Efficient Decomposition of Dense Matrices over GF(2)

Martin R. Albrecht*[1] and Clément Pernet[2]

[1] Information Security Group, Royal Holloway, University of London
Egham, Surrey TW20 0EX, United Kingdom
M.R.Albrecht@rhul.ac.uk
[2] INRIA-MOAIS LIG, Grenoble Univ. ENSIMAG, Antenne de Montbonnot 51,
avenue Jean Kuntzmann, F-38330 MONTBONNOT SAINT MARTIN, France
clement.pernet@imag.fr

Abstract. In this work we describe an efficient implementation of a hierarchy of algorithms for the decomposition of dense matrices over the field with two elements ($\mathbb{F}_2$). Matrix decomposition is an essential building block for solving dense systems of linear and non-linear equations and thus much research has been devoted to improve the asymptotic complexity of such algorithms. In this work we discuss an implementation of both well-known and improved algorithms in the M4RI library. The focus of our discussion is on a new variant of the M4RI algorithm – denoted MMPF in this work – which allows for considerable performance gains in practice when compared to the previously fastest implementation. We provide performance figures on x86_64 CPUs to demonstrate the viability of our approach.

1 Introduction

We describe an efficient implementation of a hierarchy of algorithms for PLS decomposition of dense matrices over the field with two elements ($\mathbb{F}_2$). The PLS decomposition is closely related to the well-known PLUQ and LQUP decompositions. However, it offers some advantages in the particular case of $\mathbb{F}_2$. Matrix decomposition is an essential building block for solving dense systems of linear and non-linear equations (cf. [11, 10]) and thus much research has been devoted to improve the asymptotic complexity of such algorithms. In particular, it has been shown that various matrix decompositions such as PLUQ, LQUP and LPS are essentially equivalent and can be reduced to matrix-matrix multiplication (cf. [**?**]). Thus, we know that these decompositions can be achieved in $\mathcal{O}(n^\omega)$ where ω is the exponent of linear algebra[3]. In this work we focus on matrix decomposition in the special case of $\mathbb{F}_2$ and discuss an implementation of both well-known and improved algorithms in the M4RI library [3]. The M4RI library implements dense linear algebra over $\mathbb{F}_2$ and is used by the Sage [15] mathematics software and the PolyBoRi [9] package for computing Gröbner bases. It is also the linear algebra library used in [14, 13].

Our implementation focuses on 64-bit x86 architectures (x86_64), specifically the Intel Core 2 and the AMD Opteron. Thus, we assume in this chapter that each native CPU word has 64 bits. However it should be noted that our code also runs on 32-bit CPUs and on non-x86 CPUs such as the PowerPC.

* This author was supported by the Royal Holloway Valerie Myerscough Scholarship.

[3] For practical purposes we set $\omega = 2.807$.

Element-wise operations over $\mathbb{F}_2$ are relatively cheap compared to loads from and writes to memory. In fact, in this work we demonstrate that the two fastest implementations for dense matrix decomposition over $\mathbb{F}_2$ (the one presented in this work and the one found in Magma [8] due to Allan Steel) perform worse for sparse matrices despite the fact that fewer field operations are performed. This indicates that counting raw field operations is not an adequate model for estimating the running time in the case of $\mathbb{F}_2$.

This work is organised as follows. We will start by giving the definitions of reduced row echelon forms (RREF), PLUQ and PLS decomposition in Section 2 and establish their relations. We will then discuss Gaussian elimination and the M4RI algorithm in Section 3 followed by a discussion of cubic PLS decomposition and the MMPF algorithm in 4. We will then discuss asymptotically fast PLS decomposition in Section 5 and implementation issues in Section 6. We conclude by giving empirical evidence of the viability of our approach in Section 7.

2 RREF and PLS

Proposition 1 (PLUQ decomposition). *Any $m \times n$ matrix A with rank r, can be written $A = PLUQ$ where P and Q are two permutation matrices, of dimension respectively $m \times m$ and $n \times n$, L is $m \times r$ unit lower triangular and U is $r \times n$ upper triangular.*

Proof. See [**?**].

Proposition 2 (PLS decomposition). *Any $m \times n$ matrix A with rank r, can be written $A = PLS$ where P is a permutation matrix of dimension $m \times m$, L is $m \times r$ unit lower triangular and S is an $r \times n$ matrix which is upper triangular except that its columns are permuted, that is $S = UQ$ for U $r \times n$ upper triangular and Q is a $n \times n$ permutation matrix.*

Proof. Write $A = PLUQ$ and set $S = UQ$.

Another way of looking at PLS decomposition is to consider the $A = LQUP$ decomposition [12]. We have $A = LQUP = LSP$ where $S = QU$. We can also write $A = LQUP = SUP$ where $S = LQ$. Applied to A^T we then get $A = P^T U^T S^T = P' L' S'$. Finally, a proof for Proposition 2 can also be obtained by studying any one of the Algorithms 9, 3 or 4.

Definition 1 (Row Echelon Form). *An $m \times n$ matrix A is in echelon form if all zero rows are grouped together at the last row positions of the matrix, and if the leading coefficient of each non zero row is one and is located to the right of the leading coefficient of the above row.*

Proposition 3. *Any $m \times n$ matrix can be transformed into echelon form by matrix multiplication.*

Proof. See [**?**]

Note that while there are many PLUQ decompositions of any matrix A there is always also a decomposition for which we have that $S = UQ^T$ is a row echelon form of A. In

this work we compute $A = PLS$ such that S is in row echelon form. Thus, a proof for Proposition 3 can also be obtained by studying any one of the Algorithms 9, 3 or 4.

Definition 2 (Reduced Row Echelon Form). *An $m \times n$ matrix A is in reduced echelon form if it is in echelon form and each leading coefficient of a non zero row is the only non zero element in its column.*

3 Gaussian Elimination and M4RI

Gaussian elimination is the classical, cubic algorithm for transforming a matrix into (reduced) row echelon form using elementary row operations only. The "Method of the Four Russians" Inversion (M4RI) [7] reduces the number of additions required by Gaussian elimination by a factor of $\log n$ by using a caching technique inspired by Kronrod's method for matrix-matrix multiplication.

3.1 The "Method of the Four Russians" Inversion (M4RI)

The "Method of the Four Russians" inversion was introduced in [5] and later described in [6] and [7]. It inherits its name and main idea from the misnamed "Method of the Four Russians" multiplication [4, 1].

To give the main idea consider for example the matrix A of dimension $m \times n$ in Figure 3.1. The $k \times n$ ($k = 3$) submatrix on the top has full rank and we performed Gaussian elimination on it. Now, we need to clear the first k columns of A for the rows below k (and above the submatrix in general if we want the reduced row echelon form). There are 2^k possible linear combinations of the first k rows, which we store in a table T. We index T by the first k bits (e.g., $011 \to 3$). Now to clear k columns of row i we use the first k bits of that row as an index in T and add the matching row of T to row i, causing a cancellation of k entries. Instead of up to k additions this only costs one addition due to the pre-computation. Using Gray codes (or similar techniques) this pre-computation can be performed in 2^k vector additions and the overall cost is $2^k + m - k + k^2$ vector additions in the worst case (where k^2 accounts for the Gauss elimination of the $k \times n$ submatrix). The naive approach would cost $k \cdot m$ row additions in the worst case to clear k columns. If we set $k = \log m$ then the complexity of clearing k columns is $\mathcal{O}\left(m + \log^2 m\right)$ vector additions in contrast to $\mathcal{O}(m \cdot \log m)$ vector additions using the naive approach.

This idea leads to Algorithm 1. In this algorithm the subroutine GaussSubmatrix (cf. Algorithm 8) performs Gauss elimination on a $k \times n$ submatrix of A starting at position (r, c) and searches for pivot rows up to m. If it cannot find a submatrix of rank k it will terminate and return the rank $\overline{k}$ found so far. Note the technicality that the routine GaussSubmatrix and its interaction with Algorithm 1 make use of the fact that all the entries in a column below a pivot are zero if they were considered already.

The subroutine MakeTable (cf. Algorithm 7) constructs the table T of all 2^k linear combinations of the k rows starting a row r and a column c, i.e. it enumerates all elements of the vector space $\mathsf{span}(r, ..., r + \overline{k} + 1)$ spanned by the rows $r, \ldots, r + \overline{k} - 1$. Finally, the subroutine AddRowsFromTable (cf. Algorithm 6) adds the appropriate row from T – indexed by k bits starting at column c – to each row of A with index $i \notin \{r, \ldots, r + \overline{k} - 1\}$. That is, it adds the appropriate linear combination of the rows $\{r, \ldots, r + \overline{k} - 1\}$ onto a row i in order to clear k columns.

$$A = \left(\begin{array}{c|c} 1\ 0\ 0 & 1\ 0\ 1\ 1\ 1\ \ldots \\ 0\ 1\ 0 & 1\ 1\ 1\ 1\ 0\ \ldots \\ 0\ 0\ 1 & 0\ 0\ 1\ 1\ 1\ \ldots \\ \ldots & \\ 0\ 0\ 0 & 1\ 1\ 0\ 1\ 0\ \ldots \\ \mathbf{1\ 1\ 0} & 0\ 1\ 0\ 1\ 1\ \ldots \\ 0\ 1\ 0 & 0\ 1\ 0\ 0\ 1\ \ldots \\ \ldots & \\ \mathbf{1\ 1\ 0} & 1\ 1\ 1\ 0\ 1\ \ldots \end{array}\right) T = \left[\begin{array}{c|c} 0\ 0\ 0 & 0\ 0\ 0\ 0\ 0\ \ldots \\ 0\ 0\ 1 & 0\ 0\ 1\ 1\ 1\ \ldots \\ 0\ 1\ 0 & 1\ 1\ 1\ 1\ 0\ \ldots \\ 0\ 1\ 1 & 1\ 1\ 0\ 0\ 1\ \ldots \\ 1\ 0\ 0 & 1\ 0\ 1\ 1\ 1\ \ldots \\ 1\ 0\ 1 & 1\ 0\ 0\ 0\ 0\ \ldots \\ \mathbf{1\ 1\ 0} & 0\ 1\ 0\ 0\ 1\ \ldots \\ 1\ 1\ 1 & 0\ 1\ 1\ 1\ 0\ \ldots \end{array}\right]$$

Fig. 1. M4RI Idea

Note that the relation between the index id and the row j in T is static and known *a priori* because GaussSubmatrix puts the submatrix in reduced row echelon form. In particular this means that the $\overline{k} \times \overline{k}$ submatrix starting at (r, c) is the identity matrix.

Input: A – a $m \times n$ matrix
Input: k – an integer $k > 0$
Result: A is in reduced row echelon form.
begin
  $r, c \longleftarrow 0, 0$;
  while $c < n$ **do**
    if $c + k > n$ **then** $k \leftarrow n - c$;
    ;
    $\overline{k} \longleftarrow$ GaussSubmatrix(A, r, c, k, m);
    if $\overline{k} > 0$ **then**
      $T, L \longleftarrow$ MakeTable$(A, r, c, \overline{k})$;
      AddRowsFromTable$(A, 0, r, c, \overline{k}, T, L)$;
      AddRowsFromTable$(A, r + \overline{k}, m, c, \overline{k}, T, L)$;
    end
    $r, c \longleftarrow r + \overline{k}, c + \overline{k}$;
    if $k \neq \overline{k}$ **then** $c \leftarrow c + 1$;
    ;
  end
end

Algorithm 1: M4RI

When studying the performance of Algorithm 1, we expect the function MakeTable to contribute most. Instead of performing $\overline{k}/2 \cdot 2^{\overline{k}} - 1$ additions MakeTable only performs $2^{\overline{k}} - 1$ vector additions. However, in practice the fact that $\overline{k}$ columns are processed in each loop iteration of AddRowsFromTable contributes signficiantly due to the better cache locality. Assume the input matrix A does not fit into L2 cache. Gaussian elimination would load a row from memory, clear one column and likely evict that row from cache in

order to make room for the next few rows before considering it again for the next column. In the M4RI algorithm more columns are cleared per load.

We note that our presentation of M4RI differs somewhat from that in [6]. The key difference is that our variant does not throw an error if it cannot find a pivot within the first $3k$ rows in GaussSubmatrix. Instead, our variant searches all rows and consequently the worst-case complexity is cubic. However, on average for random matrices we expect to find a pivot within $3k$ rows and thus expect the average-case complexity to be $\mathcal{O}(n^3/\log n)$.

4 M4RI and PLS Decomposition

In order to recover the PLS decomposition of some matrix A, we can adapt Gaussian elimination to preserve the transformation matrix in the lower triangular part of the input matrix A and to record all permutations performed. This leads to Algorithm 9 in the Appendix which modifies A such that it contains L in below the main diagonal, S above the main diagonal and returns P and Q such that $PLS = A$ and $SQ^T = U$.

The main differences between Gaussian elimination and Algorithm 9 are:

- No elimination is performed above the currently considered row, i.e. the rows $0, \ldots, r-1$ are left unchanged. Instead elimination starts below the pivot, from row $r+1$.
- Column swaps are performed at the end of Algorithm 9 but not in Gaussian elimination. This step compresses L such that it is lower triangular.
- Row additions are performed starting at column $r+1$ instead of r to preserve the transformation matrix L. Over any other field we would have to rescale $A[r, r]$ for the transformation matrix L but over $\mathbb{F}_2$ this is not necessary.

4.1 The Method of Many People Factorisation (MMPF)

In order to use the M4RI improvement over Gaussian elimination for PLS decomposition, we have to adapt the M4RI algorithm.

Column Swaps Since column swaps only happen at the very end of the algorithm we can modify the M4RI algorithm in the obvious way to introduce them.

U vs. I Recall, that the function GaussSubmatrix generates small $\overline{k} \times \overline{k}$ identity matrices. Thus, even if we remove the call to the function AddRowsFromTable$(A, 0, r, c, \overline{k}, T)$ from Algorithm 1 we would still eliminate up to $\overline{k} - 1$ rows above a given pivot and thus would fail to produce U. The reason the original specification [5] of the M4RI requires $\overline{k} \times \overline{k}$ identity matrices is to have a *a priori* knowledge of the relationship between id and j in the function AddRowsFromTable. On the other hand the rows of any $\overline{k} \times n$ upper triangular matrix also form a basis for the $\overline{k}$-dimensional vector space $\mathsf{span}(r, \ldots, r+\overline{k}-1)$. Thus, we can adapt GaussSubmatrix to compute the upper triangular matrix instead of the identity. Then, in MakeTable1 we can encode the actual relationship between a row j of T and id in the lookup table L.

Preserving L In Algorithm 9 preserving the transformation matrix L is straight forward: addition starts in column $c+1$ instead of c. On the other hand, for M4RI we need to fix the table T to update the transformation matrix correctly; For example, assume $\overline{k} = 3$ and that the first row of the $\overline{k} \times n$ submatrix generated by GaussSubmatrix has the first $\overline{k}$ bits equal to `[1 0 1]`. Assume further that we want to clear $\overline{k}$ bits of a a row which also starts with `[1 0 1]`. Then – in order to generate L – we need to encode that this row is cleared by adding the first row only, i.e. we want the first $\overline{k} = 3$ bits to be `[1 0 0]`. Recall that in the M4RI algorithm the *id* for the row j starting with `[1 0 0]` is `[1 0 0]` if expressed as a sequence of bits. Thus, to correct the table, we add the $\overline{k}$ bits of the *a priori id* onto the first $\overline{k}$ entries in T (starting at c) as in MakeTable1.

Other Bookkeeping Recall that GaussSubmatrix's interaction with Algorithm 1 uses the fact that processed columns of a row are zeroed out to encode whether a row is "done" or not. This is not true anymore if we compute the PLS decomposition instead of the upper triangular matrix in GaussSubmatrix since we store L below the main diagonal. Thus, we explicitly encode up to which row a given column is "done" in PlsSubmatrix (cf. Algorithm 10). Finally, we have to take care not to include the transformation matrix L when constructing T.

Input: A – a $m \times n$ matrix
Input: r_{start} – an integer $0 \leq r_{\text{start}} < m$
Input: c_{start} – an integer $0 \leq c_{\text{start}} < n$
Input: k – an integer $k > 0$
Result: Retuns an $2^k \times n$ matrix T and the translation table L
begin
  $T \longleftarrow$ the $2^k \times n$ zero matrix;
  for $1 \leq i < 2^k$ **do**
    $j \longleftarrow$ the row index of A to add according to the Gray code;
    add row j of A to the row i of T starting at c_{start};
  end
  $L \longleftarrow$ an integer array with 2^k entries;
  for $1 \leq i < 2^k$ **do**
    $id = \sum_{j=0}^{k} T[i, c_{\text{start}} + j] \cdot 2^{k-j-1}$;
    $L[id] \longleftarrow i$;
  end
  for $1 \leq i < 2^k$ **do**
    $b_0, \ldots, b_{\overline{k}-1} \longleftarrow$ bits of *a priori id* of the row i;
    for $0 \leq j < \overline{k}$ **do**
      $T[i, c_{\text{start}} + j] \longleftarrow T[i, c_{\text{start}} + j] + b_j$;
    end
  end
  return T, L;
end

Algorithm 2: MakeTable1

These modifications lead to Algorithm 3 which computes the *PLS* decomposition of A in-place, that is L is stored below the main diagonal and S is stored above the main

diagonal of the input matrix. Since none of the changes to the M4RI algorithm affect the asymptotical complexity, Algorithm 3 is cubic in the worst case and has complexity $\mathcal{O}\left(n^3/\log n\right)$ in the average case.

Input: A – a $m \times n$ matrix
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Input: k – an integer $k > 0$
Result: PLS decomposition of A
begin
  $r, c \longleftarrow 0, 0$;
  for $0 \le i < n$ **do** $Q[i] \longleftarrow i$;
  ;
  for $0 \le i < m$ **do** $P[i] \longleftarrow i$;
  ;
  while $r < m$ **and** $c < n$ **do**
    if $c + k > n$ **then** $k \longleftarrow n - c$;
    ;
    $\overline{k}, d_r \longleftarrow$ PlsSubmatrix(A, r, c, k, P, Q);
    $U \longleftarrow$ the $\overline{k} \times n$ submatrix starting at $(r, 0)$ where every entry prior to the upper triangular matrix starting at (r, c) is zeroed out;
    if $\overline{k} > 0$ **then**
      $T, L \longleftarrow$ MakeTable1$(U, 0, c, \overline{k})$;
      AddRowsFromTable$(A, d_r + 1, m, c, \overline{k}, T, L)$;
      $r, c \leftarrow r + \overline{k}, c + \overline{k}$;
    else
      `// skip zero column`
      $c \leftarrow c + 1$;
    end
  end
  `// Now compress L`
  for $0 \le j < r$ **do** swap the columns j and $Q[j]$ starting at row j;
  ;
  return r;
end

Algorithm 3: MMPF

5 Asymptotically Fast PLS Decomposition

It is well-known that PLUQ decomposition can be accomplished in-place and in time complexity $\mathcal{O}(n^\omega)$ by reducing it to matrix-matrix multiplication (cf. [**?**]). We give a slight variation of the recursive algorithm from [**?**] in Algorithm 4. We compute the PLS instead of the PLUQ decomposition.

In Algorithm 4 the routine SubMatrix(r_s, c_s, r_e, c_e) returns a "view" (cf. [2]) into the matrix A starting at row and column r_s and c_s resp. and ending at row and column

Input: A – a $m \times n$ matrix
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Result: PLS decomposition of A
begin
  $n_0 \longleftarrow$ pick some integer $0 \leq n_0 < n$; `//` $n_0 \approx n/2$
  $A_0 \longleftarrow$ SubMatrix$(A, 0, 0, m, n_0)$;
  $A_1 \longleftarrow$ SubMatrix$(A, 0, n_0, m, n)$;
  $Q_0 \longleftarrow Q[0, \ldots, n_0]$;
  $r_0 \longleftarrow$ PLS(A_0, P, Q_0); `// first recursive call`
  for $0 \leq i \leq n_0$ **do** $Q[i] \leftarrow Q_0[i]$;
  ;
  $A_{NW} \longleftarrow$ SubMatrix$(A, 0, 0, r_0, r_0)$;
  $A_{SW} \longleftarrow$ SubMatrix$(A, r_0, 0, m, r_0)$;
  $A_{NE} \longleftarrow$ SubMatrix$(A, 0, n_0, r_0, n)$;
  $A_{SE} \longleftarrow$ SubMatrix(A, r_0, n_0, m, n);
  if r_1 **then**
    `// Compute of the Schur complement`
    $A_1 \longleftarrow P \times A_1$;
    $L_{NW} \longleftarrow$ the lower left triangular matrix in A_{NW};
    $A_{NE} \longleftarrow L_{NW}^{-1} \times A_{NE}$;
    $A_{SE} \longleftarrow A_{SE} + A_{SW} \times A_{NE}$;
  end
  $P_1 \longleftarrow P[r_0, \ldots, m]$;
  $Q_1 \longleftarrow Q[n_0, \ldots, n]$;
  $r_1 \longleftarrow$ PLS(A_{SE}, P_1, Q_1); `// second recursive call`
  $A_{SW} \longleftarrow P \times A_{SW}$;
  `// Update P & Q`
  for $0 \leq i < m - r_0$ **do** $P[r_0 + 1] = P_1[i] + r_0$;
  ;
  for $0 \leq i < n - n_0$ **do** $Q[n_0 + i] \leftarrow Q_1[i] + n_0$;
  ;
  $j \leftarrow r_0$;
  for $n_0 \leq i < n_0 + r_1$ **do** $Q[j] \leftarrow Q[i]$; $j \leftarrow j + 1$;
  ;
  `// Now compress L`
  $j \leftarrow n_0$;
  for $r_0 \leq i < r_0 + r_1$ **do** swap the columns i and j starting at row i;
  ;
  return $r_0 + r_1$;
end

Algorithm 4: PLS Decomposition

r_e and c_e resp. We note that that the step $A_{NE} \longleftarrow L_{NW}^{-1} \times A_{NE}$ can be reduced to matrix-matrix multiplication (cf. [**?**]). Thus Algorithm 4 can be reduced to matrix-matrix multiplication and has complexity $\mathcal{O}(n^\omega)$. Since no temporary matrices are needed to perform the algorithm, except maybe in the matrix-matrix multiplication step, the algorithm is in-place.

6 Implementation

Similarly to matrix multiplication (cf. [2]) it is beneficial to call Algorithm 4 until some "cutoff" bound and to switch to a base-case implementation (in our case Algorithm 3) once this bound is reached. We perform the switch over if the matrix fits into 4MB or in L2 cache, whichever is smaller. These values seem to provide the best performance on our target platforms.

The reason we are considering the PLS decomposition instead of either the LQUP or the PLUQ decomposition is that the PLS decomposition has several advantages over $\mathbb{F}_2$, in particular when the flat row-major representation is used to store entries.

- We may choose where to cut with respect to columns in Algorithm 4. In particular, we may choose to cut along word boundaries. For LQUP decomposition, where roughly all steps are transposed, column cuts are determined by the rank r_0.
- In Algorithm 3 rows are added instead of columns. Row operations are much cheaper than column operations in row-major representation.
- Column swaps do not occur in the main loop of either Algorithm 4 or 3, but only row swaps are performed. Column swaps are only performed at the end. Column swaps are much more expensive than row swaps (see below).
- Fewer column swaps are performed for PLS decomposition than for PLUQ decomposition since U is not compressed.

One of the major bottleneck are column swaps. In Algorithm 5 a simple algorithm for swapping two columns a and b is given with bit-level detail. In Algorithm 5 we assume that the bit position of a is greater than the bit position of b for simplicity of presentation. The advantage of the strategy in Algorithm 5 is that it uses no conditional jumps in the inner loop, However, it still requires 9 instructions per row. On the other hand, we can add two rows with $9 \cdot 128 = 1152$ entries in 9 instructions if the SSE2 instruction set is available. Thus, for matrices of size 1152×1152 it takes roughly the same number of instructions to add two matrices as it does to swap two columns. If we were to swap every column with some other column once during some algorithm it thus would be as expensive as a matrix multiplication for matrices of these dimensions.

Another bottleneck for relatively sparse matrices in dense row-major representation is the search for pivots. Searching for a non-zero element in a row can be relatively expensive due to the need to identify the bit position. However, the main performance penalty is due to the fact that searching for a non-zero entry in one column is in a row-major representation is very cache unfriendly.

Indeed, both our implementation and the implementation available in Magma suffer from performance regression on relatively sparse matrices as shown in Figure 2. We stress that this is despite the fact that the theoretical complexity of matrix decomposition is rank sensitive, that is, strictly less field operations have to be performed for low rank matrices.

Input: A – a $m \times n$ matrix
Input: a – an integer $0 \leq a < b < n$
Input: b – an integer $0 \leq a < b < n$
Result: Swaps the columns a and b in A
begin
  $M \longleftarrow$ the memory where A is stored;
  $a_w, b_w \longleftarrow$ the word index of a and b in M;
  $a_b, b_b \longleftarrow$ the bit index of a and b in a_w and b_w;
  $\Delta \longleftarrow a_b - b_b$;
  $a_m \longleftarrow$ the bit-mask where only the a_bth bit is set to 1;
  $b_m \longleftarrow$ the bit-mask where only the b_bth bit is set to 1;
  for $0 \leq i < m$ **do**
    $R \longleftarrow$ the memory where the row i is stored;
    $R[a_w] \longleftarrow R[a_w] \oplus ((R[b_w] \odot b_m) >> \Delta)$;
    $R[b_w] \longleftarrow R[b_w] \oplus ((R[a_w] \odot a_m) << \Delta)$;
    $R[a_w] \longleftarrow R[a_w] \oplus ((R[b_w] \odot b_m) >> \Delta)$;
  end
end

Algorithm 5: Column Swap

While the penalty for relatively sparse matrices is much smaller for our implementation than for Magma, it clearly does not achieve the theoretical possible performance. Thus, we also consider a hybrid algorithm which starts with M4RI and switches to PLS-based elimination as soon as the (approximated) density reaches 15%, denoted as 'M+P 0.15'.

7 Results

In Table 1 we give average running time over ten trials for computing reduced row echelon forms of dense random $n \times n$ matrices over $\mathbb{F}_2$. We compare the asymptotically fast implementation due to Allan Steel in Magma, the cubic Gaussian elimination implemented by Victor Shoup in NTL, and both our implementations. Both the implementation in Magma and our PLS decomposition reduce matrix decomposition to matrix multiplication. A discussion and comparison of matrix multiplication in the M4RI library and in Magma can be found in [2]. In Table 1 the column 'PLS' denotes the complete running time for first computing the PLS decomposition and the computation of the reduced row echelon form from PLS.

In Table 2 we give running times for matrices as they appear when solving non-linear systems of equations. The matrices HFE 25, 30 and 35 were contributed by Michael Brickenstein and appear during a Gröbner basis computation of HFE systems using PolyBoRi. The Matrix MXL was contributed by Wael Said and appears during an execution of the MXL2 algorithm [14] for a random quadratic system of equations. We consider these matrices within the scope of this work since during matrix elimination the density quickly increases and because even the input matrices are dense enough such that we expect one non-zero element per 128-bit wide SSE2 XOR on average. The columns 'M+P 0.xx' denote the hybrid algorithms which start with M4RI and switch over to PLS based echelon form computation once the density of the remaining part of the matrix reaches 15% or 20% respectively. We note that the relative performance of the M4RI and

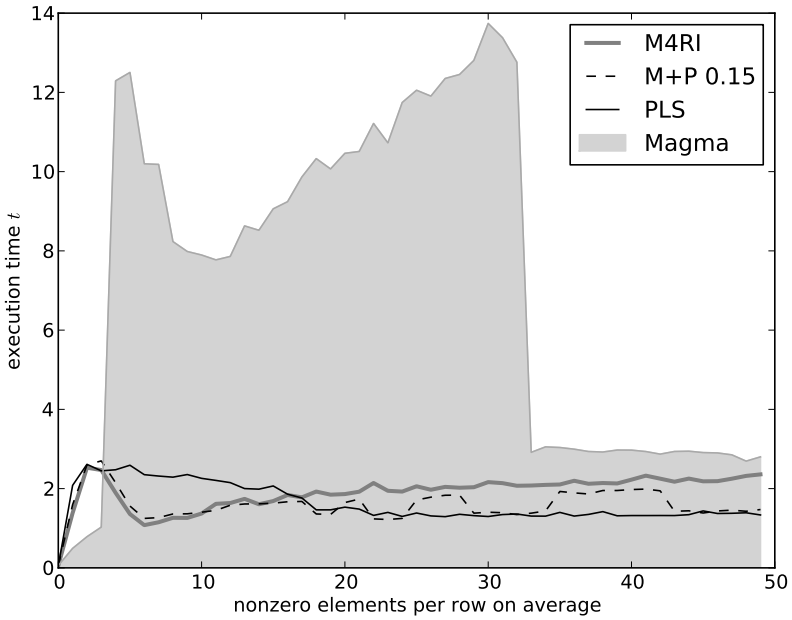

Fig. 2. Sensitivity to density for $n = 10^4$ on 2.6Ghz Opteron

	64-bit Linux, 2.6Ghz Opteron				64-bit Linux, 2.33Ghz Xeon (E5345)			
n	Magma 2.15-10	NTL 5.4.2	M4RI 20090105	PLS 20100324	Magma 2.16-7	NTL 5.4.2	M4RI 20100324	PLS 20100324
$10,000$	3.351s	18.45s	2.430s	1.452s	2.660s	12.05s	1.360s	0.864s
$16,384$	11.289s	72.89s	10.822s	6.920s	8.617s	54.79s	5.734s	3.388s
$20,000$	16.734s	130.46s	19.978s	10.809s	12.527s	100.01s	10.610s	5.661s
$32,000$	57.567s	479.07s	83.575s	49.487s	41.770s	382.52s	43.042s	20.967s
$64,000$	373.906s	2747.41s	537.900s	273.120s	250.193s	–	382.263s	151.314s

Table 1. RREF for random matrices

the PLS algorithm for these instances depends on particular machine configuration. To demonstrate this we give a set of timings for the Intel Xeon X7460 machine `sage.math`[4] in Table 2. Here, PLS always is faster than M4RI, while on a Xeon E5345 M4RI wins for all HFE examples. We note that MAGMA is not available on the machine `sage.math`. The HFE examples show that the observed performance regression for sparse matrices does have an impact in practice and that the hybrid approach does look promising for these instances.

			64-bit Fedora Linux, 2.33Ghz Xeon (E5345)				
Problem	Matrix Dimension	Density	Magma 2.16-7	M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076	3.68s	1.94s	2.09s	2.33s	2.24s
HFE 30	$19,907 \times 29,323$	0.067	23.39s	11.46s	13.34s	12.60s	13.00s
HFE 35	$29,969 \times 55,800$	0.059	–	49.19s	68.85s	66.66s	54.42s
MXL	$26,075 \times 26,407$	0.185	55.15	12.25s	9.22s	9.22s	10.22s
			64-bit Ubuntu Linux, 2.66Ghz Xeon (X7460)				
Problem	Matrix Dimension	Density		M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076		2.24s	2.00s	2.39s	2.35s
HFE 30	$19,907 \times 29,323$	0.067		27.52s	13.29s	13.78s	22.9s
HFE 35	$29,969 \times 55,800$	0.059		115.35s	72.70s	84.04s	122.65s
MXL	$26,075 \times 26,407$	0.185		26.61s	8.73s	8.75s	13.23s
			64-bit Debian/GNU Linux, 2.6Ghz Opteron)				
Problem	Matrix Dimension	Density	Magma 2.15-10	M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076	4.57s	3.28s	3.45s	3.03s	3.21s
HFE 30	$19,907 \times 29,323$	0.067	33.21s	23.72s	25.42s	23.84s	25.09s
HFE 35	$29,969 \times 55,800$	0.059	278.58s	126.08s	159.72s	154.62s	119.44s
MXL	$26,075 \times 26,407$	0.185	76.81s	23.03s	19.04s	17.91s	18.00s

Table 2. RREF for matrices from practice.

8 Acknowledgments

We would like to thank anonymous referees for helpful comments on how to improve our presentation.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Martin Albrecht, Gregory Bard, and William Hart. Algorithm 898: Efficient multiplication of dense matrices over GF(2). *ACM Trans. Math. Softw.*, 37:9:1–9:14, January 2010.

[4] Purchased under National Science Foundation Grant No. DMS-0821725.

3. Martin R. Albrecht and Gregory V. Bard. *The M4RI Library – Version 20111004*. The M4RI Team, 2011. http://m4ri.sagemath.org.
4. V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk.*, 194(11), 1970. (in Russian), English Translation in Soviet Math Dokl.
5. Gregory V. Bard. Accelerating cryptanalysis with the Method of Four Russians. Cryptology ePrint Archive, Report 2006/251, 2006. Available at http://eprint.iacr.org/2006/251.pdf.
6. Gregory V. Bard. *Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields with Applications to Cryptanalysis*. PhD thesis, University of Maryland, 2007.
7. Gregory V. Bard. Matrix inversion (or LUP-factorization) via the Method of Four Russians, in $\theta(n^3/\log(n))$ time. In Submission, 2008.
8. Wieb Bosma, John Cannon, and Catherine Playoust. The MAGMA Algebra System I: The User Language. In *Journal of Symbolic Computation 24*, pages 235–265. Academic Press, 1997.
9. Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner basis computations with Boolean polynomials. In *Electronic Proceedings of MEGA 2007*, 2007. Available at http://www.ricam.oeaw.ac.at/mega2007/electronic/26.pdf.
10. Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407, Berlin, Heidelberg, New York, 2000. Springer Verlag.
11. Jean-Charles Faugère. A new efficient algorithm for computing Gröbner basis (F4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, 1999.
12. O.H. Ibarra, S. Moran, and R. Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3:45–56, 1982.
13. Mohamed Saied Emam Mohamed, Daniel Cabarcas, Jintai Ding, Johannes Buchmann, and Stanislav Bulygin. Mxl3: An efficient algorithm for computing gröbner bases of zero-dimensional ideals. In *12th International Conference on Information Security and Cryptology (ICISC)*, 2009.
14. Mohamed Saied Emam Mohamed, Wael Said Abd Elmageed Mohamed, Jintai Ding, and Johannes Buchmann. Mxl2: Solving polynomial equations over gf(2) using an improved mutant strategy. In *Proceedings of Post-Quantum Cryptography 2008*, 2008. pre-print available at http://www.cdc.informatik.tu-darmstadt.de/reports/reports/MXL2.pdf.
15. William Stein et al. *SAGE Mathematics Software (Version 4.3)*. The Sage Development Team, 2008. Available at http://www.sagemath.org.

A Support Algorithms

Input: A – a $m \times n$ matrix
Input: r_{start} – an integer $0 \leq r_{\text{start}} < m$
Input: r_{end} – an integer $0 \leq r_{\text{start}} \leq r_{\text{end}} < m$
Input: c_{start} – an integer $0 \leq c_{\text{start}} < n$
Input: k – an integer $k > 0$
Input: T – a $2^k \times n$ matrix
Input: L – an integer array of length 2^k
begin
  for $r_{\text{start}} \leq i < r_{\text{end}}$ **do**
    $id = \sum_{j=0}^{k} A[i, c_{\text{start}} + j] \cdot 2^{k-j-1}$;
    $j \longleftarrow L[id]$;
    add row j from T to the row i of A starting at column c_{start};
  end
end

Algorithm 6: AddRowsFromTable

Input: A – a $m \times n$ matrix
Input: r_{start} – an integer $0 \leq r_{\text{start}} < m$
Input: c_{start} – an integer $0 \leq c_{\text{start}} < n$
Input: k – an integer $k > 0$
Result: Retuns an $2^k \times n$ matrix T
begin
  $T \longleftarrow$ the $2^k \times n$ zero matrix;
  for $1 \leq i < 2^k$ **do**
    $j \longleftarrow$ the row index of A to add according to the Gray code;
    add row j of A to the row i of T starting at column c_{start};
  end
  $L \longleftarrow$ integer array allowing to index T by k bits starting at column c_{start};
  return T, L;
end

Algorithm 7: MakeTable

Input: A – a $m \times n$ matrix
Input: r – an integer $0 \leq r < m$
Input: c – an integer $0 \leq c < n$
Input: k – an integer $k > 0$
Input: r_{end} – an integer $0 \leq r \leq r_{\text{end}} < m$
Result: Returns the rank $\overline{k} \leq k$ and puts the $\overline{k} \times (n - c)$ submatrix starting at $A[r, c]$ in reduced row echelon form.

```
begin
    r_s ⟵ r;
    for c ≤ j < c + k do
        found ⟵ False;
        for r_s ≤ i < r_end do
            for 0 ≤ l < j − c do // clear the first columns
                if A[i, c+l] ≠ 0 then add row r+l to row i of A starting at column c+l;
                ;
            end
            if A[i, j] ≠ 0 then // pivot?
                Swap the rows i and r_s in A;
                for r ≤ l < r_s do // clear above
                    if A[l, j] ≠ 0 then add row r_s to row l in A starting at column j;
                    ;
                end
                r_s ⟵ r_s + 1;
                found ⟵ True;
                break;
            end
        end
        if found = False then
            return j - c;
        end
    end
    return j - c;
end
```

Algorithm 8: GaussSubmatrix

```
Input: A – a m × n matrix
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Result: PLS decomposition of A. Returns the rank of A.
begin
    r, c ← 0, 0;
    while r < m and c < n do
        found ⟵ False;
        for c ≤ j < n do // search for some pivot
            for r ≤ i < m do
                if A[i, j] then found ← True and break; ;
            end
            if found then break;;
        end
        if found then
            P[r], Q[r] ⟵ i, j;
            swap the rows r and i in A;
            // clear below but preserve transformation matrix
            if j + 1 < n then
                for r + 1 ≤ l < m do
                    if A[l, j] then
                        add the row r to the row l starting at column j + 1;
                    end
                end
            end
            r, c ⟵ r + 1, j + 1;
        else
            break;
        end
    end
    for r ≤ i < m do P[i] ⟵ i ;
    ;
    for r ≤ i < n do Q[i] ⟵ i ;
    ;
    // Now compress L
    for 0 ≤ j < r do swap the columns j and Q[j] starting at row j;
    ;
    return r;
end
```

Algorithm 9: Gaussian PLS Decomposition

Input: A – a $m \times n$ matrix
Input: s_r – an integer $0 \leq s_r < m$
Input: s_c – an integer $0 \leq s_c < n$
Input: k – an integer $k > 0$
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Result: Returns the rank $\overline{k} \leq k$ and d_r – the last row considered.
Also puts the $\overline{k} \times (n - c)$ submatrix starting at (r, c) in PLS decomposition form.

```
begin
    done ⟵ all zero integer array of length k;
    for 0 ≤ r < k do
        found ⟵ False;
        for s_r + r ≤ i < m do // search for some pivot
            for 0 ≤ l < r do // clear before
                if done[l] < i then
                    if A[i, s_c + l] ≠ 0 then
                        add row s_r + l to row i in A starting at column s_c + l + 1;
                    end
                    done[l] ⟵ i;
                end
            end
            if A[i, s_c + r] ≠ 0 then
                found ⟵ True;
                break;
            end
        end
        if found = False then break;
        ;
        P[s_r + r], Q[s_r + r] ⟵ i, s_c + r;
        swap the rows s_r + r and i in A;
        done[r] ⟵ i;
    end
    d_r ⟵ max({done[i] | i ∈ {0, ..., k̄ − 1}});
    for 0 ≤ c_2 < k̄ and r + c_2 < n − 1 do // finish submatrix
        for done[c_2] < r_2 ≤ d_r do
            if A[r_2, r + c_2] ≠ 0 then
                add row r + c_2 to row r_2 in A starting at column r + c_2 + 1;
            end
        end
    end
    return r, d_r;
end
```

Algorithm 10: PLSSUBMATRIX