Specification of the large memory footprint key derivation function used for the extrinsic NADA-Cap in Academic Signature

Academic Signature is meant to offer the highest possible protection of privacy against powerful state sponsored agencies with near limitless financial resources. Such adversaries are capable of fabricating large numbers of ASICs to parallelize password cracking. In order to aggravate dictionary attacks carried out with such gear, the necessary chip surface per cracking circuit may be increased. The most straightforward method to achieve this is using a key derivation function with high memory needs.
The legacy key derivation function used in academic signature's login procedure and in symmetric encryption already has formidable memory and number crunching requirements. Using legacy elliptic curve stretching with a 1024 bit elliptic curve already requires at least 4096 bit storage space.
In order to hike up this limit further, a new key derivation function, suitable for up to two GByte block size, was designed and introduced from version 55 onwards. It is used in Academic Signature NADA-Cap protection for extrinsic caps with a block size of 500 kilobyte. Intrinsically it needs at least a second consecutive block of 500 kilobytes. Thus an attacker attempting to parallelize NADA-Cap password cracking needs to assign at least a megabyte of storage space to each circuit and has to supply fast integer arithmetics capabilities (modulo reduction and multiplication) for each cracking circuit.
For an intermediate time, Academic Signature will offer a selector box in the en/decryption dialogs to use the legacy capping key derivation function. Thus the contents of previously encrypted NADA-Capped Files will remain accessible to the legitimate user.

The C++ code of the new key derivation function and some explanations are given below. It is contained in the module `elliptic1.cpp` and is labeled "`kyprep_longblock()`".

```
/*****************************************************************************/
bool kyprep_longblock(longnumber *pky, wxString *keystr, longnumber *psalt, ellipse *pse,  int
stretch,int outlen, ulong32 memsize )

//produce outlen byte key from keystring,
//require at least memsize bytes consecutive memory
{
    longnumber tmp;
    e_p point;
    char *tmpc;
    int i;
    bool goodp;
    ulong32 li, ol_len;

    // if default return trivial
    pky->stortext(keystr);
    if((*keystr== _("default"))||( (psalt==NULL)&&(stretch==0) ))  A)
    {
        return true; //cut short if default
    }
    if(memsize <= (ulong32) outlen)
    {
              throwout(_("Warning, hugeblock smaller than keylength\nadjusting to keylength!"),2);
              memsize= (ulong32) outlen;
       }
    pky->setsize(true);  B)
    pky->shrinktofit(1);
    if(psalt != NULL) pky->appendnum(psalt); //mingle with salt prior to time consuming step
    pky->setsize(true); C)
    pky->shrinktofit(1);
    if((outlen>2000)||(outlen<5)){throwout(_("Fatal error!\nkeylength > 2000 bytes or <5 byte is
crazy\n aborting!")); return false;}
    tmp.resizelonu((unsigned long)(outlen+2),0);
    //expand key into outlen byte with oneway fun from wxstring
```

```
    tmpc=develop_o_malloc((int) pky->size, (char *)(pky->ad), 2,5,outlen,2); D)
    memcpy(tmp.ad,tmpc,outlen);
    free(tmpc);
    tmpc= (char *)malloc(memsize); //may be huge memsize E)
    if(tmpc==NULL)
    {
              throwout(_("insufficient memory error!\n Aborting"),3);
              return false;
       }
    tmp.setsize(true);
    if(pse!=NULL)
    {
//initialize Progress bar
       wxProgressDialog pbar(_("stretching"),_("elliptic curve stretching in progress\nplease
wait!"),stretch);
       for(i=0;i<stretch;i++) //time consuming operations
       {
           goodp=pbar.Update(i+1u);
           tmp.lonumodulo_qqq_e(&(pse->q)); //modulo group order q    F)
           //elliptic operations
           point.copy_ep(&(pse->d0));
           point.mult_p_qj(&tmp, pse);   G)
           //put y-coordinate into tmp
            tmp.copynum(&(point.y));  H)
            tmp.shrinktofit(1);
            tmp.appendnum(&(point.x));
            tmp.setsize(true);
            tmp.shrinktofit(1);
            ol_len= tmp.size;
            //now fill tmp cyclically into longbuffer and introduce superlong PRF
            for(li=0;li<memsize;li++) I)
            {
               *(tmpc + li)= *(tmp.ad + (li% (tmp.size-1)));
            }
            if(!develop_longblock( tmpc, memsize, 1))  J)
            {
               throwout(_("Error in develop longblock\n Aborting!"),10);
               free(tmpc);
               return false;
             }
            //pick first tmp.size bytes of tmpc for tmp
            memcpy(tmp.ad,tmpc,ol_len); K)
           tmp.setsize(true);
           tmp.shrinktofit(1);
       }  L)
    }
    //mingle with salt again
    if(psalt != NULL) tmp.randomize_o(2,psalt,(unsigned long)outlen,0,0);  M)
    else tmp.randomize_d(2,0);
    tmp.resizelonu((unsigned long)(outlen+2),0);
    tmp.setsize(true);
    //expand key into outlen byte with oneway fun from wxstring
    tmpc=develop_o_malloc((int) tmp.size, (char *) tmp.ad, 2,5,outlen,2);   N)
    memcpy(tmp.ad,tmpc,outlen);
    free(tmpc);
    tmp.setsize(true);
    pky->copynum(&tmp);  O)
    return true;
}

/*****************************************************************************/
```

Explanation:

A) Check for escape keyword "default" and skip salting and stretching altogether in this case.
B) Standardize key longnumber size to one leading zero byte and append salt:
C) Standardize resulting longnumber size to one leading zero byte
D) Use a fleas PRF to randomize and cast into outlen bytes as starter block.
E) Allocate large working memory block to tax attackers ASIC memory requirement
F) Modulo reduce starter block with respect to Group order of selected elliptic curve.

G) Multiply with gereator point of Group
H) Standardize x and y coordinate size to longnumber with one leading zero and append x to y.
I) Cyclically fill huge buffer with resulting longnumber
J) Apply special "huge variant" of a Fleas PRF that is suitable for up to 32 bit adressing within block.
K) Pick first count of bytes, that corresponds to the buffer size which was used for the cyclic fill of the huge buffer in Step I
L) Loop to step F until stretch iterations are completed
M) Apply PRF to result concatenated with salt
N) Apply a final PRF-cast to desired key size
O) Copy into key bearing longnumber

The core function "`develop_longblock()`" and its working horse "`dbytadd_a_l()`" adhere to the general pattern of the fleas pseudo random functions. Since the fundamental pattern of the fleas pseudo random functions is commented elsewhere, it will not be commented in detail here.
The 32 bit addressing within the block in "`dbytadd_a_l()`" is achieved by chained index multiplications ("`hind *= ....`"). This results in a certain entropy loss, part of which is covert and hard to exploit(e.g. primes do not occur) and part of which is plain and would facilitate attacks somewhat. This plain part results mainly in an even/odd bias at the lowest bits. Statistically in three out of four random multiplications, the result will be even. In order to prevent this bias from piling up, I added increments after index multiplications.
The code resides in the module "`helpersxx.cpp`" and is given below:

```
/*************************************************************/
bool develop_longblock( char* ad, ulong32 blolen, unsigned char spice)
{
    char *hbl;
    ulong32 i;

    hbl= (char*) malloc(blolen);
    if(hbl== NULL)
    {
        throwout(_("error in dev_longblock!\nmemory shortage?\naborting! "));
        return false;
    }
    memcpy(hbl,ad,blolen);
    if(!dbytadd_a_l(blolen,hbl,hbl,1,spice,1.5))
    {
       free(hbl);
       return false;
    }
    if(!dbytadd_a_l(blolen,ad,ad,1,spice+1,1.5))
    {
       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;
}

/*************************************************************/
bool dbytadd_a_l(ulong32 blolen, char* bladwell,  char* pertxt, int rounds, int spice, double fr)

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

{
```

```
  ulong32 ind,oi,k,hind,thresh3;
  unsigned char *blad;
  ulong32 magic=0;

  if(blolen<2)return false;
  if(fr<0||fr>10)fr=3; //default is 3 anyways

  blad=(unsigned char*)malloc(blolen);

  memcpy(blad,bladwell,blolen);

  for(k=0;k<(unsigned int)rounds;k++)
  {
      for(oi=0;oi<blolen;oi++) //systematically working up
      {
          hind = (ulong32)   *(blad +(( *(blad + oi))%blolen));  //hind is one byte
          hind+=(unsigned int)spice; //allow for differently spiced variants
          hind *= *(blad+((oi+1)%blolen)); // determine target index, hind is now 2 byte
          hind++; //mitigate even-odd bias
          hind *= *(blad+((oi+3)%blolen)); // determine target index, hind is now 3 byte
          hind++; //mitigate even-odd bias
          hind *= *(blad+((oi+7)%blolen)); // determine target index, hind is now 4 byte
          hind++; //mitigate even-odd bias
          hind %= blolen; //points to anywhere in block
          if(hind!=oi) *(blad + oi) += rotbyte(*(blad+hind),((unsigned char)(hind+oi)&7));
          *(blad+oi)+=(unsigned char)spice+(unsigned char)oi;
          if(!(oi&2)) *(blad+hind) = *(blad+hind) ^255; //entropizer step
      }
      for(oi=blolen-1;oi>0;oi--) //systematically working down
      {
          hind = (unsigned int)   *(blad +(( *(blad + oi))%blolen));
          hind+=(unsigned int)spice; //allow for differently spiced variants
          hind *= *(blad+((oi+1)%blolen)); // determine target index
          hind++; //mitigate even-odd bias
          hind *= *(blad+((oi+2)%blolen)); // determine target index, hind is now 3 byte
          hind++; //mitigate even-odd bias
          hind *= *(blad+((oi+6)%blolen)); // determine target index, hind is now 4 byte
          hind++; //mitigate even-odd bias
          hind %= blolen;
          if(hind!=oi) *(blad + oi) += rotbyte(*(blad+hind),((unsigned char)(hind+oi)&7));
          *(blad+oi)+=(unsigned char)spice+(unsigned char)oi;
          if(!(oi&2)) *(blad+hind) = *(blad+hind) ^255; //entropizer step
      }
      ind=blolen/2;
      thresh3=((ulong32)(fr*(double)blolen));
      for(oi=0;oi<thresh3;oi++) //second loop statistically jumping
      {
          hind = (ulong32)*(blad+ind); // determine target index, 1 byte long
          hind +=magic; //might be a little more than 1 byte now
          magic += *(blad + oi%blolen) +(unsigned int)spice;
          hind *= (ulong32)*(blad+(ind+2)%blolen); //now more than 2 bytes
          hind++; //mitigate even-odd bias
          hind *= (ulong32)*(blad+(ind+5)%blolen); //now more than 3 bytes
          hind++; //mitigate even-odd bias
          hind *= (ulong32)*(blad+(ind+7)%blolen); //now more than 4 bytes, possibly reentrant
          hind++; //mitigate even-odd bias
          hind %= blolen;
          if(hind!= ind) *(blad + ind) += rotbyte(*(blad+hind),(unsigned char)(ind+hind)&7); //add
rotated target byte
          *(blad+ind)+=spice+(unsigned char)oi;//kill zeros
          if(oi&2) *(blad+hind) = *(blad+hind) ^255; //entropizer
          ind = (ind +hind+oi+spice)%blolen; //reset entry index, expression possibly reentrant
      }
  }
  memcpy(pertxt,blad,blolen);
  if(blad != NULL) free( blad);
  return true;
}
/***********************************/
```

Tip for developers:
I recommend to use individualized variants of key derivation functions (e.g. by using the spice parameter differently) and explicitly NOT to adhere to standards. This is not like encryption primitives. It would be hard to get it wrong and the best idea would be to force the NSA to design a new ASIC for each single piece of security/login software in the field :-). They wouldn't!

You may use a simplified version without the elliptic curve stretching part and just use an individualized version of the "develop_longblock()" routine, if you shy away from elliptic curve algebra. Just compensate for the ECC part e.g. by doubling the round number "fr".