Solving Sparse Systems with the Block Wiedemann Algorithm
Efficient Implementation over $\text{GF}(2)$

Sonia Belaid       Sylvain Lachartre

Séminaire SALSA
2011, 8 July
Introduction

- Wiedemann’s algorithm was **introduced** by Wiedemann in 1986 [?]
- It was **extended** by Coppersmith in 1994 [?] to perform parallel computations
- It was **improved and used** by Emmanuel Thomé [?] and an international team of researchers in 2009 to break a 768 bits RSA key [?]
• Input: large sparse matrix $B$

• Computation of the linearly recurrent sequence

$$x^T \cdot B^k \cdot z$$

with $x$ and $z$ random vectors

• Computation of the minimal polynomial $\mu(X) = X \cdot \mu'(X)$ (because $B$ is singular) using Berlekamp-Massey

• Exhibition of the kernel vector $\nu$ such that

$$\nu = \mu'(B)$$
Block Wiedemann Algorithm:

- is used for the resolution of linear systems on finite fields
- takes a **sparse matrix** and exhibits a **kernel vector**
- is organized into three steps:
  1. BW1: computes a matrices series $A = (a_k)_{1 \leq k \leq L}$
  2. BW2: computes a linear generator $F_g$ of $A$
  3. BW3: exhibits a kernel vector $w$
- is probabilistic (depending on the matrix characteristics)
Algorithm 1  Wiedemann

Input:  matrix  \( B \in \mathbb{GF}(2)^{N \times N} \)
Output: vector  \( w \in \mathbb{GF}(2)^N \)

1.  \((A, z) \leftarrow BW1(B)\)  //  \(z \in \mathbb{GF}(2)^{N \times n}\)  (random)
2.  \(F_g \leftarrow BW2(A)\)  //  \(F_g \in \mathbb{GF}(2)^N\)  (linear generator)
3.  \(w \leftarrow BW3(F_g, B, A, z)\)
4.  return  \(w\)

- Input: large sparse matrix  \(B\)
- Output: kernel vector  \(w\)
- \(N\): input matrix dimension
- \(m, n\): columns number of random blocks  \(x\) and  \(z\)
BW1 : Sequence A

- The first step consists in computing the series

\[ A(X) \leftarrow \sum_k a_k \cdot X^k \quad \text{with} \quad a_k \leftarrow x^T \cdot B^k \cdot z \in GF(2)^{m \times n} \]

- Only the first \( L = \frac{N}{m} + \frac{N}{n} + 1 \) coefficients are needed

---

**Algorithm 2** BW1

**Input:** matrix \( B \)

**Output:** polynomial \( A \in GF(2)[[X]]^{m \times n} \), matrix \( z \in GF(2)^{N \times n} \)

1. \((x, z) \leftarrow \text{random matrices} \in GF(2)^{N \times m} \times GF(2)^{N \times n}\)
2. \(v \leftarrow B \cdot z\)
3. **for** \( k \leftarrow 0 \) **to** \( L \) **do**
4. \( A[k] \leftarrow x^T \cdot v\)
5. \( v \leftarrow B \cdot v\)
6. **endfor**
7. return \( A \)

---

a. \( A[k] \) represents the degree \( k \) coefficient of \( A \)
Coppersmith’s generalization of Berlekamp-Massey algorithm:

- **Initialization**: $m + n$ candidates $F_j$ for the vectorial generator

  $A(X) \cdot F(X) = G(X) + X^t \cdot E(X)$

  with $t$ the step number depending on $A$ at the beginning.

- **Iteration**: error $E(X)$ reduced by multiplying the previous equality by $P$ s.t.:

  $E[0] \cdot P = 0$

- **Termination**: generator discovered when its error is zero

  $F_g(X)$ s.t. $A(X) \cdot F_g(X) = G_g(X)$ ($E_g(X) = 0$)
BW2: Initialization

\[ d \quad \text{constant} \quad d \leftarrow \lceil \frac{N}{m} \rceil \]

\[ s \quad F^{(t_0)} \quad \text{degree} \quad \text{s.t. columns of } A[0], \ldots, A[s - 1] \text{ form a basis of } GF(2)^m \]

\[ t \quad \text{step number} \quad \text{starting with } t \leftarrow s \]

\[ \text{end} \quad \text{stop condition} \quad \text{end } \leftarrow 0 \]

\[ \delta \quad \text{quantity} \quad \text{s.t. } \delta(f, g) = \max(\deg f, 1 + \deg g) \]

\[ \Delta \quad \text{degrees bounds} \]

\[ \forall j, \quad \delta(F_j, G_j) \leq \Delta_j \]

\[ F \quad \text{generator candidates} \]

\[ \left( l_n \mid X^{s-i_1} \cdot r_1 \quad \ldots \quad X^{s-i_m} \cdot r_m \right) \]
computation of polynomial $P$ of degree 1

$$E[0] \cdot P = 0$$

the condition becomes

$$A \cdot F(t) \cdot P = G(t) \cdot P + X^t \cdot E(t) \cdot P$$

$$A \cdot F(t+1) = G(t+1) + X^{(t+1)} \cdot E(t+1)$$

update of $F$ and $E$

$$F^{(t+1)} = F^{(t)} \cdot P$$

$$E^{(t+1)} = E^{(t)} \cdot P \cdot \frac{1}{X}$$
Algorithm 3 PMatrix

Input: matrix \( E_0 \in \mathbb{GF}(2)^{m \times (m+n)} \), tab \( \Delta \in \mathbb{N}^{m+n} \)
Output: polynomial \( P \in \mathbb{GF}(2)[X]^{(m+n) \times (m+n)} \)

/* Sort */
1. \((P[0], \Delta) \leftarrow \text{Sort}(\Delta)\)
2. \(E_0 \leftarrow E_0 \cdot P[0]\)
   /* Gaussian Elimination */
3. \((E_0, P[0]) \leftarrow \text{GaussElimE0}(E_0, P[0])\)
   /* Elimination of non zero columns */
4. for \( i \leftarrow 1 \) to \( m + n \) do
5.    if \((E_0; ^a \text{ is not null})\) then
6.       \(P \leftarrow \text{MultByX}(P, i)\)
7.       \(\Delta_i \leftarrow \Delta_i + 1\)
8.    endif
9. endfor
10. return \( P \)

a. \( E_0;^a \) represents the \( i^{th} \) column of \( E_0 \)
BW2: Termination

- mean value $\Delta$ of $\Delta_j$ coefficients increases by $\frac{m}{m+n} = \frac{1}{2}$

$$t - \overline{\Delta} = (t - s) \cdot \frac{n}{m + n}$$

- for $t = s + \lceil \frac{m}{m+n}d \rceil$, $t - \overline{\Delta} \geq d$ $\Rightarrow$ $\exists j$, $t - \Delta_j \geq d$

- according to theorem 8.6 in [?]

$$\exists j, \ t - \Delta_j \geq d \Rightarrow E_j(X) = 0$$

Stop Condition

$F_j$ generator if $t - \Delta_j \geq d$ (before $s + \lceil \frac{m}{m+n}d \rceil$ steps)
Algorithm 4  BW2

Input: polynomial  $A$
Output: polynomial  $F \in \text{GF}(2)[X]^n$

/* Initialization */
1. $(d, s, t, \text{end}, \Delta, F) \leftarrow \text{Init}(A)$
2. $E \leftarrow \text{Error}(A, F, t)$

/* Iteration */
3. while (end = FALSE) do
4.  $(P, \text{end}, g) \leftarrow \text{PMatrix}(E[0], \Delta)$
5.  $F \leftarrow F \cdot P$
6.  $E \leftarrow E \cdot P \cdot \frac{1}{X}$
7.  $t \leftarrow t + 1$
8. endwhile

/* Termination */
9. return $F_g$
Kernel vector exhibition:

- coefficient $j$ of $A(X) \cdot F_g(X)$

$$(A \cdot F_g)[j] = x^T B^{j-\text{deg } F_g+1} \cdot v \quad \text{with} \quad v = \sum_{i=0}^{\text{deg } F_g} B^{\text{deg } F_g-i} \cdot z \cdot F_g[i]$$

- by construction

$$(A \cdot F_g)[j] = 0 \quad \text{for} \quad j \geq \delta(F_g)$$

- $B^{\delta(F)-\text{deg } F_g+1} \cdot v$ orthogonal to all vectors $(B^T)^i \cdot x_k$

- if these vectors form a basis of $K^N$

$$B \cdot (B^{\delta(F_g)-\text{deg } F_g} \cdot v) = 0$$

Kernel Vector
BW3 Algorithm

Algorithm 5  BW3

Input: polynomial $F_g$, matrix $B$, polynomial $A$, matrix $z$

Output: vector $w$

1. $w \leftarrow 0_N$
2. for $i \leftarrow 0$ to $\deg F_g$ do $w \leftarrow B \cdot w + z \cdot F_g[i]$ endfor
3. if ($w \neq 0_N$) then
4. for $k \leftarrow 0$ to $\delta(F_g)^a$ do
5. $u \leftarrow B \cdot w$
6. if $u = 0$ then return $w$ endif
7. $w \leftarrow u$
8. endfor
9. endif
10. return  FAILED

a. $\delta(F_g) = \max(\deg F_g, \deg (A \cdot F_g) + 1)$
Outline
RSA Attack

2009

Researchers broke a **768 bits** RSA key using **NFS** [?]}

- **NFS (Number Field Sieve)**: factorization of large numbers
  - using Wiedemann’s algorithm
  - input binary matrix:
    - 200 millions of rows
    - 150 non zeros elements by rows
  - 98 days of computations on a cluster of 576 cores
Library M4RI [?]

- Linear algebra library in C language focused on dense matrices over GF(2)
- Created by Gregory Bard
- Now maintained by Martin Albrecht

- We use it to compute operations on dense matrices
  - BW1: to compute the products of $x \cdot (B^k \cdot z)$
  - BW2: to perform all the operations involving blocks
  - BW3: to compute the products $(B \cdot z) \cdot F$
Sparse Matrix

- Structure for the input sparse matrix:
  - dimensions \( m \), \( n \)
  - number of non zeros \( nb \)
  - number of non zeros by rows \( sz \)
  - positions of non zeros \( pos \)
  - rows structures with the same characteristics \( l \)

### Matrix Example

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

- \( m,n = 5, \ nb=6 \)
- \( sz = [1,0,2,2,1] \)
- \( pos = [0,1,4,2,3,4] \)
- \( l[0] : nb=1, pos=[0] \)
- \( l[1] : nb=0, pos=[] \) ...
Sparse Dense Operations

Algorithm 6 Sparse Dense Product

Input: matrix $B$, matrix $v$
Output: matrix $res \leftarrow B \cdot v$

1. $res \leftarrow 0$
2. $p \leftarrow B_{pos}$
3. for $i \leftarrow 0$ to $N$ do
4. \hspace{1em} for $j \leftarrow 0$ to $B_{nb}$ do
5. \hspace{2em} $res_i \leftarrow res_i \oplus v_p$
6. \hspace{1em} $p \leftarrow p + 1$
7. \hspace{1em} endfor
8. endfor
9. return $res$

 Complexity: linear with the number of non zeros.
Counting Sort

array to sort

\[ \Delta = 10 \ 9 \ 9 \ 8 \ 12 \ 8 \ 9 \ 8 \ 12 \ 8 \]

counting array

<table>
<thead>
<tr>
<th>values</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>occurrences</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

sorted array

\[ \text{CountingSort}(\Delta) = 8 \ 8 \ 8 \ 8 \ 9 \ 9 \ 9 \ 10 \ 12 \ 12 \]

Complexity

Linear in \( O(m + n) \) with \( m + n \) the size of \( \Delta \)
Practical Optimization: Operations Save

Loop

1. for \( i \leftarrow 0 \) to \( a \times b \) do

Product

BW1: \( \forall k \in [1..L], \quad v \leftarrow B^k \cdot z \)

BW3: \( \forall k \in [1..\text{deg } F_g], \quad v \leftarrow B^k \cdot z \)

Loop

1. constant \( \text{tmp} \leftarrow a \times b \)
2. for \( i \leftarrow 0 \) to \( \text{tmp} \) do

Product

BW1: \( \forall k \in [1..L'], \quad v \leftarrow Bz_{\text{save}}[k] \leftarrow B^k \cdot z \)

BW3: \( \forall k \in [1..\text{deg } F_g], \quad v \leftarrow Bz_{\text{save}}[k] \)
Practical Optimization: Reduction of the Number of Tests

1. for $i \leftarrow 1$ to $n$ do
2. \text{if} ($i < \frac{n}{2}$) then Action1
3. \text{else} Action2 \text{ endif}
4. endfor

1. for $i \leftarrow 1$ to $\frac{n}{2}$ do
2. Action1
3. endfor
4. for $i \leftarrow \frac{n}{2}$ to $n$ do
5. Action2
6. endfor
Practical Optimization : Adaptation of M4RI Functions

◆ removing initial tests
  ● $A \cdot B$ : number of $A$'s columns = number of $B$'s rows
  ● equality to zero

◆ adapting to my matrices constant dimensions
  ● $BW1$ : products of matrices $(64 \times N) \times (N \times 64)$
  ● $BW2$ : products of matrices $(64 \times 128) \times (128 \times 128)$
  ● global : most matrices whose dimensions are multiples of the size a machine word

◆ improving some functions
  ● $\text{mzd\_is\_zero}$ : stopping at the first non zero word
  ● $\text{mzd\_transpose}$ : clearing before transposing

1. released 2009-05-12
Practical Optimization : Parallelization

- **BW1** : parallel
- **BW2** : sequential $\Rightarrow$ parallelization
- **BW3** : sequential $\Rightarrow$ operations save
Practical Optimization : Parallelization BW1

- sparse dense product : \( B \cdot (B^k \cdot z), \ k \in [1..L] \)
  - each thread dedicated to a number of rows
  - depending on number of non zeros by rows

![Figure: Representation of a 2688 × 2688 matrix](image)

- computations of \( a_k : a_k \leftarrow x \cdot (B^k \cdot z), \ k \in [1..L] \)
  - each thread dedicated to a number of \( a_k \)
  - same number of coefficients for each thread
polynomials products $F \cdot P$ and $E \cdot P$
- each thread dedicated to a number of coefficients of $F$ or $E$
- same number of coefficients for each thread

product of $P$ by $X$
- each thread dedicated to a number of columns of $P$
- same number of columns for each thread
Practical Optimization : Parallelization : CPU Use

1 thread

3 threads

6 threads
Execution times according to the number of threads

<table>
<thead>
<tr>
<th>Dim</th>
<th>δ(%)</th>
<th>1th</th>
<th>2th</th>
<th>3th</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
</tr>
</thead>
<tbody>
<tr>
<td>2688</td>
<td>1.3</td>
<td>0.3</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>15360</td>
<td>1.5</td>
<td>11.6</td>
<td>7.7</td>
<td>6.2</td>
<td>5.2</td>
<td>4.6</td>
<td>4.1</td>
<td>3.9</td>
<td>3.6</td>
</tr>
<tr>
<td>15360</td>
<td>5.2</td>
<td>22.3</td>
<td>12.5</td>
<td>9.6</td>
<td>7.7</td>
<td>7.1</td>
<td>6.9</td>
<td>6.4</td>
<td>5.9</td>
</tr>
<tr>
<td>15360</td>
<td>8.2</td>
<td>32.0</td>
<td>18.5</td>
<td>14.2</td>
<td>11.1</td>
<td>10.4</td>
<td>9.5</td>
<td>9.2</td>
<td>8.7</td>
</tr>
</tbody>
</table>
Theoretical Optimizations: Termination Tests

**Stop Condition**

\[ \exists j \in [1..m + n], \quad t \geq \Delta_j + d \]

\[ \Delta_{\text{min}}^{(t)} + d \text{ and } t \]

\[ s + d \]

\[ s \]

\[ \rightarrow \text{steps} \]

**New Stop Condition**

\[ \exists j \in [1..\frac{m + n}{2} + 1] \quad \text{s.t.} \quad \Delta_j^{(t)} = \Delta_j^{(t-1)}, \quad t \geq \Delta_j + d \]
Theoretical Optimizations : Error Degree Update (1/2)

- for large matrices
  - high error degree in BW2
  - product $E \cdot P$ expensive

- but only $E[0]$ used in each step

- knowing the number of reminding steps, we can:
  - determine the **useful degree** of $E$
  - decrease the number of coefficients to update
Theoretical Optimizations : Error Degree Update (2/2)

◆ Stop condition : \( t_f = \min(\Delta)^{(t_f)} + d \)

◆ Worst case
  - \( \min(\Delta) \) increases by 0.5 at each step
  - from step \( t \), we still have \( \delta_e^{(t)} \) iterations :

\[
  t + \delta_e^{(t)} = \min(\Delta)^{(t)} + \frac{\delta_e^{(t)}}{2} + d \\
  \Rightarrow \delta_e^{(t)} = 2 \cdot (\min(\Delta)^{(t)} + d - t)
\]

\[
  \Rightarrow \begin{cases} 
  \delta_e^{(t+1)} = \delta_e^{(t)} & \text{if } \min(\Delta)^{(t+1)} = \min(\Delta)^{(t)} + 1 \\
  \delta_e^{(t+1)} = \delta_e^{(t)} - 2 & \text{otherwise}
\end{cases}
\]

Error Degree Bound

\[
  \forall t, \quad \deg E^{(t)} \leq \delta_e^{(t)}
\]
Theoretical Optimizations: Candidates Degree Update (1/2)

- for large matrices
  - candidates degree increases by 1 at each step (becoming high)
  - product $F \cdot P$ becomes expensive

- but only $F_g$ (linear generator) will be kept

- knowing the number of reminding steps, we can:
  - determine the useful degree of $F$ (that is the one of $F_g$)
  - limit the number of coefficients to update
Theoretical Optimizations: Candidates Degree Update (2/2)

- $F$ degree is limited to $\delta_f$ such that

$$\delta_f^{(t)} = \min(\Delta)^{(t)} + r^{(t)}$$

with $r^{(t)} = 2 \cdot (\min(\Delta)^{(t)} + d - t)$ (maximum reminding steps)

$$\Rightarrow \delta_f^{(t)} = \min(\Delta)^{(t)} + 2 \cdot (\min(\Delta)^{(t)} + d - t)$$

$$\Rightarrow \begin{cases} 
\delta_f^{(t+1)} = \delta_f^{(t)} + 1 & \text{if } \min(\Delta)^{(t+1)} = \min(\Delta)^{(t)} + 1 \\
\delta_f^{(t+1)} = \delta_f^{(t)} - 2 & \text{otherwise}
\end{cases}$$

Candidates Degree Bound

$$\forall t, \quad \deg F^{(t)} \leq \delta_f^{(t)}$$
<table>
<thead>
<tr>
<th>Dim</th>
<th>( \delta(%) )</th>
<th>BW1</th>
<th>BW2</th>
<th>BW3</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>738</td>
<td>7.00</td>
<td>10%</td>
<td>90%</td>
<td>0%</td>
<td>0''100</td>
</tr>
<tr>
<td>3422</td>
<td>7.57</td>
<td>20%</td>
<td>73%</td>
<td>7%</td>
<td>0''450</td>
</tr>
<tr>
<td>10000²</td>
<td>5.97</td>
<td>46%</td>
<td>43%</td>
<td>11%</td>
<td>1''960</td>
</tr>
<tr>
<td>27000¹</td>
<td>2.20</td>
<td>60%</td>
<td>28%</td>
<td>12%</td>
<td>14''060</td>
</tr>
<tr>
<td>73674</td>
<td>0.61</td>
<td>65%</td>
<td>30%</td>
<td>5%</td>
<td>1'12''760</td>
</tr>
<tr>
<td>93913</td>
<td>0.20</td>
<td>51%</td>
<td>41%</td>
<td>8.3%</td>
<td>1'38''710</td>
</tr>
</tbody>
</table>

**Figure:** Performances on different matrix sizes

**Computer Details:** PC Xeon (64 bits)

- 8 Go RAM
- 2.40 GHz
- 8 processors Intel(R) Xeon(R) CPU

2. from Faugère, 2011-06-30
Comparisons with Magma and Sage

<table>
<thead>
<tr>
<th>Matrices</th>
<th>Our Implementation</th>
<th>Speed up (1th)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dim</td>
<td>δ</td>
<td>8 th</td>
</tr>
<tr>
<td>2688</td>
<td>1.27%</td>
<td>0’’200</td>
</tr>
<tr>
<td>15360</td>
<td>1.54%</td>
<td>3’’580</td>
</tr>
<tr>
<td>15360</td>
<td>5.20%</td>
<td>5’’870</td>
</tr>
<tr>
<td>15360</td>
<td>8.21%</td>
<td>8’’690</td>
</tr>
</tbody>
</table>

Figure: Temporal comparisons with Magma^4 and Sage^5

3. not enough memory (> 8 Go)
4. Magma version 2.17-1, released 2010-12-02
Summary

- Efficient implementation of the Block Wiedemann algorithm over $GF(2)$ in C language
- Practical and theoretical optimizations
- Encouraging results compared to existing methods

Further Work

- *Algebraic Cryptanalysis of HFE Cryptosystems Using Gröbner Bases* [?] using Block Wiedemann algorithm
- Use of Gröbner bases algorithm [?]
- Comparisons with LinBox
- Work on applications
- Open source?