上层的调用者了。

8.        MD5的实现  
上一节展示了init/update/final的三个步骤，对于md5算法而言，其相应的实现为md5\_init/ md5\_update/ md5\_final：

static struct shash\_alg alg = {

        .digestsize        =        MD5\_DIGEST\_SIZE,

        .init                =        md5\_init,

        .update                =        md5\_update,

        .final                =        md5\_final,

        .descsize        =        sizeof(struct md5\_ctx),

        .base                =        {

                .cra\_name        =        "md5",

                .cra\_flags        =        CRYPTO\_ALG\_TYPE\_SHASH,

                .cra\_blocksize        =        MD5\_HMAC\_BLOCK\_SIZE,

                .cra\_module        =        THIS\_MODULE,

        }

};

8.1 md5\_init  
md5初始化MD缓存冲：一个用于计算hash值的128位缓冲区。这个缓冲区由4个32位的寄存器A、B、C、D。其值分别为：  
A: 01 23 45 67  
B: 89 ab cd ef  
C: fe dc ba 98  
D: 76 54 32 10  
  
每一个调用md5算法的实列，使用一个struct shash\_desc结构的\_\_ctx成员来存储上述4个寄存器。在实际操作过程中，使用结构struct md5\_ctx来描述之：

struct shash\_desc {

        struct crypto\_shash \*tfm;

        u32 flags;

        void \*\_\_ctx[] CRYPTO\_MINALIGN\_ATTR;

};

#define MD5\_BLOCK\_WORDS               16

#define MD5\_HASH\_WORDS                4

struct md5\_ctx {

        u32 hash[MD5\_HASH\_WORDS];   //寄存器槽位

        u32 block[MD5\_BLOCK\_WORDS]; //MD5每次处理的数据块，一共是16 \* 4 \* 8 = 512位

        u64 byte\_count;          //计数器，标志一共处理了多少字节数据

};

md5\_init初始化的工作，主要就是初始化上述四个寄存器：

static int md5\_init(struct shash\_desc \*desc)

{

        struct md5\_ctx \*mctx = shash\_desc\_ctx(desc);

        mctx->hash[0] = 0x67452301;

        mctx->hash[1] = 0xefcdab89;

        mctx->hash[2] = 0x98badcfe;

        mctx->hash[3] = 0x10325476;

        mctx->byte\_count = 0;

        return 0;

}

8.2 md5\_update

static int md5\_update(struct shash\_desc \*desc, const u8 \*data, unsigned int len)

{

        struct md5\_ctx \*mctx = shash\_desc\_ctx(desc);

        const u32 avail = sizeof(mctx->block) - (mctx->byte\_count & 0x3f);

        //累计计数器

        mctx->byte\_count += len;

        if (avail > len) {

                memcpy((char \*)mctx->block + (sizeof(mctx->block) - avail),

                       data, len);

                return 0;

        }

        memcpy((char \*)mctx->block + (sizeof(mctx->block) - avail),

               data, avail);

        md5\_transform\_helper(mctx);

        data += avail;

        len -= avail;

        //md5每次按sizeof(mctx->block)大小来处理数据块（64字节或512位）

        //这个循环中，每次从data从拷贝512位出来至mctx->block，做为算法的输入项。

        //然后调用md5\_transform\_helper进行四轮转换，直接数据被处理完

        while (len >= sizeof(mctx->block)) {

                //取数据块

                memcpy(mctx->block, data, sizeof(mctx->block));

                //四轮转换

                md5\_transform\_helper(mctx);

                //data跳过该块向后移至下一个待处理的数据块开始位置

                data += sizeof(mctx->block);

                //数据长度递减块大小

                len -= sizeof(mctx->block);

        }

        //将最后没有处理完的拷贝出来，这里需要注意的是，

        //block中的数据后面是有一部份为空的（如果处理长不是刚才为块的倍数的话）

        memcpy(mctx->block, data, len);

        return 0;

}

8.3 md5\_final  
因为最后还有一部份不足512位的数据块，算法要求对其进行补齐：

static int md5\_final(struct shash\_desc \*desc, u8 \*out)

{

        struct md5\_ctx \*mctx = shash\_desc\_ctx(desc);

        //如update中所述，block中有一部份为空，则要寻找补齐位置，

        //所以，这里要计算一个offset，寻找补齐位置。

        //在update函数中，都是按512位(64字节，0x40)为块单位进行处理的。这里要计算后面为空的开始偏移，

        //只需要用总的处理数来 & (0x40 - 1 = 0x3f)计算即可。

        const unsigned int offset = mctx->byte\_count & 0x3f;

        //p指向空白待补齐部份

        char \*p = (char \*)mctx->block + offset;

        //计算需要补齐的字节数，md5算法对补齐的要求是，补齐后的消息长度比512位的整数倍少64，即448，即56字节,

        //所以这里计算是用56去减，+1是因为有一个字节要特殊处理，即紧跟着那一句。

        int padding = 56 - (offset + 1);

        //md5对于补齐填充的内容要求是，一个位的1和若干位的0，所以这里先将第一个字节置为0x80(二进制为10000000)，

        //即先填充了第一个字节

        \*p++ = 0x80;

        //pading小于0，意味着剩余的数据大于448位，但是小于512位。

        if (padding < 0) {

                //在这种情况下，仍然需要填充，pading为负值，但最大不会超过8过字节(如果是==8，

                //即刚好为512的整数部，在update中就被处理了。

                //所以，这里使用padding + sizeof(u64)，计算出从offset到512位的结束位置那片区域，将其置0。

                memset(p, 0x00, padding + sizeof (u64));

                //这样，又有一个完整的block了，再对齐进行四轮转换。

                md5\_transform\_helper(mctx);

                //置p为block开始位置，填弃总是有的：这里重置padding为56，则表示整片448字节全填充。加上前一个

                //memset，最极端的例子总共填充了 7 \* 8 + 56 \* 8。加上前面那个1字节，md5的补充操作，填充位数是1 - 512。

                p = (char \*)mctx->block;

                padding = 56;

        }

        //补齐填充0，这样，整个block的前448处理完毕，还剩最后64位，接下来附加一个长度信息

        memset(p, 0, padding);

        //置最后64位。在填充的结果后面附加一个以64位二进制表示的填充前信息长度，这样，左移右移。。。？？

        mctx->block[14] = mctx->byte\_count << 3;

        mctx->block[15] = mctx->byte\_count >> 29;

        //进行最后一轮转换，转换前要进行le32到cpu的字节序转换

        le32\_to\_cpu\_array(mctx->block, (sizeof(mctx->block) -

                          sizeof(u64)) / sizeof(u32));

        md5\_transform(mctx->hash, mctx->block);

        //将四个寄存器中的值拷贝出来，做为返回值，拷贝之前，需要进行顺序转换

        cpu\_to\_le32\_array(mctx->hash, sizeof(mctx->hash) / sizeof(u32));

        memcpy(out, mctx->hash, sizeof(mctx->hash));

        //清空mctx，留待后用

        memset(mctx, 0, sizeof(\*mctx));

        return 0;

}

8.4 著名的md5四轮转换

static inline void md5\_transform\_helper(struct md5\_ctx \*ctx)

{

        //将小头序列的数据转转换为cpu序列，可见所有的数据处理都是以cpu序列来做的，

        //所以，最后在取输出值的时候，应该做相应的逆运算

        le32\_to\_cpu\_array(ctx->block, sizeof(ctx->block) / sizeof(u32));

        //进行md5四轮转换

        md5\_transform(ctx->hash, ctx->block);

}

#define F1(x, y, z)        (z ^ (x & (y ^ z)))

#define F2(x, y, z)        F1(z, x, y)

#define F3(x, y, z)        (x ^ y ^ z)

#define F4(x, y, z)        (y ^ (x | ~z))

#define MD5STEP(f, w, x, y, z, in, s) \

        (w += f(x, y, z) + in, w = (w<<s | w>>(32-s)) + x)

static void md5\_transform(u32 \*hash, u32 const \*in)

{

        u32 a, b, c, d;

        a = hash[0];

        b = hash[1];

        c = hash[2];

        d = hash[3];

        MD5STEP(F1, a, b, c, d, in[0] + 0xd76aa478, 7);

        MD5STEP(F1, d, a, b, c, in[1] + 0xe8c7b756, 12);

        MD5STEP(F1, c, d, a, b, in[2] + 0x242070db, 17);

        MD5STEP(F1, b, c, d, a, in[3] + 0xc1bdceee, 22);

        MD5STEP(F1, a, b, c, d, in[4] + 0xf57c0faf, 7);

        MD5STEP(F1, d, a, b, c, in[5] + 0x4787c62a, 12);

        MD5STEP(F1, c, d, a, b, in[6] + 0xa8304613, 17);

        MD5STEP(F1, b, c, d, a, in[7] + 0xfd469501, 22);

        MD5STEP(F1, a, b, c, d, in[8] + 0x698098d8, 7);

        MD5STEP(F1, d, a, b, c, in[9] + 0x8b44f7af, 12);

        MD5STEP(F1, c, d, a, b, in[10] + 0xffff5bb1, 17);

        MD5STEP(F1, b, c, d, a, in[11] + 0x895cd7be, 22);

        MD5STEP(F1, a, b, c, d, in[12] + 0x6b901122, 7);

        MD5STEP(F1, d, a, b, c, in[13] + 0xfd987193, 12);

        MD5STEP(F1, c, d, a, b, in[14] + 0xa679438e, 17);

        MD5STEP(F1, b, c, d, a, in[15] + 0x49b40821, 22);

        MD5STEP(F2, a, b, c, d, in[1] + 0xf61e2562, 5);

        MD5STEP(F2, d, a, b, c, in[6] + 0xc040b340, 9);

        MD5STEP(F2, c, d, a, b, in[11] + 0x265e5a51, 14);

        MD5STEP(F2, b, c, d, a, in[0] + 0xe9b6c7aa, 20);

        MD5STEP(F2, a, b, c, d, in[5] + 0xd62f105d, 5);

        MD5STEP(F2, d, a, b, c, in[10] + 0x02441453, 9);

        MD5STEP(F2, c, d, a, b, in[15] + 0xd8a1e681, 14);

        MD5STEP(F2, b, c, d, a, in[4] + 0xe7d3fbc8, 20);

        MD5STEP(F2, a, b, c, d, in[9] + 0x21e1cde6, 5);

        MD5STEP(F2, d, a, b, c, in[14] + 0xc33707d6, 9);

        MD5STEP(F2, c, d, a, b, in[3] + 0xf4d50d87, 14);

        MD5STEP(F2, b, c, d, a, in[8] + 0x455a14ed, 20);

        MD5STEP(F2, a, b, c, d, in[13] + 0xa9e3e905, 5);

        MD5STEP(F2, d, a, b, c, in[2] + 0xfcefa3f8, 9);

        MD5STEP(F2, c, d, a, b, in[7] + 0x676f02d9, 14);

        MD5STEP(F2, b, c, d, a, in[12] + 0x8d2a4c8a, 20);

        MD5STEP(F3, a, b, c, d, in[5] + 0xfffa3942, 4);

        MD5STEP(F3, d, a, b, c, in[8] + 0x8771f681, 11);

        MD5STEP(F3, c, d, a, b, in[11] + 0x6d9d6122, 16);

        MD5STEP(F3, b, c, d, a, in[14] + 0xfde5380c, 23);

        MD5STEP(F3, a, b, c, d, in[1] + 0xa4beea44, 4);

        MD5STEP(F3, d, a, b, c, in[4] + 0x4bdecfa9, 11);

        MD5STEP(F3, c, d, a, b, in[7] + 0xf6bb4b60, 16);

        MD5STEP(F3, b, c, d, a, in[10] + 0xbebfbc70, 23);

        MD5STEP(F3, a, b, c, d, in[13] + 0x289b7ec6, 4);

        MD5STEP(F3, d, a, b, c, in[0] + 0xeaa127fa, 11);

        MD5STEP(F3, c, d, a, b, in[3] + 0xd4ef3085, 16);

        MD5STEP(F3, b, c, d, a, in[6] + 0x04881d05, 23);

        MD5STEP(F3, a, b, c, d, in[9] + 0xd9d4d039, 4);

        MD5STEP(F3, d, a, b, c, in[12] + 0xe6db99e5, 11);

        MD5STEP(F3, c, d, a, b, in[15] + 0x1fa27cf8, 16);

        MD5STEP(F3, b, c, d, a, in[2] + 0xc4ac5665, 23);

        MD5STEP(F4, a, b, c, d, in[0] + 0xf4292244, 6);

        MD5STEP(F4, d, a, b, c, in[7] + 0x432aff97, 10);

        MD5STEP(F4, c, d, a, b, in[14] + 0xab9423a7, 15);

        MD5STEP(F4, b, c, d, a, in[5] + 0xfc93a039, 21);

        MD5STEP(F4, a, b, c, d, in[12] + 0x655b59c3, 6);

        MD5STEP(F4, d, a, b, c, in[3] + 0x8f0ccc92, 10);

        MD5STEP(F4, c, d, a, b, in[10] + 0xffeff47d, 15);

        MD5STEP(F4, b, c, d, a, in[1] + 0x85845dd1, 21);

        MD5STEP(F4, a, b, c, d, in[8] + 0x6fa87e4f, 6);

        MD5STEP(F4, d, a, b, c, in[15] + 0xfe2ce6e0, 10);

        MD5STEP(F4, c, d, a, b, in[6] + 0xa3014314, 15);

        MD5STEP(F4, b, c, d, a, in[13] + 0x4e0811a1, 21);

        MD5STEP(F4, a, b, c, d, in[4] + 0xf7537e82, 6);

        MD5STEP(F4, d, a, b, c, in[11] + 0xbd3af235, 10);

        MD5STEP(F4, c, d, a, b, in[2] + 0x2ad7d2bb, 15);

        MD5STEP(F4, b, c, d, a, in[9] + 0xeb86d391, 21);

        hash[0] += a;

        hash[1] += b;

        hash[2] += c;

        hash[3] += d;

}

9.        小结  
选择hmac(md5)做为分析的切入口，主要是因为它比较简单，只有两层。但是它完成地展示了Linux加密框架的整个流程：  
        模版的注册λ  
λ        算法实例的分配及注册  
        tfm的分配和spawn的孵化λ  
        密钥设置λ  
        算法的具体执行流程λ  
  
未完，待续......