Specification of the symmetric cipher "Chimera"

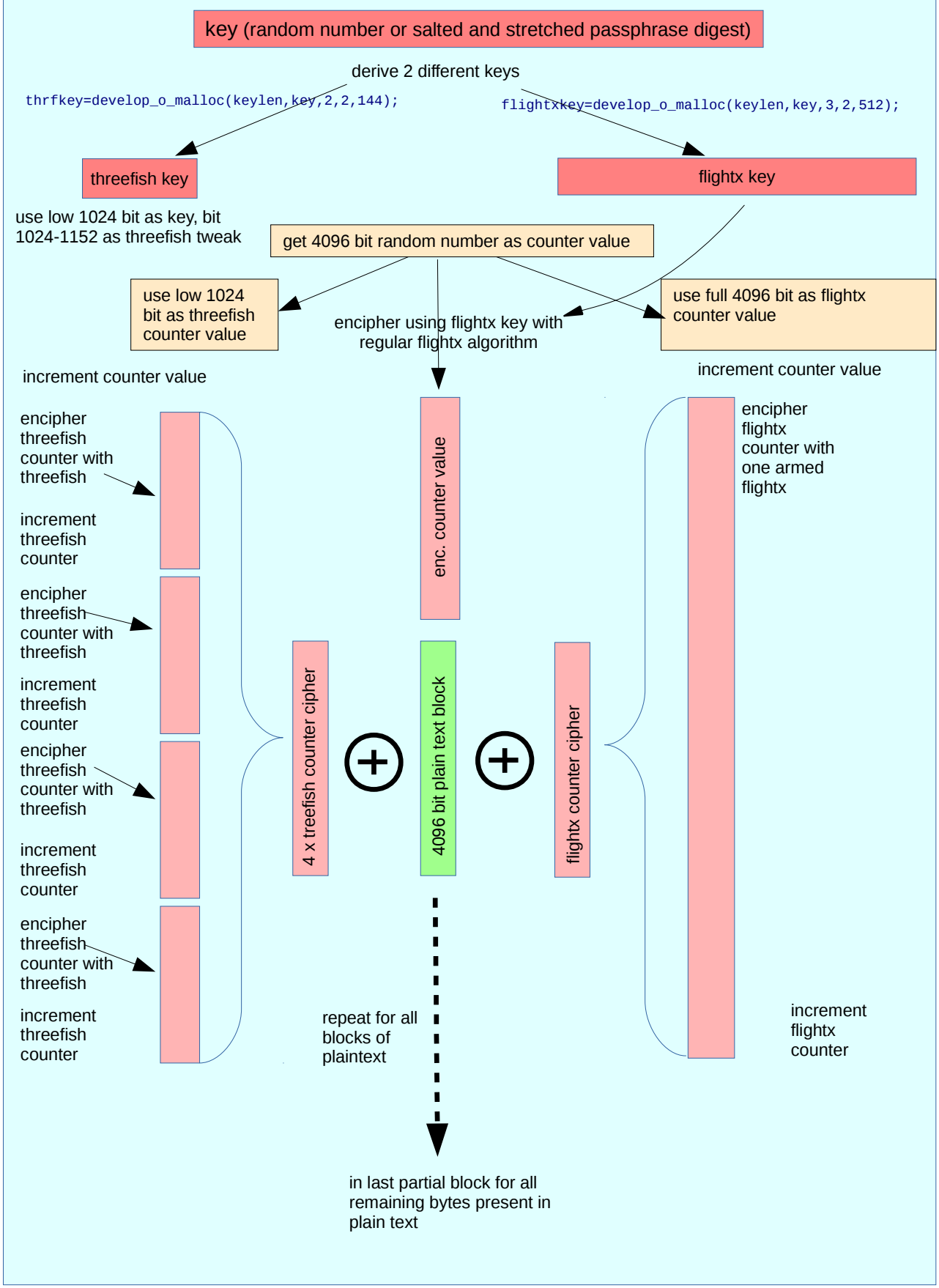
Overview

Chimera is what it says, it is a chimera of two algorithms. In fact it is a pile of the Threefish cipher [http://www.skein-hash.info/sites/default/files/skein1.3.pdf] xored with a single branch Flightx, applied in counter mode. I introduced that cipher to address several possible security concerns.
1.) Threefish is a nice cipher, yet it is a conventional substitution permutation network designed by a group of people some of which live under US jurisdiction and one of them is even employed by the company Microsoft. Let me call this a social concern.
2.) Threefish has a rather regular, simple structure and has a fixed mixing pattern, which some cryptographers view as a potential vulnerability. This point is purely technical.
3.) Although Threefish has a larger block size and a larger key space than other conventional ciphers, in my view they are still way too small.
4.) Flightx (or my other newly developed algorithms) are of a completely different type than traditional ciphers. To my knowledge they have not yet been rigorously charged by other cryptographers.
5.) I am completely unknown in the cryptographic community. This point, and partially also the last one, are social concerns again.

The Chimera in Detail

The included graphics below shows the structure of the chimera
The pile or the "chimera" of the two algorithms is at least as secure as Threefish. If my Flighx were completely trivial to crack(it is obviously not), the security would reduce to the security of Threefish applied in counter mode. If Threefish were trivial to break(it has been rigorously studied in the SHA3 competition), we had still the security of Flightx. If both of them were vulnerable to specific sophisticated attacks, the chimera would still be secure. The attacker needs to break them in conjunction! I like to think of these algorithms as fighters, covering each others back in the Chimera.
Flightx brings the large block size, the large key size (4 kilobit) and the key dependent mixing pattern. Threefish brings the fame of its developers and the speedy diffusion of a fixed pattern substitution permutation network. Flightx can be used singly pathed in the Chimera since Threefish is used as the second path. This is sufficient to block "backtracing" of Flightx. Thus Chimera is somewhat faster than a full two pathed Flightx, but (of course) is slower than its one component Threefish alone.
Remarks regarding the drawing:
A) The key derivation via "develop_o_malloc" uses a seasoned Fleas routine. Interested readers can of course look it up in the source code. The computational cost is negligible since it is done just once. It is not critical and merely serves as producer of two different key. Since its input as well as its output is unknown to the attacker, almost anything could be safely used at this place - even using the same key material for both components altogether.
B) In contrast with theoretical requirements, the IV or the start counter value is also enciphered(using the regular Flightx algorithm as round function in a feistel network). This enciphered counter value comprises the first block of cipher text. The computational cost is negligible, again, since it is done just once. And it surely does no harm to restrict the information accessible to the attacker to the bare minimum.

Schematic Drawing of the Chimera encryption Algorithm



Code Excerpts

Enciphering can be realized in the procedure "encipher_file_fleafish". As you can see, the chimera is internally called "fleafish". I decided to use the name "chimera" for the outside world to respect the term "fish" as brand of the Schneir group and to avoid encroaching on the brandname.

In order to increase efficiency I also created and now use more involved one pass versions for ecc use and advanced symmetrical enciphering, that interweave header creation or header processing and are considerably harder to read. They are compatible though and produce identical cipher files.

The single thread procedure "encipher_file_fleafish" as contained in Academic Signature ver b52 is printed below:

```
/*****************************************************/
bool encipher_file_fleafish(char* fnam, char* key, int keylen, int k_steps, int blolen, int mode,
wxString algo, unsigned char shoprog)
//k_steps: number of owf-steps,   mode:0-clear number, 1number of lonulock steps for number
//
{
  long long filen=0, blockzahl, j;
  FILE *plainfile, *cipherfile;
  char *sfnam, *iv, *ic, *ffic,*ip,hc, *thrfkey, *flightxkey, *flightxblock;
  int i,algoflag,ichar, currpos,lastmax=0;
  longnumber ivno,flivno;
  bool goodp=true;
    // Threefish cipher context data
    ThreefishKey_t keyCtx;

//check new mt version now
 if (algo!= _("chimera")) { throwout(_("error 1 in fleafish call"), 10); return false;}
 if (blolen!= 512) { throwout(_("blocklength error  in fleafish call"), 10); return false;}
 algoflag=17;

  //some primitive error checking
  if(strlen(fnam) > 398) return false;
  wxString fnwx(fnam, wxConvFile);
  if(!wxFile::Exists(fnwx))
  {
      throwout(_("File does not exist!!\nAborting."),2);
      return false;
  }
  //determine Length
  wxFile tmpfl(fnwx);
  filen=tmpfl.Length();
  tmpfl.Close();
  if((plainfile = fopen(fnam, "rb"))== NULL) return false;
  // jetzt cipherfile öffnen mit anhängsel "_1c" etc und setze algoflag
    //create new sfnam "shadow fnam" buffer
    sfnam=(char *)malloc(strlen(fnam)+10);
    if(sfnam == NULL){throwout(_(" failed malloc in enc fleafish"));return false;}
    strcpy(sfnam,fnam);
    strncat(sfnam,"_ff",3);
  if((cipherfile = fopen(sfnam, "wb"))== NULL)
  {
    free(sfnam);
    return false;
  }
  free(sfnam);
  // reform key to 128 byte
  thrfkey=develop_o_malloc(keylen,key,2,2,144); //A(1)
  if(thrfkey==NULL) {throwout(_(" failed malloc in enc fleafish(2)"));return false;}
  threefishSetKey(&keyCtx, Threefish1024, (uint64_t *)thrfkey, (uint64_t *)(thrfkey+128));  //use
first 128 byte of key as key, next 16 as tweak
  flightxkey=develop_o_malloc(keylen,key,3,2,512);//A(2)
  if(flightxkey==NULL) {free(thrfkey); throwout(_(" failed malloc in enc fleafish(3)"));return
false;}
//initialize Progress bar
```

```
  wxProgressDialog pbar(_("Enciphering Progress"),_("Computer is working hard :-)"),100);
  //determine blockzahl
  blockzahl=filen/blolen;
  //create IV from random then lock with key
  iv=(char *) malloc(blolen);
    if(iv==NULL)
      { free(thrfkey);free(flightxkey); return false;}
  ic=(char *) malloc(blolen);
      if(ic==NULL)
      { free(iv); free(thrfkey);free(flightxkey); return false;}
  ffic=(char *) malloc(blolen);
      if(ffic==NULL)
      { free(ic); free(iv); free(thrfkey);free(flightxkey); return false;}
  ip=(char *) malloc(blolen);
      if(ip==NULL)
      { free(ffic); free(ic); free(iv); free(thrfkey); free(flightxkey);return false;}
  flightxblock= (char *) malloc(blolen+512);
      if(flightxblock==NULL)
      { free(ip); free(ffic); free(ic); free(iv); free(thrfkey);free(flightxkey); return false;}

  zuf.push_bytes(blolen, (unsigned char *)iv); //B)
  //wxString primalgo;
  //primalgo=_("Fleas_lc");

  if(mode>0)
  {
      dofeistel(blolen, 512, iv,ic, flightxkey, 4, mode, 0, NULL, _("flightx"));//4Steps, mode-fold
owf, no cpa blocker
  }
  else memcpy(ic,iv,blolen);
  //write locked IV
  for(i=0;i<blolen;i++) //C)
  {
      fputc( *(ic + i),cipherfile) ;
  }
  //put IV into longnumbers  //D)
  ivno.resizelonu(128+2);
  flivno.resizelonu(blolen+2);
  memcpy(ivno.ad, iv, 128);
  memcpy(flivno.ad, iv, blolen);

  ivno.setsize(true);
  ivno.inc();
  flivno.setsize(true);
  flivno.inc();
  //loop complete blocks with progbar update
  for(j=0;j<blockzahl;j++) //E)
  {
       if((shoprog>0)&&(blockzahl>1))
       {
         //prog bar propagation in main loop
         currpos=(int)( (100*j)/(blockzahl-1));
         if((currpos >lastmax)&&goodp)
         {
            goodp=pbar.Update(currpos);
            lastmax=currpos;
         }
       }
      //read in plaintext block
      for(i=0;i<blolen;i++)
      {
          *(ip + i)= (char) fgetc(plainfile);  //E1)
      }
      //get key-developed number block from ivno
      threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) ic); //E2)
      ivno.inc();
      threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) (ic+128));
      ivno.inc();
      threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) (ic+256));
      ivno.inc();
      threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) (ic+384));
      ivno.inc();
      //now develop flightx-half
      //fflivno.ad nach ffic mit flightxkey
      memcpy(flightxblock,flightxkey,512); //copy
      memcpy(flightxblock+512,flivno.ad,512); //copy
      develop_flightx_onearm( flightxblock, blolen+512, 1);  //E3)
      flivno.inc();
      //pick center section
      memcpy(ffic,flightxblock+(blolen+512)/2,512);
```

```
      //xor with plaintext block
      //write to cipher file
      for(i=0;i<blolen;i++) //E4)
      {
          hc=*(ic + i);     // get plain byte
          hc ^= *(ip+i);    // xor with threefish cipher byte
          hc ^= *(ffic+i);  // xor with flightx-cipher byte
          fputc( hc,cipherfile) ; //writeout to cipher file
      }
  }
  //get rest block from plainfile
  //get key-developed number block from ivno
  //xor bytewise and write to cipherfile
  //F)
  threefishEncryptBlockBytes(&keyCtx, (uint8_t *) ivno.ad, (uint8_t *) ic);
      ivno.inc();
  threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) (ic+128));
      ivno.inc();
  threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) (ic+256));
      ivno.inc();
  threefishEncryptBlockBytes(&keyCtx, (uint8_t *)(ivno.ad), (uint8_t *) (ic+384));
      ivno.inc();
  i=0;
      memcpy(flightxblock,flightxkey,512); //copy
      memcpy(flightxblock+512,flivno.ad,512); //copy
      develop_flightx_onearm( flightxblock, blolen+512, 1);
      flivno.inc();
      //pick center section
      memcpy(ffic,flightxblock+(blolen+512)/2,512);
  do
  {
      ichar=fgetc(plainfile);
      if(ichar==EOF) break;
      hc=(char) ichar;
      hc ^= *(ic + i);
      hc ^= *(ffic+i);
      fputc( hc,cipherfile) ;
      i++;
  } while(ichar != EOF);
  //close files
  fclose(cipherfile);  //G)
  fclose(plainfile);
    //free buffers
    free(iv); free(ip); free(ic); free(thrfkey); free(ffic); free(flightxkey); free(flightxblock);
  return true;
}
/*****************************************************/
```

I will now comment on the code, the enumeration is added in red to the code listing:

A) After some error, checking, memory allocation and the like the core routine starts with deriving different and differently sized keys for Threefish and Flightx. Threefish gets 144 byte, where 128 bit are key material and the remaining bytes are called "tweak". I suspect crypto-political reasons for limiting the "key" to 128 and regard the construction as one 144 byte key. Flightx gets 512 byte key material.

Then the number of blocks is determined and some buffers needed later on are allocated.

B) Then the 512 byte (4096 bit) counter value is taken from the PRNG and enciphered using a Feistel network and the standard Flightx algorithm.

C) The enciphered counter value is written as first block into the cipherfile.

D) Two longnumbers are set up. One for the Threefish algorithm containing the 128 byte counter and another one for the one armed Flightx with the 512 byte counter. The Threefish counter is set to the low 128 byte of the random number.

E) A loop running through all complete blocks of the plain text file is started. It performs the

following tasks:
E1 -> read in the plain text block
E2 -> encipher 4 successive 128 byte Threefish counter blocks using Threefish.
E3 -> encipher the 512 bit Flightx counter using a one armed Flightx variant.
E4 -> XOR plaintext with the Threefish 4-block counter cipher chunk and additionally XOR with the Flightx enciphered counter value.

F) Do the same thing as in the last loop once more, but do the XOR and writeout bytewise until EOF is encountered.

G) Cleanup, free allocated pointers and close files.

Now I will give the code of the "one armed" Flightx algortihm:

```
/*****************************************************************************************/
bool develop_flightx_onearm( char* ad, int blolen, unsigned char spice)
//only ever use in conjunction with other xored cipher!, backtrace vulnerabilty in solo use!
{
    char *hbl;
    int i;
    unsigned int blolenx;

    if(blolen&3) blolenx = (blolen - (blolen&3) + 4);
    else blolenx=blolen;
    hbl= (char*) malloc(blolenx);  //make sure to allocate full integer from last address

    if(hbl== NULL)
    {
        throwout(_("error in dev_flight!\naborting! "));
        return false;
    }
    memcpy(hbl,ad,(unsigned int)(blolen)); //only copy what's given
    for(i=blolen;i<blolenx;i++)  //set possible additional bytes to zero
    {
      *(hbl+i)=0;
    }

    if(!dbytadd_ax(blolenx,hbl,hbl,1,spice,1.5)) // A)
    {
       free(hbl);
       return false;
    }
    for(i=0;i< blolen-3;i+=4)
    {
        *((ulong32*)(ad+i)) ^= *((ulong32*)(hbl + i));
    }
    while(i<blolen)
    {
        *(ad + i) ^= *(hbl + i);
        i++;
    }
    free(hbl);
    return true;
}
/*************************************************************/
```

Apart from calling "dbytadd_ax(blolenx,hbl,hbl,1,spice,1.5)" at A) it is trivial and self explaining.Thus I will not comment it and now give the code of dbytadd_ax().

```
/***********************************/
bool dbytadd_ax(int blolen, char* bladwell,  char* pertxt, int rounds, int spice, double fr)

//spice is individualizer(def 1 set in header),fr is flea loop multiplier(def 3 set in header)

{
  unsigned int oi,k,thresh3,blolenx,modlen;
  ulong32 hind;
  unsigned char *blad;
  ulong32 magic=0;
  unsigned short shind,ind;

  if(blolen<4)return false;
```

```
  if(fr<0||fr>10)fr=3; //default is 3 anyways
  if(blolen&3) blolenx=(unsigned int) (blolen - (blolen&3) + 4); //using access as integer requires
integer frame
  else blolenx=blolen;
  modlen=blolenx-4;
  blad=(unsigned char*)malloc(blolenx);
  memcpy(blad,bladwell,blolen);
  for(oi=(unsigned int)blolen;oi<blolenx;oi++)  //set possible additional bytes to zero
  {
      *(blad+oi)=0;
  }

  for(k=0;k<(unsigned int)rounds;k++) // A)
  {
      for(oi=0;oi<(unsigned int)blolen-3;oi+=4) //systematically working up upper -3
      {
          hind =  *((ulong32*)  (blad +( *((unsigned short*)(blad + oi))%modlen) ));
          hind+=(ulong32)spice; //allow for differently spiced variants
          hind *= *(blad+(((unsigned short)oi+1)%modlen)); // determine target index
          shind = (unsigned short) hind;
          shind %= modlen;
          if(shind!=(unsigned short)oi) *((ulong32*)(blad + oi)) += rot32(*((ulong32*)(blad+shind)),
((unsigned char)(shind+oi)&31));
          *((ulong32*)(blad+oi))+=(ulong32)spice+oi+hind;
          if(!(oi&2)) *((ulong32*)(blad+shind)) = *((ulong32*)(blad+shind)) ^0xffffffff;
//entropizer step
      }
      //do highest block // B1)
      {
          oi=blolen-4;
          hind =  *((ulong32*)  (blad +( *((unsigned short*)(blad + oi))%modlen) ));
          hind+=(ulong32)spice; //allow for differently spiced variants
          hind *= *(blad+(((unsigned short)oi+1)%modlen)); // determine target index
          shind=(unsigned short) hind;
          shind %= modlen;
          if(shind!=(unsigned short)oi) *((ulong32*)(blad + oi)) += rot32(*((ulong32*)(blad+shind)),
((unsigned char)(hind+oi)&31));
          *((ulong32*)(blad+oi))+=(ulong32)spice+oi+hind;
          if(!(oi&2)) *((ulong32*)(blad+shind)) = *((ulong32*)(blad+shind)) ^0xffffffff;
//entropizer step
      }
      for(oi=(unsigned int)blolen-4;oi>3;oi-=4) //systematically working down // C)
      {
          hind = *((ulong32*)   (blad +( *((unsigned short*) (blad + oi))%modlen)));
          hind+=(ulong32)spice; //allow for differently spiced variants
          hind *= *(blad+(((unsigned short)oi+1)%modlen)); // determine target index
          shind=(unsigned short) hind;
          shind %= modlen;
          if(shind!=oi) *((ulong32*)(blad + oi)) += rot32(*((ulong32*)(blad+shind)),((unsigned char)
(hind+oi)&31));
          *((ulong32*)(blad+oi))+=(ulong32)spice+(ulong32)oi+hind;
          if(!(oi&2)) *((ulong32*)(blad+shind)) = *((ulong32*)(blad+shind)) ^0xffffffff;
//entropizer step
      }
      //do lowest block // B2)
      {
          hind = *((ulong32*)   (blad +( *((unsigned short*) (blad))%modlen)));
          hind+=(ulong32)spice; //allow for differently spiced variants
          hind *= *(blad+1); // determine target index
          shind=(unsigned short) hind;
          shind %= modlen;
          if(shind!=0) *((ulong32*)(blad)) += rot32(*((ulong32*)(blad+shind)),((unsigned char)
(hind)&31));
          *((ulong32*)(blad))+=(ulong32)spice+hind;
      }
      ind=modlen/2;
      thresh3=((unsigned int)(fr*(double)blolen))/2;
      for(oi=0;oi<thresh3;oi++) //third loop statistically jumping 4byte chunks // D)
      {
          hind =  *((ulong32*)  (blad +( *((unsigned short*)(blad + ind))%modlen) )); // determine
target index
          hind +=magic;
          shind=(unsigned short) hind;
          shind %= modlen;
          magic += *((ulong32*)(blad + oi%modlen)) +(unsigned int)spice;
          if(shind!= ind) *((ulong32*)(blad + ind)) += rot32(*((ulong32*)(blad+shind)),((unsigned
char)(hind+oi)&31)); //add rotated target byte
          *((ulong32*)(blad+ind)) += (ulong32)spice+(ulong32)oi+hind;//kill zeros
          if(oi&3) *((ulong32*)(blad+shind)) = *((ulong32*)(blad+shind)) ^0xffffffff; //entropizer
step
```

```
          ind = (ind +shind+(unsigned short)oi+(unsigned short)spice)%modlen; //reset entry index
      }
       for(oi=0;oi<thresh3;oi++) //fourth loop statistically jumping one byte version // E)
       {
          shind = (unsigned short)*(blad+ind)%blolen; // determine target index
          shind +=(unsigned short)magic;

          magic += *(blad + oi%blolen) +(unsigned int)spice;
          shind %= blolen;
          if(shind!= ind) *(blad + ind) += rotbyte(*(blad+shind),(unsigned char)(ind+shind)&7);
//add rotated target byte
          *(blad+ind)+=spice+(unsigned char)oi;//kill zeros
          if(oi&2) *(blad+shind) = *(blad+shind) ^0xff; //entropizer
          ind = (ind +shind+oi+spice)%blolen; //reset entry index
      }
   }
   memcpy(pertxt,blad,blolen);
   if(blad != NULL) free( blad);
   return true;
}
/*****************************************/
```

The routine has the same structure as the fleas pseudorandom functions explained in detail in the respective section(Fleas Pseudorandom Functions) of the specifications. Thus I will only comment on the differences.
First the block(blolen bytes, cipher block concatenated with key block) is worked on ascendingly(A) and descendingly(C) in 4 byte chunks(Standard fleas works on bytes). This requires some additional, hard to read special treatment for leftover bytes, which is necessary when "blolen", the sum of cipher block length (512 byte) and key length(512 byte) may not be a multiple of four(B1 and B2). In flightx, this is done in any case, also for the regular case of a multiple of 4.
Then the block is worked through erratically, first in four byte chunks(D) starting in the middle, and then in single byte chunks(E). Both the single byte and the four byte loop are repeated for 3/4 times "blolen", amounting to (0.75*4 + 0.75) times "blolen" byte touches. So statistically, each byte of the sum of key length and block length is hit 3.75 times. There may be a small number of bytes which are not hit by the erratic loops(About 2,3 % to be precise). All had been hit at least twice before, however, in the two systematic loops.
The content of all the loops has been explained in detail, line by line, in the previous specification" Fleas Pseudorandom Functions" and will not be repeated here.