Secure Distributed Computation of the Square Root and Applications

Manuel Liedel

Fakultät für Wirtschaftswissenschaften, University of Regensburg,
Manuel.Liedel@wiwi.uni-regensburg.de

Abstract. The square root is an important mathematical primitive whose secure, efficient, distributed computation has so far not been possible. We present a solution to this problem based on Goldschmidt's algorithm. The starting point is computed by linear approximation of the normalized input using carefully chosen coefficients. The whole algorithm is presented in the fixed-point arithmetic framework of Catrina/Saxena for secure computation. Experimental results demonstrate the feasibility of our algorithm and we show applicability by using our protocol as a building block for a secure QR-Decomposition of a rational-valued matrix.
Keywords: Square Root, Fixed-Point Arithmetic, Secure Computation, QR-Decomposition

1 Introduction

Secure Multi-Party-Computation (SMPC) is an important branch of cryptography which enables a number of distinct entities (or parties) to securely evaluate any function without any of them having to reveal their particular input. The problem was first presented in [16] and (mostly) theoretically solved in ([1], [2], [6], [9]). However due to their high complexity these protocols are unsuitable for all but the most elementary computations.

In 2010 in [3], [4] and [5], Catrina et al. presented a framework for secure computation with fixed-point numbers. It can be used in conjunction with any linear Secret Sharing Scheme with a multiplication protocol such as Shamir's ([14]) and is the most versatile and practical scheme for secure computations with non-integer numbers developed so far. We describe how it can be extended by a protocol that securely computes the square root. It is based on Goldschmidt's algorithm for square root rather than Newton-Raphson iterations mainly because each iteration contains fewer dependent multiplications for virtually identical computation complexity. However, since Goldschmidt's algorithm is not self-correcting, the last iteration is Newton-Raphson to correct for accumulated rounding errors ([13]). The starting point - correct up to 5.4 bits - is computed by linear approximation.

We view our protocol not so much as a stand-alone application, but rather as a building block for more intricate algorithms. One such application is the secure computation of the QR-Decomposition of matrices, which can be used to securely solve linear systems of equations[1] and is an important building block in many other numerical algorithms such as optimization algorithms and finding zeroes of functions.

In section 2 we will define cryptographic primitives and terminology. In sections 3 and 4 we will describe our algorithm and its implementation. In section 5 we will apply our algorithm to the QR-Decomposition of matrices and in section 6 we will present our experimental results. Lastly in section 7 we will draw a conclusion.

2 Cryptographic Primitives and Definitions

The cryptographic primitive underlying our algorithms is a linear Secret Sharing Scheme (LSSS), such as Shamir's, with a multiplication protocol. Any secret shared number x will be written with braces $[x]$, while any public constant c will be written without braces. To signify a secret-shared vector v we will add an arrow: $[\overrightarrow{v}]$. A secret-shared matrix A will be written $[[A]]$. All matrices - unless stated differently - will be assumed to be quadratic with n rows and columns.

On top of the LSSS we employ the fixed-point arithmetic presented in [3],[4] and [5] to facilitate computations with non-integer numbers. We assume that all numbers have total bit-length k of which f are fractional, i.e. are elements of $\mathbb{Q}_{<k,f>}$ (cf. [5]). In order to be able to represent these in a Secret Sharing Scheme all fixed-point numbers are scaled by 2^f before being secret-shared yielding the set $\mathbb{Z}_{<k,f>}$. Any number in $\mathbb{Z}_{<k,f>}$ representing $\frac{x}{2^f}$ will be denoted by $\bar{x}$. Since secret-sharing requires a finite field we will treat $\mathbb{Z}_{<k,f>}$ as if it were part of $\mathbb{Z}/q\mathbb{Z}$ for a very large q (e.g. $\log_2 q \approx l = 1024$, $k = 110$). Note that because of this no wrap-around will occur and thus computations will *not* be affected by the fact that numbers are actually part of the much bigger $\mathbb{Z}/q\mathbb{Z}$. At some points in our protocols (pseudo-)random sharings of zero (PRSZ) need to be computed. We refer the reader to [8] for details.

We aim to develop algorithms secure in the so-called honest-but curious scenario in which parties may not deviate from the protocol. In addition we only require statistical and not information-theoretic security, i.e. the protocols can be simulated such that the distributions of the real and the simulated view are statistically indistinguishable ([5]).

[1] This has been done already (cf. [7] and others), but only for fields with characteristic > 0.

In the analysis of our algorithm we measure computation complexity using the unit of one secure multiplication.

3 Mathematical Foundations

Both algorithms - Goldschmidt's as well as Newton-Raphson's - work by iterative approximation, i.e. iteratively improving an initial estimate. They converge quadratically: If a good initial estimate is given, the number of correct digits doubles in every iteration.

3.1 Newton-Raphson Method

The aim of the Newton-Raphson method is to approximate the zero of a continuous, once differentiable function f. Starting with iterate x_0 a new iterate is given by (cf. [15]).

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{1}$$

We assume that the input is always greater than 0 and apply (1) to the function $f(R) = \frac{1}{R^2} - x$, whose zero is given by $\frac{1}{\sqrt{x}}$. The iterating function is thus $R_{j+1} = \frac{1}{2} \cdot R_j \cdot \left(3 - x \cdot R_j^2\right)$. At the end we multiply by x and gain $\sqrt{x}$.

3.2 Goldschmidt's Algorithm

If $x > 0$ is the number whose radicand is desired, Goldschmidt's algorithm ([10]) iteratively computes approximations of $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$[2]. The description of the software-friendly version can be seen in Fig. 1. An initial estimate y_0 of $\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{x_0}}$, such that

$$\frac{1}{2} < x_0 \cdot y_0^2 < \frac{3}{2} \tag{2}$$

is assumed to be given. We set $g_0 = x_0 \cdot y_0$ and $h_0 = \frac{y_0}{2}$. The iterates for $\sqrt{x}$ and $\frac{1}{2\sqrt{x}}$ are given by g_i and h_i respectively. Note that the multiplications in lines 3 and 4 are independent.

[2] It is actually a variation of the Newton-Raphson method described above; this can be shown using the original definition of the algorithm shown in [13]. Thus the convergence properties of Newton-Raphson iterations also apply here

Algorithm 1: Goldschmidt's algorithm for square root

1 While $|g_i - g_{i-1}| > \varepsilon$
2 $\quad r_{i-1} = \frac{1}{2} - g_{i-1} \cdot h_{i-1}$
3 $\quad g_i = g_{i-1} \cdot (1 + r_{i-1})$
4 $\quad h_i = h_{i-1} \cdot (1 + r_{i-1})$

Fig. 1. Goldschmidt's algorithm for square root with starting point y_0

3.3 Computation of the Starting Value

We compute the starting value by linear approximation:

$$L(x) = \alpha \cdot x + \beta. \tag{3}$$

Since the domain of our linear approximating function is the interval $[\frac{1}{2}, 1[$, we first have to normalize the input x_0 to this range giving x_{normal}. This is done in such a way that the resulting value is actually a very close approximation of $\frac{1}{\sqrt{x_0}}$ (see section 4)! To compute the coefficients α and β the idea (cf. [12][3]) is to minimize the relative error function

$$E(x) = \frac{\alpha \cdot x + \beta - \frac{1}{\sqrt{x}}}{\frac{1}{\sqrt{x}}} \tag{4}$$

Differentiating E gives its maximum at $x_{max} = -\frac{\beta}{3\alpha}$. Evaluating at x_{max} we get

$$M := E(x_{max}) = \frac{\sqrt{3}}{3} \cdot \sqrt{\frac{-\beta}{\alpha}} \cdot \left(\frac{2}{3} \cdot \beta - \frac{\sqrt{3}}{\sqrt{\frac{-\beta}{\alpha}}} \right) \tag{5}$$

Plugging this back into E and solving the system

$$E\left(\frac{1}{2}\right) = -M \tag{6}$$

$$E(1) = -M \tag{7}$$

for α and β gives us the values $\alpha = -0.8099868542$ and $\beta = 1.787727479$ which allow us to compute a linear approximation to $\frac{1}{\sqrt{x}}$ for $\frac{1}{2} \leq x < 1$ with relative error no more than 0.0222593752. This means the result is exact to almost 5.5 bits.

4 Description and Analysis of the Algorithms

We approximate $\frac{1}{\sqrt{x}}$ by first normalizing the input value to the interval $[\frac{1}{2}, 1[$ and then applying function (3) using the constants computed in section 3.3. The final result is computed by Goldschmidt and Newton-Raphson iterations.

[3] The coefficients used in [5] can be computed in a similar way.

Protocol 2: $([c],[v],[m],[w]) \leftarrow \mathsf{NormSQ}([x],k,f)$

$\left([x_{k-1}]^{\mathbb{F}_2^8},\ldots,[x_0]^{\mathbb{F}_2^8}\right) \leftarrow \mathsf{BitDec}([x],k,k)$
$\left([y_{k-1}]^{\mathbb{F}_2^8},\ldots,[y_0]^{\mathbb{F}_2^8}\right) \leftarrow \mathsf{PreOR}\left([x_{k-1}]^{\mathbb{F}_2^8},\ldots,[x_0]^{\mathbb{F}_2^8}\right)$
foreach $i \in [0,\ldots,k-1]$**do parallel**
    $[y_i] \leftarrow \mathsf{BitF2MtoZQ}\left([y_i]^{\mathbb{F}_2^8}\right)$
foreach $i \in [0,\ldots,k-2]$**do**
    $[z_i] \leftarrow [y_i] - [y_{i+1}]$
$[z_{k-1}] \leftarrow [y_{k-1}]$
$\overrightarrow{[W]} \leftarrow \mathsf{HalfIndex}\left(\overrightarrow{[z]},k\right)$
$[w] \leftarrow \sum_{i=0}^{\frac{k}{2}} 2^i \cdot [W_i]$
$[m] \leftarrow \sum_{i=0}^{k-1} 2^i \cdot [z_i]$
$[v] \leftarrow \sum_{i=0}^{k-1} 2^{k-i-1}[z_i]$
$[c] \leftarrow [x][v]$
return$([c],[v],[m],[w])$

Fig. 2. Modified protocol NormSQ

4.1 Norm

Protocol Norm from [5] returns values $2^{k-1} \leq [c] < 2^k$ and $[v]$ such that $[x]\cdot[v] = [c]$. If $2^{m-1} \leq [x] < 2^m$ then $[v] = \left[2^{k-m}\right]$. We modify Norm so that in addition it also returns $[m]$ as well as $[w] = [2^{\frac{m}{2}}]$, if m is even and $[w] = [2^{\frac{m-1}{2}}]$, if m is odd. $[w]$ is computed by sub-protocol HalfIndex, which works by rearranging the entries of $\overrightarrow{[z]}$ and can thus be implemented without additional expense. The modified protocol NormSQ is depicted in Fig. 2. Note that - in contrast to Norm - we leave out computation of the sign, since we assume the radicand to be greater than zero.

4.2 Approximation

Correctness: Let us assume that m is even. After evaluating the linear approximating function (3) we get $\alpha\cdot 2^{2k} + \beta\cdot 2^k \cdot [c]$. Multiplication by $[v] = [2^{k-m}]$ then yields $\alpha \cdot 2^{3k-m} + \beta \cdot 2^{2k-m} \cdot [c]$. But since $[c]$ is nothing but the Secret-Sharing of $\frac{\bar{x}}{2^m} \cdot 2^k = \frac{x\cdot 2^f}{2^m} \cdot 2^k$ this equals[4] $\alpha \cdot 2^{3k-m} + \beta \cdot 2^{3k-m} \cdot \frac{x\cdot 2^f}{2^m}$. After truncating[5] this by $3k-2f$ Bits, we get

$$2^{2f-m} \cdot \left(\alpha + \beta \cdot \frac{x \cdot 2^f}{2^m}\right), \tag{8}$$

which is a linear approximation to the inverse square root of the normalized value $\frac{x\cdot 2^f}{2^m}$ of $[x]$, scaled by the factor 2^{2f-m}. This means equation (8) equals

[4] for ease of presentation we will drop braces from here

[5] we neglect rounding errors at this point; for computational reasons the order in Protocol 3 is slightly different

Protocol 3: $[w] \leftarrow \mathsf{LinAppSQ}([b], k, f)$

1 $\alpha \leftarrow \mathsf{fld}_k(-0.8099868542)$
2 $\beta \leftarrow \mathsf{fld}_{2k}(1.787727479)$
3 $([c], [v], [m], [W]) \leftarrow \mathsf{NormSQ}([b], k, f)$
4 $[w] \leftarrow \alpha[c] + \beta$
5 $[m] \leftarrow \mathsf{Mod2}([m], \lceil \log_2 k \rceil)$
6 $[w] \leftarrow [w] \cdot [W] \cdot [v]$
7 $[w] \leftarrow \mathsf{DivConst}\left([w], 2^{\frac{f}{2}}\right)$
8 $[w] \leftarrow \mathsf{TruncPr}\left([w], 3k, 3k - 2f\right)$
9 $[w] \leftarrow (1 - [m]) \cdot [w] \cdot 2^f + \left(\sqrt{2} \cdot 2^f\right) \cdot [m] \cdot [w]$
10 $[w] \leftarrow \mathsf{TruncPr}([w], k, f)$
11 **return** $[w]$

Fig. 3. Linear approximation of $\frac{1}{\sqrt{x}}$

$$K \cdot 2^{2f-m} \cdot \frac{1}{\sqrt{\frac{x \cdot 2^f}{2^m}}} = \frac{2^{\frac{3f}{2}}}{2^{\frac{m}{2}}} \cdot K \cdot \frac{1}{\sqrt{x}}, \tag{9}$$

where K is factor very close to 1, determined by the approximation. Multiplication by $[W] = 2^{\frac{m}{2}}$ and division by $2^{\frac{f}{2}}$ thus yields an approximation to $\frac{1}{\sqrt{x}}$ scaled by 2^f which is just what is needed. If m is odd (to distinguish between even and odd we employ the protocol Mod2 from [3]), function NormSQ returns $[W] = 2^{\frac{m-1}{2}}$. Thus multiplication by $[W] \cdot 2^{-\frac{f}{2}}$ only gives $K \cdot \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{x}}$. In this case we thus subsequently multiply the equation by $\sqrt{2}$ and get the desired result.

Complexity: The cost is dominated by the protocol NormSQ and - to a lesser degree - by the protocol $\mathsf{TruncPr}([w], 3k, 3k - 2f)$. All other steps only add a small constant number of multiplications. The complexity of LinAppSQ can be seen in Table 1.

4.3 Goldschmidt's Algorithm

Given an approximation $[y_0]$ that fulfills the requirement (2) all we we have left to do is turn Algorithm 1 into a Secure Multi-Party Algorithm (Fig. 4).

In contrast to Algorithm 1 the number of iterations is fixed at $\theta = \left\lceil \log_2 \left(\frac{k}{5.4}\right) \right\rceil$ which ensures accuracy to k bits. We have replaced the last iteration (lines 19-23) by a Newton-Raphson iteration, because - in contrast to a Goldschmidt

Protocol 4: $[g] \leftarrow \mathbf{SQR}([x], k, f)$

1	$\theta \leftarrow \lceil \log_2 \left(\frac{k}{5.4}\right) \rceil$	**16**	$[r] \leftarrow \frac{3}{2} \cdot 2^f - [gh]$
2	$[y_0] \leftarrow \mathsf{LinAppSQ}([x], k, f)$	**17**	$[h] \leftarrow [h] \cdot [r]$
3	$[g_0] \leftarrow [y_0] \cdot [x]$	**18**	$[h] \leftarrow \mathsf{TruncPr}([h], k, f)$
4	$[g_0] \leftarrow \mathsf{TruncPr}([g_0], k, f)$	**19**	$[H] \leftarrow (2 \cdot [h])^2$
5	$[h_0] \leftarrow \mathsf{DivConst}([g_0], 2)$	**20**	$[H] \leftarrow [H] \cdot [x]$
6	$[gh] \leftarrow [g_0] \cdot [h_0]$	**21**	$[H] \leftarrow \left(3 \cdot 2^{2f}\right) - [H]$
7	$[gh] \leftarrow \mathsf{TruncPr}([gh], k, f)$	**22**	$[H] \leftarrow [h] \cdot [H]$
8	**For** $i = 1, \ldots, \theta - 2$	**23**	$[g] \leftarrow [H] \cdot [x]$
9	$[r] \leftarrow \frac{3}{2} \cdot 2^f - [gh]$	**24**	$[g] \leftarrow \mathsf{DivConst}([g], 2)$
10	$[g] \leftarrow [g] \cdot [r]$	**25**	$[g] \leftarrow \mathsf{TruncPr}([g],4k,4f)$
11	$[h] \leftarrow [h] \cdot [r]$	**26**	**return**$([g])$
12	$[g] \leftarrow \mathsf{TruncPr}([g], k, f)$		
13	$[h] \leftarrow \mathsf{TruncPr}([h], k, f)$		
14	$[gh] \leftarrow [g] \cdot [h]$		
15	$[gh] \leftarrow \mathsf{TruncPr}([gh], k, f)$		

Fig. 4. Goldschmidt's square root algorithm for SMPC

iteration - it is self-correcting and accumulated errors can be eliminated[6] ([13], except perhaps for the last bit which may be wrong due to the inexactness of probabilistic rounding). Since $[g]$ and $[gh]$ are no longer needed at this point, their computation is omitted in the last Goldschmidt iteration (lines 16-18). Note that computation (and truncation) of $[g]$ and $[h]$ in the loop can be parallelized. Complexity again can be read off from Table 1.

4.4 Security

All our protocols consist of building blocks that have been proven secure either perfect or statistical ([3],[4],[5]). No information is revealed in our additional protocols. All counters are public parameters and thus do not leak any information. We conclude our protocols are secure in the honest-but-curious scenario.

5 Application to QR-Decomposition

The QR-Decomposition of a matrix is an important numerical primitive, that can be used to solve linear systems of equations and is part of many numerical algorithms. For any matrix A, the goal is to compute an orthogonal matrix Q and an upper-triangular matrix R such that $Q \cdot R = A$. For details see [11].

[6] Due to accumulated rounding errors it is theoretically possible that the result after the last Goldschmidt iteration has less than $\frac{k}{2}$ correct bits, and thus one Newton-Raphson iteration might not suffice to eliminate them. However this did not occur once in our experiments. See section 6.2 for more details.

	Secure Multiplications	Rounds	Field
LinAppSqr	10	$\frac{k}{2}+l+9$	$\mathbb{Z}_q$
	$7k+1$	7	$\mathbb{Z}_{q_1}$
	$2k^2-2k+kl$	$l+1$	$\mathbb{F}_{2^8}$
SQR	$6\theta+11$	$\frac{k}{2}+l+4\theta+14$	$\mathbb{Z}_q$
	$3f\cdot(\theta+1)+7k+1$	$2\theta+9$	$\mathbb{Z}_{q_1}$
	$2k^2-2k+kl$	$l+1$	$\mathbb{F}_{2^8}$
House	$6\theta+15$	$\frac{k}{2}+l+4\theta+17$	$\mathbb{Z}_q$
	$8k+4f+3\theta f$	$2\theta+10$	$\mathbb{Z}_{q_1}$
	$2k^2+kl-4$	$l+1$	$\mathbb{F}_{2^8}$
Pre-Mult-House	$2n^2+3n+3\theta+9$	$l+3\theta+13$	$\mathbb{Z}_q$
	$k\cdot(2\theta+6)+nf\cdot(n+2)-1$	12	$\mathbb{Z}_{q_1}$
	$2k^2+kl$	$2l+2$	$\mathbb{F}_{2^8}$
QRDecomp	$\mathcal{O}(n^3+\theta n)$	$(n-1)\cdot\left(\frac{k}{2}+2l+7\theta+30\right)$	$\mathbb{Z}_q$
	$\mathcal{O}(n^3f+\theta kn)$	$(n-1)\cdot(2\theta+22)$	$\mathbb{Z}_{q_1}$
	$\mathcal{O}(+kln+k^2n)$	$3(n-1)\cdot(l+1)$	$\mathbb{F}_{2^8}$

Table 1. Complexity of the protocols. The bit-length of k is assumed to be a power of 2, e.g. $k=2^l$. All vectors are assumed to be of length n and all matrices are assumed to be quadratic with n rows and n columns.

5.1 Secure Computation of the QR-Decomposition

We compute the QR-Decomposition by the sequential application of *Householder-Matrices*. Each such Householder-Matrix is responsible for computing one column of R. Their product forms Q. To compute a Householder-Matrix one first needs to compute the *Householder-Vector* $\overrightarrow{\tilde{v}}$ from the respective column $\overrightarrow{v}$. The Householder-Matrix P is then defined by $P=\left(Id-2\cdot\frac{\tilde{v}\tilde{v}^t}{\tilde{v}^t\tilde{v}}\right)$. The secure version of the algorithm used to compute the Householder-Vector $\overrightarrow{\tilde{v}}$ from a vector $\overrightarrow{v}$ is based on the one described in [11], but differs in that the first component is not normalized to one which saves one division. It can be seen in Fig. 5.

We assume the vector to be non-zero.[7] In steps 1-3 the norm of the input-vector is computed using the routine described in [3] that reduces the cost of the inner-product to one secure multiplication. Protocol SQR is utilized in step 3. In step 5 the sign of $[x_1]$ is computed. Steps 1-3 and 4-5 can be parallelized.

5.2 Multiplication with a Householder-Matrix

If a secret-shared Householder-Vector $\overrightarrow{[v]}$ and a matrix $[[A]]$ are given, the algorithm for Pre-Multiplication of $[[A]]$ by the respective Householder-Matrix is

[7] This condition could be checked prior to the computation, but this should rarely be necessary. For $\overrightarrow{[x]}=0$ the respective Householder-Vector is not defined

Protocol 5: $\overrightarrow{[v]} \leftarrow$**House**$\left(\overrightarrow{[x]}, n\right)$

$[\mu] \leftarrow \mathsf{Inner}\left(\overrightarrow{[x]}, \overrightarrow{[x]}\right)$
$[\mu] \leftarrow \mathsf{TruncPr}([\mu], k, f)$
$[\mu] \leftarrow \mathsf{SQR}([\mu], k, f)$
$\overrightarrow{[v]} \leftarrow \overrightarrow{[x]}$
$[\sigma] \leftarrow 1 - 2 \cdot \mathsf{LTZ}([x_1], k) \qquad \backslash\backslash \sigma = [sign([x_1])]$
$[\beta] \leftarrow [x_1] + [\sigma] \cdot [\mu]$
$[v_0] \leftarrow [\beta]$
return $\overrightarrow{[v]}$

Fig. 5. Computation of a Householder-Vector

Protocol 6: $[[A]] \leftarrow$ **Pre-Mult-House** $\left([[A]], \overrightarrow{[v]}, m, n\right)$

$[\tilde{v}] \leftarrow \mathsf{Inner}\left(\overrightarrow{[v]}, \overrightarrow{[v]}\right)$
$[\tilde{v}] \leftarrow \mathsf{TruncPr}([\tilde{v}], k, f)$
$[\beta] \leftarrow -2 \cdot \mathsf{DivNR}(1, [\tilde{v}], k, f)$
$\overrightarrow{[v]} \leftarrow \mathsf{Matrix\text{-}Mult\text{-}Vector}([[A]]^t, \overrightarrow{[v]})$
$\overrightarrow{[w]} \leftarrow [\beta] \cdot \overrightarrow{[v]}$
$\overrightarrow{[w]} \leftarrow \mathrm{TruncPr}(\overrightarrow{[w]}, 2k, 2f)$
$[[V]] \leftarrow \mathsf{Matrix\text{-}Matrix\text{-}Multiply}\left(\overrightarrow{[v]}, \overrightarrow{[w]}^t\right)$
$[[V]] \leftarrow \mathsf{TruncPr}([[V]], 3k, 3f)$
$[[A]] \leftarrow [[A]] + [[V]]$
return $[[A]]$

Fig. 6. Pre-Multiplication of A by the Householder-Matrix determined by the Householder vector $\overrightarrow{[v]}$

described in Fig. 6. Correctness can be easily verified using the equation in section 5.1. Post-Multiplication is only slightly different.

5.3 Computation of the QR-Decomposition

With these tools it is easy to describe the QR-Decomposition based on Householder-Multiplications (Fig. 7). The matrix R is saved in the upper-triangular part, while the Householder-Vectors - except for the first component, which is stored in an additional vector $[\delta]$ - are stored below the diagonal. If necessary the matrix Q can then be computed by repeated application of protocol 6.

Protocol 7: $\left([[A]], \overrightarrow{[\delta]}\right) \leftarrow \mathbf{QR}\left([[A]], n\right)$

1 For $(j = 1, \ldots, n-1)$
2 $\quad \overrightarrow{[v(j:n)]} \leftarrow \mathsf{House}\left(\overrightarrow{[A(j:n,j)]}, n-j+1\right)$
3 $\quad [[A(j:n, j:n)]] \leftarrow \mathsf{Pre\text{-}Mult\text{-}House}\left([[A(j:n, j:n)]], \overrightarrow{[v(j:n)]}, n-j, n-j\right)$
4 $\quad$ **If** $(j < n)$
5 $\quad\quad [[A(j+1:n, j)]] \leftarrow \overrightarrow{[v(j+1:m)]}$
6 $\quad\quad [\delta_j] \leftarrow [v_1]$
7 return $\left([[A]], \overrightarrow{[\delta]}\right)$

Fig. 7. Secure Computation of the QR-Decomposition of a square matrix using Householder-Matrices

6 Experimental Results

6.1 The Setup

All protocols were tested with an underlying (5,2) Shamir Secret Sharing Scheme. In contrast to "real" Multi-Party-Computations all computations were performed on one machine so network-latency is not included in the computation times. We tested our protocols using our own C++-implementation of the fixed-point arithmetic from [3], [4] and [5]. For computations with very large numbers we employed the GNU MP 5.0.2. The machine was running Linux Mint 10 with an Athlon II Quad-Core CPU @2.6GHz and 4GB RAM.

6.2 Computation of the Square Root

In our experiments we computed 8 square roots from numbers $a = 0.008585937$, $b = 0.146234375$, $c = 0.6326875$, $d = 11.19$, $e = 197.04$, $f = 3110.4$, $g = 489,291.776$, $h = 3,701,997.568$. Since in our protocol numbers are normalized first, the size of the number should matter less than how close or how far a number $2^{k-1} \leq s < 2^k$, $s \in \{a, b, c, d, e, f, g, h\}$ is to the respective 2^k and 2^{k-1}. Care was taken that the numbers are evenly distributed, i.e. irrespective of size there is one number for each eighth of the interval $[2^{k-1}, 2^k[$. We used fixed-point numbers with 110 bits of which 80 were fractional. The absolute value of the absolute error (the difference between the exact result and the computed result) was always less than 2^{-80}, i.e. exact in our fixed-point setting. Computation times were $\approx 4.89s$ (≈ 0.98s per player) for all numbers. One full Goldschmidt iteration took about 0.57s (≈ 0.114s p.p.) to compute (the abbreviated one took 0.19s (≈ 0.038s p.p.)). What stands out is that the Newton-Raphson iteration at the end at about 0.19s p.p. was more than 60% more expensive! Even though communication did not actually take place, this is remarkable and vindicates our decision to use Goldschmidt iterations for all but one iteration. Figures for average precision gained from testing our algorithm on 1400 random numbers can be seen in Table 2.

x	a	b	c	d	e	f	g	h
abs. error	$< 2^{-81}$	$< 2^{-82}$	$< 2^{-81}$	$< 2^{-82}$	$< 2^{-81}$	$< 2^{-80}$	$< 2^{-81}$	$< 2^{-82}$
rel. error	$< 2^{-78}$	$< 2^{-81}$	$< 2^{-81}$	$< 2^{-84}$	$< 2^{-85}$	$< 2^{-86}$	$< 2^{-90}$	$< 2^{-93}$

Table 2. Exactness for computation of the square root

6.3 Computation of the QR-Decomposition

We tested our secure implementation of the QR-Decomposition on a symmetric positive definite 3×3-matrix A and a random 5×5-matrix B. To quantify the exactness of our results $(\tilde{Q}, \tilde{R})$ we compared them to the exact ones using the Frobenius-Norm:

$$\|A\|_F := \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}^2} \tag{10}$$

The results can be found in Table 3. Note that the Frobenius-Norm is just one of a number of (equivalent) matrix-norms. Using another norm could yield slightly smaller or bigger numbers.

Matrix	**A**	**B**
$\|\tilde{Q}\|_F - \|Q\|_F$	$4.7 \cdot 10^{-20}$	$\approx 1.6 \cdot 10^{-24}$
$\|\tilde{R}\|_F - \|R\|_F$	$3.2 \cdot 10^{-14}$	$\approx 7 \cdot 10^{-12}$
$\|\tilde{Q} \cdot \tilde{R}\|_F - \|A, B\|_F$	$3.1 \cdot 10^{-14}$	$\approx 7 \cdot 10^{-12}$

Table 3. Experimental results of the QR-Decomposition

7 Conclusion and further work

We have for the first time described a practical way to securely compute the square root of a shared value. We have demonstrated the feasibility of our approach experimentally and applied it to the QR-Decomposition of a square-matrix which can be used to securely solve linear systems of equations and can serve as a building block for many other numerical algorithms.

Acknowledgment. The author is funded by "Ausbau der Kompetenzpartnerschaft zum Themenschwerpunkt 'IT-Sicherheit' an den Standorten Passau und Regensburg" which is co-funded by the European Regional Development Fund (EFRE).

References

1. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.
2. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.
3. O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. Garay and R. De Prisco, editors, *Security and Cryptography for Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199. Springer Berlin / Heidelberg, 2010.
4. O. Catrina and S. de Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security - ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. Springer Berlin / Heidelberg, 2010.
5. O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In R. Sion, editor, *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin / Heidelberg, 2010.
6. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.
7. R. Cramer and I. Damgård. Secure distributed linear algebra in a constant number of rounds. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 119–136. Springer Berlin / Heidelberg, 2001.
8. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In J. Kilian, editor, *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer Berlin / Heidelberg, 2005.
9. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.
10. R. E. Goldschmidt. Applications of division by convergence. Master's thesis, M.I.T., 1964.
11. G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.
12. M. Ito, N. Takagi, and S. Yajima. Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification. *IEEE Transactions on Computers*, 46:495–498, 1997.
13. P. Markstein. Software division and square root using goldschmidt's algorithms. In *In 6th Conference on Real Numbers and Computers*, pages 146–157, 2004.
14. A. Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.
15. J. Stoer and R. Bulirsch. *Introduction to numerical analysis*. Texts in applied mathematics. Springer, Berlin/Heidelberg, 2002.
16. A. C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.