## *E.) Specifications of the Fleas Hash Algorithm*

### *Overview*

The need for US/NSA-independence applies to all cryptographic functions of Academic Signature. It has largely been met by developing the cryptographic primitives of the "Fleas" family, all basically depending on the Fleas pseudo random functions. These functions make use of the naturally occurring tendency for entropy to increase on the self referential development of a sufficiently large, initially ordered system.

The natural entropy enhancement, however, might possibly be poisoned by feeding a maliciously crafted input block to the basic Fleas pseudo random function.

This problem is irrelevant for designing a PRNG, because the adversary has no control over the input to the algorithms. The probability of encountering such a "poisonous" input block by chance is practically zero.

It is somewhat relevant for Fleas ciphers in CBC-mode, since the adversary may have a chance to select the plain text, which in conjunction with the key is fed to the algorithms. (It is no problem in Counter Mode though.) Care had to be taken to harden against such chosen plain text attacks. In Academic Signature Ciphers this problem was addressed by employing key whitening prior to feeding the material to the algorithms.

This input poisoning danger is definitely relevant in cryptographic hash functions. The adversary has complete control over all of the input to the algorithm and can observe each and any bit flip that occurs during the calculations. There are no secrets. Thus I cannot exclude that a sophisticated adversary might be able to find a poisonous input block. This in turn would allow for existential forgery. Consequently additional elements are needed and are supplied, which protect the input to the Fleas pseudo random functions.

The Fleas Hash functions are based on a modified Merkle Damgard construction. In the plain Merkle Damgard construction, the new block is fed in as the key while the chaining variable is fed in as plain text to a cipher(=keyed pseudo random function). The main countermeasure against input poisoning is what I call a "longitudinal xor" in an input processing chain absent in plain Merkle Damgard.

### *Implementation*

Only the most recent implementation "Fleas_d" of the hash algorithms will be dealt with here. "Fleas_b and c are very similar except they use 2 and 3 parallel paths instead of 4 in their pseudo random functions.

The corresponding code is contained in the module "helpersxx.cpp" of academic signature. Hash evaluation takes place in the procedures "*get_lonu_hash_from_filename_x(..)*" and "*get_lonu_hmac_from_filename_x(..)*". The procedures contain three large switch structures each.

The first switch is dealing with single files smaller than a single block, the second switch deals with the first block of larger files, the third switch with all remaining blocks. The switch statements are rather extensive, since the cases take care of almost all of the implemented hash algorithms. The following excerpts are from the Fleas_d cases for the first and the second switch, respectively.

First block:

```
// ersten Block verarbeiten, reentrant auffüllen, so dass nur noch glatte bloecke bleiben
       readno=plaintext.Read(plainblock,blono - startbyte); //restzahl vom ersten block einlesen
       if(readno != (blono - startbyte)) {if(shobar) throwout(_("Error reading File 8-O"));}
       for(indx=readno;indx<blono;indx++)
          *(plainblock+indx)=(*(plainblock+(indx%readno)))^((char) indx);
       switch(algonum)
       {
       .
       .
       .
       case 13:
       case 10: //Fleas_ld and Fleas_d
          //to prevent "early cancellation attack" mirror-xor output of first round with input
          //keyblock is allocated, still free and can be used as temp storage
          memcpy(keyblock,plainblock,blono);
          for(i=0;i<rounds+1;i++)
          {
            if(!co_develop_ln2(plainblock,blono,(char *)&startbyte,sizeof(unsigned int),rounds+1,4))
              {if(shobar) throwout(_("Err2 in get_lonu_hash_.._lean_x3"));}
            for(j=0;j<blono;j++) //mirrored xoring step
            {
               *(plainblock+j) ^= *(keyblock+blono-j-1);
            }
            develop_l(plainblock,blono,2);
          }
       break;
       .
       .
       }
```

As usual in the Fleas family, padding is done at the head: First, the remaining number of bytes by which file length exceeds a multiple of block size are read in.

The rest of the first block is filled by cycling through the bytes again and additionally Xoring with the loop index.

In the Fleas_d case(algonum==13) the block is first copied to a buffer misleadingly called "keyblock".

The block is co-randomized with the variable "startbyte" uniquely identifying the file's length (modulo block length).

The "longitudinal Xor" is applied in reverse order.

Finally the block is randomized once more using the procedure "develop_l".

Repeat from the second to last step for another two times.

The procedure for the first block is schematically shown in the figure below.

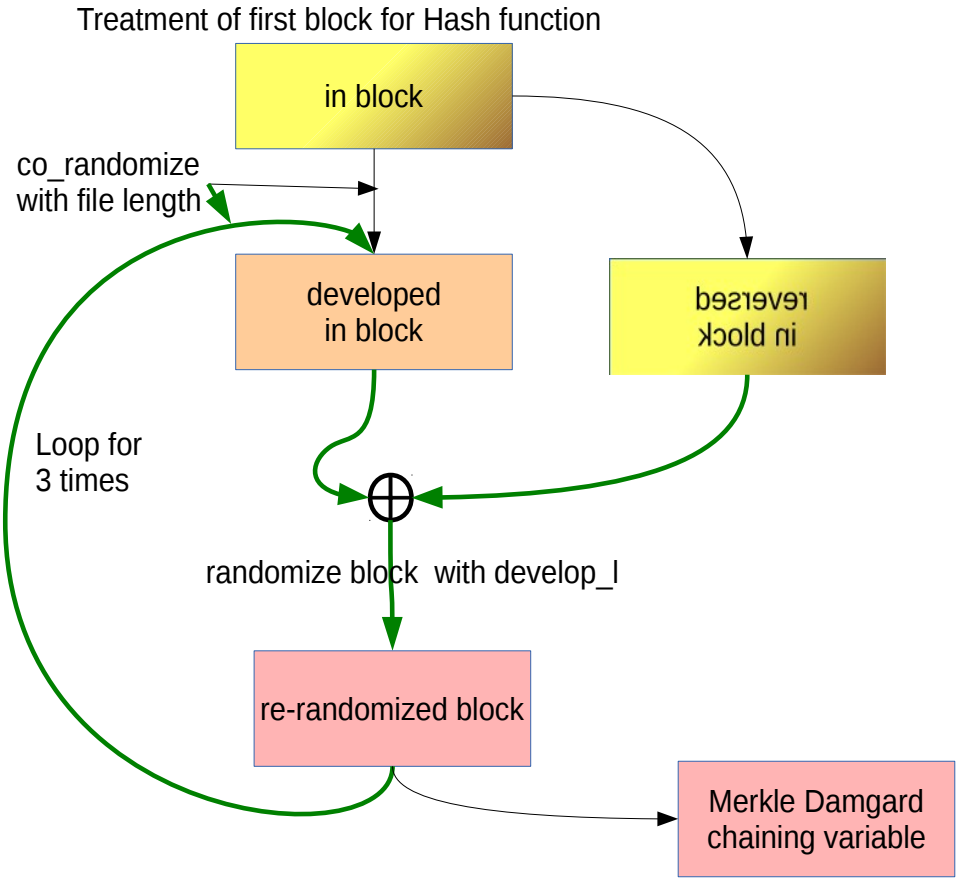


Figure 1: Drawing of the tasks performed for preparing the initial value of the chaining variable. The fat green arrows show repeated tasks, the thin black ones denote singly executed tasks.

Case for remaining blocks:

```
//jetzt weitere Bloecke
   memcpy(keyblock,plainblock,blono);
   for(indx=1;indx<blockzahl;indx++)
   {
     readno=plaintext.Read(plainblock,blono); //einlesen von naechstem plainblock
     if(readno != blono) {if(shobar) throwout(_("Error reading File 8-O"));}
     if(indx+1 ==blockzahl) //im letzten block eine runde mehr
     { rounds++;}
     switch(algonum)
     {
       .
       .
        case 13:
            //to prevent "early cancellation attack" mirror-xor output of last block with input
            memcpy(hlbl,plainblock,blono); //keep reference for longitudinal xor
            if(!develop_ln2_mt(plainblock,blono,rounds,algonum-9,1)) // 13(d)-> 4 paths
                {if(shobar) throwout(_("Unknown Error 3a in Hashing Fleas_c"));}
            for(j=0;j<blono;j++) //mirrored xoring step
            {
                *(plainblock+j) ^= *(hlbl+blono-j-1);
            }
            if(!k_develop_f4(blono,keyblock,0,NULL,blono,plainblock,rounds,algonum))
                  //no cpa blocking! to keep chaining var untouched
                  {if(shobar) throwout(_("Unknown Error 3b in Hashing Fleas_d"));}
        break;
           .
           .
     }
```

The chaining variable is set to the processed first block.

The loop for the remaining blocks is started.

The next plain block is read in.

(In case the last block is encountered the "rounds" variable is incremented.)

in the case "Fleas_d" (algonum==13) a copy of the plain block is put into intermediate storage "hlbl" (= helper block ;-).

The plainblock is randomized using a multi threaded version of "develop_ln2".

The "longitudinal Xor" is applied in reverse order.

The result is used as key in the keyed pseudo random function "k_develop_f4", while the chaining variable("keyblock") is used as "plain text". I apologize for the misleading(=inverted) naming of the block variables.

The result conveniently sits in "keyblock" and can be used as chaining variable for the next block without relocation.

All procedures called in these code snippets are also included in Academic Signature's module "helpers_xx.cpp".

The procedure for the subsequent blocks is schematically shown in the figure below.

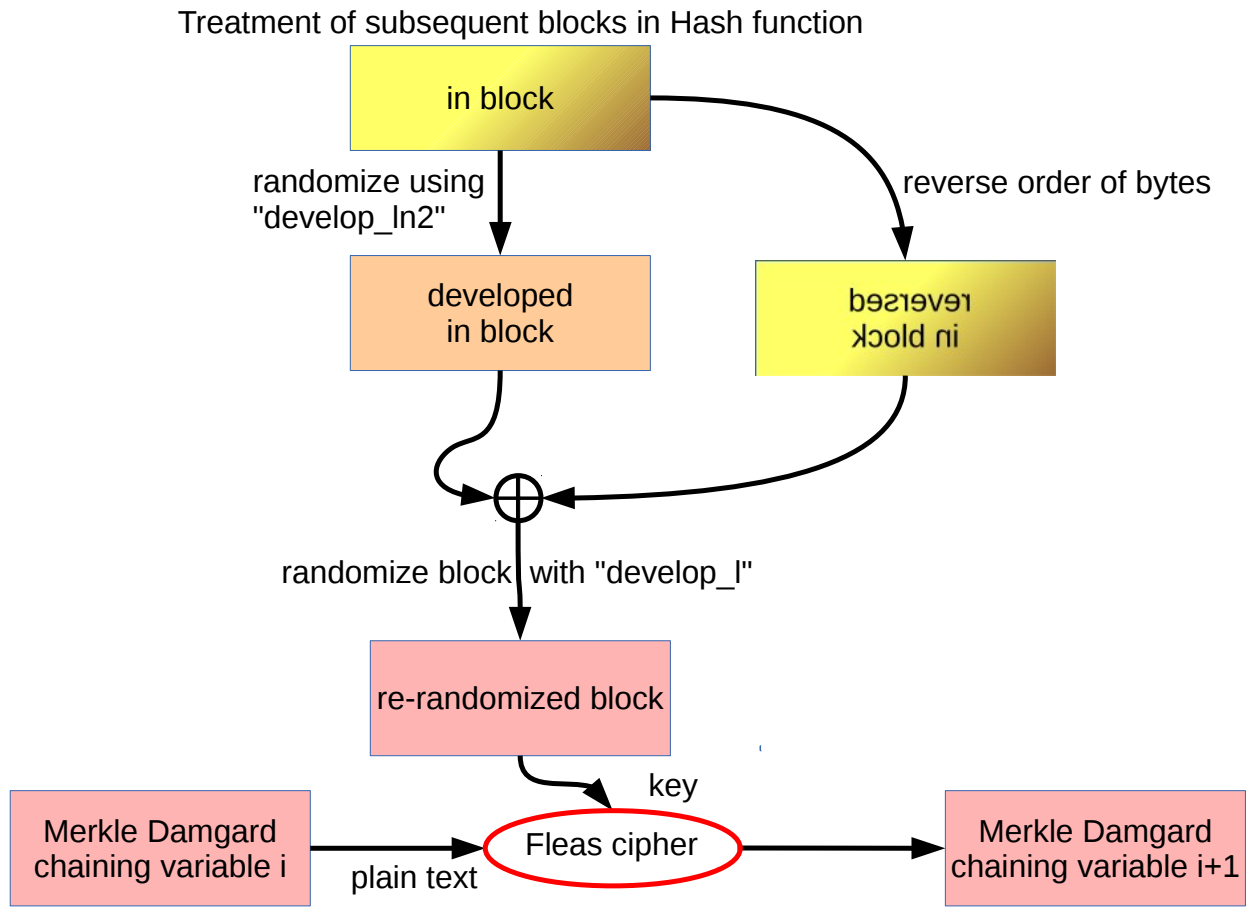


Figure 2: Drawing of the tasks performed for propagating the chaining variable. All subtasks denoted by arrows are performed just once.
The preparation of the in-block is computationally more expensive than the enciphering step to process the chaining variable.

Let me state here again that designing cryptographically secure hash functions was by far the hardest challenge I encountered during my work on the project "Academic Signature".