A history of the application of MDS matrices in cryptography

Ana Clara Zoppi Serpa
Prof. Dr. Ricardo Dahab
Dr. Jorge Nakahara Jr.

November 23, 2021

Contents

Chapter 1

A history of the application of MDS matrices in cryptography

MDS matrices have been widely used in the construction of diffusion layers for block ciphers such as SHARK [14], SQUARE [6], BKSQ [7], KHAZAD [3], ANUBIS [2], Hierocrypt-3 [5], Rijndael (AES) [8] and Curupira [4]. They have also been applied in the design of hash functions (e.g Whirlwind [1] and Grøstl [9]). The choice is due to the fact that MDS codes provide transformations with optimal linear and differential branch numbers (see e.g [6] or [14]), thus contributing to security against Differential and Linear Cryptanalysis attacks.

Due to the computational cost of matrix multiplication, there is an interest in finding MDS matrices with coefficients as small as possible, in order to minimize the required amount of **xor** and **xtime** operations required by the implementations. However, the complexity of finding MDS matrices through random search increases proportionally to the dimension, which led to the investigation of methods to construct (or find) MDS matrices. One possible avenue is trying to find direct mathematical constructions which ensure the MDS property, and another is to impose restrictions to limit the random search space (e.g imposing the matrix should be circulant, as was done by the authors of [6]). Furthermore, there is an interest in finding involutory MDS matrices (as pointed by [3] and [2]), so that the encryption and the decryption computational cost are the same.

In this chapter, we aim at providing a history of the application of MDS matrices in cryptography, listing the matrices, the ciphers in which they have been applied, the respective Finite Fields (order and irreducible polynomial), and their cost (amount of **xor** and **xtime** operations).

Note: this is a partial report. For the moment, it contains only Preliminaries and information about the ciphers SHARK and SQUARE. It will be expanded in the future.

We assume the reader is familiar with:

- Linear branch number (see Chapter X)
- Differential branch number (see Chapter X)
- Differential Cryptanalysis (see Chapter X)
- Linear Cryptanalysis (see Chapter X)
- MDS codes (see Chapter X and, for further detail, reference [10])
- Diffusion property in cryptography (see Chapter X)
- Groups, rings and fields in abstract algebra (see Chapter X)

1.1 Notation

- $det(A)$: determinant of the matrix A
- A^{-1}: inverse matrix of A
- n, k, d: parameters of a code
- $\mathcal{C}$: a code
- G: generator matrix of a code
- I: identity matrix
- $[IB]$: matrix obtained by placing the $n \times n$ matrix B to the right of the $n \times n$ identity matrix I. For example, for $B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $[IB] = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 3 & 4 \end{bmatrix}$

1.2 Acronyms

- MDS: Maximum Distance Separable
- SHARK: refers to the SHARK cipher [14]
- SQUARE: refers to the SQUARE cipher [6]
- BKSQ: refers to the BKSQ cipher [7]
- KHAZAD: refers to the KHAZAD cipher [3]
- ANUBIS: refers to the ANUBIS cipher [2]
- Hierocrypt-3: refers to the Hierocrypt-3 cipher [5]
- Rijndael: refers to the Rijndael cipher which later became AES [8]
- AES: Advanced Encryption Standard (Rijndael's name after being chosen by NIST)

- Curupira: refers to the Curupira cipher [4]
- Whirlwind: refers to the Whirlwind hash function [1]
- Grøstl: refers to the Grøstl hash function [9]
- **xor**: bitwise XOR between two integers
- **xtime**: refers to the multiplication by the x polynomial in the cipher's Finite Field

1.3 Preliminaries

1.3.1 Matrices

Obs: eu escrevi essas primeiras definições (matriz singular, involutória, circulante, circulante à esquerda, circulante à direita) com base no que eu lembrava de matemática mesmo, então por hora ainda não coloquei uma referência bibliográfica, já que são definições mais gerais e não chegam a ser específicas de cripto. Mas posso colocar depois se necessário.

(Singular matrix) *A square matrix A is singular if $det(A) = 0$. Respectively, A is non-singular if $det(A) \neq 0$.*

(Involutory matrix) *A square matrix A is involutory if $A \times A^{-1} = I$, where I is the identity matrix.*

(Circulant matrix) *A square matrix A is involutory if $A \times A^{-1} = I$, where I is the identity matrix.*

(Right circulant matrix) *An $n \times n$ matrix A is circulant if each row i is formed by a cyclical shift of i positions of the same set of elements $\{a_0, a_1, a_2, ..., a_{n-1}\}$, i.e*

$$A = \begin{bmatrix} a_0 & a_1 & ... & ... & a_{n-1} \\ a_{n-1} & a_0 & a_1 & ... & a_{n-2} \\ ... & ... & ... & ... & ... \\ a_1 & ... & a_{n-2} & a_{n-1} & a_0 \end{bmatrix}.$$

(Left circulant matrix) *A circulant matrix in which the shift is a cyclical shift to the left.*

(Right circulant matrix) *A circulant matrix in which the shift is a cyclical shift to the right.*

Theorem 1 (MDS matrices and singularity [6]) *An (n, k, d)-code $\mathcal{C}$ with generator matrix $G = [IB]$ is MDS if and only if every square submatrix of B is non-singular.*

Evaluating a matrix for MDS property

If Theorem 1 is used, the determinant of the matrix itself must be calculated, as well as the determinants of its submatrices, and we must check that they are non-zero. This can be done in $O(n!^2)$ time with a recursive implementation to compute the determinants.

Computational complexity to obtain the determinant

The determinant of a 2×2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ can be computed in constant time by means of the formula $ad - bc$. The determinant of a 3×3 matrix can be computed by calculating cofactors of 2×2 submatrices obtained by removing a row i and a column j. There are $3^2 = 9$ such submatrices, therefore 3^2 determinants of 2×2 matrices are calculated. The determinant of a 4×4 matrix can be computed similarly by calculating the cofactors of $4^2 = 16$ submatrices which are 3×3 and thus require 3^2 determinants of 2×2 submatrices each, therefore totalizing $4^2 \times 3^2$ determinants of 2×2 submatrices. For a 5×5 matrix, the total amount is $5^2 \times 4^2 \times 3^2$, and so forth. Therefore, for an $n \times n$ matrix, the complexity is bounded by $n!^2$ operations (in asymptotic notation, $O(n!^2)$).

1.3.2 Abstract algebra

Aqui pretendo colocar definições de grupo, grupo abeliano, corpo etc. A parte de álgebra abstrata que não é específica de corpos finitos e que a gente geralmente vê na faculdade

1.3.3 Finite fields — GF(2^m)

(Finite field [8]) *A* finite field *is a field with a finite number of elements. The number of elements in the set is called the* order *of the field.*

(Characteristic and order [8]) *A field with order m exists if and only if m is a prime power, i.e $m = p^n$ for some integer n, where p is a prime integer. p is called the* characteristic *of the field. For each prime power there is exactly one finite field, denoted by GF(p^n).*

(Representing finite fields with prime order [8]) *Elements of a finite field GF(p) can be represented by the integers $0, 1, ..., p-1$, and the field operations are integer addition modulo p and integer multiplication modulo p.*

(Representing finite fields with non-prime order [8]) *For finite fields with an order that is not prime, addition and multiplication cannot be represented by addition and multiplication modulo a number. One of the possible representations for GF(p^n) is by means of* polynomials over GF(p).

In this chapter, we focus particularly on fields with characteristic $p = 2$, due to their wide application in cryptography.

(Polynomial [8]) *A polynomial over a field $\mathbb{F}$ is an expression of the form*

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + ... + b_2x^2 + b_1x + b_0,$$

where x is the indeterminate *and $b_i \in \mathbb{F}$ are the coefficients. The* degree *of the polynomial equals l if $b_j = 0$ for all $j > l$ and l is the smallest number with this property.*

Addition and multiplication are defined on polynomials as follows.

(Polynomial addition [8]) *Summing two polynomials $a(x)$ and $b(x)$ consists of summing the coefficients with equal powers of x, with the sum occuring in the underlying field $\mathbb{F}$. The neutral element is 0 (the polynomial with all coefficients equal to 0). The inverse element can be found by replacing each coefficient by its inverse element in $\mathbb{F}$. The degree of $a(x) + b(x)$ is at most the maximum of the degrees of $a(x)$ and $b(x)$, therefore addition is closed.*

For polynomials over GF(2) stored as integers in a cryptographic software implementation, addition can be implemented with a bitwise XOR instruction.

(Polynomial multiplication [8]) *In order to make multiplication closed, we select a polynomial $m(x)$ of degree l, called the* reduction polynomial*. Multiplication of $a(x)$ and $b(x)$ is then defined as the algebraic product of the polynomials modulo the reduction polynomial $m(x)$.*

The neutral element is 1 (the polynomial of degree 0 and with coefficient of x^0 equal to 1). The inverse element of $a(x)$ is $a^{-1}(x)$ such that $a(x) \times a^{-1}(x) = 1$.

For polynomials over GF(2) stored as integers in a cryptographic software implementation, multiplication by x can be implemented as a logical bit shift followed by conditional XOR (i.e subtraction) of the reduction polynomial (the **xtime** operation). Multiplication by other polynomials can be implemented as a series of **xtime**.

The reduction polynomial is usually chosen as an irreducible polynomial.

(Irreducible polynomial [8]) *A polynomial $d(x)$ is irreducible over the field GF(p) if and only if there exist no two polynomials $a(x)$ and $b(x)$ with coefficients in GF(p) such that $d(x) = a(x) \times b(x)$, where $a(x)$ and $b(x)$ are of degree greater than 0.*

For further reference on abstract algebra and Finite Fields, the reader may refer to [11], [13] and [12].

1.3.4 Computational cost unit

Computational cost of multiplication in GF(2^8)

Consider T a state byte, which we multiply by the polynomial $2e_x = 00101110_2 = x^5 + x^3 + x^2 + x$ in GF(2^8). Note that

$$T \cdot 2e_x = T \cdot x^5 + T \cdot x^3 + T \cdot x^2 + T \cdot x = T \cdot x \cdot x \cdot x \cdot x \cdot x + T \cdot x \cdot x \cdot x + T \cdot x \cdot x + T \cdot x,$$

where $\cdot$ denotes multiplication and $+$ denotes addition (which, in GF(2^8), is equivalent to a bitwise XOR). Multiplication by the x polynomial is performed by **xtime**, and addition is performed by **xor**.

Let $T \cdot x = Y$. Then $T \cdot 2e_x = Y + Y \cdot x + Y \cdot x \cdot x + Y \cdot x \cdot x \cdot x \cdot$.

Let $Y \cdot x = W$. Then $T \cdot 2e_x = Y + W + W \cdot x + W \cdot x \cdot x \cdot x$.

Let $W \cdot x = Z$. Then $T \cdot 2e_x = Y + W + Z + Z \cdot x \cdot x$.

The total number of **xtime** operations in this process is 5 (1 to obtain Y from T, 1 to obtain W from Y, 1 to obtain Z from W, 2 to compute $Z \cdot x \cdot x$), since we can reuse intermediate **xtime** calls. The total number of **xor** operations is 3. For multiplication in GF(2^8), in the worst case, 7 **xtime** would be necessary, since the maximum degree of polynomials in GF(2^8) is 7.

Computational cost of a matrix

The computational cost of an n matrix A is given by the necessary **xor** and **xtime** operations when multiplying a $n \times 1$ column vector by A. As an example, we calculate the cost of Matrix 1.4, used in the SQUARE [6] and AES [8] ciphers.

A row of Matrix 1.4 contains the elements $01_x = 1, 02_x = x$ and $03_x = x + 1$ only. Multiplying by 01_x does not require **xtime** or **xor**, since $01_x \cdot T = T$. Computing $02_x \cdot T = x \cdot T$ requires 1 **xtime**. Computing $03_x \cdot T = (x + 1) \cdot T = T \cdot x + T$ requires 1 **xtime** and 1 **xor**. Furthermore, adding the row multiplication results costs 3 **xor**. Therefore, the cost of a row is 2 **xtime** and 4 **xor**. Equation 1.1 illustrates this, with t_1, t_2, t_3 and t_4 being bytes of the state column vector.

$$\begin{bmatrix} 02_x & 01_x & 01_x & 03_x \end{bmatrix} \cdot \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{bmatrix} = 02_x \cdot t_1 + 01_x \cdot t_2 + 01_x \cdot t_3 + 03_x \cdot t_4 \qquad (1.1)$$

Note that Matrix 1.4 contains 4 rows, yielding a total cost of 8 **xtime** and 16 **xor**.

1.4 MDS matrix catalogue

In Table 1.1, the **Inv** column refers to whether they are involutory or not, **#xor** refers to the necessary amount of **xor** operations, **#xtime** refers to the necessary amount of **xtime** operations.

Year	**Order**	**Type**	**Inv**	**Use**	$GF(2)[x]/p(x)$	**#xor**	**#xtime**	**Matrices**
1996	8	—	no	SHARK [14]	$x^8 + x^7 + x^6 + x^5$ $+x^4 + x^2 + 1$	235, inverse: 223	369, inverse: 393	1.2, inverse: 1.3

1997	8	right circulant	no	SQUARE [6]	$x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$	16, inverse: 40	8, inverse: 48	1.4, inverse: 1.5
1998				BKSQ [7]				
2000			yes	KHAZAD [3]				
2000			yes	ANUBIS [2]				
2000				Hierocrypt-3 [5]				
2002				Rijndael (AES) [8]				
2007				Curupira [4]				
2009				Grøstl [9]				
2010				Whirlwind [1]				

Table 1.1: MDS matrix usage and cost

Matrix 1.2 and its inverse (1.3) are used in the SHARK [14] cipher.

$$\begin{bmatrix} ce_x & 95_x & 57_x & 82_x & 8a_x & 19_x & b0_x & 01_x \\ e7_x & fe_x & 05_x & d2_x & 52_x & c1_x & 88_x & f1_x \\ b9_x & da_x & 4d_x & d1_x & 9e_x & 17_x & 83_x & 86_x \\ d0_x & 9d_x & 26_x & 2c_x & 5d_x & 9f_x & 6d_x & 75_x \\ 52_x & a9_x & 07_x & 6c_x & b9_x & 8f_x & 70_x & 17_x \\ 87_x & 28_x & 3a_x & 5a_x & f4_x & 33_x & 0b_x & 6c_x \\ 74_x & 51_x & 15_x & cf_x & 09_x & a4_x & 62_x & 09_x \\ 0b_x & 31_x & 7f_x & 86_x & be_x & 05_x & 83_x & 34_x \end{bmatrix} \tag{1.2}$$

$$\begin{bmatrix} e7_x & 30_x & 90_x & 85_x & d0_x & 4b_x & 91_x & 41_x \\ 53_x & 95_x & 9b_x & a5_x & 96_x & bc_x & a1_x & 68_x \\ 02_x & 45_x & f7_x & 65_x & 5c_x & 1f_x & b6_x & 52_x \\ a2_x & ca_x & 22_x & 94_x & 44_x & 63_x & 2a_x & a2_x \\ fc_x & 67_x & 8e_x & 10_x & 29_x & 75_x & 85_x & 71_x \\ 24_x & 45_x & a2_x & cf_x & 2f_x & 22_x & c1_x & 0e_x \\ a1_x & f1_x & 71_x & 40_x & 91_x & 27_x & 18_x & a5_x \\ 56_x & f4_x & af_x & 32_x & d2_x & a4_x & dc_x & 71_x \end{bmatrix} \tag{1.3}$$

Matrix 1.4 and its inverse (1.5) are used in the SQUARE [6] cipher.

$$\begin{bmatrix} 02_x & 01_x & 01_x & 03_x \\ 03_x & 02_x & 01_x & 01_x \\ 01_x & 03_x & 02_x & 01_x \\ 01_x & 01_x & 03_x & 02_x \end{bmatrix} \tag{1.4}$$

$$\begin{bmatrix} 0e_x & 09_x & 0d_x & 0b_x \\ 0b_x & 0e_x & 09_x & 0d_x \\ 0d_x & 0b_x & 0e_x & 09_x \\ 09_x & 0d_x & 0b_x & 0e_x \end{bmatrix} \tag{1.5}$$

1.5 Computing xtime and xor of the matrices

One can compute the costs manually, however, the following following C code can also be used for this purpose, considering that polynomials in GF(2^8) are stored in integers (a set bit means coefficient equal to 1, a zero bit means coefficient equal to 0).

For e.g SHARK and SQUARE, the field order is equal to 8, therefore `ORDER` must be set to 8 and `DEGREE_LIMIT_MASK` must be set to x^8.

```
#define DEGREE_LIMIT_MASK 0x100
#define ORDER 8
```

The following function obtains the amount of **xtime** required to multiply by the polynomial.

```
unsigned int poly_xtime_cost(unsigned int poly) {
        unsigned int degree_mask = DEGREE_LIMIT_MASK;
        unsigned int degree = ORDER;
        while ((poly & degree_mask) == 0) {
                degree_mask >>= 1;
                degree--;
        }
        return degree;
}
```

The following function obtains the amount of **xor** required to multiply by the polynomial.

```
unsigned int poly_xor_cost(unsigned int poly) {
        unsigned int mask = 1;
        unsigned int set_bits = 0;
        unsigned int current_bit = 0;
        while (current_bit <= ORDER) {
                set_bits += ((poly & mask) != 0);
                mask <<= 1;
                current_bit++;
        }
        return set_bits - 1;
}
```

And, to compute the **xtime** and **xor** costs of a matrix, we must sum the costs of each row, which is accomplished by the following functions. Note that for e.g SHARK `DIM` must be set to 8, for SQUARE, to 4, and so forth.

```
#define DIM 8

unsigned int matrix_xtime_cost(unsigned int mat [DIM][DIM]) {
        unsigned int total_cost = 0;
        for (int row = 0; row < DIM; row++) {
                unsigned int row_cost = 0;
                for (int col = 0; col < DIM; col++) {
                        row_cost += poly_xtime_cost(mat[row][col]);
                }
                printf("Row %d costs %d xtime\n", row, row_cost);
                total_cost += row_cost;
        }
        printf("The full matrix costs %d xtime\n", total_cost);
        return total_cost;
}

unsigned int matrix_xor_cost(unsigned int mat[DIM][DIM]) {
        unsigned int total_cost = 0;
        for (int row = 0; row < DIM; row++) {
                unsigned int row_cost = DIM - 1; //sum elements
                for (int col = 0; col < DIM; col++) {
                        row_cost += poly_xor_cost(mat[row][col]);
                }
                printf("Row %d costs %d xor\n", row, row_cost);
                total_cost += row_cost;
        }
        printf("The full matrix costs %d xor\n", total_cost);
        return total_cost;
}
```

1.5.1 SQUARE manual calculation example

The computational cost for Matrix 1.2, used in the SQUARE cipher, was explained in Section 1.3.4.

For Matrix 1.5, used in SQUARE's decryption process, each row contains elements from $\{0e_x, 09_x, 0d_x, 0b_x\}$.

$$0e_x = 00001110_2 = x^3 + x^2 + 1 \text{ requires 3 } \textbf{xtime} \text{ and 2 } \textbf{xor}$$

$$09_x = 00001001_2 = x^3 + 1 \text{ requires 3 } \textbf{xtime} \text{ and 1 } \textbf{xor}$$

$$0d_x = 00001101_2 = x^3 + x^2 + 1 \text{ requires 3 } \textbf{xtime} \text{ and 2 } \textbf{xor}$$

$$0b_x = 00001011_2 = x^3 + x + 1 \text{ requires 3 } \textbf{xtime} \text{ and 2 } \textbf{xor}$$

There are 3 **xor** to add the intermediate row multiplication results, totalizing 12 **xtime** and 10 **xor** per row. There are 4 rows, hence 48 **xtime** and 40 **xor**.

1.6 Conclusions

yet to be written

Bibliography

[1] Paulo S. L. M. Barreto, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Elmar Tischhauser. Whirlwind: a new cryptographic hash function. *Des. Codes Cryptogr.*, 56(2-3):141–162, 2010.

[2] Paulo S. L. M. Barreto and Vincent Rijmen. The ANUBIS block cipher. In *First NESSIE Workshop, Heverlee, Belgium*, 2000.

[3] Paulo S. L. M. Barreto and Vincent Rijmen. The KHAZAD Legacy-Level block cipher. In *First NESSIE Workshop, Heverlee, Belgium*, 2000.

[4] PSLM Barreto and M Simplicio. Curupira, a block cipher for constrained platforms. *Anais do 25o Simpsio Brasileiro de Redes de Computadores e Sistemas Distribudos-SBRC*, 1:61–74, 2007.

[5] Toshiba Corporation. Specification of Hierocrypt-3. In *First NESSIE Workshop, Heverlee, Belgium*, 2000.

[6] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher square. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[7] Joan Daemen and Vincent Rijmen. The block cipher BKSQ. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, volume 1820 of *Lecture Notes in Computer Science*, pages 236–245. Springer, 1998.

[8] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[9] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl - a SHA-3 candidate. In Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway, editors, *Symmetric Cryptography, 11.01.*

- *16.01.2009*, volume 09031 of *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.

[10] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North-holland Publishing Company, 2nd edition, 1978.

[11] A.M. Masuda and D. Panario. *Topicos de corpos finitos com aplicacoes em criptografia e teoria de codigos.* Publicações matemáticas. IMPA, 2007.

[12] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., USA, 1st edition, 1996.

[13] G.L. Mullen and D. Panario. *Handbook of Finite Fields.* Discrete Mathematics and Its Applications. CRC Press, 2013.

[14] Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik De Win. The cipher SHARK. In Dieter Gollmann, editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 1996.