H.) The Academic Signature NADA-Cap

1) Overview

NADA stands for No Access Data Available. A NADA-Cap conceals all information in the hybrid cipher file and renders the cipher indistinguishable from noise in toto. The adversary cannot extract any information from the cipher file except its file size, which necessarily gives an upper bound for contained entropy.
In fact the adversary can't even conclude that the file is ciphertext at all. Only if the adversary knows the NADA-Cap key, he/she can expose formatting and cipher info, which are traditionally given in the clear and learn about the nature of the file as an enciphered document.
This is achieved by an additional layer of symmetric encryption for the plaintext header. The encryption uses the cipher Fleas_d in counter mode ("F_cnt_ld"). Academic Signature hybrid cipher header files are all closed by the terminating sequence "X:" after which the nude symmetric cipher is appended. NADA-Cap encryption consequently terminates after the terminating sequence is encountered and enciphered.

Description

There are two different methods to derive the 4096 bit NADA-Cap key. The intrinsic method derives the NADA-Cap key from public key information. This is done using the procedure "cap_prep_from_point", which is shown and commented below.
The extrinsic method derives the NADA-Cap key from a human manageable password or passphrase. In order to ward off dictionary attacks, Academic Signature uses extensive salting and stretching in this case. The used methods are explained in the previous standards-section "***J. Academic Signature Stretching***".

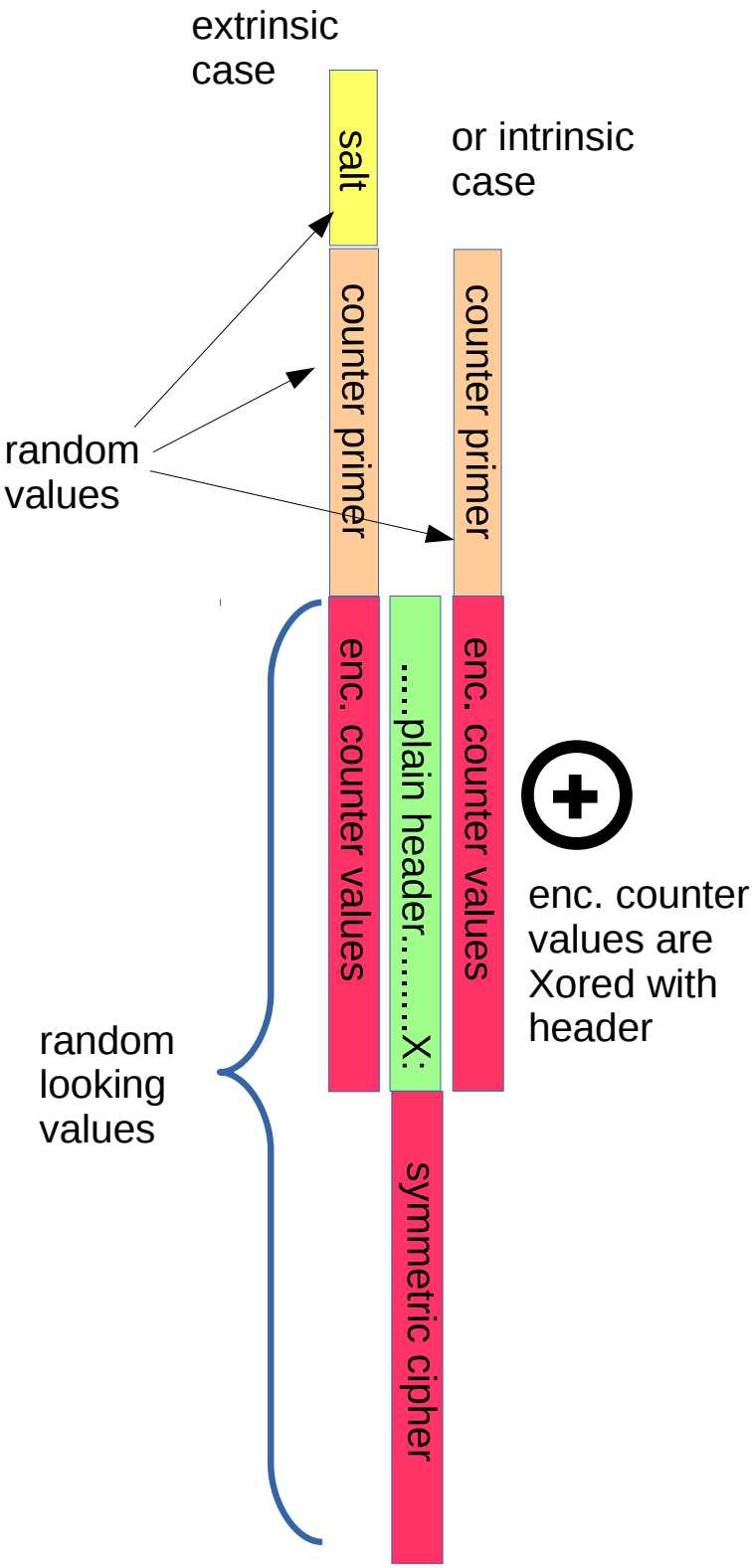
Intrinsic case

A 4096-bit random counter primer value is requested from Academic Signatures PRNG and prepended to the hybrid cipher file. This counter primer is enciphered using the NADA-Cap key via the cipher flightx to receive the true counter. The counter is incremented by one and subsequently used with NADA-Cap key and the cipher F_cnt_ld to encrypt the header of the hybrid cipher file up to and including the end mark of the header.

Extrinsic case

A 2048 bit random salt and a 4096-bit random counter primer value are requested from Academic Signatures PRNG and prepended to the hybrid cipher file. Human manageable keyword, salt and counter primer are given to the function "cap_prep_from_String". This routine derives the 4096 bit NADA-Cap key from the 2048-bit salt and the arbitrary length password string using ten rounds of stretching using 1024 bit elliptic curve "anders_1024_2" as described in "***J. Academic Signature Stretching***". Furthermore, the counter primer is enciphered using the NADA-Cap key via the cipher "flightx" to receive the true counter.
The true counter is subsequently used with NADA-Cap key and the cipher "F_cnt_ld" to encrypt the header of the hybrid cipher file up to and including the end mark of the header.

2) Diagram of the Structure of a NADA-capped Academic Signature Hybrid Cipher



3) Commented Implementation of NADA-Cap Preparation from a Public Key in C++ (intrinsic NADA-Capping)

```
/*****************************************************************/
bool cap_prep_from_point(e_p *pp, longnumber *pcounter, ulong32 cntlen, longnumber *pkey, int outlen)
{
   char *pbuff;

   //concatenate x and y coordinate without leading zeroes A)
   pkey->copynum(&(pp->x));
   pkey->setsize(true); //determine real size
   pkey->resizelonu(pkey->size,0); //shrink to non null bytes
   pkey->appendnum(&(pp->y));
   pkey->setsize(true); //determine real size
   pkey->resizelonu((pkey->size)+2,0); //shrink to non null bytes + 2 zeroes
   //flighxtmap to right length key

   if(!develop_flightx( (char *)pkey->ad,(int)(pkey->length), 3, 1.5)) //security here not critical  B)
   {
      throwout(_("Alert!!\ncouldn't develop counter in cap_prep!\nAborting"),15);
      return false;
   }
   pkey->setsize(true); //determine real size

   pkey->resizelonu((ulong32)outlen+2,0); //truncate or extend to outlength +2 C)
   //develop true counter
   pbuff = (char *) malloc(outlen+cntlen);
   if(pbuff==0)
   {
      throwout(_("Alert!!\ncouldn't alloc buffer in cap_prep!\nAborting"),15);
      return false;
   }
   //prepare counter from primer
   pcounter->setsize(true);
   if(pcounter->length <= cntlen) pcounter->resizelonu(cntlen+2);
   //copy all to buffer

   memcpy(pbuff,pcounter→ad,cntlen); D)
   memcpy(pbuff+cntlen,pkey->ad,outlen); //use first outlen bytes from key
   if(!develop_flightx( pbuff,(int)(cntlen+outlen), 3, 3.5))
   {
      throwout(_("Alert!!\ncouldn't develop counter in cap_prep!\nAborting"),15);
      free(pbuff);
      return false;
   }
   //put result into counter

   memcpy(pcounter→ad,pbuff+outlen/2,cntlen);  E)
   free(pbuff);
   pcounter->setsize(true);
   if(pcounter->size > cntlen)
   {
      throwout(_("Warning, this should not happen in cap_prep!"),10);
      pcounter->resizelonu(cntlen);
      pcounter->shrinktofit(2);
   }
   if(pcounter->size < cntlen)
   {
      if(pcounter->length < cntlen)
      {
         throwout(_("Warning, this should not happen in cap_prep!"),10);
         pcounter->resizelonu(cntlen);
     }
   }
   return true;
}
/*****************************************************************/
```

A) Concatenate x and y coordinate of a point in an elliptic curve. Usually this will be a public ECC key.

B) Use the algorithm flightx as pseudorandom function on the concatenation. The result

C) Extend or truncate to desired keylength. Usually this will be a truncation to 4096 bit.

D) Concatenate counter primer and key. Use flightx as pseudorandom function on the concatenation.

E) Pick the center 4096 bit of the result as start counter value.

4) Commented Implementation of NADA-Cap Preparation from a Keyword in C++ (extrinsic NADA-Capping)

```
/*****************************************************************/
bool cap_prep_from_String(wxString *keyin, longnumber *psalt, longnumber *pcounter, ulong32 cntlen, longnumber *pkey,  ellipse *pse,
int stretch,int outlen)
{
   char *pbuff;
   ulong32 indx;
   //derive real key salted and stretched
   if(!kyprep(pkey, keyin, psalt, pse,  stretch , outlen ))  A)
   {
      throwout(_("Alert!!\nkeyprep for cap failed!\nAborting"),15);
      return false;
   }
   if(pkey->length<outlen) pkey->resizelonu(outlen+2);
   pbuff = (char *) malloc(outlen+cntlen);
   if(pbuff==0)
   {
      throwout(_("Alert!!\ncouldn't alloc buffer in cap_prep!\nAborting"),15);
      return false;
   }
   //prepare counter from primer
   pcounter->setsize(true);
   if(pcounter->length <= cntlen) pcounter->resizelonu(cntlen+2);
   //copy all to buffer
   memcpy(pbuff,pcounter->ad,cntlen);
   memcpy(pbuff+cntlen,pkey->ad,outlen);
   if(!develop_flightx( pbuff,(int)(cntlen+outlen), 3, 3.5))  B)
   {
      throwout(_("Alert!!\ncouldn't develop counter in cap_prep!\nAborting"),15);
      free(pbuff);
      return false;
   }
   //put result into counter
   memcpy(pcounter→ad,pbuff+outlen/2,cntlen);    C)
   free(pbuff);
   pcounter->setsize(true);
   if(pcounter->size > cntlen)
   {
      throwout(_("Warning, this should not happen in cap_prep!"),10);
      pcounter->resizelonu(cntlen);
      pcounter->shrinktofit(2);
   }
   if(pcounter->size < cntlen)
   {
      if(pcounter->length < cntlen)
      {
         throwout(_("Warning, this should not happen in cap_prep!"),10);
         pcounter->resizelonu(cntlen);
     }
   }
   return true;
}
/*****************************************************************/
```

A) Call the routine keyprep with the human manageable password and salt to perform salting and stretching to obtain the resulting 4096 bit key. This routine is described in detail in section "*J. Academic Signature Stretching*".

B) Concatenate counter primer and key. Use flightx as pseudorandom function on the concatenation.

C) Pick center 4096 bit of the result and deposit for later use as first counter value.

5) Excerpt of a Sample Routine to place a Cap on any File with a specified Marker for the End of the Header to be concealed

The routine "OnEnCap" shown below is not contained in the published source code of Academic Signature.
Academic Signature v53 uses more involved, but fully compatible versions, that allow one pass enciphering and deciphering. The respective distributed code fragments, which are not shown here, are necessarily less clearly arranged and are interspersed in the regular en/deciphering routines. They run substantially faster for long files on slow storage media.
The code snippet shown here is more clearly arranged and fairly self explaining to readers familiar with C++. Since it is not contained in the officially published source code, however, it is not commented here.
The code shown here can be used to encap a non NADA-Capped Academic Signature hybrid cipher file such that the regular deciphering routines of Academic Signature will be able to decap and subsequently decipher.

```
/*****************************************************************************/
void CapDialog::OnEnCap(wxCommandEvent& event)
{
   wxString newfnam, messg, keyid;
   longnumber salt, counter, key;
   int endmsiz,i, readflag;
   int saltlen=256, counterlen = 512;
   ellipse el1024;
   char buff[400],cbuf[400],c,*otpbuff;
   ulong32 usecnt=0,useblock=0;
   int mode=13; //10 for fleas ld mt, 13 for monothread
   bool intrinsic=false;
   el_pub pbky;
   FILE *pfile;

   //get file name
   fnam = fnam_ctrl->GetLineText(0);
   //check for existence
   if(!wxFile::Exists(fnam))
   {
      throwout(_("file to cap does not exist!\nDo Nothing."),3);
      return;
   }
   newfnam=fnam+_(".cap");
   //get separator
   endmark = sep_ctrl->GetLineText(0);
   //get Cap Key
   keyword = Capky_ctrl->GetLineText(0);
   if(keyword==_("intrinsic")) intrinsic=true;
   else intrinsic=false;
   //open file to read
   wxFFile capfile(fnam, _("r"));
   if(!capfile.IsOpened())
   {
      throwout(_("Error\ncould not open file to cap!\nDo Nothing."),3);
      return;
   }
   //open capped file to write
   wxFFile nadafile(newfnam,_("w"));
   if(!nadafile.IsOpened())
   {
      throwout(_("Error\ncould not open cappedfile!\nDo Nothing."),3);
      capfile.Close();
      return;
```

```
    }
    if(!intrinsic)
    {
       //write salt and counter to outfile
       salt.makerandom(saltlen);
       readflag=nadafile.Write(salt.ad,saltlen);
    }
    counter.makerandom(counterlen);
    readflag=nadafile.Write(counter.ad,counterlen);
    if(!intrinsic)
    {
        el1024.ellipse_set_stretch1024();
        //prepare cap data structure for extrinsic cap
        if(!cap_prep_from_String(&keyword, &salt, &counter, (ulong32)counterlen, &key,  &el1024, mode,counterlen))
        {
           throwout(_("Error in cap preparation!\naborting!"),5);
           nadafile.Close(); capfile.Close();
          return;
        }
    }
    else //if intrinsic
    {
          //determine public key id
            // read kid: ID
            pfile=capfile.fp();
            readflag = fscanf(pfile,"kid: %100s", buff);
            if(readflag != 1)
            {
               throwout(_("cannot cap intrinsically!\nno key ID found in file"),5);
               nadafile.Close(); capfile.Close();
               return;
            }
            //convert to wxString
            keyid=wxString::FromUTF8(buff);
            if(readflag==1)
              {
                 throwout(_("Capping for pubkey: ")+keyid, 3);
              }
           //rewind capfile
            capfile.Seek(0);
        //load public key
       if(!load_pub_ell_ky(&pbky, &keyid))
       {
          throwout(_("could not find public key \nin internal storage!\nAborting."),10);
          nadafile.Close(); capfile.Close();
          return;
       }
        //determine cap key and true counter from pubkey
       //cap_prep_from_pubky is a simple wrapper of  cap_prep_from_point
       if(!cap_prep_from_pubky(&pbky, &counter, (ulong32) counterlen, &key, counterlen))
       {
          throwout(_("could not determine cap key \nfrom public key\nAborting."),10);
          nadafile.Close(); capfile.Close();
          return;
       }
    }
    counter.inc();
    //set up endmark buffer
    endmsiz= endmark.Len();
    if(endmsiz > 50)
    {
        throwout(_("Are you Nuts!\More than 50 chars Endmark is crazy\nAborting!"),5);
        nadafile.Close(); capfile.Close();
        return;
   }
   strcpy(buff, (const char*) endmark.mb_str(wxConvUTF8));
   strcpy(cbuf, (const char*) endmark.mb_str(wxConvUTF8));//easiest way to ensure \0 termination
   endmsiz=strlen(buff);
   //fill compare buffer
   capfile.Read(cbuf,endmsiz); //overwrite content
   //fill xorblock
   otpbuff=(char*)malloc(counterlen);
   memcpy(otpbuff,counter.ad,counterlen);
   if(! k_develop_f4(counterlen, otpbuff, 0, NULL, counterlen, (char*) key.ad, 2, mode ))  //0,NULL means no cpa blocker, mode= 10 or 13
specifies algo as  Ld( 4 Path)
   {
      throwout(_("Alert, keydevelop failed for cap!"),5);
      free(otpbuff);
      nadafile.Close(); capfile.Close();
```

```
      return;
   }
   // writeout xored compbuffer first
   for(i=0;i<endmsiz;i++)
   {
      if(useblock==counterlen) //refresh bufer
      {
         counter.inc();
         memcpy(otpbuff,counter.ad,counterlen);
         if(! k_develop_f4(counterlen, otpbuff, 0, NULL, counterlen, (char *)key.ad, 2, mode ))  //0,NULL means no cpablocker, algo 10 is
fleas Ld 4 Path
         {
            throwout(_("Alert, keydevelop failed for cap!"),5);
            free(otpbuff);
            nadafile.Close(); capfile.Close();
            return;
         }
         useblock=0;
      }
      c= *(cbuf+i);
      c ^= *(otpbuff+useblock);
      useblock++; usecnt++;
      readflag=nadafile.Write(&c,1);
   }
    //read in byte , check end mark match, xor and writeout
   while((strcmp(buff,cbuf)!=0)&&(usecnt<4000))
   {
      memmove(cbuf,cbuf+1,endmsiz-1); //moveone down
      readflag=capfile.Read(&c,1);  //get next byte
      if(readflag != 1) //EOF reached or error
         {
            throwout(_("Error, file to cap exhausted!\n end mark not encountered!"),5);
            free(otpbuff);
            nadafile.Close(); capfile.Close();
            return;
         }
      *(cbuf+endmsiz-1)=c;  //put into compare buffer
      if(useblock==counterlen) //refresh buffer if necessary
      {
         counter.inc();
         memcpy(otpbuff,counter.ad,counterlen);
         if(! k_develop_f4(counterlen, otpbuff, 0, NULL, counterlen, (char *)key.ad, 2, mode ))  //0,NULL means no cpablocker, algo 10 is
fleas Ld 4 Path
         {
            throwout(_("Alert, keydevelop failed for cap!"),5);
            free(otpbuff);
            nadafile.Close(); capfile.Close();
            return;
         }
         useblock=0;
      }
      c ^= *(otpbuff+useblock); //xor with OTP
      useblock++; usecnt++;
      readflag=nadafile.Write(&c,1);  //writeout cap byte
   }
   free(otpbuff);
   if(usecnt<4000)
   {
      messg.Printf(_("Capping done\nEnd Mark found at byte %i"), usecnt);
   }
   else
   {
      messg.Printf(_("Could not encap, \nend mark not found after 4000 byte!\nAborted!"));
      throwout(messg,3);
      nadafile.Close(); capfile.Close();
      return;
   }
    //after match transfer rest
    while(readflag==1)
    {
       readflag=capfile.Read(&c,1);  //get next byte
       if(readflag==1)nadafile.Write(&c,1);
    }
    nadafile.Close(); capfile.Close();
    throwout(messg);
    return;
}
/*****************************************************************************/
```

The corresponding inverse routine "OnDeCap" is not shown here.