LARGE SPARSE MATRIX-VECTOR MULTIPLICATION OVER FINITE FIELDS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

CEYDA MANGIR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

FEBRUARY 2019

Approval of the thesis:

LARGE SPARSE MATRIX-VECTOR MULTIPLICATION OVER FINITE FIELDS

submitted by **CEYDA MANGIR** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Cryptography Department, Middle East Technical University** by,

Prof. Dr. Ömür Uğur
Director, Graduate School of **Applied Mathematics** _____

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** _____

Assoc. Prof. Dr. Murat Cenk
Supervisor, **Cryptography, METU** _____

Assoc. Prof. Dr. Murat Manguoğlu
Co-supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Ferruh Özbudak
Mathematics Department, METU _____

Assoc. Prof. Dr. Murat Cenk
Institute of Applied Mathematics, METU _____

Assoc. Prof. Dr. Ali Doğanaksoy
Mathematics Department, METU _____

Assoc. Prof. Dr. Sedat Akleylek
Computer Engineering Department, 19 Mayıs University _____

Assist. Prof. Dr. Oğuz Yayla
Mathematics Department, Hacettepe University _____

Date: _____

iv

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: CEYDA MANGIR

Signature   :

ABSTRACT

LARGE SPARSE MATRIX-VECTOR MULTIPLICATION OVER FINITE FIELDS

Mangır, Ceyda

Ph.D., Department of Cryptography

Supervisor : Assoc. Prof. Dr. Murat Cenk

Co-Supervisor : Assoc. Prof. Dr. Murat Manguoğlu

February 2019, 76 pages

Cryptographic computations such as factoring integers and computing discrete logarithms require solving a large sparse system of linear equations over finite fields. When dealing with such systems iterative solvers such as Wiedemann or Lanczos algorithms are used. The computational cost of both methods is often dominated by successive matrix-vector products. In this thesis, we introduce a new algorithm for computing a large sparse matrix-vector multiplication over finite fields. The proposed algorithm is implemented and its performance is compared with a classical method. Our algorithm exhibits a significant improvements between 34% and 77%.

Keywords: Discrete Logarithms, Sparse Matrix-Vector Multiplication

ÖZ

SONLU CİSİMLER ÜZERİNDE BÜYÜK SEYREK MATRİS-VEKTÖR ÇARPIMI

Mangır, Ceyda

Doktora, Kriptografi Bölümü

Tez Yöneticisi   : Doç. Dr. Murat Cenk

Ortak Tez Yöneticisi   : Doç. Dr. Murat Manguoğlu

Şubat 2019, 76 sayfa

Çarpanlara ayırma ve ayrık logaritma hesaplama gibi kriptografik işlemler sonlu cisimler üzerinde büyük ve seyrek denklem sistemlerinin çözümünü gerektirmektedir. Bu işlemler için Wiedemann ve Lanczos gibi yinelemeli yöntemler benimsenmektedir. Her iki algoritmada da matris-vektör çarpımlarının baskın olduğu hesaplamalar kullanılmaktadır. Bu tezde, sonlu cisimler üzerinde büyük seyrek matris-vektör çarpma işlemine yönelik bir algoritma önerilmiştir. Söz konusu algoritmanın performansı klasik yöntemle kıyaslanmış ve $\%34$ ile $\%77$ arasında hızlanma sağlanmıştır.

Anahtar Kelimeler: Ayrık Logaritma, Seyrek Matris-Vektör Çarpımları

To My Family

ACKNOWLEDGMENTS

My deep gratitude goes first to my thesis supervisor Assoc. Prof. Dr. Murat Cenk for his guidance, encouragement and valuable advices during the development and preparation of this thesis. His willingness to give his time and to share his experiences has brightened my path.

I would also like to extend my gratitude to my co-supervisor Assoc. Prof. Dr. Murat Manguoğlu for his constructive advices and ingenious suggestions.

In special, I am indebted to my parents, Ferda and Afet, and my brother, Cem, for love, guidance and inspiration throughout my life.

I am deeply grateful to my husband, Görkem, for encouraging me in all of my pursuits and giving me the extra strength throughout my study.

And the last word goes for Begüm, my daughter, who has been the light of my life since she was born and who has given me the motivation to complete this thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

The problem of solving a large sparse system of linear equations over finite fields arises in different areas such as computer algebra, number theory, and cryptography. In cryptographic computations, factoring integers and solving discrete logarithms over finite fields are two well-known problems that require solving a large sparse system of linear equations [1, 18, 31, 35, 36]. Two of the widely used methods for solving large sparse systems of linear equations in cryptographic computations are the Wiedemann [42] and the Lanczos algorithms [27] and their block variants [2, 30]. These methods compute the iterative matrix-vector products $A^t \boldsymbol{v}$ for $t = 1, \ldots, m$ where A is an $n \times n$ matrix, $\boldsymbol{v}$ is an n dimensional column vector, and m is a positive integer.

1.1   Motivation

Matrices that are encountered when solving discrete logarithms or factoring integers are usually very large. Therefore, iterative matrix-vector products used in Wiedemann and Lanczos algorithms can take considerable time. Fortunately, the sparse structure of these matrices let us take advantage of sparse matrix storage formats and reduce

not only the size of the memory needed for the matrix but also irregular access to matrix entries. Besides, matrix-vector multiplication over large finite fields uses multiple precision arithmetic which is considerably slower than the single precision arithmetic using numbers that are limited by the size of the processor register. Although simple algorithms exist for multiple precision addition, subtraction and comparison, the multiplication and modular reduction are more complex. Reducing the number of modular reductions enhances the performance of matrix-vector multiplication over finite fields significantly [8, 10].

1.2   Objectives

In this thesis, we propose a new algorithm for large sparse matrix-vector multiplications over finite fields. The purpose of the algorithm is to reduce the number of big integer multiplications and modular reductions. We mainly target discrete logarithm computations, particularly, index calculus algorithms for large prime fields. The proposed algorithm consists of two stages, namely, the preprocessing stage and the permutational matrix-vector multiplication stage. In the first part of our work, we describe the two stages of the algorithm along with pseudocodes, examples and a brief analysis. In the second part, an implementation of the algorithm is introduced and the performance results with the comparison to a classical method are given.

1.3   Organization

The rest of the thesis is organized as follows: Chapter 2 gives the preliminaries to the subject. In Chapter 3, The discrete logarithm problem is described and the algo-

rithms to solve discrete logarithm problem are investigated. Chapter 4 is about the large sparse linear systems. The proposed algorithm and its analysis are explained in Chapter 5. Chapter 6 is about the implementation details and the performance results of the proposed algorithm. Finally, the conclusion is presented in Chapter 7.

CHAPTER 2

PRELIMINARIES

In this chapter, the preliminaries for the proposed algorithm are given. First, the linear algebra background, with an introduction to matrices and vector spaces, is presented. Second, the time complexities of algorithms and order notations will be introduced. Sorting algorithms are also explained for easy comprehension of the proposed algorithm.

2.1 Linear Algebra

The linear algebra is one of the main topics in applied mathematics but usually is studied for the field of real and complex numbers. This section is dedicated to the basic concepts of linear algebra in order to show that the techniques remain valid over an arbitrary field.

2.1.1 Matrices

$m \times n$ matrix A over a field $\mathbb{F}$ is an arrangement of mn elements of $\mathbb{F}$ in a rectangular array with m rows and n columns.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

If most of the elements of a matrix is zero then it is called a *sparse matrix*. By contrast, if most of the elements is nonzero then it is considered as a *dense matrix*. The ratio of the number of zero elements to the total number of elements is said to be the *sparsity* of the matrix and the ratio of the number of nonzero elements to the total number of elements is called the *density* of the matrix. If m is equal to n then the matrix A is called a *square matrix*. The set which consists of a_{ii}, for $i = 1, 2, \cdots, n$, forms the *main diagonal* of a square matrix. If the entries of a square matrix are symmetric with respect to main diagonal then the matrix is called *symmetric matrix*. When the row and column indices of A are switched, an $n \times m$ matrix *A-transpose* is formed and it is denoted by A^T. The rows of A becomes the columns of A^T, that is

$$A^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}.$$

A matrix that consists of a single column with n elements is called a *column vector*

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}.$$

Similarly, a matrix with a single row of n elements is called a *row vector*. A row vector is a transpose of a column vector and vice versa

$$\begin{bmatrix} a_1 & a_2 & \ldots & a_n \end{bmatrix}^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}.$$

Throughout the thesis, boldface notation is used to denote all the vectors and they are considered as column vectors unless otherwise stated.

2.1.2 Vector Spaces

Let $\mathbb{F}$ be a field and the elements of $\mathbb{F}$ are called as *scalars*. A vector space V over $\mathbb{F}$ is a set of vectors $\{v_1, v_2, \cdots, v_n\}$ where addition of two vectors and multiplication of a vector by a scalar satisfy the following properties.

1. V is an Abelian group under vector addition.

2. Distributive law is hold, that is, for any vectors $\boldsymbol{u}, \boldsymbol{v} \in V$ and any scalars $\alpha, \beta \in \mathbb{F}$

$$\alpha(\boldsymbol{u} + \boldsymbol{v}) = \alpha\boldsymbol{u} + \beta\boldsymbol{v}, \tag{2.1}$$

$$(\alpha + \beta)\boldsymbol{u} = \alpha\boldsymbol{u} + \beta\boldsymbol{u}. \tag{2.2}$$

3. Associative law is also hold, i.e., for any vector $\boldsymbol{v}$ and any scalars $\alpha, \beta \in \mathbb{F}$

$$(\alpha\beta)\boldsymbol{v} = \alpha(\beta\boldsymbol{v}). \tag{2.3}$$

The zero element of V is called the *origin* of V and denoted as $\mathbf{0}$.

The *inner product* of two vectors $\boldsymbol{u} = \begin{bmatrix} \alpha_1 & \alpha_2 & \ldots & \alpha_n \end{bmatrix}^T$ and $\boldsymbol{v} = \begin{bmatrix} \beta_1 & \beta_2 & \ldots & \beta_n \end{bmatrix}^T$ is a scalar defined as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \alpha_1\beta_1 + \alpha_2\beta_2 + \cdots + \alpha_n\beta_n \tag{2.4}$$

If the inner product of two vectors is zero then they are said to be *orthogonal*. A nonzero vector over a finite field $\mathbb{F}_q$ can be orthogonal to itself.

If a vector $\boldsymbol{v} \in V$ can be written as

$$\boldsymbol{v} = \alpha_1\boldsymbol{v_1} + \alpha_2\boldsymbol{v_2} + \cdots + \alpha_k\boldsymbol{v_k} \tag{2.5}$$

for any set of scalars $\alpha_1, \alpha_2, \cdots, \alpha_k \in \mathbb{F}$ then it is said to be a *linear combination* of the set of the vectors $\{\boldsymbol{v_1}, \boldsymbol{v_1}, \cdots, \boldsymbol{v_k}\}$. If there exist elements $\alpha_1, \alpha_2, \cdots, \alpha_k$, not all zero, such that

$$\alpha_1\boldsymbol{v_1} + \alpha_2\boldsymbol{v_2} + \cdots + \alpha_k\boldsymbol{v_k} \neq 0 \tag{2.6}$$

then $\{\boldsymbol{v_1}, \boldsymbol{v_2}, \cdots, \boldsymbol{v_k}\}$ are said to be *linearly independent*. If every vector $\boldsymbol{v} \in V$ can be expressed as at least one linear combination of a set of vectors $\{\boldsymbol{v_1}, \cdots, \boldsymbol{v_k}\} \in V$, then this set of vectors is said to *span* the vector space V. The number of linearly independent vectors in a set that spans a vector space V is called the *dimension* of V. A set of k linearly independent vectors that spans a k-dimensional vector space is called a *basis* of the space.

2.1.3   Matrix Equations

Matrices are convenient tools for working with systems of linear equations. If A is an $m \times n$ matrix and $\boldsymbol{v}$ is an n-vector then the elementary method for multiplying A by

$\boldsymbol{v}$ computes the inner product of the ith row vector of A and $\boldsymbol{v}$ as follows

$$A\boldsymbol{v} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + \ldots + a_{1n}v_n \\ a_{21}v_1 + a_{22}v_2 + \ldots + a_{2n}v_n \\ \vdots \\ a_{m1}v_1 + a_{m2}v_2 + \ldots + a_{mn}v_n \end{bmatrix} \tag{2.7}$$

Therefore, a system of m linear equations with n unknown variables $x_1, x_2, \cdots, x_n$ given as

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \tag{2.8}$$

is equivalent to the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$ where $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix}^T$ and $\boldsymbol{b} = \begin{bmatrix} b_1 & b_2 & \ldots & b_m \end{bmatrix}^T$

$$A\boldsymbol{v} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \boldsymbol{b}. \tag{2.9}$$

2.2 Algorithms and Complexity

The performance of an algorithm is measured by its running time. However, because of the run-time conditions can differ from architecture to architecture, the theoretical cost usually cannot be characterized easily and precisely. Therefore it is usually expressed with order notations. Order notations compare the rates of growth of functions.

It should be noted that the notation $\log n$ is used for binary logarithm (i.e. $\log_2 n$) throughout the thesis.

2.2.1 Order Notations and Time Complexities

Big-Oh Notation. A function $f(n)$ is of the order of $g(n)$, denoted $f(n) = O(g(n))$, if there exist a positive real constant c and a non-negative integer n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$. $f(n) = O(g(n))$ implies that f does not grow faster than g up to multiplication by a positive constant value. For example, if $f = 5n^3 + 7n^2 + 1$ then $f = O(n^3)$. Commonly encountered time complexities are listed below.

1. If $f(n) = O(1)$ then it is said to be *constant time*.

2. If $f(n) = O(\log n)$ then it is said to be *logarithmic time*.

3. If $f(n) = O(n)$ then it is said to be *linear time*.

4. If $f(n) = O(n^k)$ then it is said to be *polynomial time*.

5. If $f(n) = O(2^k)$ for some constant k then it is said to be *exponential time*.

Big-Omega Notation. Big-Omega indicates lower bound. If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$.

Big-Theta Notation. If f and g exhibit the same rate of growth then $f(n) = \Theta(g(n))$. This implies $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Small-o Notation. We say $f(n) = o(g(n))$ if g is an upper bound on f but it is not tight. For example, for any non-negative integer d and real constant $a > 1$, $n^d =$

$o(a^n)$, i.e. any exponential function asymptotically grows faster than any polynomial function.

Small-omega Notation. If $f(n) = o(g(n))$ then $g(n) = \omega(f(n))$.

L-Notation. In order to express the complexity of number theoretical problems such as integer factoring or solving discrete logarithms, the L-notation is used. It is defined as

$$L_n[\alpha, c] = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}} \tag{2.10}$$

where c is a positive constant, and α is a constant $0 \leq \alpha \leq 1$. When $\alpha = 0$ then

$$L_n[0, c] = e^{(c+o(1)) \ln \ln n} = (\ln n)^{(c+o(1))} \tag{2.11}$$

is a polynomial function of $\ln n$. When $\alpha = 1$ then

$$L_n[\alpha, c] = e^{(c+o(1))(\ln n)} = n^{c+o(1)} \tag{2.12}$$

is a fully exponential function of $\ln n$. If $0 < \alpha < 1$ then the function is subexponential of $\ln n$. When n is clear from the context $L_n[\alpha, c]$ can be abbreviated as $L[\alpha, c]$.

2.2.2 Sorting Algorithms

As we will see in the following sections, the proposed algorithm employs a sorting algorithm in the preprocessing stage. Therefore, sorting algorithms are reviewed in this section.

A sorting algorithm is an algorithm that rearranges the elements of a list according to a comparison operator. The sorting algorithms can be classified as comparison

11

and non-comparison based algorithms. Comparison based sorting algorithms work by comparing values. If the array to be sorted has no structural properties to exploit than one of these algorithms can be preferred. The most well-known comparison sort algorithms are selection sort, insertion sort, merge sort and quick sort [4]. At worst case, an array of size n can be sorted in $O(n^2)$ time by using selection, insertion or quick sort and in $O(n \log n)$ time by using merge sort. However quick sort can have $O(n \log n)$ time on average. If the size of the array to be sorted is relatively large when compared to available memory then the memory usage becomes more critical.

However, under some special conditions having to do with the values to be sorted, it is possible to implement non-comparison based algorithms that have linear complexity $O(n)$. Three such algorithms are bucket sort, counting sort and radix sort [4]. If the values to be sorted are evenly distributed in some range and it is possible to divide the range into n equal parts, each of size k then bucket sort can be used. When the array contains values each of a sequence of length k, the radix sort becomes appropriate. If the values are integers less than some upper bound k and may have some associated information such as reordered indices to be carried along with them then the counting sort is more applicable.

2.2.2.1 Bucket Sort

Bucket sort is a divide and conquer algorithm. First, a finite number of buckets are created by partitioning the array. Then a sorting algorithm which can be either a different algorithm or recursively implemented bucket sort is applied on each bucket. The pseudocode of the algorithm is given in Algorithm 1 and an example is provided in Example 1.

12

Algorithm 1 The Bucket Sort Algorithm

    Input: n dimensional array A

    Output: Sorted array B

1: **for** $i \leftarrow 1, n$ **do**

2:     Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

3: **end for**

4: **for** $i \leftarrow 0, n-1$ **do**

5:     Sort list $B[i]$ using the appropriate sorting algorithm

6: **end for**

7: Concatenate the lists $B[0], B[1], \cdots, B[n-1]$ together in order

---

Example 1: Sorting the array $A = \{56, 29, 8, 5, 24, 48, 77, 62\}$ using bucket sort.

1. Create an empty buckets array B of size $10n$ for $n = 8$.

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]

2. Go through array A by putting each value in the appropriate bucket.

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]
8, 5		29, 24		48	56	62	77

3. Sort each bucket.

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]
5, 8		24, 29		48	56	62	77

4. Concatenate the buckets into a single array $B = \{5, 8, 24, 29, 48, 56, 62, 77\}$.

If n input values are distributed evenly in some range min to max then there should be one value per bucket and the time to insert each value into a bucket will be $O(1)$.

If the values are not evenly distributed then all the values go into the same bucket and the worst case time complexity for inserting the values into the buckets will be $O(n^2)$. Passing through the buckets array in order to place the sorted values into the output array requires $O(n)$ time. So the total time is $O(n)$ if the values are evenly distributed and, $O(n^2)$, if they are not evenly distributed.

2.2.2.2   Counting Sort

Counting sort algorithm can be used for sorting a collection of small integers $k \leq n$. The pseudocode for the algorithm is given in Algorithm 2.

Algorithm 2 The Counting Sort Algorithm

    Input: n dimensional array A

    Output: Sorted array B

  1: $C \leftarrow$ empty count array

  2: **for** $i \leftarrow 1, n$ **do**

  3:     $C[A[i]] \leftarrow C[A[i]] + 1$

  4: **end for**

  5: **for** $i \leftarrow 1, k$ **do**

  6:     $C[i] \leftarrow C[i] + C[i - 1]$

  7: **end for**

  8: **for** $i \leftarrow n, 1$ **do**

  9:     $B[C[A[i]]] \leftarrow A[i]$

  10:     $C[A[i]] \leftarrow C[A[i]] - 1$

  11: **end for**

Counting sort algorithm uses an auxiliary array to store the number of counted ele-

ments. An example is given in Example 2.

Example 2: Sorting the array $A = \{1, 5, 3, 7, 4, 1, 2, 3\}$ using counting sort.

1. Set $C[A[i]]$ to $C[A[i]] + 1$

C:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	0	2	1	2	1	1	0	1	0	0

2. Set $C[i]$ to $C[i] + C[i-1]$

C:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	0	2	3	5	6	7	0	8	0	0

3. Set $B[C[A[i]]]$ to $A[i]$ and set $C[A[i]]$ to $C[A[i]] - 1$

For $i = 0$

B:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
		1						

C:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	0	1	3	5	6	7	0	8	0	0

For $i = 1$

B:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
		1					5	

C:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	0	1	3	5	6	6	0	8	0	0

For $i = 2$

B:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
		1			3		5	

C:	0	1	2	3	4	5	6	7	8	9
	0	1	3	4	6	6	0	8	0	0

$\vdots$

For $i = 6$

B:	0	1	2	3	4	5	6	7
	1	1	2		3	4	5	7

C:	0	1	2	3	4	5	6	7	8	9
	0	0	2	4	5	6	0	7	0	0

For $i = 7$

B:	0	1	2	3	4	5	6	7
	1	1	2	3	3	4	5	7

C:	0	1	2	3	4	5	6	7	8	9
	0	0	2	3	5	6	0	7	0	0

The count array is of size k and initializing it together with the second for loop which is used to apply a prefix sum on the count array, require $O(k)$ time. The operation of counting the elements of the array takes and the last for loop which assigns the sorted elements to the output array require $O(n)$ time. Therefore, the whole algorithm takes $O(n + k)$ time.

16

2.2.2.3 Radix Sort

Radix sort is an integer algorithm. The data is sorted using keys grouped by the individual digits which share the same significant position and the value. The pseudocode for the algorithm is given in Algorithm 3.

Algorithm 3 The Radix Sort Algorithm

    Input: n dimensional array A

    Output: Sorted array B

  1: $k \leftarrow$ the maximum number of digits that the values to be sorted have

  2: **for** $i \leftarrow 1, k$ **do**

  3:     The digit i of $B \leftarrow$ Sort A on digit i

  4: **end for**

As can be seen from Algorithm 3, each digit of the sequence is processed once. Since processing one digit involves sorting the n values at that position, this can be done by using a non-comparison based sorting algorithm such as bucket sort or counting sort. There exist k digits, thus the total time becomes $O(k * n)$.

If there exist values that are shorter than the maximum number of digits k then padding can be done in order to have an equal number of digits for each value. An example is given in Example 3.

Example 3: Sorting the $A = \{125, 5, 98, 546, 357, 11, 23, 19\}$ using radix sort for $k = 3$.

1. Set B by sorting A on the digit 1

0	1	2	3	4	5	6	7
011	023	005	125	546	357	098	019

2. Set B by sorting A on the digit 2

0	1	2	3	4	5	6	7
005	011	019	023	125	546	357	098

3. Set B by sorting A on the digit 3

0	1	2	3	4	5	6	7
005	011	019	023	098	125	357	546

4. Sorted array $B = \{5, 11, 19, 23, 98, 125, 357, 546\}$.

CHAPTER 3

DISCRETE LOGARITHMS

Let G be a finite Abelian group of size ℓ. Assume that G is cyclic and g is a generator of G. Any element $\alpha \in G$ can be uniquely expressed as $\alpha = g^\beta$ for some integer β in the range $0 \leq \beta \leq \ell - 1$. The integer β is called the discrete logarithm of α with respect to g. The discrete logarithm problem (DLP) is the problem of computing β when G, g and α are known.

Depending upon the group G, the computational complexity of DLP varies from easy (polynomial time) to intractable (exponential time). If G is taken as $\mathbb{F}_q^*$ then the problem is considered as finite field discrete logarithm problem and the computation of the discrete logarithm of α with respect to g appears to be an intractable computational problem for sufficiently large finite fields. It has to be noted that finite fields of order q exist only when q is of the form $q = p^r$, where p is a prime number and r is a natural number, and that there is a unique field of any such order up to isomorphism.

In this thesis, we are interested in finite field discrete logarithm problem with the field having large prime characteristic. Due to recent developments, discrete logarithms in extension fields such as $\mathbb{F}_p^r$ with p being a small prime are much easier to compute when compared to large prime fields $\mathbb{F}_p$ [31, 12, 14, 15, 20]. Hence, the fields $\mathbb{F}_p$ with

p being large prime appear to offer relatively high levels of security depending on the bit size of p.

In the first section of this chapter, cryptosystems that rely on DLP are addressed. In the second section, the algorithms for solving DLP, including the generic algorithms along with the index calculus algorithms, are summarized.

3.1 Cryptosystems Based on DLP

In secure communication, the cryptosystems that are based on computationally difficult problems are used. The most known algorithms predicate their security on DLP.

3.1.1 Diffie-Hellman Key Exchange

Diffie-Hellman was one of the first public-key protocols. It was originally conceptualized by Ralph Merkle and named after Whitfield Diffie and Martin Hellman [29, 7].

Two parties A and B can agree on a secret key by using Diffie-Hellman key exchange. Let G be a cyclic group of order n and g be a generator of this group. Suppose that G and g are made public to both A and B. When A and B need to agree on a secret key

1. A chooses an element $priv_A$ as private key, computes $g^{priv_A} \pmod{n}$ and sends it to B.

2. B chooses an element $priv_B$ as private key, computes $g^{priv_B} \pmod{n}$ and sends it to A.

3. A gets $g^{priv_B} \pmod{n}$, raises it to the power $priv_A$, and computes the secret key

$$K_{AB} = (g^{priv_B})^{priv_A} = g^{priv_A priv_B} \pmod{n}. \qquad (3.1)$$

4. B gets $g^{priv_A} \pmod{n}$, raises it to the power $priv_B$ and computes the secret key

$$K_{AB} = (g^{priv_A})^{priv_B} = g^{priv_A priv_B} \pmod{n}. \qquad (3.2)$$

Diffie-Hellman key exchange is the basis of many popular internet protocols such as TLS [6, 37]. In general, it does not provide authentication of the communicating parties, thus can be subject to man-in-the middle attacks. The protocol is considered secure if G and g are chosen properly and authentication is used. In addition, the order n of G must be large enough.

3.1.2 The ElGamal Encryption and Signature Scheme

ElGamal encryption is a public key encryption method derived from Diffie-Hellman key exchange. It is first described by Taher El Gamal in 1985 [9].

Let B wants to send a ciphertext to A:

1. A generates his public-private key pair by first choosing a cylic group G of order n and a generator g of the group G. Second, he chooses an element $priv_A < p-1$ as a private key. Finally, he constructs his public key as a 3-tuple

$$pub_A = (G, g, h = g^{priv_A} \pmod{n}) \qquad (3.3)$$

and sends his public key to B.

2. B gets A's public key and encrypts the message m as a composition of $c_0 = mh^r$ and $c_1 = g^r$ where r is a random integer $< n$.

3. A decrypts the message by computing

$$c_0(c_1^{priv_A})^{-1} = mh^r((g^r)^{priv_A})^{-1} = m. \qquad (3.4)$$

ElGamal encryption can also be used to sign messages. If A wants to send a signed message to B:

1. A generates his public-private key pair by first choosing a prime p and a generator g of the group $\mathbb{Z}_p^*$. Second, he chooses an element $priv_A < n$ as a private key. Finally, he constructs his public key as a 3-tuple

$$pub_A = (p, g, h = g^{priv_A} \, (mod \, p - 1)) \qquad (3.5)$$

and sends his public key to B.

2. A generates the signature s of the message m by first choosing an integer e with $gcd(e, p) = 1$. Then he computes

$$r = g^e \, (mod \, p), \qquad (3.6)$$

$$s = (m - priv_A r)e^{-1} \, (mod \, p - 1) \qquad (3.7)$$

and sends B the triple (m, r, s).

3. B verifies the signature (r, s) by comparing $h^r r^s \, (mod \, p)$ and $g^m \, (mod \, p)$. If they are equal then it is validated that m is signed by A. During the signature generation of A, the equality

$$m = priv_A r + se \, (mod \, p - 1) \qquad (3.8)$$

is hold and by Fermat's little theorem

$$g^m = g^{priv_A r} g^{se} = h^r r^s \, (mod \, p). \qquad (3.9)$$

3.1.3 Digital Signature Algorithm

The Digital Signature Algorithm (DSA) was proposed in 1991 by the National Institute of Standards and Technology [22]. It differs from ElGamal signature scheme in the way that it uses groups with order $p - 1$ where $p - 1$ has a large prime factor. Let g_q be a generator of the subgroup of order q of the group $\mathbb{F}_p^*$. Then

$$g_q = g^{(p-1)/q} \, (mod \, p). \tag{3.10}$$

The private key of A, $priv_A$, is less than q and his public key, pub_A, is little bit modified compared to the ElGamal scheme. It is composed of the parameters

$$pub_A = (p, q, g_q, h = g_q^{priv_A} \, (mod \, p)). \tag{3.11}$$

When A wants to sign a message m:

1. A generates the signature s of the message m by first choosing an integer $e < q$. Then he computes

$$r = (g_q^e \, (mod \, p)) \, (mod \, q), \tag{3.12}$$

$$s = (m + priv_A r) \, (mod \, q) \tag{3.13}$$

and sends B the triple (m, r, s).

2. B verifies the signature (r, s) by computing

$$w = s^{-1} \, (mod \, q), \tag{3.14}$$

$$u_0 = mw \, (mod \, q), \tag{3.15}$$

$$u_1 = rw \, (mod \, q), \tag{3.16}$$

$$v = (g_q^{u_1} h^{u_2} \, (mod \, p)) \, (mod \, q). \tag{3.17}$$

If $v = r$ then the signature is validated.

DSA is faster than ElGamal since it uses subgroups where the computations are faster.

3.2 Solving DLP

Let G be a finite cyclic multiplicative group of size ℓ, and g is a generator of G. Solving the discrete logarithm of α with respect to g is to find β such that $\beta = \log_g \alpha$.

Several algorithms are proposed for computing discrete logarithms. These algorithms can be categorized into two groups, namely, the generic algorithms and the index calculus algorithms. While the generic algorithms work in arbitrary groups, the index calculus algorithms are known to be the most powerful methods for solving DLP on certain groups [28, 19].

3.2.1 Generic Algorithms

Generic algorithms are also called as square root methods. Baby-step-giant-step, Pollard's rho, and Pohlig-Hellman algorithms fall in the category of generic algorithms.

3.2.1.1 Baby-Step-Giant-Step Algorithm

The baby-step-giant-step algorithm refers to a class of algorithms, proposed by Shanks [39]. Let $t = \lceil \sqrt{n} \rceil$. In this method, g^i for $i = 0, 1, 2, \cdots, t-1$ are computed and stored in a table as ordered pairs (i, g^i) by sorting with respect to the second element. Then αg^{-jt} is searched among the second elements of the table. If such an i is found for a particular j that is $\alpha = g^{jt+i}$, then $\log_g \alpha \equiv jt + i$. The determination of j

is called the giant-step whereas the determination of i is called the baby-step. Since $\log_g \alpha \in \{0, 1, 2, \cdots, \ell - 1\}$ and $\ell \leq t^2$, $\log_g \alpha$ can always be expressed as $jt + i$ for some $i, j \in \{0, 1, \cdots, t - 1\}$.

3.2.1.2 Pollard's Rho Algorithm

Pollard's Rho algorithm is first introduced by John Pollard [34] in order to solve the integer factorization problem. The algorithm can also be adapted for the discrete logarithms. A random element $w_0 = g^{s_0} \alpha^{t_0}$ is computed and the sequence $w_i = g^{s_i} \alpha^{t_i}$ is calculated consecutively. By the birthday paradox it is expected to arrive at a collision $w_i = w_j$ after $O(\sqrt{n})$ iterations. That is, we get

$$\alpha^{t_i - t_j} = g^{s_j - s_i}. \tag{3.18}$$

Since the order of g is ℓ, we have

$$(t_i - t_j) log_g \alpha \equiv s_j - s_i \, (mod \, \ell). \tag{3.19}$$

If $gcd(t_i - t_j, \ell) = 1$ then

$$log_g \alpha \equiv (t_i - t_j)^{-1} s_j - s_i \, (mod \, \ell) \tag{3.20}$$

is obtained.

3.2.1.3 Pohlig-Hellman Algorithm

The Pohlig-Hellman algorithm was introduced in [33]. Suppose that the complete prime factorization of $\ell = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$ is known. If the largest prime divisor of ℓ is small then the algorithm is quite efficient.

If $\delta = \log_g \alpha \,(mod\, p_i^{e_i})$ is known for all $i = 1, 2, \cdots, r$ then the Chinese remainder theorem (CRT) gives the desired value of $\delta \,(mod\, \ell)$. First, p^e is taken to be a divisor of n and $\delta \,(mod\, \ell)$ is obtained by solving a DLP in the subgroup of G of size p. Then this value is lifted to $\delta \,(mod\, p^2)$, $\delta \,(mod\, p^3)$, and so on. Each lifting involves computing a discrete logarithm in the subgroup of size p.

Let $\gamma = g^{n/p^e}$ and $\zeta = \alpha^{n/p^e}$. $alpha = g^\beta$ implies that

$$\zeta = \alpha^{n/p^e} = (g^{n/p^e})^\beta = \gamma^\beta = \gamma^{\beta \, mod \, p^e}. \tag{3.21}$$

In order to compute $\xi = \beta \, mod \, p^e$ we write

$$\xi = \beta_0 + \beta_1 p + \beta_2 p^2 + \ldots + \beta_{e-1} p^{e-1}. \tag{3.22}$$

We keep on computing p-ary digits $\beta_0, \beta_1, \beta_2, \cdots$ of ξ one by one. If $\beta_0, \beta_1, \cdots, \beta_{i-1}$ are already computed then we know

$$\lambda = \beta_0 + \beta_1 p + \cdots + \beta_{i-1} p^{i-1}. \tag{3.23}$$

Now, $\zeta = \gamma^\xi$ gives

$$\zeta \gamma^{-\lambda} = \gamma^{\xi - \lambda} = \gamma^{\beta_i p^i + \beta_{i+1} p^{i+1} + \cdots + \beta_{e-1} p^{e-1}}. \tag{3.24}$$

Raising both sides of the equality to the power p^{e-i-1} gives

$$(\zeta \gamma^{-\lambda})^{p^{e-i-1}} = \gamma^{\beta_i p^{e-1} + \beta_{i+1} p^e + \cdots + \beta_{e-1} p^{2e-i-2}} = (\gamma^{p^{e-1}})^{\beta_i}. \tag{3.25}$$

The order of $\gamma^{p^{e-1}}$ is p and β_i is the discrete logarithm of $(\zeta \gamma^{-\lambda})^{p^{e-i-1}}$ to the base $\gamma^{p^{e-1}}$. If it is rephrased in terms of g and α we see that

$$\beta_i = \log_{g^{n/p}}[(\alpha g^{-(\beta_0 + \beta_1 p + \cdots + \beta_{i-1} p^{i-1})})^{n/p^{i+1}}] \tag{3.26}$$

for $i = 0, 1, 2, \cdots, e - 1$. These calculations are done in the subgroup of size p, generated by $g^{n/p}$.

26

3.2.2 Index Calculus Algorithms

The index calculus method is a probabilistic method for computing the discrete logarithms. It is first invented in 1922 by Kraitchik [25] for the integer factorization. The basic idea of using the index calculus method in the discrete logarithms was introduced by Western and Miller [41].

In order to give the general description for the index calculus algorithms, let G be a cyclic group of order ℓ generated by g and let $\beta = \log_g \alpha$. When α is known, β is calculated by relying on three main steps: the sieving phase, linear algebra phase, and individual logarithm phase. Different index calculus methods vary mostly in the way the relations are found during the sieving phase and the linear algebra phase remains the same for most of them. In this section some of the index calculus algorithms, namely, the basic index calculus, linear sieve, Gaussian integer and number field sieve algorithms are introduced.

3.2.2.1 Basic Index Calculus Algorithm

In practice, two kinds of groups namely the finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^\ell}$ are widely used in cryptographic computations. The basic index calculus method can be applied to both fields.

When the basic index calculus algorithm is applied to finite field $\mathbb{F}_p$ for prime p, the factor base B is determined by choosing the first n primes $p_1, p_2, \cdots, p_n$. Then a random integer $u < p - 1$ is taken and the integer k with $k = g^u \pmod{p}$ is calculated. If k can be written as a product of primes in the factor base, then a relation is obtained.

27

If we intend to collect m relations then the set of identities of the form

$$k_i \equiv g^{u_i} \, (mod \, p) = \prod_{j=1}^{n} p_j^{a_{ij}} \tag{3.27}$$

are gathered for $i = 0, 1, \cdots, m$.

In practice, the exact choice of B is implementation dependent. If B has small size then testing the smoothness and collecting the relations are fast. On the other hand, choosing a small value for the size of B makes difficult to find enough number of relations. From a computational point of view, finding the necessary relations is trivially parallelizable and requires a relatively small amount of storage.

When all of the prime factors of k are less than a given bound S, k can be completely factored at most $S + \log k$ divisions. If u is chosen from a uniform distribution, $x = L_p[\alpha, c]$ and $S = L_p[\alpha_S, c_S]$ then the probability that a random $k \leq X$ will factor as a product of primes less than S is

$$\frac{\Psi(X, S)}{X} = L_p[\alpha - \alpha_B, -(\alpha - \alpha_B)(c/c_B)] \tag{3.28}$$

where $\Psi(X, S)$ is defined to be the number of positive integers less than or equal to X that have no prime factors exceeding S. One can refer to [40] for an experimental analysis of the parameters X and S.

It is trivial that the exponents a_{ij} coming from the prime factorization of k in the calculation of ith relation constitute a row of A. Thus each row in A is an exponent vector. Since $k < p$, the exponents and consequently the entries of A are limited by the size of the prime p.

In the linear algebra phase of the index calculus algorithms, the relations collected in

28

the first phase are written as a set of linear congruences

$$\sum_{j=1}^{n} a_{ij} \log_g p_j \equiv u_i \,(mod\, p - 1).$$ (3.29)

The linear system (3.29) is solved for $x_i = \log_g p_i$ and can be interpreted as

$$A\boldsymbol{x} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} \log_g p_1 \\ \log_g p_2 \\ \vdots \\ \log_g p_n \end{bmatrix} \equiv \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} (mod\, p - 1) = \boldsymbol{u}.$$ (3.30)

The existence of a unique solution depends on the rank of the matrix A. If A has full rank then the system (3.30) has a unique solution. In order to ensure the existence of a solution, the number of relations m is chosen to be greater than the size n of the factor base. In Chapter 4, solution techniques for large sparse linear systems that arise in this phase are given in detail.

Once the logarithms of p_i have been computed, then $\beta = \log_g \alpha$ can be calculated by constructing a relation of the form

$$\prod_{j=1}^{n} p_j^{e_j} = \alpha g^e$$ (3.31)

from which $\beta = \sum_{j=1}^{n} e_j \log_g p_j - e$ can be obtained. In practice, the third stage takes considerably less time than the first two stages.

When the basic index calculus algorithm is applied over the finite field $\mathbb{F}_{2^\ell}$, the factor base B is chosen to be consisted of non-constant irreducible polynomials $\omega(x) \in \mathbb{F}_2[x]$ of degrees at most some prescribed bound s. The finite field $\mathbb{F}_{2^\ell}$ has the polynomial-basis representation $\mathbb{F}_2(\theta)$ with $f(\theta) = 0$, where $f(x) \in \mathbb{F}_2[x]$ is an irreducible polynomial of degree ℓ. In the first stage, given a primitive element

29

$g(\theta) \in \mathbb{F}_{2^\ell}^*$, the discrete logarithms of all $\omega(\theta)$ to the base $g(\theta)$ are computed. To this end, $g(\theta)$ is raised to a random exponent u and it is checked whether the canonical representatives of $g(\theta)^u \in \mathbb{F}_{2^\ell}$ can be expressed as the products of the polynomials $\omega(\theta)$ with $\omega(x) \in B$. If a factoring attempt is successful, the relation of the form

$$g(x)^u = \prod_{\omega(x) \in B} \omega(x)^{\gamma_\omega} \, (mod\,(f(x))) \tag{3.32}$$

is obtained. It can also be written as

$$g(\theta)^u = \prod_{\omega(x) \in B} \omega(\theta)^{\gamma_\omega} \in \mathbb{F}_{2^\ell}. \tag{3.33}$$

When the discrete logarithm of both sides are taken, we have

$$u \equiv \sum_{\omega \in B} \gamma_\omega \log_{g(\theta)} \omega(\theta) \, (mod\,(2^\ell - 1)). \tag{3.34}$$

If $|B| = n$ then m relations with $n \le m \le 2n$ are generated. In the linear algebra stage the resulting $m \times n$ system is solved modulo $2^\ell - 1$ to obtain the logarithms of elements in the factor base. At the third stage individual logarithms are computed. Suppose that $\beta = \log_g \alpha(\theta)$ is going to be computed. Random u is selected and decomposing $\alpha(\theta)g(\theta)^u$ into irreducible factors of degrees $\le s$ gives

$$\alpha(\theta)g(\theta)^u = \prod_{\omega \in B} \omega(\theta)^{\delta_\omega}. \tag{3.35}$$

Then $\beta = \log_g \alpha(\theta)$ can be computed as

$$\beta = \log_g \alpha(\theta) \equiv -u + \sum_{\omega \in B} \delta_\omega \log_g \omega(\theta) \, (mod\, 2^\ell - 1). \tag{3.36}$$

3.2.2.2   Linear Sieve Algorithm

Linear sieve algorithm [26] is an adaptation of the quadratic sieve algorithm [35] for factoring integers. It is an $L[1/2, 1]$-time algorithm for the computation of discrete

logarithms over prime fields and $L[1/2, 0.8325...]$-time for the finite fields of characteristic 2 [5].

Let $H = \lceil \sqrt{p} \rceil$ and $J = H^2 - p$. Suppose that for small integers c_1 and c_2, the integer

$$T(c_1, c_2) = J + (c_1 + c_2)H + c_1 c_2 \tag{3.37}$$

factors completely over the first n primes $p_1, p_2, \cdots, p_t$. This leads to

$$J + (c_1 + c_2)H + c_1 c_2 \equiv H^2 + (c_1 + c_2)H + c_1 c_2$$

$$\equiv (H + c_1)(H + c_2)$$

$$\equiv p_1^{e_1} p_2^{e_2} \dots p_t^{e_t} \, (mod \, p). \tag{3.38}$$

By taking the base g discrete logarithms of $(H+c_1)(H+c_2)$ and $p_1^{e_1} p_2^{e_2} \dots p_t^{e_t} \, (mod \, p)$ we get

$$e_1 \log_g p_1 + \dots + e_t \log_g p_t \equiv \log_g (H + c_1) + \log_g (H + c_2) \, (mod \, p - 1). \tag{3.39}$$

This implies that as well as the small primes $p_1, p_2, \cdots, p_t$, the integers $H + c$ for small values of c should be in the factor base B. Hence, c_1 and c_2 are chosen to be between $-M$ and M. Moreover, for certain values of c_1 and c_2, $T(c_1, c_2)$ takes negative values. Thus the factor base B is taken as

$$B = \{-1\} \cup \{p_1, p_2, \cdots, p_t\} \cup \{H + c \mid -M \le c \le M\}. \tag{3.40}$$

The size of the factor base is $n = 2M + t + 2$. By choosing proper M and t and letting c_1, c_2 vary in the range $-M \le c_1 \le c_2 \le M$, we generate $m - 1$ relations. We assume that the base g of the discrete logarithms itself a small prime p_i in the factor base. This gives us a free relation

$$\log_g p_i \equiv 1 \, (mod \, p - 1). \tag{3.41}$$

31

The resulting system has m relations with n unknowns thus the $m \times n$ dimensional coefficient matrix is generated.

After applying the linear algebra step, individual logarithms can be calculated as done in the basic method by searching a smooth value of $\alpha g^u \, (mod \, p)$ for randomly chosen u.

When the linear sieve method is applied over $\mathbb{F}_{2^l}$, it uses a defining polynomial $f(x)$ of the form $x^n + f_1(x)$ where $f_1(x)$ has low degree and constructs a factor base B with two parts B_1 and B_2. B_1 contains non-constant irreducible polynomials of degree $\leq t$ and B_2 contains polynomials of the form $x^{\lceil n/2 \rceil} + c(x)$ with $c(x) \in \mathbb{F}_2[x]$ of degrees $\leq t$. $T(c_1, c_2)$ is obtained by multiplying two polynomials $x^{\lceil n/2 \rceil} + c_1(x)$ and $x^{\lceil n/2 \rceil} + c_2(x)$ from B_2 and it is factored over the irreducible polynomials of B_1

$$(x^{\lceil n/2 \rceil} + c_1(x))(x^{\lceil n/2 \rceil} + c_2(x)) \equiv \prod_{\omega(x) \in B_1} \omega(x)^{\gamma_\omega} \, (mod \, (f(x)) \qquad (3.42)$$

where $\epsilon = 2\lceil n/2 \rceil - n$ is 0 if n is even or 1 otherwise. This is equivalent to

$$\theta^\epsilon f_1(\theta) + (c_1(\theta)c_2(\theta))\theta^{\lceil n/2 \rceil} + c_1(\theta)c_2(\theta) = \prod_{\omega \in B_1} \omega(\theta)^{\gamma_\omega}. \qquad (3.43)$$

By taking the logarithm of both sides, we obtain a linear congruence. As c_1 and c_2 range over all polynomials in B_2, m relations are generated by choosing t such that all (c_1, c_2) pairs lead to an expected number s of relations, satisfying $|B| \leq s \leq 2|B|$. The linear system is solved modulo $2^n - 1$.

The linear algebra and the individual logarithms stages are the same as the linear sieve method in $\mathbb{F}_p$.

3.2.2.3 Gaussian Integer Algorithm

Let d be a small nonzero positive integer such that $-d$ is a quadratic residue in $\mathbb{F}_p$ and let s be a square root of $-d$ modulo p. The integers u, v such that $p = u^2 + dv^2$ can be computed. Either $u + vs \equiv 0 \,(mod\, p)$ or $u - vs \equiv 0 \,(mod\, p)$. Suppose that $u + vs \equiv 0 \,(mod\, p)$. Here, $p = u^2 + dv^2$ implies that the sizes of u and v are $O(\sqrt{p})$. If

$$T(c_1, c_2) = c_1 u + c_2 v \tag{3.44}$$

where c_1 and c_2 small integers, then $T(c_1, c_2)$ can be treated as an element of $\mathbb{Z}[\sqrt{-d}]$ and

$$c_1 u + c_2 v = c_1(u + v\sqrt{-d}) + v(c_2 - c_1\sqrt{-d}) \tag{3.45}$$

can be written. Applying the ring homomorphism

$$\Phi : \mathbb{Z}[\sqrt{-d}] \to \mathbb{F}_p \tag{3.46}$$

gives

$$T(c_1, c_2) \equiv c_1 u + c_2 v \equiv c_1(u + vs) + v\Phi(c_2 - c_1\sqrt{-d}) \,(mod\, p). \tag{3.47}$$

Since $u + vs \equiv 0 \,(mod\, p)$ then the equivalence becomes

$$T(c_1, c_2) \equiv c_1 u + c_2 v \equiv v\Phi(c_2 - c_1\sqrt{-d}) \,(mod\, p). \tag{3.48}$$

If $T(c_1, c_2)$ can be factored completely over the set of all rational primes $B_1 = \{p_1, p_2, \cdots p_n\}$ and $(c_2 - c_1\sqrt{-d})$ can be factored completely over the set of all complex primes $B_2 = \{q_1, q_2, \cdots q_{n'}\}$ then

$$p_1^{e_1} p_2^{e_2} \ldots p_n^{e_n} \equiv v\Phi(q_1)^{t_1} \Phi(q_2)^{t_2} \ldots \Phi(q_{n'})^{t_{n'}} \,(mod\, p). \tag{3.49}$$

When the logarithm of both sides are taken, the relation

$$e_1 \log_g p_1 e_2 \log_g p_2 \ldots e_n \log_g p_n \equiv$$

$$\log_g v + t_1 \log_g \Phi(q_1) t_2 \log_g \Phi(q_2) \ldots t_{n'} \log_g \Phi(q_{n'}) \, (mod \, p) \quad (3.50)$$

is generated in the Gaussian integer method [26]. The size of the factor base $B = B_1 \cup B_2 \cup -1 \cup v$ is $n + n' + 2$. The Gaussian integer method produces smaller systems when compared to the linear sieve method. The resulting system of congruences can be solved in $L[1/2, 1]$ time.

3.2.2.4 Number Field Sieve Algorithm

The number field sieve [13] is known as the fastest algorithm for computing discrete logarithms over prime fields with $L[1/3, (64/9)^{1/3}]$ time complexity. It can be regarded as a generalization of the Gaussian integer algorithm.

Let $\mathfrak{N}_K$ be the number ring of integers of $K = \mathbb{Q}(\theta)$ where θ is a root of $f \in \mathbb{Z}_x$ which is an irreducible polynomial over $\mathbb{Q}$. For simplicity, it is assumed that $\mathfrak{N}_K$ supports unique factorization of elements which can be written as polynomials in θ with rational coefficients. The map $\Phi : \theta \mapsto m \, (mod \, p)$ extends to a ring homomorphism $\mathfrak{N}_K \to \mathbb{F}_p$.

Suppose that

$$T_1(c_1, c_2) = c_1 + c_2\theta \in \mathfrak{N}_K \quad (3.51)$$

is smooth with respect to small primes q_i of $\mathfrak{N}_K$, and

$$T_2(c_1, c_2) = \Phi(T_1(c_1, c_2)) = c_1 + c_2 m \in \mathbb{Z} \quad (3.52)$$

34

is smooth with respect to small rational primes p_j. Applying the homomorphism Φ gives

$$\Phi(T_1(c_1, c_2)) \equiv \Phi(\prod_i q_i^{t_i}) \equiv \prod_i \Phi(q_i)^{t_i} \equiv T_2(c_1, c_2) \equiv \prod_j p_j^{e_j} \,(mod\,p). \quad (3.53)$$

Taking the discrete logarithm leads to the following relation

$$\sum_i t_i \log_g (\Phi(q_i)) \equiv \sum_j e_j \log_g (p_j) \,(mod\,p - 1). \quad (3.54)$$

The rational primes p_j and $\Phi(q_i)$ for small primes q_i of $\mathfrak{N}_K$ should be included in the factor base B. In addition, a set of generators of the group of units of $\mathfrak{N}_K$ is considered and for each such generator u, $\Phi(u)$ is also included.

CHAPTER 4

LARGE SPARSE LINEAR SYSTEMS

In index calculus computations, as the size of the finite field increases, the size of the system of linear equations that are encountered during the linear algebra phase also increases. Besides, the revealed linear systems are desired to have full rank in order to ensure the existence of a unique solution, so they are arranged to have much more rows than columns, i.e. $m \gg n$ for $m \times n$ matrix A. Since n can be millions, the huge size of the matrix can cause storage problems. However, the high sparsity of the matrix permits us to store the matrix by utilizing a sparse matrix storage format. In addition, the size of the matrix can be also reduced by applying a filtering step that involves structured Gaussian elimination [32]. This step reduces the matrix size without much increasing the density.

Despite the high sparsity of the matrices, ordinary Gaussian elimination is insufficient for solving these systems since the cost depends on a complicated relationship between the sparsity structure and the fill-in. Thus rather than direct methods, iterative methods such as Lanczos and Wiedemann methods are used in order to benefit from the sparsity. The recent discrete logarithm records employ either Lanczos or Wiedemann algorithms or their block variants. For instance, the latest discrete loga-

rithm record solves the linear system of 24 million columns and rows with an average row weight of 135 using the block Wiedemann algorithm [24].

In this chapter, first, the structured Gaussian elimination and the sparse matrix storage formats are introduced in separate sections. Second, under the section of iterative solvers, Lanczos and Wiedemann algorithms and their block variants are explained. Finally, sparse matrix-vector multiplication is given in detail.

Throughout this chapter, the linear system of congruences of the form

$$A\boldsymbol{x} \equiv \boldsymbol{u} \,(mod\, M) \tag{4.1}$$

is considered. Here A is an $m \times n$ matrix, $\boldsymbol{x}$ is an n-vector, $\boldsymbol{u}$ is a nonzero m-vector and M is the modulus.

4.1 Structured Gaussian Elimination

The structured Gaussian elimination (SGE) eliminates some of the rows and columns of the matrix in order to reduce the dimension. In some situations, SGE can reduce the size of the matrix to such an extent that standard Gaussian elimination becomes practical for solving the system.

The basic strategy implemented by structured Gaussian elimination is to name the columns that have the largest number of nonzero elements *heavy* and to operate on the remaining *light* columns by preserving the sparsity. In practice, the matrix is stored by using one of the sparse matrix storage formats. However, SGE can be understood better considering the full-matrix. The algorithm employs the following steps:

Step 1. The columns of weight zero and one are deleted. The corresponding row for

the column that has one nonzero coefficient is also removed. After this step, all the columns have weight ≥ 2. Therefore, it is essential to declare some light columns heavy, obviously by choosing those have the highest weights.

Step 2. The rows of weight zero and one are deleted. A row of weight zero corresponds to the equation $0 = 0$ thus can be eliminated. Let the row R_i has weight one and suppose that the nonzero entry is the value at he j^{th} column, x_j. The value of x_j is substituted in all the rows where the x_j occurs and the row R_i and the corresponding j^{th} column is deleted.

Step 3. The rows of weight one in the light part is deleted. These are the rows whose intersection with all the light columns contain exactly one nonzero entry. Let this row be R_i and the entry be the variable x_j. R_i allows x_j to be written in terms of some variables corresponding to heavy columns. The value of x_j is substituted in all the rows where x_j occurs. Then the row R_i and the corresponding j^{th} column is deleted. It may be better to redefine the heavy and light columns after this step.

Step 4. The redundant rows are deleted. If the matrix still have more rows than columns after above steps then some redundant rows may be removed. Rows with the largest number of nonzero entries in the light columns may be a good choice since this removal can make light parts even lighter.

Above steps are usually applied in sequence. For instance after steps $2 - 4$ are executed, maybe some columns with weight one and zero can occur again. Thus the first step needs to be repeated.

Although structured Gaussian elimination is a heuristic algorithm, in [32], the size of the matrix is reduced by structured Gaussian elimination on variable data sets and an

improvement of 90% is recorded.

4.2 Sparse Matrix Storage Formats

In order to take advantage of the sparsity of the matrices, specialized algorithms and data structures are used when storing and manipulating them. If regular dense matrix structures and algorithms are implemented for large sparse matrices then both processing and memory are wasted on the zero-valued elements. On the other hand, sparse data can easily be compressed and thus requires significantly less storage and one does not often need to do arithmetic operations with zeros.

A dense matrix is typically stored as a two-dimensional array. Each element of the array corresponds to the a_{ij}^{th} element of the matrix and it is accessed by the row index i and the column index j. For a $m \times n$ matrix, the amount of memory needed to store the matrix in this format is proportional to mn.

The objective of the storage formats for sparse matrices is to exploit certain matrix properties by storing the nonzero elements in contiguous memory locations. This strategy not only reduces the memory space but also provides efficient execution of the subroutines on the matrix data. Coordinate format (COO), compressed sparse row (CSR) and Block CSR are the most well-known sparse matrix storage formats that have been proposed so far [38].

In general, matrices over finite fields do not have any structural properties that can be taken advantage of to improve performance. In [10], the underlying matrix representation and the corresponding arithmetic are selected by a code generation tool employing heuristics.

4.2.1 Coordinate Format (COO)

The most basic storage format for a sparse matrix is the so-called coordinate format. The nonzero values are stored with the associated coordinates in terms of row and column indices. Three arrays are used by COO. First array is to store the nonzero values, the second array is for the row indices and the third array is for the column indices. Each array is of size nnz, the number of nonzero elements.

4.2.2 Compressed Sparse Row (CSR)

CSR format stores the compressed data according to the row indices. Three arrays are needed, first one is the array for storing the nonzero values in the matrix, the second one is for the column indices and the third one stores the number of nonzero entries per row. If A is a $m \times n$ sparse matrix and the rows of A are denoted by $A_1, \cdots, A_m$ then CSR format represents A as the set of vector $\{r_1, \cdots, r_{m+1}\}$, where the ith row A_i is represented by the vector r_i and r_{m+1} is an imaginary $(m + 1)^{\text{th}}$ row which is included for computational convenience during matrix-vector multiplication operations. Each element r_i is the ordered pair of the form $\{j, a_{ij}\}$, where j denotes the column which contains a nonzero element a_{ij}.

In CSR format, the values and column indices are stored in an array of size nnz and the row pointers are stored in an array of size $m+1$. Therefore the storage requirement is lower than COO format. CSR format allows fast row access. A similar format is compressed sparse column (CSC) storage, where the column pointers are stored instead of row pointers which allow fast column access if a matrix is stored in CSC format.

41

4.2.3 Block CSR

If the sparse matrix A consists of dense blocks of nonzero elements, the CSR can be modified to exploit such block patterns in order to improve the number of cache hits during sparse matrix-vector multiplications relying on small dense matrix operations with those blocks. The matrix is partitioned in small blocks and each block is treated as a dense matrix, even though it may have some zeros. It could be still advantageous to treat those zeros as nonzero values within the blocks.

The Block CSR format needs three arrays as in the CSR format. These arrays are an integer array storing the column indices of the original matrix A, an array storing the nonzero blocks in row-wise fashion, a pointer array in which the entries point to the beginning of each block row in other two arrays. Similarly, one could also use a block CSR format as well.

4.3 Iterative Methods

Solving large sparse linear systems by Gaussian elimination is not affordable since it requires $O(n^3)$ operations for a dense coefficient matrix. In order to decrease the running time of the process and also to take advantage of the sparsity of the matrices, some iterative methods are developed. Conjugate gradient [17], Lanczos [27] and Wiedemann [42] algorithms are widely used algorithms to solve such systems. Although conjugate gradient and Lanczos methods are originally designed for systems with real numbers, their adaptations for finite fields exist in the literature [3].

Both Lanczos and Wiedemann algorithms are applicable on square matrices. In order

to transform the rectangular matrix into a square one, the both sides of the linear equation given by (4.1) is multiplied from left by A^T and it is transformed into the equation

$$B\boldsymbol{x} \equiv \boldsymbol{c}\,(mod\,p). \tag{4.2}$$

Here, $B = A^T A$ is an $n \times n$ diagonal matrix and $c = A^T u$ is an n-vector. It is clear that a solution to the equation (4.1) is also a solution to the equation (4.2). However, it is not feasible to compute $B = A^T A$ explicitly since B may be denser than A and it is instead preferable to apply sparse matrix-vector multiplication without explicitly forming B.

The choice of the solver heavily depends on the modulus M. Block methods are usually suited for $M = 2$. If $M \geq 3$ the non-block variants are employed. The systems obtained from index calculus algorithms need not to be prime. If it is the situation and the complete prime factorization of M is known then the system is solved for the prime divisors of M and then the solution is lifted to appropriate powers of these primes. If the factorization of the modulus M is not known then M is assumed as a prime. If the system solver fails when attempting to invert an element then a non-trivial factor of M is discovered. M is divided into its factors M_1 and M_2 and the system is solved modulo both M_1 and M_2 again pretending as if they were primes.

4.3.1 Lanczos Algorithm

The Lanczos algorithm is an iterative algorithm which is originally proposed for solving equations over real numbers, but also adapted to work over finite fields.

Lanczos method is a Krylov space method. Krylov space methods build up Krylov

43

subspaces

$$\mathcal{K}_j(A, \boldsymbol{b}) = span\{\boldsymbol{b}, A\boldsymbol{b}, A^2\boldsymbol{b}, ...A^{j-1}\boldsymbol{b}\} \tag{4.3}$$

in order to find good approximations to eigenvectors and invariant subspaces within Krylov spaces.

The Lanczos algorithm solves the linear system $A\boldsymbol{x} = \boldsymbol{u}$ for symmetric $A \in \mathbb{R}^{nxn}$ by computing the eigenspace of A. A-conjugent basis $\{\boldsymbol{w_0}, \boldsymbol{w_1}, \ldots, \boldsymbol{w_{j-1}}\}$ of the Krylov space $\mathcal{K}_j(A, \boldsymbol{u})$ is computed for $d \leq n$. Thus

$$\boldsymbol{w_i}^T A \boldsymbol{w_j} = 0 \text{ when } i \neq j. \tag{4.4}$$

If a solution $\boldsymbol{x}$ to $A\boldsymbol{x} = \boldsymbol{u}$ exists in $\mathcal{K}_j(A, \boldsymbol{u})$, it can be found by projecting $\boldsymbol{u}$ onto $\{\boldsymbol{w_0}, \boldsymbol{w_1}, \ldots, \boldsymbol{w_{j-1}}\}$:

$$\boldsymbol{x} = \sum_{i=0}^{j-1} \frac{\boldsymbol{w_i}^T \boldsymbol{u}}{\boldsymbol{w_i}^T A \boldsymbol{w_i}} \boldsymbol{w_i}. \tag{4.5}$$

The algorithm computes $\{\boldsymbol{w_0}, \boldsymbol{w_1}, \ldots, \boldsymbol{w_{j-1}}\}$ by modifying the Gram-Schmidt orthogonalization in order to induce A-conjugacy. Let

$$\boldsymbol{w_0} = \boldsymbol{u}, \tag{4.6}$$

$$\boldsymbol{v_1} = A\boldsymbol{w_0}, \tag{4.7}$$

$$\boldsymbol{w_1} = \boldsymbol{v_1} - \frac{\boldsymbol{v_1}^T \boldsymbol{v_1}}{\boldsymbol{w_0}^T \boldsymbol{v_1}} \boldsymbol{w_0}. \tag{4.8}$$

Then for $i \geq 1$

$$\boldsymbol{v_{i+1}} = A\boldsymbol{w_i}, \tag{4.9}$$

$$\boldsymbol{w_{i+1}} = \boldsymbol{v_{i+1}} - \frac{\boldsymbol{v_{i+1}}^T \boldsymbol{v_{i+1}}}{\boldsymbol{w_i}^T \boldsymbol{v_{i+1}}} \boldsymbol{w_i} - \frac{\boldsymbol{v_{i+1}}^T \boldsymbol{v_i}}{\boldsymbol{w_{i-1}}^T \boldsymbol{v_i}} \boldsymbol{w_{i-1}}. \tag{4.10}$$

The iteration stops when

$$\boldsymbol{w_j}^T A \boldsymbol{w_j} = 0 \tag{4.11}$$

i.e $\boldsymbol{w_j}$ is A-conjugate to itself.

In particular, the matrices that are generated by index calculus algorithms are not symmetric. One way to make these matrices symmetric is to multiply both sides of the equality from left with the transpose of the coefficient matrix and instead of solving $A\boldsymbol{x} = \boldsymbol{u}$, the equation

$$(A^T A)\boldsymbol{x} = (A^T \boldsymbol{u}) \tag{4.12}$$

is solved. If A is of rank close to minimum of m and n then the two systems are equivalent with high probability. As mentioned earlier $A^T A$ is not computed explicitly since $A^T A$ is denser than A. The iteration (4.9) can be carried out by first multiplying $\boldsymbol{v'_{i+1}} = A\boldsymbol{w_i}$ and then computing $\boldsymbol{v_{i+1}} = A^T \boldsymbol{v'_{i+1}}$. Therefore instead of one matrix-vector multiplication, two multiplications are computed in each iteration.

Another problem that arise when utilizing Lanczos algorithm to solve a system of linear equations over finite fields is that there can be a nonzero vector which is conjugate to itself over finite fields. One possible solution to these problem is given in [32]. In their method, the field over which the equations to be solved is embedded in a considerably larger field. This reduces the possibility of encountering a self-conjugate nonzero vector. Then a diagonal random $m \times m$ matrix D with entries from $\mathbb{F}_{q^d}$ is chosen and the scaled system $(DA)^T (DA)\boldsymbol{x} = (DA)^T D\boldsymbol{u}$ is solved which has the same solution vector as the original system. Here, A and DA have nonzero entries at same locations. Although the sparsity is not effected by defining D, the storage of DA may take more space since D is from a larger field. Hence, in (4.9) $\boldsymbol{v_{i+1}}$ can be computed as $A^T (D^2 (A\boldsymbol{w_i}))$ by adding one more matrix-vector multiplication and a matrix squaring to complexity.

45

4.3.1.1   Block Lanczos Algorithm

In the block Lanczos method, higher dimensional subspaces are generated instead of individual vectors. This helps us in two ways. First, we process multiple dimensions simultaneously by block operations on words. Second, the number of iterations is reduced by a factor of the word size. The word size can be 32-bit or 64-bit depending on the platform.

We work on pairwise A-orthogonal subspaces $W_0, W_1, \cdots W_s$. They are generated by the method satisfying the conditions:

1. W_i is A-invertible for all $i = 0, 1, 2, \cdots, s - 1$,

2. W_i and W_j are A-orthogonal for $i \neq j$,

3. $AW \subseteq W$ where $W = W_0 + W_1 + \cdots + W_{s-1}$.

The iterations stop as soon as the zero space W_s is obtained.

4.3.2   Wiedemann Algorithm

Let $\mathbb{K}$ be a field. Wiedemann algorithm solves the linear system $A\boldsymbol{x} = \boldsymbol{u}$ by probabilistically determining the minimal polynomial $\mu_A(x)$ of A in $\mathbb{K}[x]$ using Berlekamp-Massey algorithm. Let $a_0, a_1, a_2, \cdots$ be an infinite sequence of elements from a field $\mathbb{K}$. When the first d initial terms are given, for all $k \geq d$, we have

$$a_k = c_{d-1}a_{k-1} + c_{d-2}a_{k-2} + \cdots + c_1 a_{k-d+1} + c_0 a_{k-d} \tag{4.13}$$

for some constant $c_0, c_1, \cdots c_{d-1} \in \mathbb{K}$. Given the first $2d$ terms of the sequence, the constants $c_0, c_1, \cdots c_{d-1} \in \mathbb{K}$ can be computed by

$$C(x) = 1 - c_{d-1}x - c_{d-2}x^2 - \cdots - c_0 x^d \qquad (4.14)$$

using Berlekamp-Massey algorithm. Cayley-Hamilton theorem states that $n \times n$ matrix A satisfies its characteristic equation $\chi_A(x) = det(xI - A)$ where I is the identity matrix. Clearly, $\mu_A(x) \mid \chi_A(x)$ in $\mathbb{K}[x]$. Let

$$\mu_A(x) = x^d - c_{d-1}x^{d-1} - c_{d-2}x^{d-2} - \cdots - c_1 x - c_0 \in K[x], \qquad (4.15)$$

$$\mu_A(A) = 0 \qquad (4.16)$$

with $d = deg\mu_A(x) \le n$. For any non zero $n-$vector $\boldsymbol{v}$ and for any integer $k \ge d$, $\mu_A(x) = 0$:

$$A^{k-d}\boldsymbol{v}\mu_A(A) = A^k\boldsymbol{v} - c_{d-1}A^{k-1}\boldsymbol{v} - \cdots - c_1 A^{k-d+1}\boldsymbol{v} - c_0 A^{k-d}\boldsymbol{v} = 0. \qquad (4.17)$$

Let v_k be the element of $A^k\boldsymbol{v}$ at some particular position. The sequence v_k satisfies the recurrence relation

$$v_k = c_{d-1}v_{k-1} + \cdots + c_1 v_{k-d+1} + c_0 v_{k-d}. \qquad (4.18)$$

Using Berlekamp-Massey algorithm, the polynomial $C(x)$ of degree $d' \le d$ is computed and $x^{d'}C(1/x)$ is obtained. Trying several sequences that correspond to different position in $A^k\boldsymbol{v}$ many polynomials $x^{d'}C(1/x)$ are gathered whose least common multiple (lcm) is expected to be the minimal polynomial of A. If the upper bound for d is chosen to be n and $2n$ vector elements are supplied then the Berlekamp-Massey algorithm works correctly. Hence, $2n$ matrix-vector multiplications are computed in Wiedemann algorithm.

4.3.2.1 Block Wiedemann Method

In Wiedemann's algorithm, the minimal polynomial of the sequence $\boldsymbol{u}^T A^i \boldsymbol{v}$ is computed for a randomly chosen vector $\boldsymbol{v}$ and for a projection vector $\boldsymbol{u}$. In block Wiedemann method, the block of μ vectors are taken as U and a block of ν vectors as V. That is, U is an $n \times \mu$ matrix, whereas V is an $n \times \nu$ matrix. If we take $W = AV$ and consider the sequence

$$M_i = U^T A^i W \ for \ i \geq 0 \tag{4.19}$$

of $\mu \times \nu$ matrices. There exist coefficient vectors $\boldsymbol{c_0}, \boldsymbol{c_1}, \cdots, \boldsymbol{c_d}$ with $d = \lceil n/\nu \rceil$ such that the sequence M_i is linearly generated as

$$M_k \boldsymbol{c_d} + M_{k-1} \boldsymbol{c_{d-1}} + \ldots + M_{k-d+1} \boldsymbol{c_1} + M_{k-d} \boldsymbol{c_0} = 0 \tag{4.20}$$

for all $k \geq d$, let $e = \lceil \nu(d+1)/\mu \rceil$. Applying the (4.20) for $k = d, d+1, \cdots, d+e-1$ yields the system

$$\begin{bmatrix} M_d & M_{d-1} & M_{d-2} \ldots & M_1 & M_0 \\ M_{d+1} & M_d & M_{d-1} \ldots & M_2 & M_1 \\ \vdots & \vdots & \ddots & & \vdots \\ M_{d+e-1} & M_{d+e-2} & M_{d+e-3} \ldots & M_e & M_{e-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{c_d} \\ \boldsymbol{c_{d-1}} \\ \boldsymbol{c_{d-2}} \\ \vdots \\ \boldsymbol{c_1} \\ \boldsymbol{c_0} \end{bmatrix} = 0 \tag{4.21}$$

In order to solve the above system, Coppersmith [2] proposes a generalization of the Berlekamp-Massey algorithm. However, in [5], a simpler version of the algorithm proposed by Kaltofen [21] is explained. Kaltofen's algorithm uses the fact that the coefficient matrix in (4.21) is in the Toeplitz form.

4.4 Sparse Matrix-Vector Multiplication

Sparse matrices are usually involved in the applications which use very large matrices. Typically the dimensions of the matrices that are encountered in the cryptographic applications are in the range one million and 25 millions with approximately 150 nonzeros per row.

Sparse matrix-vector multiplication (SpMVM) is one of the basic operations needed for solving sparse linear systems. In the previous chapter, two iterative methods namely Lanczos and Wiedemann algorithms are explained. As concluded, both methods need at least n matrix-vector multiplications for n dimensional square matrices. Therefore the performance of the SpMVM affects the speed of the linear algebra phase of the index calculus algorithm at the first place.

The ordinary method for multiplying $m \times n$ matrix A by an n-vector $\boldsymbol{v}$ is given in Algorithm 4.

Algorithm 4 The ordinary matrix-vector multiplication algorithm

---

Input: $A = \{\{A_{0,0}, \cdots, A_{0,n-1}\}, \cdots, \{A_{m-1,0}, \cdots, A_{m-1,n-1}\}\}$: The array for the matrix elements, $v = \{v_0, \cdots, v_{n-1}\}$: The array for the vector elements.

Output: $u = \{u_0, u_1, \cdots, u_{m-1}\}$: The array for the result vector.

1: $u_i \leftarrow$ An empty array of length m

2: **for** $i \leftarrow 0, m - 1$ **do**

3:     **for** $j \leftarrow 0, n - 1$ **do**

4:         $u_i \leftarrow u_i + A_{i,j}.v_j$

5:     **end for**

6: **end for**

If the matrix is dense then the ordinary matrix-vector multiplication applies $O(mn)$ operations on $O(mn)$ amount of data. The cost is $O(nnz)$ operations and storage for the sparse case where nnz is the number of nonzero elements. Even though the cost is lower in the sparse case, the operation is not cache friendly, i.e., there would be a lot of cache misses. Thus, better sparse matrix-vector multiplication algorithms can be used depending on the storage format of the matrix and if available, the special structure of the matrix should be exploited for optimization.

If COO format is used to store the sparse matrix then the matrix-vector multiplication can be implemented as given in Algorithm 5.

Algorithm 5 The sparse matrix-vector multiplication algorithm using COO format

---

Input: nnz: The number of the nonzero elements, $a = \{a_0, a_1, \cdots, a_{nnz-1}\}$: The array for values of the nonzero elements, $r = \{r_0, r_1, \cdots, r_{nnz-1}\}$: The array for the row indices of the nonzero elements, $c = \{c_0, c_1, \cdots, c_{nnz-1}\}$: The array for the column indices of the nonzero elements.

Output: $u = \{u_0, u_1, \cdots, u_{m-1}\}$: The array for the result vector.

1: $u_i \leftarrow$ An empty array of length m
2: **for** $i \leftarrow 0, (nnz - 1)$ **do**
3:      $u_{r_i} \leftarrow u_{r_i} + a_i.v_{c_i}$
4: **end for**

If the CSR storage format is used then Algorithm 6 can be applied when multiplying A by v. Applying sparse matrix-vector multiplication using CSR format is advantageous to COO format. Although the number of operations are the same for both algorithms, the number of memory accesses is two times better in CSR method.

Algorithm 6 The sparse matrix-vector multiplication algorithm using CSR format

Input: nnz: The number of the nonzero elements, $a = \{a_0, a_1, \cdots, a_{nnz-1}\}$: The array for values of the nonzero elements, $r = \{r_0, r_1, \cdots, r_{nnz}\}$: The array for the row index range of the nonzero elements, $c = \{c_0, c_1, \cdots, c_{nnz-1}\}$: The array for the column indices of the nonzero elements

Output: $u = \{u_0, u_1, \cdots, u_{m-1}\}$: The array for the result vector.

1: $u_i \leftarrow$ An empty array of length m

2: **for** $i \leftarrow 0, m - 1$ **do**

3:      **for** $j \leftarrow r_i, r_{i+1}$ **do**

4:         $u_i \leftarrow u_i + a_j.v_{c_j}$

5:      **end for**

6: **end for**

4.4.1 Structured Matrices

Another approach for efficient SpMVM is to optimize the operation by exploiting the structure of the matrices. There are many different structures that one can encounter in practice. Some examples can be given as Toeplitz, Hankel, Vandermonde and Cauchy matrices.

Let the matrices $A \in \mathbb{F}^{m \times m}$ and $B \in \mathbb{F}^{n \times n}$ are given. Let $X \in \mathbb{F}^{m \times n}$ be a matrix satisfying a equation of the form (Sylvester equation)

$$\triangle_{A,B}(X) = AX - XB = YZ \qquad (4.22)$$

for some matrices $Y \in \mathbb{F}^{m \times \alpha}$ and $Z \in \mathbb{F}^{\alpha \times n}$, where $\alpha < n$. The pair of matrices Y, Z is referred to as the $\{A, B\}$-generator of X and the smallest possible inner size

α among all $\{A, B\}$-generators is called a A, B displacement rank of X. This is the so-called Toeplitz-like displacement operator.

When the displacement has low rank, the generators are utilized as a compact data structure and this approach is the basic idea behind the algorithms of structured matrices. Generally, diagonal matrices and cyclic down shift matrices are selected for matrices A and B. As computing matrix vector products with such structured matrices have close algorithmic correlation to computations with polynomials and rational functions, these matrices can be multiplied by vectors nearly in linear time.

In general, the matrices that are generated by the index calculus algorithms for discrete logarithms have no such structural properties to take advantage.

PERMUTATIONAL MATRIX-VECTOR MULTIPLICATION

WITH PREPROCESSING

In this chapter, we present a new algorithm for computing matrix-vector multiplications over prime fields. The proposed algorithm consists of two stages. The first one is the preprocessing stage which constructs the permutation tables. The second stage is the actual matrix-vector multiplication using the permutation tables.

The underlying idea behind the proposed algorithm can be understood by rearranging the vector-vector multiplications that are employed for each row of the matrix. Let the ith row vector of $n \times n$ matrix A be $\boldsymbol{a} = \begin{bmatrix} a_0, a_1, \cdots, a_{n-1} \end{bmatrix}$ and let s_1 be the index of the smallest entry of $\boldsymbol{a}$. If we take the inner product of $\boldsymbol{a}^T$ and the $n-$vector $\boldsymbol{v} = \begin{bmatrix} v_0, v_1, \cdots, v_{n-1} \end{bmatrix}^T$ then the result $\boldsymbol{a}^T \cdot \boldsymbol{v} = a_0 v_0 + a_1 v_1 + \cdots + a_{n-1} v_{n-1}$ can be written as:

$$
\begin{aligned}
\boldsymbol{a}^T \cdot \boldsymbol{v} &= (a_0 - a_{s_1})v_0 + a_{s_1}v_0 + (a_1 - a_{s_1})v_1 + a_{s_1}v_1 + \cdots + a_{s_1}v_j + \cdots \\
&\quad + (a_{n-1} - a_{s_1})v_{n-1} + a_{s_1}v_{n-1} \\
&= a_{s_1} \sum_{j=0}^{n-1} v_j + \sum_{\substack{j=0 \\ j \neq s_1}}^{n-1} (a_j - a_{s_1})v_j
\end{aligned}
$$

If the set $\{a_j - a_{s_1} | j = 0, 1, \cdots, n-1 \text{ and } j \neq s_1\}$ has $(a_{s_2} - a_{s_1})$ as the smallest

element then the product can be rearranged as:

$$\boldsymbol{a^T} \cdot \boldsymbol{v} = a_{s_1} \sum_{j=0}^{n-1} v_j + (a_{s_2} - a_{s_1}) \sum_{\substack{j=0 \\ j \neq s_1}}^{n-1} v_j + \sum_{\substack{j=0 \\ j \notin \{s_1,s_2\}}}^{n-1} (a_j - a_{s_2}) v_j$$

In the same way, by picking the smallest element and excluding it from the last summation, the equality becomes:

$$\begin{aligned}
\boldsymbol{a^T} \cdot \boldsymbol{v} = \; & a_{s_1} \sum_{j=0}^{n-1} v_j + (a_{s_2} - a_{s_1}) \sum_{\substack{j=0 \\ j \neq s_1}}^{n-1} v_j + (a_{s_3} - a_{s_2}) \sum_{\substack{j=0 \\ j \notin \{s_1,s_2\}}}^{n-1} v_j + \cdots \\
& + (a_{s_{n-2}} - a_{s_{n-3}})(v_{s_{n-2}} + v_{s_{n-1}}) + (a_{s_{n-1}} - a_{s_{n-2}}) v_{s_{n-1}}
\end{aligned} \tag{5.1}$$

Above rearrangement can be regarded as the first step and successive steps can be applied until the subtractions $(a_{s_k} - a_{s_{k-1}})$ are minimized to zero. The algorithm for this process is separated into two stages as subtraction on matrix entries given in Section 5.1 namely the preprocessing stage and the vector additions given in Section 5.2 namely the PMVM stage.

In the preprocessing stage of the proposed algorithm, the entries of the matrix are sorted in order to make the subtraction operations in (5.1) sequential. As soon as a matrix entry is subtracted from the preceding entry, the corresponding element in the vector should be excluded from the addition. Therefore the same sorting permutation is applied to the vector in PMVM stage.

5.1 Preprocessing Stage

In this stage, each row is sorted in ascending order and then minimized by applying sequential subtractions until desired number of nonzero elements $\bar{z}$ is left at the row.

The main objective of the preprocessing stage is to save the permuted indices of the sorted row into a table in every sorting step. Each row vector $\boldsymbol{A_i}$ of the $m \times n$ matrix A

has $(c_i + 1)$ number of permutation tables P_i. How the value $\bar{z}$ is chosen is discussed in Chapter 6. Algorithm 7 shows the pseudocode for preprocessing and Table 5.1 gives an example of an application of the algorithm to a 3×3 matrix.

Algorithm 7 The Preprocessing Algorithm

Input: $m \times n$ matrix A

Output: Permutation table P, the number of permutations per row c_i, preprocessed matrix $\hat{A}$

1: **for** $i \leftarrow 0, m - 1$ **do**

2:     $c_i \leftarrow 0, sort \leftarrow true, \bar{z} \leftarrow$ the number of nonzero elements left

3:     **while** $sort$ is $true$ **do**

4:         $\boldsymbol{A_i} \leftarrow$ the elements of $\boldsymbol{A_i}$ sorted in ascending order

5:         $z_i \leftarrow$ index of first nonzero element

6:         $P_{i,c_i} \leftarrow$ the permutation of the indices of the nonzero elements

7:         **if** $z_i < (n - \bar{z})$ **then**

8:             **for** $j \leftarrow n - 1, z_i + 1$ **do**

9:                 $A_{i,j} \leftarrow A_{i,j} - A_{i,j-1}$

10:             **end for**

11:             $c_i \leftarrow c_i + 1$

12:         **else**

13:             $sort \leftarrow false$

14:         **end if**

15:     **end while**

16: **end for**

17: $\hat{A} \leftarrow A$

Table 5.1: An example for Algorithm 1

i	c_i	A_i	After Sorting A_i	z_i	P_{i,c_i}	After Subtraction A_i	$\hat{A}_i$
0	0	$\begin{bmatrix}3 & 1 & 4\end{bmatrix}$	$\begin{bmatrix}1 & 3 & 4\end{bmatrix}$	0	$\begin{bmatrix}1 & 0 & 2\end{bmatrix}$	$\begin{bmatrix}1 & 2 & 1\end{bmatrix}$	
	1	$\begin{bmatrix}1 & 2 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 1 & 2\end{bmatrix}$	0	$\begin{bmatrix}0 & 2 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 0 & 1\end{bmatrix}$	
	2	$\begin{bmatrix}1 & 0 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	1	$\begin{bmatrix}0 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$	
	3	$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$	$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$	2	$\begin{bmatrix}1\end{bmatrix}$		$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$
1	0	$\begin{bmatrix}0 & 2 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 2\end{bmatrix}$	1	$\begin{bmatrix}2 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	
	1	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	1	$\begin{bmatrix}1 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$	
	2	$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$	$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$	2	$\begin{bmatrix}1\end{bmatrix}$		$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$
2	0	$\begin{bmatrix}1 & 4 & 0\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 4\end{bmatrix}$	1	$\begin{bmatrix}0 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 3\end{bmatrix}$	
	1	$\begin{bmatrix}0 & 1 & 3\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 3\end{bmatrix}$	1	$\begin{bmatrix}1 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 2\end{bmatrix}$	
	2	$\begin{bmatrix}0 & 1 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 2\end{bmatrix}$	1	$\begin{bmatrix}1 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	
	3	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1 & 1\end{bmatrix}$	1	$\begin{bmatrix}1 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$	
	4	$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$	2	$\begin{bmatrix}1\end{bmatrix}$		$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$

An example of a preprocessing stage is given in Table 5.1 for the inputs

$$m = n = 3, \ \bar{z} = 1 \text{ and } A = \begin{bmatrix} 3 & 1 & 4 \\ 0 & 2 & 1 \\ 1 & 4 & 0 \end{bmatrix}.$$

The first column shows the indices of the rows of the matrix A and the second column c_i is the number of permutations produced for the row until the current step. When the last sort is applied for a row then the total number of permutations i.e sorts will be $c_i + 1$. In the columns titled as *After Sorting*, the sorted row A_i, the index of the first nonzero element z_i and the resulting permutation P_{i,c_i} are given. *After Subtraction* column presents the ith row of A after consequent subtractions over the elements of sorted A_i are applied.

For example, the first row of the table, $A_0 = \begin{bmatrix}3 & 1 & 4\end{bmatrix}$ is sorted and the vector $\begin{bmatrix}1 & 3 & 4\end{bmatrix}$ is revealed. The index of first nonzero element encountered in sorted A_i is 0 thus $z_i = 0$ and the permutation produced by sorting is $\begin{bmatrix}1 & 0 & 2\end{bmatrix}$. Since the required number of nonzero elements for ending the preprocessing of a row is $\bar{z} = 1$

and we have no nonzero elements then the subtraction step is applied. Beginning

with the last element, the previous element is subtracted. As a result A_0 becomes

$\begin{bmatrix} 1 & (3-1) & (4-3) \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$ and the preprocessing continues with the next

sorting step. When c_0 becomes 3, the A_0 is reduced to the vector $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ and

the desired number of nonzero elements is reached. The each row of the matrix is

preprocessed in the same manner. The algorithm stops when the preprocessing of all

the rows is finished. If the number of sorts is desired to be limited then the required

number of nonzero elements can be decreased. The affect of the number of sorts to

the performance of the algorithm is analysed in Section 5.3

5.2 Permutational Matrix-Vector Multiplication Stage

$A\boldsymbol{v} = \boldsymbol{u}$ is computed by successively permuting $\boldsymbol{v}$ according to permutation tables

P_{i,c_i} constructed at preprocessing stage and adding the permuted elements consec-

utively. Algorithm 8 shows how PMVM stage works and an example of a PMVM

stage is given in Table 5.2.

Table 5.2: An example for Algorithm 2

| | | | Permutation | | | Addition | |
i	c_i	w_i	P_{i,c_i}	w_i		w_i	u_i
0	0	$\begin{bmatrix} 4 & 10 & 8 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 2 \end{bmatrix}$	$\begin{bmatrix} 10 & 4 & 8 \end{bmatrix}$		$\begin{bmatrix} 0 & 1 & 8 \end{bmatrix}$	
	1	$\begin{bmatrix} 0 & 1 & 8 \end{bmatrix}$	$\begin{bmatrix} 0 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 8 & 1 \end{bmatrix}$		$\begin{bmatrix} 9 & 9 & 1 \end{bmatrix}$	
	2	$\begin{bmatrix} 9 & 9 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 2 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 9 & 1 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 10 & 1 \end{bmatrix}$	
	3	$\begin{bmatrix} \varnothing & 10 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 \end{bmatrix}$	$\begin{bmatrix} \varnothing & \varnothing & 10 \end{bmatrix}$			10
1	0	$\begin{bmatrix} 4 & 10 & 8 \end{bmatrix}$	$\begin{bmatrix} 2 & 1 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 8 & 10 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 7 & 10 \end{bmatrix}$	
	1	$\begin{bmatrix} \varnothing & 7 & 10 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 7 & 10 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 6 & 10 \end{bmatrix}$	
	2	$\begin{bmatrix} \varnothing & 6 & 10 \end{bmatrix}$	$\begin{bmatrix} 1 \end{bmatrix}$	$\begin{bmatrix} \varnothing & \varnothing & 6 \end{bmatrix}$			6
2	0	$\begin{bmatrix} 4 & 10 & 8 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 4 & 10 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 3 & 10 \end{bmatrix}$	
	1	$\begin{bmatrix} \varnothing & 3 & 10 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 3 & 10 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 2 & 10 \end{bmatrix}$	
	2	$\begin{bmatrix} \varnothing & 2 & 10 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 2 & 10 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 1 & 10 \end{bmatrix}$	
	3	$\begin{bmatrix} \varnothing & 1 & 10 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \end{bmatrix}$	$\begin{bmatrix} \varnothing & 1 & 10 \end{bmatrix}$		$\begin{bmatrix} \varnothing & 0 & 10 \end{bmatrix}$	
	4	$\begin{bmatrix} \varnothing & 0 & 10 \end{bmatrix}$	$\begin{bmatrix} 1 \end{bmatrix}$	$\begin{bmatrix} \varnothing & \varnothing & 0 \end{bmatrix}$			0

In the example, $A\boldsymbol{v} = \boldsymbol{u}$ is calculated over $\mathbb{F}_{11}$ for $\boldsymbol{v} = \begin{bmatrix} 4 & 10 & 8 \end{bmatrix}^T$ using permutation tables coming from Table 5.1. The notation '$\varnothing$' is used for unused vector entries. The first and second column in Table 5.2 show the indices i and c_i of the permutation tables, the other columns show the permutations that will be applied to the vectors and the state of the vectors after permutation and addition.

Algorithm 8 The PMVM Algorithm

Input: Preprocessed matrix $\hat{A}$, Permutation table P, the number of permutations per row c_i, $n-$vector $\boldsymbol{v}$

Output: $n-$vector $\boldsymbol{u} = A\boldsymbol{v}$

1: **for** $i \leftarrow 0, m - 1$ **do**

2:      $\boldsymbol{w} \leftarrow \boldsymbol{v}$

3:      $\hat{A} \leftarrow$ Preprocessed A

4:      **for** $j \leftarrow 0, c_i$ **do**

5:          $\boldsymbol{w} \leftarrow \boldsymbol{w}$ is permuted according to $P_{i,j}$

6:          **for** $l \leftarrow (n-2), (n-(\text{the number of elements in } P_{i,j})$ **do**

7:              $w_l \leftarrow w_l + w_{l+1}$

8:          **end for**

9:      **end for**

10:      $u_i \leftarrow 0$

11:      **for** $j \leftarrow 0, \bar{z}$ **do**

12:          $u_i \leftarrow u_i + \left(w_{n-1-j} \hat{A}_{n-1-j} \right)$

13:      **end for**

14: **end for**

For instance, the first permutation $P_{0,0} = \begin{bmatrix} 1 & 0 & 2 \end{bmatrix}$ is applied to vector $\begin{bmatrix} 4 & 10 & 8 \end{bmatrix}$ The

permuted vector is $\begin{bmatrix} 10 & 4 & 8 \end{bmatrix}$. After adding the elements over $\mathbb{F}_{11}$ successively, the vector $\boldsymbol{w_0}$ becomes $\begin{bmatrix} 0 & 1 & 8 \end{bmatrix}$ and it is assigned to $\boldsymbol{w_1}$. Similarly, $\boldsymbol{w_2}$ is obtained by applying the second permutation $P_{0,1} = \begin{bmatrix} 0 & 2 & 1 \end{bmatrix}$ and successive additions to the vector $\boldsymbol{w_1}$. The algorithms stops when all of the permutations are applied consecutively. The result of the product $\boldsymbol{u} = \begin{bmatrix} 10 & 6 & 0 \end{bmatrix}$ is given at the last column of the Table 5.2.

5.3 Analysis of The Algorithm

The performance of the algorithm is highly affected by the total number of sorts applied to the matrix. For each row of the matrix $c_i + 1$ number of sorts are applied. Every sorting operation runs over less elements than the previous one because of the subtraction step that takes place after a sorting process. Therefore the number of sorts is always less than the number of nonzero elements per row.

If the sorting algorithm chosen for the preprocessing stage involves comparing pairs of values then the worst-case time complexity for sorting can not be better than $O(n \log n)$. The matrices generated in the sieving phase of index calculus family of algorithms have entries that are limited by the bit size of the prime for prime fields. Therefore the entries of the matrix are all integers less than $k \leq n$. Moreover, the indices after sorting should be kept in order to use in the PMVM stage. Thus, the counting sort algorithm with time complexity $O(n)$ becomes the appropriate algorithm. Besides, if the CSR format is used when storing the matrix then the run time further decreases to $O(\bar{z})$.

In its simplest form, the index calculus algorithm chooses a random a and then calculates $g^a \in \mathbb{F}_p$. If g^a can be written as the product of prime powers of a factor base

$B = \{p_1, p_2, \cdots, p_n\}$ then exponents coming from the prime factorization constitute a row of A. Thus each row in A is an exponent vector. Since $a < p$, the exponents and consequently the entries of A are limited by the size of the prime p. If p is b-bits in size and the dimension of the matrix A is n then it can be observed that n is always much larger than b. Therefore the matrix A can be preprocessed efficiently by using the counting sort algorithm.

At the worst case, $(c_i+1)n$ additions are computed in PMVM stage. However according to the sparsity of the matrices only $\bar{z}$ element of the $n-$vector will be affected in first step and the number of elements that are affected decreases rapidly. As a result, the performance of the PMVM algorithm becomes faster than classical matrix-vector multiplication (CMVM) algorithm in which each row vector of the $m \times n$ matrix is multiplied with the n-vector. Thus CMVM needs n multiplications and $n-1$ additions per row.

CHAPTER 6

IMPLEMENTATION AND PERFORMANCE

In this chapter, implementation details of the proposed algorithm are explained. In addition, the performance test results are given and they are compared against a classical method, the sparse matrix-vector multiplication using CSR storage format (CSR-MVM) algorithm.

6.1 Implementation

In this section, the details about the development environment of the implementation and the generation procedures of the input matrices, vectors, and prime numbers are explained.

6.1.1 Development Environment

The code for the proposed algorithm is written in C++ language. MS Visual Studio 2017 environment is preferred for development. MS Visual Studio 2017 permits use of compilers like Clang or GNU Compiler Collection (GCC) other than Microsoft Visual C++ Compiler (MSVC) but since Windows operating system is targeted for

performance tests, MSVC is used as the compiler.

The preprocessing stage does not use multi-precision arithmetic but PMVM stage operates over finite fields and big integer support is needed for modular arithmetic. C++ standard library does not include a big integer class but some useful multi-precision arithmetic libraries can be found in open source. GMP [16] and MPIR [11] are two of them.

GMP is one of the widely used free libraries for arbitrary precision arithmetic performing operations on signed integers, rational numbers, and floating point numbers. In order to provide the speed, full words as the basic arithmetic type, and fast algorithms are used. In addition, the most common inner loops for variable CPUs are highly optimized by using assembly code. GMP's main target platforms are Unix-type systems but it also is known to work on Windows in both 32-bit and 64-bit mode.

The other portable library written in C for arbitrary precision arithmetic is MPIR that performs the arithmetic on integers, rational numbers, and floating-point numbers. The goal of MPIR is to provide high performance arithmetic for all applications entailing higher precision. MPIR began as a fork of GMP so they have a lot of code in common.

Although big integer libraries other than MPIR and GMP exist in open source, the support and sustainability issues lead us to choose one of them. We prefer MPIR over GMP in our implementation since it can be compiled by MS Visual Studio with optimized assembly language support.

6.1.2   Input Data Generation

The proposed algorithm takes the matrix and the vector as input and outputs the matrix-vector product. While generating the inputs, two decisions has to be made. The first one is the selection of the size n of the matrices and vectors, and the second one is the generation method. Since we desire to produce matrices such as those generated in the linear algebra phase of the index calculus algorithm, we began by implementing the sieving phase of the basic index calculus method for small prime fields. At most 64-bit primes and factor base size of 1000 were used when generating the relations. We observed that when the bit size doubles, the number of nonzero elements nearly doubles.

However as the field gets larger, the factor base and, accordingly, the size of the matrix grows too and the sieving phase for large finite fields becomes unaffordable in a single machine. Therefore, we chose to generate the matrices synthetically by using our observation for small fields as well as attempting to approximate the sparsity of the matrices used in [24] and [23].

For this purpose, we generated 1024-bit and 2048-bit prime numbers p_1 and p_2. Then random n-vectors for $n = 10,000$, $50,000$ and $100,000$ are generated over the fields $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$. All of the multi-precision integers are generated using MPIR library functions. $n \times n$ matrices with the number of nonzero elements $\bar{z} = 200$ and 400 are generated randomly using a pseudo random number generator. The values of the entries of the matrices are limited by the bit size of the primes.

For each set of $\{n, \bar{z}, b\}$, 25 matrices are generated and tested. After the generation of the sparse matrices, the CSR format is used to store them.

6.2 Performance Results

Performance results are obtained on a single core of a 3.7 GHz Xeon E3-1281 processor. PMVM with preprocessing algorithm is run on the matrices and the vectors generated randomly as described in the previous section. It is tested against the CSR-MVM method.

Figure 6.1 shows the increase in running time of PMVM with preprocessing algorithm versus the increase in matrix dimension n. It is obvious from the figure that, the running time of the algorithm grows with the dimension nearly in a linear trend.
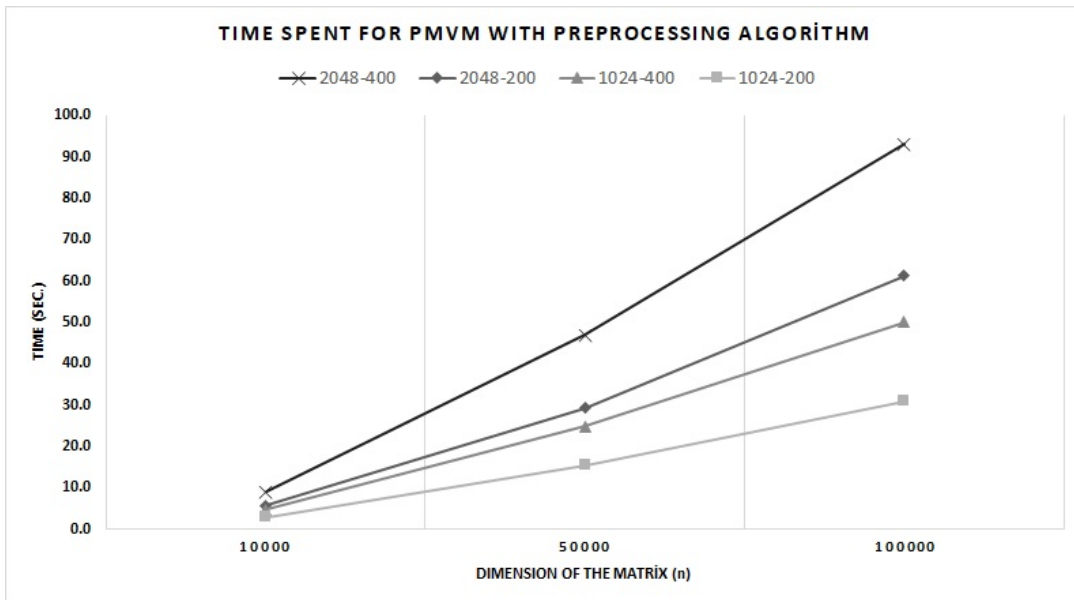
Figure 6.1: The speed of PMVM with preprocessing

Table 6.1 shows the time (in seconds) required by PMVM with preprocessing and CSRMVM for various $n, \bar{z}$ and b. As used in earlier notation n represents the dimension of the matrix and the vector, $\bar{z}$ is the number of nonzero elements per row and b is the bit size of the primes used to generate prime fields. The time required by the preprocessing stage and the PMVM stage is shown separately in the fourth and fifth

columns. The sixth column is the total time needed for the proposed algorithm. The sparse matrix-vector multiplication using CSR format time requirement is given in the seventh column. Finally the eight column shows the speed improvement gained by the PMVM with preprocessing against CSRMVM.

As can be seen from Table 6.1, the PMVM with preprocessing algorithm is at least 34% faster than CSRMVM for all the matrices tested. It can be observed that as the number of nonzeros increases, the algorithm performs better. While the speed improvement for $\bar{z} = 200$ ranges between 34% and 61%, the interval is 56%-77% for $\bar{z} = 400$.

In the linear algebra phase of the index calculus algorithm, either Lanczos or Wiedemann algorithms apply at least n iterative matrix-vector multiplications. According to the results obtained by the proposed algorithm, for $n = 100,000$ iterations nearly 47 days can saved when compared to CSRMVM. It should be noted that the preprocessing stage is run only once even if the iteration count is bigger than 1.

Table 6.1: Time required by PMVM with Preprocessing and CSRMVM

n	$\bar{z}$	b	Running Time (s)			
			Preprocessing	**PMVM**	**PMVM with Preprocessing**	**CSRMVM**
10000	200	1024	0.2	2.7	2.9	4.7
		2048	0.3	5.3	5.6	8.1
	400	1024	0.2	4.5	4.7	8.2
		2048	0.3	8.6	9	14.5
50000	200	1024	1.2	14.1	15.3	24.3
		2048	2.3	26.8	29.1	41.8
	400	1024	1.4	23.4	24.8	41.9
		2048	2.3	44.5	46.9	73.5
100000	200	1024	2.4	28.3	30.8	48
		2048	4.5	56.6	61.2	82.1
	400	1024	2.6	47.2	49.9	88.4
		2048	4.4	88.5	92.9	148.6

CHAPTER 7

CONCLUSION

In this chapter, the work reported in this thesis is summarized and the need for further research in this area is described.

In this thesis, we proposed a new algorithm for computing large sparse matrix-vector multiplications over finite fields. The main focus is the matrices that are involved in the linear systems revealed by the basic index calculus algorithm. We began with the preliminaries prior to the subject. The details of the discrete logarithm problem and the basic index calculus method are provided. In addition, The basic knowledge for sorting algorithms is given. In the third chapter, the characteristics of the linear systems that are induced by index calculus algorithm are given and the two solvers namely Lanczos and Wiedemann algorithms are explained.

The fourth chapter is devoted to the subject of the sparse matrix-vector multiplication. the difficulties of the SpMVM operation on modern processors are described along with the solutions.

In the fifth chapter we proposed new algorithm for matrix-vector multiplication over finite-fields. The proposed algorithm, permutational matrix-vector multiplication with

preprocessing, consists of two stages. The first stage is the preprocessing stage. The preprocessing sequentially sorts the rows of the matrix and applies addition on sorted elements in order to reveal permutation tables which are used to permute the vector elements in the second stage called permutational matrix-vector multiplication. The algorithm has advantages over the classical method by means of computational performance. The use of additions instead of multiplications and computing less modular reductions make the algorithm more efficient. We obtained improvements between 7% and 61% in comparative performance results. The implementation details and the performance results are detailed in chapter six.

Sparse matrix-vector multiplication is a basic kernel in many applications. It has special importance in cryptographic computations since the security of many widely used systems such as secure communication systems depends on the difficulty of discrete logarithm problem and the best known method for solving discrete logarithm problem, the index calculus algorithm, deals with many iterative sparse matrix-vector multiplications. As a result, developing faster algorithms for sparse matrix-vector multiplications can eliminate the difficulty of the linear algebra phase that takes place in the index calculus algorithms.

For further research, both stages of the proposed algorithm can be parallelized using CPUs and GPUs. For practical purposes, we imitate the index calculus matrices in our performance test. Using more processors, the real life index calculus matrices with bigger sizes can be generated and tested. Since our study targets the matrices generated with index calculus algorithms, we focused on sparse matrices. But the performance results showed us that the proposed algorithm can exhibit better performance results for dense matrices as well. The PMVM with preprocessing algorithm

can also be adapted and tested against the algorithms developed for structured matrices.

REFERENCES

[1] L. M. Adleman and J. DeMarrais, A subexponential algorithm for discrete logarithms over all finite fields, in *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, pp. 147–158, 1993.

[2] D. Coppersmith, Solving homogeneous linear equations over GF(2) via block wiedemann algorithm, Math. Comput., 62(205), pp. 333–350, January 1994.

[3] D. Coppersmith, A. M. Odlzyko, and R. Schroeppel, Discrete logarithms in GF(p), Algorithmica, 1(1), pp. 1–15, Nov 1986, ISSN 1432-0541.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, MIT Press, 2nd edition, 2001, ISBN 0262032937.

[5] A. Das, *Computational Number Theory*, Chapman and Hall-CRC, 2013, ISBN 9781439866153.

[6] T. Dierks and E. Rescorla, The Transport Layer Security (TLS) protocol version 1.2, in *RFC 5246*, 2008.

[7] W. Diffie and M. Hellman, New directions in cryptography, IEEE Trans. Inf. Theor., 22(6), pp. 644–654, September 2006, ISSN 0018-9448.

[8] J. G. Dumas, T. Gautier, P. Giorgi, and C. Pernet, Dense linear algebra over finite fields: the FFLAS and FFPACK packages, CoRR, abs/cs/0601133, 2006.

[9] T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, in G. R. Blakley and D. Chaum, editors, *Advances in Cryptology*, pp. 10–18, Springer Berlin Heidelberg, Berlin, Heidelberg, 1985, ISBN 978-3-540-39568-3.

[10] P. Giorgi and B. Vialla, Generating optimized sparse matrix vector product over finite fields, in H. Hong and C. Yap, editors, *Mathematical Software – ICMS 2014*, pp. 685–690, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, ISBN 978-3-662-44199-2.

[11] B. Gladman, W. Hart, J. Moxham, et al., MPIR: Multiple Precision Integers and Rationals, 2017, version 3.0.0, `http://mpir.org`.

[12] F. Göloğlu, R. Granger, G. McGuire, and J. Zumbrägel, Solving a 6120-bit dlp on a desktop computer, in T. Lange, K. Lauter, and P. Lisoněk, editors, *Selected*

Areas in Cryptography – SAC 2013, pp. 136–152, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, ISBN 978-3-662-43414-7.

[13] D. M. Gordon, Discrete logarithms in GF(P) using the number field sieve, SIAM J. Discrete Math., 6, pp. 124–138, 1993.

[14] R. Granger, T. Kleinjung, and J. Zumbrägel, Breaking '128-bit secure' supersingular binary curves, in J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pp. 126–145, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, ISBN 978-3-662-44381-1.

[15] R. Granger, T. Kleinjung, and J. Zumbrägel, On the discrete logarithm problem in finite fields of fixed characteristic, volume 370, pp. 3129–3145, 2017.

[16] T. Granlund et al., GNU-multiple precision arithmetic library 6.1.2, December 2016, https://gmplib.org/.

[17] M. R. Hestenes and E. Stiefel, Method of conjugate gradients for solving linear systems, J. Res. Nat. Bur. Standards, 49, pp. 409–436, 1952.

[18] A. Joux, *Algorithmic Cryptanalysis*, Chapman & Hall/CRC, 1st edition, 2009, ISBN 1420070029, 9781420070026.

[19] A. Joux, A. Odlyzko, and C. Pierrot, *The Past, Evolving Present, and Future of the Discrete Logarithm*, pp. 5–36, Springer International Publishing, Cham, 2014, ISBN 978-3-319-10683-0.

[20] A. Joux and C. Pierrot, Technical history of discrete logarithms in small characteristic finite fields, Designs, Codes and Cryptography, 78(1), pp. 73–85, Jan 2016, ISSN 1573-7586.

[21] E. Kaltofen, Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems, Math. Comput., 64(210), pp. 777–806, April 1995, ISSN 0025-5718.

[22] C. Kerry and P. Gallagher, FIPS PUB 186-4: Digital Signature Standard (DSS), NIST, 2013.

[23] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. J. J. te Riele, A. Timofeev, and P. Zimmermann, Factorization of a 768-bit RSA modulus, in *Advances in Cryptology - CRYPTO 2010*, pp. 333–350, 2010.

[24] T. Kleinjung, C. Diem, A. K. Lenstra, C. Priplata, and C. Stahlke, Computation of a 768-bit prime field discrete logarithm, in *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pp. 185–201, 2017.

[25] M. Kraitchik, *Théorie Des Nombres*, Paris : Gauthier-Villars, 1922.

[26] B. A. LaMacchia and A. M. Odlyzko, Computation of discrete logarithms in prime fields, Des. Codes Cryptography, 1, pp. 47–62, 1991.

[27] C. Lanczos, Solution of systems of linear equations by minimized iterations, J. Res. Natl. Bur. Stand, 49, pp. 33–53, 1952.

[28] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996, ISBN 0849385237.

[29] R. C. Merkle, Secure communications over insecure channels, Commun. ACM, 21(4), pp. 294–299, April 1978, ISSN 0001-0782.

[30] P. L. Montgomery, A block lanczos algorithm for finding dependencies over GF(2), in *Advances in Cryptology - EUROCRYPT '95, International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France, May 21-25, 1995, Proceeding*, pp. 106–120, 1995.

[31] A. M. Odlyzko, Discrete logarithms in finite fields and their cryptographic significance, in *Advances in Cryptology: Proceedings of EUROCRYPT 84, A Workshop on the Theory and Application of of Cryptographic Techniques, Paris, France, April 9-11, 1984, Proceedings*, pp. 224–314, 1984.

[32] L. Odlyzko, B. A. Lamacchia, and A. M. Odlyzko, Solving large sparse linear systems over finite fields, pp. 109–133, Springer, 1991.

[33] S. Pohlig and M. Hellman, An improved algorithm for computing logarithms over and its cryptographic significance (corresp.), IEEE Trans. Inf. Theor., 24(1), pp. 106–110, September 2006, ISSN 0018-9448.

[34] J. M. Pollard, Monte carlo methods for index computation (mod p), volume 32, pp. 918–924, 1978.

[35] C. Pomerance, The quadratic sieve factoring algorithm, in *Advances in Cryptology: Proceedings of EUROCRYPT 84,*, pp. 169–182, 1984.

[36] C. Pomerance, A tale of two sieves, volume 43, pp. 1473–1485, 1996.

[37] E. Rescorla, The transport layer security (tls) protocol version 1.3, in *RFC 8446*, 2018.

[38] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003, ISBN 0898715342.

[39] D. Shanks, Class number, a theory of factorization and genera, Proceedings of Symposium of Pure Mathematics, 20, pp. 415–440, 1969.

[40] C. Studholme, Discrete logarithm problem, in *Research paper requirement (milestone) of the Ph.D. program at the University of Toronto*, 2002.

[41] A. E. Western and J. C. P. Miller, Tables of indices and primitive roots., in *Royal Society Mathematical Tables*, volume 9, Cambridge University Press, Cambridge, 1968.

[42] D. H. Wiedemann, Solving sparse linear equations over finite fields, volume 32, pp. 54–62, January 1986.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Mangır, Ceyda

Nationality: Turkish (TC)

Date and Place of Birth: 16.10.1979, İstanbul

Marital Status: Married

EDUCATION

Degree	Institution	Year of Graduation
M.S.	Institute of Applied Mathematics, METU	2007
B.S.	Mathematics Engineering, Yıldız Technical University	2002
High School	Üsküdar Fen Lisesi	1998

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2004-	The Research and Development Institute, Ankara, TURKEY	Specialist

PUBLICATIONS

International Conference Publications

C. Mangır, M. Cenk, M. Manguoğlu. *An Improved Algorithm for Iterative Matrix-Vector Multiplications over Finite Fields*. In: Lanet JL., Toma C. (eds) Innovative Security Solutions for Information Technology and Communications. SECITC 2018. Lecture Notes in Computer Science, vol 11359. Springer, Cham, 2018