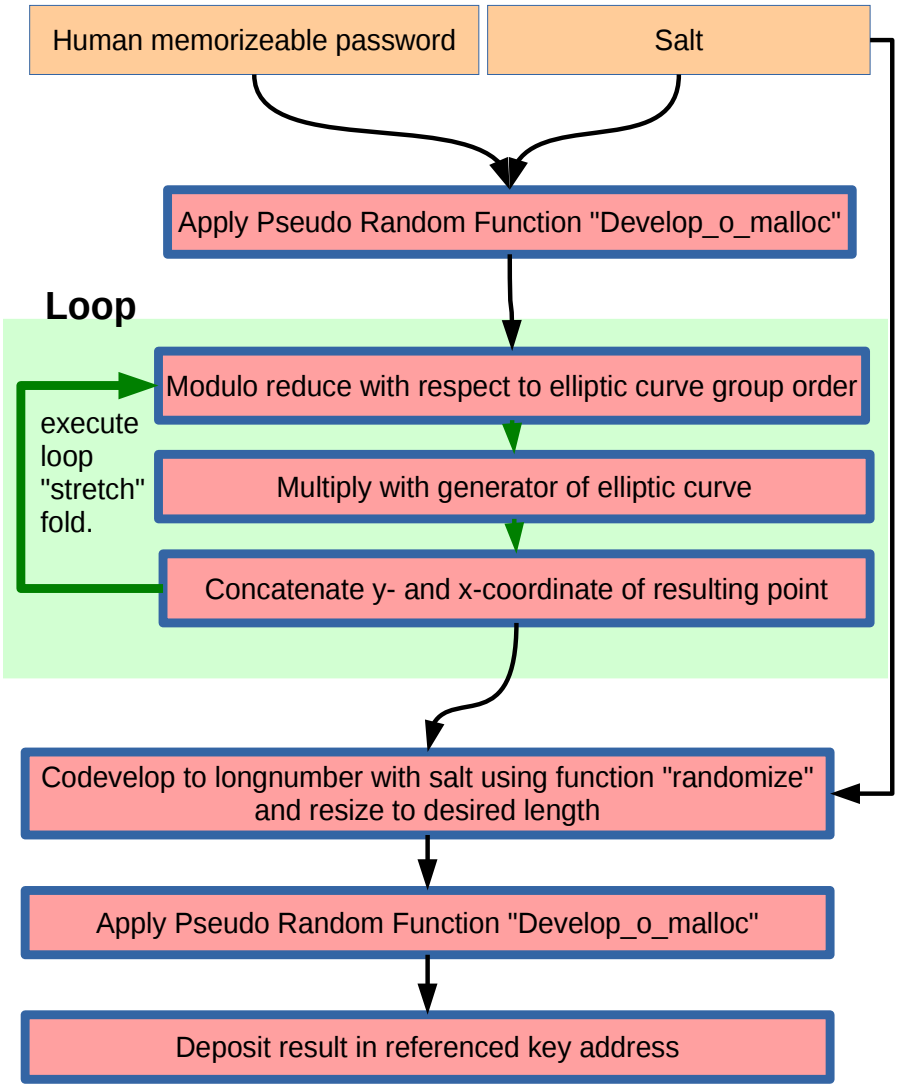
J.) Academic Signature Stretching

1) Overview

Since Academic Signature needs human manageable passwords in several contexts i.e. for the login, for direct symmetric encryption or for applying an extrinsic NADA-Cap. Salting and stretching is necessary to render dictionary attacks unfeasible in these contexts.
Below you see the C++ code of the procedure “kyprep“, which is used for deriving a salted and stretched key from the human manageable password.
The procedure employs elliptic curve point multiplications as main time consuming step. A lot of research has been done to find efficient ways to perform such operations. So it is unlikely that a hostile agency might know of a method to perform this operation substantially faster than with methods known to the public. The one way property of elliptic curve point multiplication is not needed for this use.

2) Block Diagram of the procedure



3) Commented Implementation in C++

```
/*****************************************************************/
bool kyprep(longnumber *pky, wxString *keystr, longnumber *psalt, ellipse *pse,  int stretch,int
outlen, bool legacy )
//produce outlen byte key from ky,
//pky is the pointer to the longnumber the processed key will be deposited in,
//psalt is self-explaining, pse points to the elliptic curve to be used for stretching,
//stretch is the number of reps of the stretching loop
//outlen is the number of desired bytes in the resulting key
//legacy, if true, branches to a legacy password reading routine for backwards compatibility
{
    longnumber tmp;
    e_p point;
    char *tmpc;
    int i;
```

```
    bool goodp;

    // if default return trivial
    if(legacy) pky->stortext_old(keystr);
    else pky->stortext(keystr);
    if((*keystr== _("default"))||( (psalt==NULL)&&(stretch==0) ))      A)
    {
        return true; //cut short if default
    }
    pky->setsize(true); B)
    pky->shrinktofit(1);
    if(psalt != NULL) pky->appendnum(psalt); //mingle with salt prior to time consuming step
    pky->setsize(true);
    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);   C)
    //expand key into outlen byte with oneway fun from string
    tmpc=develop_o_malloc((int) pky->size, (char *)(pky->ad), 2,5,outlen,2); D)
    memcpy(tmp.ad,tmpc,outlen);  E)
    free(tmpc);
    tmp.setsize(true);
    if(pse!=NULL)   F)
    {
//initialize Progress bar
       wxProgressDialog pbar(_("stretching"),_("elliptic curve stretching in progress\nplease
wait!"),stretch);
       for(i=0;i<stretch;i++) //time consuming operations  G)
       {
           goodp=pbar.Update(i+1u);
         tmp.lonumodulo_qqq_e(&(pse->q)); //modulo group order q  G1)
           //elliptic operations
           point.copy_ep(&(pse->d0));   G2)
           point.mult_p_qj(&tmp, pse);
            //put y-coordinate into tmp
            tmp.copynum(&(point.y));   G3)
            tmp.shrinktofit(1);
            tmp.appendnum(&(point.x));
            tmp.setsize(true);
            tmp.shrinktofit(1);
       }
    }
    if(psalt != NULL) tmp.randomize_o(2,psalt,(unsigned long)outlen,0,0);//mingle with salt again H)
    else tmp.randomize_d(2,0);
    tmp.resizelonu((unsigned long)(outlen+2),0);
    tmp.setsize(true);
    //expand/develop key into outlen byte with oneway fun
    tmpc=develop_o_malloc((int) tmp.size, (char *) tmp.ad, 2,5,outlen,2);  I)
    memcpy(tmp.ad,tmpc,outlen);  E)
    free(tmpc);
    tmp.setsize(true);
    pky->copynum(&tmp);  J)
    return true;
}
/*********************************************************/
```

A) If the keyword given is "default", skip salting and stretching.
B) Cast a concatenation of the keyword and the salt into a longnumber.
C) Allocate a temporary longnumber "temp" to hold the result of salting and stretching.
D) Employ the function " develop_o_malloc" to produce a buffer filled with the product of a fleas type one way function of the contents of "temp". Parameters: 2 rounds, 5 paths, the desired output length "outlen", algoflag "2". This is a rather seasoned and, due to excessive "security overhead", slow version from the fleas family. Algoflag 2 means all operations are carried out, no shortcuts taken. Since this function is used in the context of stretching, it is deliberately used in a time consuming variant. The function develop_o_malloc is not shown here but can be found in Academic Signature's module "helpersxx.cpp".
E) The result is transferred to the longnumber tmp and the buffer tmpc is freed right away.
F) The progress bar accompanying the stretching process is initialized.
G) The loop to be repeated for "stretch" times is started
G1) The number "tmp" is modulo reduced with respect to the group order of the selected

elliptic curve.
G2) The generator point of the elliptic curve is loaded and multiplied by tmp.
G3) tmp is filled with the concatenation of the y coordinate and the x-coordiante of the resulting point with one zero byte after each coordinate's highest nonzero byte (1 is the parameter of the function "shrinktofit").
H) After completion of the loop, remix with the salt again and bring to desired output length.
I) Use the same operation as in Step D) as a finalizing step.
J) Copy the result to the longnumber address designated to hold the processed key.

4) Usage

The function "kyprep" described above is mostly called directly and mostly used with a 253 bit elliptic curve. In NADA-Cap relevant routines it is wrapped in "cap_prep_from_String" for extrinsic cap-keys and is used with a 1024 bit elliptic curve. Both curves have been created exclusively for Academic Signature and are used in the Weierstrass form. For cap keys in the intrinsic mode, salting and stretching is not employed, since this mode doesn't employ human memorizable pre-keys.