ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Generic Homomorphic Undeniable Signature Scheme: Optimizations

Yvonne Anne Oswald

SIN

Semester Project

February 2005

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Jean Monnerat
EPFL / LASEC

LASEC

Contents

List of Tables

1 Introduction

Undeniable signatures, which have been introduced by Chaum and van Antwerpen in [3], differ from classical digital signatures in the verification process. Contrary to classical digital signatures, where anyone holding the public key of the signer is able to verify whether a given signature is valid or not, one has to interact with the signer to be convinced of the validity of the signature. This interaction, the confirmation protocol, gives the signer the control over the distribution of the verification, as no entity can verify a signature without the signer. To prevent a dishonest signer from falsely claiming a signature to be invalid, we need a second interactive protocol, the denial protocol. With this protocol, a honest signer can prove that a forged signature is not valid. A complete undeniable signature scheme therefore consists of a key generation and signature algorithm, as well as an interactive confirmation and denial protocol.
A new undeniable signature scheme called MOVA was proposed in [1] and generalized to a generic homomorphic undeniable signature scheme in [2]. These signature schemes allow signatures to be arbitrarily short (typically around 20–30 bits), depending on the required security level.
In 2004, a demonstrator for MOVA signatures has been implemented. The aim of this project is to optimize the existing implementation and to implement 3 additional homomorphisms and compare them with each other.
The mathematical techniques used are based on group homomorphisms. One can transform a private group homomorphism from public groups G and H into an undeniable signature scheme. The homomorphism used in the demonstrator is the quartic residue symbol. We tried to optimize the computation of the quartic residue symbol as well as trying out different algorithms. Furthermore we implemented the variant of the generalized signature scheme involving an homomorphism based on the discrete logarithm. As the demonstrator was implemented in C, all our optimizations and new implementations are written in C as well. The main algorithms of the signature scheme are based on operations with large integers which are up to 1024 bits long. To handle numbers that large we used the GNU Multiple Precision Arithmetic Library (GMP) [7]. This library provides highly optimized basic and number theoretic functions.
We begin with a survey of the mathematical theory necessary for the scheme in Section 2. Section 3 contains the description of algorithms for the quartic residue symbol and their implementation. Details regarding the variant using discrete logarithms are provided in Section 4, Section 5 is dealing with the RSA homomorphism. Our results of comparing the different variants and implementations are discussed in Section 6.

2 Notation and Background

2.1 GHI–Problem

Definition 2.1. Given two groups G, H, a mapping $\phi : G \rightarrow H$ is called a *homomorphism* if $\forall\ x, y \in G : \ \phi(xy) = \phi(x)\phi(y)$.

The following definitions are taken from [2].

Definition 2.2. Let G, H be Abelian groups and $S := \{(x_1, y_1), \ldots, (x_s, y_s)\} \subseteq G \times H$. We say that S *interpolates in a group homomorphism* if there exists a homomorphism $\phi : G \rightarrow H$ such that $\phi(x_i) = y_i$ for $i = 1, \ldots, s$. Given two sets $A, B \subseteq G \times H$, we say B *interpolates in a group homomorphism with* A if $A \cup B$ interpolates in a group homomorphism.

Definition 2.3. GHI – Problem $\big($Group Homomorphism Interpolation Problem$\big)$
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 $\big(x, y\big)$ interpolates with S in a group homomorphism.

The GHI-Problem is a generalization of many problems in cryptography, e.g. the discrete logarithm problem, the Diffie-Hellman problem or the RSA decryption problem. The security of the generic homomorphic signature scheme is based on the difficulty of solving the GHI-Problem.

2.2 Generic Homomorphic Undeniable Signature Scheme

Key Generation :
The signer chooses two Abelian groups G, H and a group homomorphism $\phi : G \rightarrow H$. To construct the public key he computes the order d of H and a set $K := \{(x_{key1}, \phi(x_{key1}), \ldots, (x_{keyk}, \phi(x_{keyk}))\} \subseteq G \times H$, where the x_i are generated from a seed ρ using a deterministic pseudorandom generator. More precisely, we need that K interpolates in unique group homomorphism with high probability. The private key consists of the homomorphism.

Signature and Protocols :
To sign a message m, the signer generates $(x_1, \ldots, x_s)$ from m, using a deterministic pseudorandom generator , m serving as a seed. The set $S := \{(x_{sig1}, \phi(x_{sig1}), \ldots, (x_{sigs}, \phi(x_{sigs}))\}$ constitutes the signature. The signature is valid if K interpolates with S in a group homomorphism. Proving this interactively forms the confirmation protocol. The denial protocol is composed of proving that S and K do not interpolate in a group homomorphism.

The values of s and k depend on the security level we want to ensure. For details regarding the confirmation and denial protocols, additional setup variants and proofs we refer to [1] and [2].

2.3 Quartic Residue Symbol

We saw that the signer can choose a homomorphism according to his needs. As a next step, we will have a closer look at some homomorphisms. We begin with the introduction of the quartic (biquadratic) residue symbol χ. This homomorphism is set in the ring of Gaussian Integers $\mathbb{Z}\big[i\big] = \{a+bi|a,b \in \mathbb{Z}\}$. Here is a sketch of the necessary facts of $\mathbb{Z}\big[i\big]$ (for more details and proofs see [4]):

- units: $\pm 1, \pm i$,
- norm: $\alpha \in \mathbb{Z}[i]$, $N(\alpha) = Re(\alpha)^2 + Im(\alpha)^2$
- $1+i$ is a prime, $N(1+i) = 2$
- $\alpha \in \mathbb{Z}[i]$ is called primary iff either
$$Re(\alpha) \equiv 1 \pmod 4,\ Im(\alpha) \equiv 0 \pmod 4$$
or
$$Re(\alpha) \equiv 3 \pmod 4,\ Im(\alpha) \equiv 2 \pmod 4$$
- if $\alpha \in \mathbb{Z}[i]$ is not divisible by $1+i$, then α is associated to a primary number
- $\forall\ \alpha \in \mathbb{Z}[i]$ there is a unique representation of the form $i^j \cdot (1+i)^k \cdot \alpha'$, with α' primary
- $\pi \in \mathbb{Z}[i]$ is prime iff either π or one of its associates fulfills one of the following conditions:
$$\pi = 1+i$$
$$\pi \text{ is a prime in } \mathbb{Z} \text{ and } \pi \equiv 3 \pmod 4$$
$$\pi\bar{\pi} \text{ is a prime in } \mathbb{Z} \text{ and } \pi\bar{\pi} \equiv 1 \pmod 4$$

Definition 2.4. Let $\alpha, \beta \in \mathbb{Z}[i]$ be such that $(1+i) \nmid \beta$ and $\gcd(\beta, \alpha) = 1$. The *quartic residue symbol* is defined as $\chi_\beta : \mathbb{Z}[i] \to \{0, \pm 1, \pm i\}$

$$\chi_\beta(\alpha) = \begin{cases} \big(\alpha^{\frac{N(\beta)-1}{4}}\big) \bmod \beta & \text{if } \beta \text{ prime} \\ \prod_i \ \chi_{\beta_i}(\alpha) & \text{if } \beta = \prod_i \beta_i,\ \beta_i \text{ prime} \end{cases}$$

In addition, the quartic residue symbol satisfies

- Modularity: $\chi_\beta(\alpha) = \chi_\beta(\alpha \bmod \beta)$
- Multiplicativity: $\chi_\beta(\alpha\alpha') = \chi_\beta(\alpha)\chi_\beta(\alpha')$
- Reciprocity Law: if $\alpha,\ \beta$ primary:
$$\chi_\beta(\alpha) = \chi_\alpha(\beta) \cdot (-1)^{\frac{N(\alpha)-1}{4}\cdot\frac{N(\beta)-1}{4}}$$
- Complementary Laws: if β primary:
$$\chi_\beta(i) = i^{\frac{N(\beta)-1}{4}} \qquad \text{and} \qquad \chi_\beta(1+i) = i^{\frac{Re(\beta)-Im(\beta)-Im(\beta)^2-1}{4}}$$

If we want to use the quartic residue symbol χ_β as a homomorphism for the undeniable signature scheme, we have different setup variants to choose from. Let p, q be two rational primes such that $p \equiv q \equiv 1 \pmod 4$. There exist π, σ such that $p = \pi\bar{\pi}$ and $q = \sigma\bar{\sigma}$. π and σ can be computed with the help of the algorithms by Tonelli and Cornacchia (For more details see [5]). If we select $G := \mathbb{Z}[i]/\beta\mathbb{Z}[i]$, $G \cong \mathbb{Z}_n^*$ when $\beta = \pi\sigma$ or $G \cong \mathbb{Z}_p^*$ when $\beta = \pi$, we can adapt the size of p and q to fulfill our security requirements.

2.4 Discrete Logarithm

Another homomorphism suitable for the generic homomorphic signature scheme is based on the discrete logarithm.
Let n be such that $n = pq$ with $p = rd + 1$, q, d prime, $\gcd(q-1, d) = 1$, $\gcd(r, d) = 1$ and g generating a subgroup of $\mathbb{Z}_p^*$. We obtain g by choosing a random element $h \in \mathbb{Z}_n^*$ until h satisfies $h^r \bmod p \neq 1$ and we set $g = h^r \bmod p$. Like this we find a homomorphism suitable for the generic homomorphic signature scheme by computing a discrete logarithm in a small subgroup of $\mathbb{Z}_n^*$:
$$\phi : \mathbb{Z}_n^* \to \mathbb{Z}_d \quad \phi(x) = \log_g(x^r \bmod\ p)$$

3 Quartic Residue Symbol

3.1 Basic Algorithm

3.1.1 Description

To compute the quartic residue symbol directly, one has to know the factorization of β into primes over $\mathbb{Z}[i]$ and the computation contains an exponentiation. To avoid this factorization as well as the exponentiation we apply the properties of the quartic residue symbol iteratively. First we find the unique representation $i^j \cdot (1+i)^k \cdot \alpha'$, α' primary, of α and employ the multiplicativity property and the complementary laws of the quartic. Next, we use the modularity property and interchange α and β according to the law of reciprocity and start again. We stop the iteration process when α or β is a unit.

Algorithm 1 Basic Algorithm Quartic Residuosity in $\mathbb{Z}[i]$

```
Require: $\alpha, \beta \in \mathbb{Z}[i] \setminus \{0\}$, $\gcd(\alpha, \beta) = 1$ and $(1+i) \nmid \beta$
Ensure: $c = \chi_\beta(\alpha)$  ($c = 0 \Leftrightarrow \chi_\beta(\alpha)$ is not defined)
 1: $\alpha \leftarrow \alpha \bmod \beta$
 2: if $\alpha = 0$ then $c = 0$ end if
 3: let primary $\alpha_1, \beta_1 \in \mathbb{Z}[i]$ be defined by
    $\alpha = (i)^{i_1} \cdot (1+i)^{j_1} \cdot \alpha_1$ and
    $\beta = (i)^{i_2} \cdot \beta_1$
 4: let $m, n \in \mathbb{Z}$ be defined by $\beta_1 = m + ni$
 5: $t \leftarrow \frac{m-n-n^2-1}{4} j_1 + \frac{m^2+n^2-1}{4} i_1 \bmod 4$
 6: replace $\alpha$ with $\beta_1$, $\beta$ with $\alpha_1$
 7: $t \leftarrow t + \frac{(N(\alpha)-1)(N(\beta)-1)}{8} \bmod 4$
 8: while $N(\alpha) > 1$ do
 9:     (LOOP INVARIANT: $\alpha, \beta$ are primary)
10:     let primary $\alpha_1$ be defined by $\alpha \bmod \beta = (i)^{i_1} \cdot (1+i)^{j_1} \cdot \alpha_1$
11:     let $m, n \in \mathbb{Z}$ be defined by $\beta = m + ni$
12:     $t \leftarrow t + \frac{m-n-n^2-1}{4} j_1 + \frac{m^2+n^2-1}{4} i_1 \bmod 4$
13:     replace $\alpha$ with $\beta$, $\beta$ with $\alpha_1$
14:     $t \leftarrow t + \frac{(N(\alpha)-1)(N(\beta)-1)}{8} \bmod 4$
15: end while
16: if $N(\alpha) \neq 1$ then $c \leftarrow 0$ else $c \leftarrow i^t$ end if
```

3.1.2 Implementation

For this algorithm we have to implement a few functions for calculating basic operations in the ring of Gaussian Integers as they are not provided by gmp (let $\alpha, \beta \in \mathbb{Z}[i]$):

1. Multiplication: $\alpha \cdot \beta$
2. Modulo: $\alpha \bmod \beta$
3. Norm: $N(\alpha)$
4. Division by $(1+i)^r$
5. Primarization: transforms α into its primary associate if possible

Some of these functions existed already in the implementation of the demonstrator, but there was a considerable amount of speed to be gained.

First of all, we scrutinized every line of the existing code closely and checked if there is a faster algorithm we could apply. Our next steps were to remove unnecessary function calls, use some of the more sophisticated gmp functions, reduce the number of `mpz_t` variables used whenever possible, examine different implementation variants and apply general optimization techniques described e.g. in [9],[10]. We tested carefully where it is better to work with `int` instead of `mpz_t`. In addition, we used profiling and tried out different compiler optimization levels.

In two cases we reimplemented the functions completely:

The division of α by $(1+i)^r$ was done by first raising $(1+i)$ to the power of r and then dividing α by the result. We found a way of achieving the same by only using shift operations, additions and interchanging the imaginary and real part if necessary. The following equations demonstrate our procedure:

$$\frac{\alpha}{(1+i)} = \frac{Re(\alpha)+Im(\alpha)}{2} + \frac{Im(\alpha)-Re(\alpha)}{2} i$$

$$\frac{\alpha}{(1+i)^r} = \frac{i^{3k}\left(\frac{Re(\alpha)}{2^k} + \frac{Im(\alpha)}{2^k} i\right)}{(1+i)^b} \quad , \quad r = 2k + b$$

If $r = 2k,\ k \in \mathbb{N}$ we shift the real and the imaginary parts of α by k to the right and multiply them by -1 and/or interchange them depending on the value of $3k$. If r is odd, there is an additional subtraction and addition to perform.

For the primarization function we discovered a much simpler and faster method as well. It consists of a few congruency tests and it also determines the number of times we have to multiply α by i to get the primary associate of α.

The calculation of $\alpha \bmod \beta$ is done according to [6] using an Euclidean division and rounding appropriately.

To find the representation of α we proceed as follows: First calculate the norm of α, $N(\alpha)$. Then find j maximal such that $2^j \mid N(\alpha)$. Divide α by $(1+i)^j$ and transform the result into its primary associate.

In the implementation of the algorithm we need to ensure that $(1+i) \nmid \beta$ and $\gcd(\alpha, \beta) = 1$. The first requirement is taken care of by applying the primarization function on β. If we cannot find a primary associate, β is divisible by $(1+i)$ and we terminate. For the second condition we check in every iteration if $\alpha \bmod \beta = 0$. This would imply $\gcd(\alpha, \beta) \neq 1$ and we terminate.

3.2 Damgård's Algorithm

3.2.1 Description

The most expensive operation used in the algorithm described above is $\alpha \bmod \beta$. Damgård and Frandsen present in [11] an efficient algorithm for computing the cubic residue symbol in the ring of Eisenstein integers $\mathbb{Z}[\zeta]$. Their algorithm can be transformed into an algorithm for the quartic residue symbol in the ring of Gaussian integers.

Algorithm 2 Damgård's Algorithm Quartic Residuosity in $\mathbb{Z}[i]$

```
Require: $\alpha, \beta \in \mathbb{Z}[i] \setminus \{0\}$, $\gcd(\alpha, \beta) = 1$ and $(1+i) \nmid \beta$
Ensure: $c = \chi_\beta(\alpha)$  ($c = 0 \Leftrightarrow \chi_\beta(\alpha)$ is not defined)
 1: let primary $\alpha_1, \beta_1 \in \mathbb{Z}[i]$ be defined by
    $\alpha = (i)^{i_1} \cdot (1+i)^{j_1} \cdot \alpha_1$ and
    $\beta = (i)^{i_2} \cdot \beta_1$
 2: let $m, n \in \mathbb{Z}$ be defined by $\beta_1 = m + ni$
 3: $t \leftarrow \frac{m-n-n^2-1}{4} j_1 + \frac{m^2+n^2-1}{4} i_1 \bmod 4$
 4: replace $\alpha$ with $\alpha_1$, $\beta$ with $\beta_1$
 5: if $\tilde{N}(\alpha) < \tilde{N}(\beta)$ then
 6:    interchange $\alpha$ and $\beta$ and adjust $t$
       $t \leftarrow t + \frac{(N(\alpha)-1)(N(\beta)-1)}{8} \bmod 4$
 7: end if
 8: while $\alpha \neq \beta$ do
 9:    (LOOP INVARIANT: $\alpha, \beta$ are primary)
10:    let primary $\alpha_1$ be defined by $\alpha - \beta = (i)^{i_1} \cdot (1+i)^{j_1} \cdot \alpha_1$
11:    let $m, n \in \mathbb{Z}$ be defined by $\beta = m + ni$
12:    $t \leftarrow t + \frac{m-n-n^2-1}{4} j_1 + \frac{m^2+n^2-1}{4} i_1 \bmod 4$
13:    replace $\alpha$ with $\alpha_1$
14:    if $\tilde{N}(\alpha) < \tilde{N}(\beta)$ then
15:       interchange $\alpha$ and $\beta$ and adjust $t$
          $t \leftarrow t + \frac{(N(\alpha)-1)(N(\beta)-1)}{8} \bmod 4$
16:    end if
17: end while
18: if $\alpha \neq 1$ then $c \leftarrow 0$ else $c \leftarrow i^t$ end if
```

There are three main differences to the basic algorithm: Instead of using $\alpha \bmod \beta$ to reduce α, they suggest using $\alpha - \beta$. This takes much less time but increases the number of iterations needed. Furthermore they only interchange α and β, if $N(\alpha) < N(\beta)$. It is not necessary to calculate $N(\cdot)$ exactly for this purpose, an approximation $\tilde{N}(\cdot)$ suffices. They demonstrate how one can compute an approximate norm $\tilde{N}(\alpha)$ in linear time: Instead of adding up the squares of the real and the imaginary part of α, one replaces all but the 8 most significant bits of the real and the imaginary part of α with zeroes and computes the norm of the resulting Gaussian number.

Their algorithm takes $O(\log^2 N(\alpha\beta))$ time to compute $\chi_\beta(\alpha)$

3.2.2 Implementation

The structure of the algorithm is similar to the basic algorithm so we were able to reuse parts of our previous work.
We implemented both the standard norm and the norm Damgård and Frandsen suggest. The standard norm consists of only two gmp functions: one multiplication and one combined addition/multiplication whereas the approximate norm involves one bit scan to determine the size of the real part, one shift operation to extract the 8 most significant bits, one multiplication for the squaring of these 8 bits and another shift operation to put the result back to its correct position. We apply the same procedure on the imaginary part and we add the two approximate squarings up. In short, we need four additional operations to reduce the size of the numbers we have to multiply. As gmp is a highly optimized library, computing the standard norm takes little time and the additional operations of the approximate norm only amortize if the real and the imaginary part are larger than 2048 bits. This and the fact that the norm of α and β decreases with each iteration convinced us to use the standard norm instead.

3.3 Other Algorithms

In addition to the above, we studied papers concerning algorithms for the quartic residue symbol by Weilert. In [12],[13] he presents fast gcd algorithms for Gaussian integers. Based on these gcd algorithms and using some properties of the Hilbert symbol he demonstrates in [14] how to construct an algorithm for the quartic residue symbol. This algorithm involves calculating an Euclidean descent and storing some intermediate results for later use.
Another algorithm we considered adapting to the computation of the quartic residue symbol, is the k-ary right/left shift algorithm for the Jacobi symbol presented by Meyer and Sorendson in [15]. Their algorithm involves performing an extended gcd computation in each iteration.
When we timed our implementations of the quartic residue symbol we found that they are much slower than e.g. the computation of the Jacobi symbol for input of the same size. This is mostly due to the fact that basic functions in $\mathbb{Z}[i]$ are more time consuming than their equivalent in $\mathbb{Z}$. For example multiplication in $\mathbb{Z}[i]$ consists of four multiplications in $\mathbb{Z}$. Moreover, they are not optimized on assembly level, unlike gmp functions.
Therefore we decided not to implement neither Weilert's nor Meyer's and Sorendson's algorithms. Even if their asymptotic running time is very fast, we probably would not be able to achieve a significant speed up of the computation of the quartic residue symbol.

4 Discrete Logarithm

As mentioned in Section 2.3, a suitable homomorphism for the generic homomorphic signature scheme is the following:

$$\phi : \mathbb{Z}_n^* \to \mathbb{Z}_d \quad \phi(x) = \log_g(x^r \bmod p)$$

There already exists a function in gmp for exponentiation modulo a prime, `mpz_powm`. Our task remained to provide a fast method for computing the discrete logarithm. We implemented three algorithms of different speed and different requirements for storage. Depending on the platform where the signature scheme is used, one of them will be preferable to the others.
The algorithms described are adaptations of the algorithms in [17].

4.1 Precomputed Table

4.1.1 Description

The simplest and fastest way of computing the discrete logarithm uses a table with precomputed entries. Given p prime, g a generator of a cyclic group G, subgroup of $\mathbb{Z}_p^*$, and $d = |G|$, we construct a table with entries (g^j, j) for $0 \leq j \leq d$. Building this table is a very time and memory consuming task, but once the table exists, finding the discrete logarithm consists of a simple look up operation.
There are several ways of constructing such a table. One can use a two dimensional array and sorting it by the first component. Finding the discrete logarithm is then reduced to a binary search. Alternatively, one can use conventional hashing on the first component to store the entries in a hash table, in which case placing an entry and searching for an entry in the table takes constant time. Another advantage is the fact, that we do not need space for g^i. Especially when $p \gg d$, this can save an enormous amount of memory. The only difficulties are finding a suitable hash function and dealing with collisions without losing too much time.
Time complexity of the construction of the table is $O(d)$ multiplications (plus $O(d \log d)$ comparisons to sort). Space complexity is $O(d(\log d + \log p))$ for the sorted table, resp $O(d \log d)$ for the hash table. The running time for the sorted table is $O(\log d)$, for the hash table O(1).

4.1.2 Implementation

In this suggested variant of the generic homomorphic signature scheme, p is typically a 512 bit and d a 20 bit prime. Creating a table with d entries of size 532 bits is impossible on a usual desktop computer. Therefore we decided to use a hash table (key 512 bits, data 20 bits, 2^{20} entries). We found some existing hash table data structures written in C, but they do

not fulfill our requirements. They are either too slow, support C types only, do not allow tables that large and/or they store the key as well.
To avoid problems, we did not adapt any of the existing data structures, but implemented a hash table ourselves providing enough storage and a collision handling mechanism suitable for our needs. Our solution is a hash table consisting of an array of unsigned integers. This array is of maximal length (2^{24}) to reduce collisions.
An unsigned integer is 32 bits long, so it was possible to store the data for the logarithm as well as using one of the higher order bits as a flag for collisions. Because the key is large and we wanted to avoid any unnecessary computation, we chose to use the 24 least significant bits of the key as the index into the hash table, in case of collision the next 24 bits, etc. By selecting 24 bits instead of the possible 20 bits, we minimize the occurrence of collisions. Tests have shown that most collisions are resolved by choosing the next 24 bits. We tried out other hash functions, but we did not achieve a gain of speed. This way, the size of the table is 64 MB.
To find the correct discrete logarithm for $y \in G$, one has to check if the collision flag at the corresponding array field is set, to decide if one can return the logarithm stored in the field or if one has to continue with the next field.
If we want to adapt the developed data structure to d being a 30 bit prime, we meet quite a hard problem, as our environment does not allow us to allocate an integer array containing more than 2^{24} elements. For any d smaller than 2^{24} we can reduce the size of the hash table easily by adjusting the putEntry, getEntry and hash function dynamically.

4.2 Baby Step Giant Step Algorithm

4.2.1 Description

One can write $y = g^x = g^{im+j}$, where $m = \lceil \sqrt{d} \rceil,\ 0 \leq i, j < m$, which implies $y(g^{-m})^i = g^j$. This suggests the following algorithm (BSGS) for computing the discrete logarithm x of y.
Instead of storing all powers of g in a table, storing the results of $O(\sqrt{d})$ multiplications by g (baby steps) followed by a maximum $O(\sqrt{d})$ of table look ups and multiplications by γ (giant steps) suffice (see Algorithm 3). We need less time and memory for constructing the table, but more time for computing the discrete logarithm. This algorithm requires storage for $O(\sqrt{d})$ group elements. The asymptotic running time of the the baby step giant step algorithm is $O(\sqrt{d})$.

4.2.2 Implementation

While implementing the BSGS algorithm no problems occurred and we were even able to use the hash table of the previous implementation. As we place

Algorithm 3 Baby Step Giant Step Discrete Logarithm

```
Require: $p$ prime, $g$ generator of cyclic group $G$, subgroup of $\mathbb{Z}_p^*$,
    $d = |G|,\ y \in G$
Ensure: $x = \log_g(y)$
 1: set $m \leftarrow \lceil \sqrt{d} \rceil$
 2: construct a hash table with entries $(g^j, j)$ for $0 \leq j \leq m$
 3: compute $g^{-m}$, set $\gamma \leftarrow y$
 4: for $i$ from 0 to $m - 1$ do
 5:     check if there is an entry $j$ for $\gamma$
 6:     if $\gamma = g^j$ then
 7:         set $x \leftarrow im + j$
 8:         return x
 9:     end if
10:     set $\gamma \leftarrow \gamma \cdot g^{-m}$
11: end for
```

less entries in the table, collisions hardly ever occur. In all the tests we performed, there was never a collision.

4.3 Pollard's Rho Algorithm

4.3.1 Overview

In environments with restricted memory we need an algorithm which requires an negligible amount of memory. Pollard's rho algorithm has this property and its asymptotic running time is the same as for the baby step giant step algorithm. For this reason it is far more preferable to the baby step giant step algorithm for many problems of practical interest.
The group G is partitioned into three sets S_0, S_1, S_2 of roughly equal size, based on some easily testable property. Some care must be exercised in selecting the partition; for example, $1 \notin S_1$.
We define a sequence of group elements $x_0, x_1, x_2, \ldots$ and integers $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ satisfying $x_i = g^{a_i} y^{b_i}$ for $i \geq 0$ by $x_0 = 1,\ a_0 = 0,\ b_0 = 0$, and for $i \geq 0$

$$(x_{i+1}, a_{i+1}, b_{i+1}) = \begin{cases} (yx_i \bmod p, & a_i, & b_i + 1 \bmod d) & \text{if } x_i \in S_0 \\ (\ x_i^2 \bmod p, & 2a_i \bmod d, & 2b_i + 1 \bmod d) & \text{if } x_i \in S_1 \\ (gx_i \bmod p, & a_i \bmod d, & b_i \bmod d) & \text{if } x_i \in S_2 \end{cases}$$

In every iteration of the algorithm we compute x_i and x_{2i} using the previously computed values until $x_i = x_{2i}$. Hence $g^{a_i} y^{b_i} = g^{a_{2i}} y^{b_{2i}}$ and so $y^{b_i - b_{2i}} = g^{a_{2i} - a_i}$. Taking logarithms to the base g of both sides of the last equation yields

$$(b_i - b_{2i}) \cdot \log_g(y) \equiv (a_{2i} - a_i) \pmod d$$

Algorithm 4 Pollard's Rho Discrete Logarithm

```
Require: $p$ prime, $g$ generator of cyclic group $G$, subgroup of $\mathbb{Z}_p^*$,
    $d = |G|, y \in G$
Ensure: $x = \log_g(y)$
 1: set $x_0 \leftarrow 1,\ a_0 \leftarrow 0,\ b_0 \leftarrow 0$
 2: for $i = 1, 2, \ldots$ do
 3:   compute $x_i, a_i, b_i$ using $x_{i-1}, a_{i-1}, b_{i-1}, x_{2i-2}, a_{2i-2}, b_{2i-2}$
```

$$(x_{i+1}, a_{i+1}, b_{i+1}) = \begin{cases} (yx_i \bmod p, & a_i, & b_i + 1 \bmod d) & \text{if } x_i \in S_0 \\ (\ x_i^2 \bmod p, & 2a_i \bmod d, & 2b_i + 1 \bmod d) & \text{if } x_i \in S_1 \\ (gx_i \bmod p, & a_i \bmod d, & b_i \bmod d) & \text{if } x_i \in S_2 \end{cases}$$

```
 4:   if $x_i = x_2$ then
 5:     set $r \leftarrow b_i - b_{2i} \mod d$
 6:     if $r \neq 0$ then
 7:       set $x \leftarrow r^{-1}(a_{2i} - a_i) \bmod d$
 8:     else
 9:       terminate algorithm with failure
10:     end if
11:   end if
12: end for
```

Provided $b_i \not\equiv b_{2i} \pmod{d}$, this equation can be efficiently solved to determine $\log_g(y)$. In the rare case that the algorithm terminates with failure ($b_i \equiv b_{2i} \pmod{d}$ occurs with negligible probability), the procedure can be repeated by selecting a_0, b_0 randomly in the interval $[1, d-1]$ and starting with $x_0 = g^{a_0} y^{b_0}$.

4.3.2 Implementation

This algorithm was straightforward to implement. For the partition we examined two variants:

$$S_i = \{x \in G | x - 1 \bmod 3 = i\} \text{ and } S_i = \{\lceil (ip)/3 \rceil, \ldots, \lceil (2ip)/3 \rceil\}$$

We did not find any significant differences comparing the running times. In addition to this, we measured the running time when selecting random numbers a_0, b_0 in the interval $[1, d-1]$ and starting with $x_0 = g^{a_0} y^{b_0}$. It proved to be slower than choosing $x_0 = 1,\ a_0 = 0,\ b_0 = 0$. Termination with failure occurred hardly ever in any of the cases and did not deteriorate the average running time.

5 RSA

5.1 Description

In [16] R. Gennaro, T. Rabin and H. Krawczyk propose an undeniable signature scheme based on RSA. As mentioned when introducing the GHI problem, the RSA problem is an example of the GHI problem. Therefore we can use the RSA exponentiation as a homomorphism for the generic homomorphic signature scheme.
By implementing this scheme we were able to compare the implementations of the homomorphic undeniable signature scheme based on the quartic residue symbol and the discrete logarithm with a well known signature scheme, which in turn is also an example of a homomorphic signature scheme. We followed the description of the algorithm for the RSA homomorphism in [17].

5.2 Implementation

We implemented the RSA exponentiation using gmp functions `mpz_nextprime, mpz_invertmod, mpz_powm` without any optimizations.

6 Results

In this section we present the results of the timing measurements we conducted to determine how well the different algorithms perform. We implemented the algorithms using Visual Studio 2003 and we tested and timed them under Linux and Windows.
In order to measure the running time precisely under both operating systems, we used functionalities offered by `frequence_cpu.h` by Victor Stinner [8]
Our Windows platform is an Intel(R)4 1.4 GHz Desktop Computer with 256 MB RAM running Windows XP. We will only present the results obtained under Windows here.
Unless otherwise stated all our results are average values produced by test series of 1000 tests, each using 512 bit random numbers generated with the gmp function `mpz_urandomb`. When dealing with Gaussian integers, both the real and the imaginary part are 512 bit random numbers.

6.1 Homomorphisms

6.1.1 Quartic Residue Symbol

We start with comparing the improved subfunctions needed for the computation of the quartic residue symbol. Optimizing and/or reimplementing them cut the running time to a fraction of their original running time. We

managed to reduce the running time for the calculation of $\alpha \cdot \beta$ and $\alpha \bmod \beta$, $\alpha, \beta \in \mathbb{Z}[i]$, by more than a third.
The functions for the division by $(1+i)^r$ and the primarization were also changed algorithmically. This lead to a remarkable speed up, especially in the latter case, which is now more than ten times faster.

Subfunctions	
Multiplication in $\mathbb{Z}[i]$	time in ms
Not optimized	0.078
Optimized	0.049
gmp `mpz_mul`	0.010
Modulo in $\mathbb{Z}[i]$	time in ms
Not optimized	0.141
Optimized	0.104
gmp `mpz_mod`	0.001
Division by $(1+i)^r$	time in ms
Not optimized	0.061
Optimized	0.015
gmp `mpz_tdiv_q_2exp`	0.001
Primarization	time in ms
Not optimized	0.071
Optimized	0.006

Table 1: Results Subfunctions

As a reference we timed the equivalent gmp functions. Take into account that they operate in $\mathbb{Z}$ not $\mathbb{Z}[i]$ when regarding Table 1.
We tested our implementations with input of different size as well and they showed the expected behaviour.

For α, β being random numbers of the same size we stated the following behaviour: The version of the basic algorithm using the improved basic functions is twice as fast as it was before. The implementation of Damgård's algorithm is on average about a third faster than the optimized version of the basic algorithm. See Table 2.
The average number of iterations performed by the basic algorithm is 249.30, whereas Damgård's algorithm needs 512.84 iterations. These numbers grow linearly with the size of α and β, if we change both, α and β. However, by changing only the size of either α or β we find the following dependencies: Concerning the basic algorithm the number of iterations is linearly related

Quartic Residue Symbol	
Running time	**time in ms**
Basic algorithm	62.79
Basic algorithm optimized	31.57
Damgård's algorithm	24.22
Iterations	
Basic algorithm	249.27
Damgård's algorithm	512.84

Table 2: Results Quartic Residue Symbol

with the smaller of the two arguments. In the case of Damgård's algorithm the larger argument is mainly responsible for the number of iterations. This is natural, as the reduction of α by modulo β guarantees a number with a norm smaller or equal to $N(\beta)$, which is not the case for a reduction by subtraction.
In the MOVA signature scheme, the real and the imaginary parts of α and β are typically 1024, 512 bits long respectively. Unfortunately, this implies that Damgård's algorithm is less suitable for the MOVA signature scheme. However, if we construct a mixed algorithm by first computing $\alpha' = \alpha \bmod \beta$ and then calculating $\chi_\beta(\alpha')$ using Damgård's algorithm, we obtain the same running time Damgård's algorithm achieves when α and β are of the same size. The fact that the MOVA signature scheme demands β to be of the form $\beta = \pi\sigma$, with π, σ prime numbers in $\mathbb{Z}[i]$ has no influence on the running time. An illustration of the above can be found in Table 3 (100 tests per Setup variant).

As a reference problem we compared the running time of `mpz_jacobi`, the gmp function for calculating the Jacobi symbol $(\frac{a}{b})$, with our own implementation of the ordinary algorithm for computing the Jacobi symbol (gmp uses the binary algorithm). The Jacobi symbol is the equivalent of the quartic residue symbol in $\mathbb{Z}$ and the structure of the ordinary algorithm is the same as the structure of the basic algorithm for the quartic residue symbol. It also uses modularity to reduce the size of the arguments and applies iteratively multiplicativity and a reciprocity law very similar to the one of the quartic residue symbol. For our measurements we chose a, b to be random integers of 1024, 512 bits respectively.
Our implementation is about ten times slower than gmp's but roughly twenty times faster than the quartic residue symbol. The number of iterations performed is 187.71 on average, about three quarters of the respective number for the quartic residue symbol. See Table 4.

Parameter Choices		
α: 1024 bits, random number, β: 512 bits, random number		
	time in ms	iterations
Basic algorithm	32.77	249.66
Damgård's algorithm	51.18	76.18
Mixed algorithm	24.86	510.19
α: 1024 bits, random number, $\beta = \pi\sigma$: 512 bits		
	time in ms	iterations
Basic algorithm	32.12	248.81
Damgård's algorithm	50.63	766.12
Mixed algorithm	24.65	511.92
α: 1024 bits, random number, $\beta = \pi$: 256 bits		
	time in ms	iterations
Basic algorithm	14.31	123.87
Damgård's algorithm	38.59	640.71
Mixed algorithm	9.03	255.95

Table 3: Results Parameter Choices

The fact that the arguments for the computation of the quartic residue symbol are in $\mathbb{Z}[i]$, entails that we cannot use the available and optimized gmp functions for integers. This prevents our implementation from being competitive with implementations of functions in $\mathbb{Z}$.

6.1.2 Discrete Logarithm

The homomorphism based on the discrete logarithm consists of one exponentiation modulo a large prime and computing the discrete logarithm in a small subgroup. The exponentiation is a very time consuming operation, so

Jacobi Symbol	
Running time	time in ms
Jacobi	1.261
`mpz_jacobi`	0.116
Iterations	
Jacobi main loop	187.71

Table 4: Results Jacobi Symbol

the running time of this homomorphism is dominated by the running time of the gmp function `mpz_powm`.

Discrete Logarithm	
Construction	**time in s**
Logarithm table construction	16.616
BSGS table construction	6.023
Running time	**time in ms**
Logarithm table	9.66
BSGS	19.47
Pollard's rho	74.92
(`mpz_powm`	9.44)
Iterations	
Logarithm table look up	1.04
BSGS main loop	388.36
Pollard's rho main loop	1037.49

Table 5: Results Discrete Logarithm

As expected the version using a precomputed table very fast, once the table is constructed. It is only marginally slower than `mpz_powm`. The construction of the hash table takes rather much time which is also due to the collision handling procedure. There are about 14 000 collisions when constructing a table for p, q, d being 512 bits respectively 20 bits long primes.
While filling the BSGS table no collisions occurred during our tests. Computing the homomorphism with the BSGS implementation took about twice as long as with the precomputed table.
Pollard's rho algorithm, which has the same asymptotic running time as BSGS but requires almost no storage, is about 4 times slower than BSGS. This is due to the fact that there are more iterations of the main loop and we need to perform more steps per iteration.
In addition, we conducted tests to determine the average number of table look ups for the first algorithm and the numbers of main loop iterations for the BSGS and Pollard's rho algorithm. For details see Table 5.

6.1.3 RSA

With p, q of size 512 bits, our implementation of RSA exponentiation takes 33.87 ms.

RSA	
	time in ms
Running time	33.87

Table 6: Results RSA

6.2 Signature Generation

So far we only considered the time the preprocessing and the computation of one homomorphism takes. In this section we present our results from comparing the time to compute a complete signature.
In the generic homomorphic signature scheme a signature consists of the set $S := \{(x_{sig1}, \phi(x_{sig1}), \ldots, (x_{sigs}, \phi(x_{sigs}))\}$. The size $s = |S|$ depends on the homomorphism ϕ we choose and the level of security we want to offer. Typical values would be $s = 20$ in the case of the quartic residue symbol with $\beta = \pi\sigma$ $(\beta = \pi)$, $\alpha \in \mathbb{Z}_n^*$, n 1024 bits long and the Jacobi symbol $(\frac{a}{p})$ with $n = pq$, p, q prime and $a \in \mathbb{Z}_n^*$, or $s = 1$ when using the homomorphism based on the discrete logarithm with d 20 bits, n 1024 bits long or RSA with n 1024 bits long. For this comparison, we only use the mixed algorithm to calculate the quartic residue symbol and we do not take into consideration the time it takes to generate the x_i from the message m.

Signature Generation Time	
	time in ms
Quartic Residue Symbol ($\beta = \pi\sigma$)	493.01
Quartic Residue Symbol ($\beta = \pi$)	180.64
Jacobi Symbol (ordinary algorithm)	25.22
Jacobi Symbol (`mpz_jacobi`)	2.32
Discrete Logarithm (Precomputed Table)	9.66
Discrete Logarithm (BSGS)	19.47
Discrete Logarithm (Pollard's rho)	74.93
RSA	33.87

Table 7: Results Comparison Signature Schemes

When examining Table 7, we observe that the variant using the quartic residue symbol is much slower than any of the implementations of the homomorphism based on the discrete logarithm or RSA. The reason for this result is mainly the fact that the computation of the quartic residue symbol is in $\mathbb{Z}[i]$, which is discussed in detail above.
The running time of the variant using the homomorphism based on the discrete logarithm is in the same order as RSA, the implementation using the precomputed table and BSGS are even considerably faster than RSA. The implementation with the precomputed table takes less than 1/3, BSGS 2/3 of the time needed by RSA. Pollard's rho algorithm is slower than RSA,

but it has the advantage over the other two of neither requiring a large amount of storage nor time for constructing a data structure.
By far the fastest running time has the version using the gmp function `mpz_jacobi` for the computation of the Jacobi symbol. Even though we have to calculate 20 Jacobi symbols, it takes less than 1/4 of the time the fastest implementation of the homomorphism based on the discrete logarithm needs. When using our own implementation, the running time is comparable to the running time of BSGS. Furthermore, neither of the two functions require a large amount of storage or preprocessing.

7 Conclusion

In the course of the project we achieved to gain insight in the field of the GHI–Problem and its applications, especially undeniable signatures. We studied some of the group homomorphisms suitable for the generic homomorphic undeniable signature scheme proposed in [2] and implemented and/or improved them.
We successfully decreased the running time of the computation of the quartic residue symbol by trying out different algorithms and applying general optimization techniques. The calculation takes now less than half of the time it used to. We compared the results with the running time of the computation of the Jacobi symbol and we state that due to the fact that the quartic residue symbol is in $\mathbb{Z}[i]$, we will not be able to achieve a similar running time.
Furthermore we implemented the variant of the generic homomorphic undeniable signature scheme based on the discrete logarithm and compared it with the quartic residue symbol and RSA. We found that our version using a table with precomputed values for the discrete logarithm is much faster than the others including RSA. We verified that BSGS and Pollard's rho algorithm also perform very well with the advantage of requiring less storage and preparation time.
During the implementation process of the homomorphism based on the discrete logarithm we developed a reusable hash table data structure capable of containing a large number of entries without any key overhead. Moreover, it is possible to have arbitrarily long keys and collision information is stored together with the data.
As a future improvement concerning the Demonstrator for the MOVA signature scheme one could apply multi-exponentiation optimization methods to accelerate the confirmation and denial protocol for the verifier.

8 Acknowledgments

We wish to thank I. Suarez for the helpful discussions concerning the algorithms for the quartic residue symbol and J. Monnerat and S. Vaudenay for the support during this project.

References

[1] J. Monnerat and S. Vaudenay, *Undeniable Signatures Based on Characters: How to Sign with One Bit*, PKC '04, LNCS 2947, pp.69–85, Springer, 2004.

[2] J. Monnerat and S. Vaudenay, *Generic Homomorphic Undeniable Signatures*, Asiacrypt 2004, LNCS 3329, pp. 354 – 371, Springer, 2004.

[3] D.Chaum and H. van Antwerpen, *Undeniable Signatures*, Advances in Cryptology - Crypto '89, LNCS, 435, pp. 212 – 217, Springer, 1989.

[4] K. Ireland and M. Rosen, *A Classical Introduction to modern Number Theory: Second Edition*, Graduate Texts in Mathematics 84, Springer, 1990.

[5] H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Mathematics 138, Springer, 2000.

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

[7] The GNU multiple precision arithmetic library, http://www.swox.com/gmp/

[8] V. Stinner, `frequence_cpu.h, frequence_cpu.c`, http://www.haypocalc.com/, 2003.

[9] M.E. Lee, *Optimization of Computer Programs in C*, http://vision.eng.shu.ac.uk/bala/c/c/optimisation/l/optimization.html, 2001.

[10] S. Garg, *How to optimize C/C++ Source - Performance Programming*, http://bdn.borland.com/article/0,1410,28278,00.html, 2002.

[11] I.B. Damgård and G.S. Frandsen *Efficient Algorithms for GCD and Cubic Residuosity in the Ring of Eisenstein Integers*, FCT 2003, LNCS 2751, pp. 109–117, Springer, 2003.

[12] A.Weilert, *(1+i)-ary GCD Computation in Z[i] is an ue to the Binary GCD Algorithm*, J. Symbolic Comput. 30(5), pp. 605–617, 2000.

[13] A.Weilert, *Asymptotically fast GCD Computation in Z[i]*, Algorithmic Number Theory, LNCS 1838, pp.595–613, Springer, 2000.

[14] A. Weilert, *Fast Computation of the Biquadratic Residue Symbol*, Journal of Number Theory 96, pp. 133–151, 2002.

[15] S.M. Meyer and J.P. Sorendson *Efficient Algortihms for Computing the Jacobi Symbol* J. Symbolic Comput. 26(4), pp. 509–523, 1998.

[16] R. Gennaro, T. Rabin, H. Krawczyk, *RSA-Based Undeniable Signatures*, Journal of Cryptology, 13, pp. 397 – 416, Springer, 2000.

[17] A.Menezes, P. van Oorschot, and S Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

A User Manuals

A.1 Tester

This program allows you to execute the test series described above and many more. You can chose what sort of tests and how many you want to perform as well as how large the random numbers used shall be. It works under Linux and Windows.

A.1.1 Windows

1. To start the program double click on ”tester.exe”. A console window opens and asks you to enter the filename where the results of your tests are written to.
2. Type ”filename.txt”, where ”filename” is any chain of chars allowed in filenames and press ENTER.
3. Enter the number of tests you want to execute and the size of the random numbers to be generated in bits. The default values are 100 tests, 512 bits.
4. Choose between the test options by entering the respective value. Depending on the test you have more options to choose from or the test series start. If you want more information concerning the options, enter ”999” for help. There is no output on the console during the test, the results will be written to the file chosen previously.
5. After running the tests, you can select if you would like to continue testing or if you want to end the program.
6. To look at your test results open ”filename.txt” with you favourite text editor.

A.1.2 Linux

1. Open a terminal, go to the directory ”tester”.
2. Compile the program by typing ”make”.
3. Type ”./run” to start the program.
4. Continue at the description for Windows at 2.

A.2 Demonstrator for the MOVA Signature Scheme

M. Hammoutène has developed a software which implements the MOVA signature scheme based on the quartic residue symbol. We replaced its computation of the quartic residue symbol by our optimized version. With the help of this program you can generate public and private keys of your desired length as well as construct signatures for any file and execute the confirmation and denial protocol. To run it, simply double click on "crypt1.exe" and follow the instructions. When generating a public and private key pair, observe that secret key files have the ending ".smova" , public key files ".pmova". This software is currently only available under Windows.