A Demonstrator for the MOVA Undeniable Signatures

Malik Hammoutène, malik.hammoutene@epfl.ch

October 26, 2004

1 Introduction

Undeniable signatures are digital signatures which protect the privacy of the signer: a signer can sign any digital document, and the signature can be verified through an interactive protocol together with the signer. A new undeniable scheme was proposed by EPFL. It makes possible to have very short signatures (typically: 20 to 30 bits). This scheme is called the MOVA scheme (MO stands for Monnerat Jean, and VA for Vaudenay Serge). Please, read [1] for more information about this protocol.

During this 12 weeks internship, I had to program a software in C, implementing a variant of this scheme (signer without expert group knowledge). This implementation is a Windows API, using sockets (TCP/IP).

2 Quartic Residue Symbol

Let $\mathbb{Z}$ be the set of integers, $i = \sqrt{-1}$ and $\mathbb{Z}[i] = \{a + bi | a, b \in \mathbb{Z}\}$. We recall one proposition and one definition from [2]:

Proposition 2.1. *Consider an irreducible π in $\mathbb{Z}[i]$. If $\pi \nmid \alpha, (\pi) \neq (1+i)$ there exists a unique integer j, $0 \leq j \leq 3$ such that*

$$\alpha^{(N(\alpha)-1)/4} \equiv i^j(\pi).$$

Definition 2.2. *If π is an irreductible, $N(\pi) \neq 2$, then the biquadratic (or quartic) residue character of α, for $\pi \nmid \alpha$, is defined by $\chi_\pi(\alpha) = i^j$ where j is determined by Proposition 2.1. If $\pi|\alpha$ then $\chi_\pi(\alpha) = 0$.*

Definition 2.3. *Let n be an integer. A character χ on $\mathbb{Z}_n^*$ is a map from $\mathbb{Z}_n^*$ to $\mathbb{C} - \{0\}$ satisfying $\chi(ab) = \chi(a)\chi(b)$ for all a,b $\in \mathbb{Z}_n^*$*

For instance, take $\chi_\pi : \mathbb{Z}_n^* \to \{1, -1, i, -i\}$, where $\pi|n$.

We note that quartic symbols and classical Jacobi symbols have the following relation:

$$\left(\frac{\alpha}{\delta}\right)_4^2 = \left(\frac{\alpha}{N(\delta)}\right)_2, \; if \; \alpha \in \mathbb{Z}$$

which is useful to rebuild the quartics after a compression.

2.1 Primarity

For any Gaussian Integer α we denote the real part, resp. imaginary part of α as α_R, resp. α_I. α is said to be *primary* if $\alpha_R + \alpha_I \equiv 0 \pmod 4$ and $\alpha_I \equiv 1 \pmod 4$.

2.2 Algorithm

The goal of this algorithm is to compute $(\alpha/\pi\sigma)_4$, where $\pi = \pi_R + i\pi_I$ and $\sigma = \sigma_R + i\sigma_I$ are Gaussian integers such that $N(\pi)$ and $N(\sigma)$ are prime numbers equal to 1 modulo 4. For more information, see [3] and [4]

Let $\delta = \pi\sigma$. We assume that δ is be a *primary* Gaussian Integer. We successively apply some properties of the quartic residue symbol (see [2], p. 121-127).

1.
$$\left(\frac{\alpha}{\delta}\right)_4 = \left(\frac{\alpha \bmod \delta}{\delta}\right)_4 = \left(\frac{\alpha'}{\delta}\right)_4$$

2.
$$\left(\frac{\alpha'}{\delta}\right)_4 = \left(\frac{1+i}{\delta}\right)_4^r \left(\frac{\alpha''}{\delta}\right)_4,$$

$$\left(\frac{1+i}{\delta}\right)_4 = i^{\left(\frac{\delta_R - \delta_I - \delta_I^2 - 1}{4}\right)},$$

Choose s such that $i^s\alpha''$ is primary:

$$\left(\frac{\alpha''}{\delta}\right)_4 = \left(\frac{(-i)^s(i^s\alpha'')}{\delta}\right)_4 = \left(\frac{i}{\delta}\right)_4^{3s} \left(\frac{\alpha'''}{\delta}\right)_4, \alpha''' \; primary,$$

$$\left(\frac{i}{\delta}\right)_4 = i^{\left(\frac{N(\delta)-1}{4}\right)}$$

3.
$$\left(\frac{\alpha'''}{\delta}\right)_4 = \left(\frac{\delta}{\alpha'''}\right)(-1)^{\left(\frac{(N(\alpha''')-1)(N(\delta)-1)}{16}\right)}$$

4.

$$\alpha := \delta,$$
$$\delta := \alpha''',$$
$$if\ \alpha \neq i^s \rightarrow \text{Step 1}$$

To perform $\alpha \pmod{\delta}$, we do an Euclidian division in $\mathbb{Z}[i]$ [5]:

$$\frac{\alpha}{\delta} = u + iv, (u, v) \in \mathbb{Q}^2 \rightarrow \exists(u_o, v_o) \in \mathbb{Z}^2 s.t. |u - u_0| \leq \frac{1}{2}\ and\ |v - v_0| \leq \frac{1}{2}$$

Then

$$\alpha = \delta(u_0 + iv_0) + r_0,\ with\ N(r_0) < N(\delta)$$

To find r at point 2, we find the maximum integer r such that $N(\alpha) = 2^r\beta$, where $\beta \in \mathbb{Z}$.

Some simulations have shown that, if α and δ are 512 bits the quartic residue symbol $(\alpha/\delta)_4$ takes 28 milliseconds to be computed. In comparaison, a modular inversion takes 0,186721 milliseconds and a modular exponentiation (square-and-multiply) takes 13,25 milliseconds. Note that the two last results come from optimized routine of GMP.

The output of this algorithm is 1, -1, i or -i. In the software, we use a compressed version of the quartic and its value is 0 if the output is 1 or i, and 1 if the output is -1 or -i. To rebuild the quartic, we compute $(\alpha/N(\delta))_2$ then,

if $quartic = 0$ and $jacobi$ = 1 $\longrightarrow$ $quartic$ = 1
if $quartic = 1$ and $jacobi$ = 1 $\longrightarrow$ $quartic$ = -1
if $quartic = 0$ and $jacobi$ = -1 $\longrightarrow$ $quartic$ = i
if $quartic = 1$ and $jacobi$ = -1 $\longrightarrow$ $quartic$ = -i

3 Proofs and Signature

3.1 Problems

3.1.1 S-GHI Problem (Group Homomorphism Interpolation Problem)

Parameters: two Abelian groups G and H, a set of s points $S \subseteq G \times H$
Input: $x \in G$

Problem: find $y \in H$ such that(x, y) interpolates with S in a group homomorphism.

3.1.2 S-GHID Problem
(Group Homomorphism Interpolation Decisional Problem)

Parameters: two Abelian groups G and H, a set of s points $S \subseteq G \times H$
Input: a point $(x, y) \in G \times H$

Problem: does (x, y) interpolate with S in a group homomorphism?

3.2 The GHI Proof

The prover wants to convince a verifier he knows that some pairs (x, y) interpolate in a group homomorphism. Therefore, he performs the GHIProof [1]. We note that, assuming the fact the prover and the verifier are honest, the protocol always succeeds.

3.3 The coGHI Proof

Let $(x_i, z_i) \in G \times H$, $i = 1, ..., Lsig$, where x_i's are equal to the $Xsig_i$'s. Note that z_i's should be equal to the $Ysig_i$'s, but are not because they are from the alleged non-signature. The prover wants to convince a verifier that for at least one i the answer of the GHID problem with (x_i, z_i) is negative. We suppose an honest verifier enters an alleged non-signature.

3.4 The signature

The message is used to generate $Xsig_1$, ..., $Xsig_{Lsig}$ from a pseudo random number generator $Gen2(M)$. Then the signer computes $Ysig_k = Hom(Xsig_k)$ for $k = 1$, ..., Lsig. The signature is $(Ysig_1, ..., Ysig_{Lsig})$.

4 The software

4.1 User manual

The software is written in C with Microsoft Visual C++ 6.0 and is called $crypt1.exe$. They are 6 parts:

1. The public and secret keys generator

   Choose the size of the random numbers, then insert one name for the public key and another one for the secret key. Click on *Ok* and the two keys will be generated and saved on the desktop. Note that the extension of the public key is *.pmova* and the extension of the secret key is *.smova*. Note that at the same time, a folder called $MovaDir$ has been created on the Desktop.

2. The file signer

Indicate the path of the secret key, the path of the file you want to sign and then click OK. Two words will appear. These two words are the signature of your file by the specific secret key.

Afterwards, you can send by email, for example, the public key, the signed file and the signature to a verifier. Once you have a public key and a secret key, you can sign files.

3. The GHIProof as verifier

   You want to verify the validity of a signature. For this, you have to enter the DNS address of the prover, the port you will be connected to, the path of the public key and the path of the signed file, and the signature. All these elements were given by the prover. As soon as you are sure that the prover is connected, click OK. The GHIProof protocol will be executed and, if it succeeds, a message box will tell it to you. Else, the protocol will be aborted.

4. The GHIProof as prover

   For this implementation, we use quartic residue symbols as a group homomorphism:

   $$f : \mathbb{Z}_n^* \to \{-1, 1, -i, i\}$$

   You simply have to enter the path of the secret key and to indicate the port number on which you will wait for the verifier. Then, click Ok. Once the verifier is connected, the protocol starts and if the GHIProof succeeds, a message box will tell it to you. Else, the protocol will be aborted.

5. The coGHIProof as verifier

   It works exactly the same as the $GHIProof\ as\ verifier$. It simply creates other files.

6. The coGHIProof as prover

   It works exactly the same as the $GHIProof\ as\ prover$. It simply creates other files.

During the execution of GHIProof and coGHIProof, several files are created and saved in the MovaDir folder. You can delete them after the execution of the protocol.

The software has been prepared to choose automatically if it executes a confirmation or a denial protocol, depending if the signature is valid or invalid, but this release doesn't do it. It **can't be executed on a LINUX or MAC OS !** It has been developped as a Windows Application. It doesn't need any special installation: just copy the file on your computer and double-click on the icon to execute it. The only imperative need is to have a known opened port.

4.2 Libraries

4.2.1 GMP

GMP [6] is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision, except the one implied by the available memory in the machine GMP runs on. The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc. This library is systematically used in this program to manipulate numbers around 1024 bits.

4.2.2 LibTomCrypt

LibTomCrypt [7] is a fairly comprehensive, modular and portable cryptographic toolkit that provides developers with a vast array of well known published block ciphers, one-way hash functions, chaining modes, pseudo-random number generators, public key cryptography and a plethora of other routines. MD-5 and SHA-1, which are used to hash the files to sign and the commitments, were taken from this library.

4.3 The protocol

1. The public and secret keys generator

   The secret key is a primary $\delta = \delta_R + i\delta_I$, such that $N(\delta) = pq$ where $p \equiv 1 \pmod 4$, $q \equiv 1 \pmod 4$ are two prime numbers in $\mathbb{Z}$.

   The public key is

   (a) $X_{group} = \mathbb{Z}_n^*$, represented by the norm of δ

   (b) $Y_{group} = 0, 1, 2, 3$

   (c) $d = 4$

   (d) seedK, which is a random number

   (e) (Y_{Key_1},...,$Y_{Key_{LKey}}$), where $Y_{Key_i} = Quartic(X_{Key_i})$. Each X_{Key_i} are pseudo random numbers that are generated by $seedK$.

2. The file signer

   The file to be signed is hashed with SHA-1 and the output is used as a seed for a pseudo random number generator to generate the $X_{Sig_i} \in$

$\mathbb{Z}_n^*$. The signature is $Y_{Sig_i} = Quartic(X_{Sig_i})$ and is transformed in intelligible words taken from the RFC1760 dictionnary [8].

3. The GHIProof

   The Verifier picks random $r_i \in \mathbb{Z}_n^*$ and $a_{i,j} \in \mathbb{Z}_4$ to compute $u_i = r_i^4 \prod_{k=1}^{s} g_i^{a_{i,k}} \pmod n$ and $w_i = \sum_s a_{i,s} e_s \pmod 4$, where $(g_1, ..., g_s) = (Xkey_1, ..., Xkey_{Lkey}, Xsig_1, ..., Xsig_{Lsig})$ and
   $(e_1, ..., e_s) = (Ykey_1, ..., Ykey_{Lkey}, Ysig_1, ..., Ysig_{Lsig})$, $1 \leq i \leq k$. He then sends the u_i to the prover through a socket (TCP/IP).
   Once he receives the u_i, the prover computes $v_i = Quartic(u_i)$ and sends a commitment to v_i: commitment $= MD5(v_i|seed)$. After having received the commitment, the verifier sends all r_i and $a_{i,j}$ which are the content of the ra_file.crypt file. Then, the prover checks that the u_i's computations are correct. He then sends the seed to open his commitment and the verifier check that $v_i = w_i$.

4. The coGHIProof

   The verifier picks random $r_{i,k} \in G$, $a_{i,j,k} \in \mathbb{Z}_4$ and $\lambda_i \in \{0, 1\}$. then he computes $u_{i,k} := \left(r_{i,k}^d \prod_s g_j^{a_{i,j,k}} x_k^{\lambda_i}\right) \pmod n$ and
   $(w_{i,k} := \sum_s a_{i,j,k} e_j + \lambda_i z_k) \pmod 4$, where $g_j = Xkey_j$, $e_j = Ykey_j$. He then sends u_i's and w_i's to the prover. Once he received the $u_{i,k}$, the prover computes $v_{i,k} = Quartic(u_{i,k})$. Since $\lambda_i(z_k - y_k) = (w_{i,k} - v_{i,k})$, he should be able to find every λ_i if the signature is invalid. Otherwise, he sets λ_i to a random value and sends a commitment to λ to the verifier who sends all $r_{i,k}$'s and $a_{i,j,k}$'s to the prover. Once he has received the ra_file.crypt file, the prover verifies that u and w were correctly computed. He then opens the commitment. Finally, once the commitment is opened, the verifier checks that the prover could find the right λ. Otherwise, the protocol is stopped and the validity of the signature remains undeterminated.

4.4 The functions

Each of the 6 parts of the protocol uses several functions. Here are some explanations.

The public and secret keys generator

The function $generateKeys(pkname, skname, bitnb)$ has as entry the names the user wishes for the public and the secret key, and the size in bits of the random numbers used to build δ: it first generates two random numbers p and q of $bitnb$ bits with $getrand2(randNbX, bitnb)$, then it creates the files $pkname$ and $skname$, it generates two prime number such that they are congruent to $1 \pmod 4$ with $genPrime(prime\_n, randNbX)$ and, using

the cornacchia algorithm, it finds $p = \pi_R^2 + \pi_I^2$ and $q = \sigma_R^2 + \sigma_I^2$. This two prime numbers are π and σ and $\delta = \pi\sigma$ is saved in $skname$.
Then, a seed is computed with $getrand()$ and saved in $pkname$ with $N(\delta)$, the norm of δ. Using $genrand\_int32()$, 80 X_{Keys} are computed and $Y_{Keys} = (X_{Keys}/\delta)_4$ are computed with $quartic(piXY, alphaXY)$ and saved in $pkname$.

The file signer

The source file is hashed with SHA-1, using $encode(szData, out, dwFileSize)$, and the outuput is used has a seed to compute 20 numbers, the X_{Sigs}, of 1024 bits. Then, using $quartic(piXY, alphaXY)$, $Y_{Sigs} = (X_{Ksigs}/\delta)_4$ are computed, the result (something like 001001011010101110010) is separated in two 10bits parts, one bit (checksum) is added, the number is converted in base 10 and the two words in that positions in $dico[z1]$ are taken and are the signature (for example: $MANYADD$).

The GHIProof (Verifier)

First, $computing\_the\_s\_file(hWnd, n\_pk, yKeys\_pk, seedK\_pkr, conc, szData)$ function is called and computes
$S = ((X_{Key_1}, Y_{Key_1}), ..., (X_{Key_{80}}, Y_{Key_{80}}), (X_{Sig_1}, Y_{Sig_1}), ..., (X_{Sig_{20}}, Y_{Sig_{20}}))$

This function returns n_pk, the norm of δ, $Ykeys$, which are the 80 quartics from the public key, $seedK$, conc, which is a concatenation of $Ykeys$ and of Y_{Sigs}, and szData which is the S file itself. $hWnd$ is simply the name of the window where we are.

Once S is built, the verifier picks r_i's, using $getrand3(r, zn)$ (it computes a random number r in $\mathbb{Z}_{zn}$), a_i's, using $getrand()$, computes u_i's and w_i's and stocks these values in $u\_file.crypt$. The r_i's and a_i's are saved in $ra\_file.crypt$.

With $ClientConnect(req\_host, PORT, hWndconf)$, the verifier connects himself to a port number and checks if he finds the prover. If he succeeds, he sends the $u\_file.crypt$ to the prover, using $csSendFile(pathDesktop(\text{“}MovaDir/u\_file.crypt''))$. Then once he gets the commitment $(csGetFile(pathDesktop(\text{“}MovaDir/commit\_file.crypt'')))$, he sends $ra\_file.crypt$ and receives the seed to open the commitment. He then computes $commit(concat2, com2, strlen(concat2) + 1)$ and compares if what he has computed is the same as $commit\_file.crypt$. If not, the GHIProof failed.

The GHIProof (Prover)

First, *getting_the_u_file(hWndconfp,szData2ref,szDataref)* is called to receive the u_i's. This function returns the u_i's and the secret key. Using the secret key and the u_i's, the prover can compute $v_i = f(u_i)$, using $quartic(u_part, alphaXY)$. Then, he sent a commitment, com, to the prover and wait for the $ra\_file.crypt$.

Once $csGetFile(pathDesktop("MovaDir/ra\_file.crypt")$ is executed, he opens the file and check that the u_i's he recomputes with the $ra\_file.crypt$ and the two temporary files saved in $MovaDir$ are the same as the ones from $getting\_the\_u\_file$. If not, the protocol is aborted, else, the prover sends the seed, $concat80$, to open the commitment.

The coGHIProof (Verifier)

Following the same method as for the $GHIProof$, the verifier first computes
$S = ((X_{Key_1}, Y_{Key_1}), ..., (X_{Key_{80}}, Y_{Key_{80}}), (X_{Sig_1}, Z_{Sig_1}), ..., (X_{Sig_{20}}, Z_{Sig_{20}}))$. The $computing\_the\_s\_file$ function returns n_pk, the norm of δ, $yKeys$, and szData which is the S file itself, as in the $GHIProof$.

Then, the verifier picks $r_{i,k}$'s using $getrand3(r, zn)$, $a_{i,j,k}$'s and λ_i's , using $getrand()$, computes $u_{i,k}$'s and $w_{i,k}$'s, and saves the results in $u\_denifile.crypt$, before sending this file to the prover,with the help of the $csSendFile$ function.

Once he receives the commitment $csGetFile(pathDesktop("MovaDir/commit\_file.crypt"))$ from the prover, he sends the $ra\_file.crypt$ where all the $r_{i,k}$'s and $a_{i,j,k}$'s were saved and once he gets the key to open the commitment, he checks if the prover could find the right λ_i. If not, the protocol is stopped and the invalidity of the signature remains undetermined.

The coGHIProof (Prover)

As for the $GHIProof$, the prover first calls the function $getting\_the\_u\_file$, which creates $u\_file.crypt$ and returns the secret key too. The prover then computes $v_{i,k} = f(u_{i,k})$, using $quartic(u\_part, alphaXY)$, and try to compute the λ_i: from the secret key, he recomputes y from the temporary files, the true signature, then, bit by bit he compares it with z, the alleged non-signature. If $y_k - z_k = 0$, he can't say anything about λ. If $w_{i,k} - v_{i,k} = y_k - z_k \neq 0$ and $= z_k - y_k$, for all k, he sets λ_i to 1. If $y_k - z_k \neq 0$ and $w_{i,k} - v_{i,k} = 0$, for all k, he sets λ_i to 0. Otherwise, he chooses it randomly with $getrand()$.

Once the 20 λ_i's are computed, he sends a commitment to the verifier and then recomputes all $u_{i,k}$ and $w_{i,j,k}$ with the help of $ra\_file.crypt$ and $s\_file.crypt$. He then compares the results to the content of the $u\_file.crypt$. If everything is al right, he opens the commit and the coGHIProof is finished for the prover.

5 Thanks

I wish to thank *Ivan Suarez Atias* at CSAG-EPFL, for his help to write the quartic computation algorithm.
I'd like to thank *Fabien Ezber* from Sylog Consulting, all the people at LASEC-EPFL, and all the "geeks" on www.cppfrance.com!

Thank you to all of you for your help ☺

References

[1] J. Monnerat and S. Vaudenay, *Generic Homomorphic Undeniable Signatures*, 2004.

[2] K. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, pp. 121-122, 1998.

[3] Z.H. Sun, *Supplements to the theory of quartic residues*, Acta America97, pp. 361-377, 2001.

[4] F. Lemmermeyer, *Reciprocity Laws*, pp. 194-207, Springer, 2000.

[5] V. Lèfevre, *Entiers de Gauss (sujet d'étude XM')*, 1993.

[6] T. Grandlund, *GNU MP*, 2004.

[7] T. St Denis, *LibTomCrypt Version 0.97b*, 2004.

[8] N. Haller, *RFC 1760, The S/Key One-Time Password System*, Network Working Group, Bellcore, 1995.