

Grøstl - funkcja skrótu BDAN

Mateusz Plichta, Karol Żelazowski

Czerwiec 2023

Spis treści

1	Czym jest funkcja skrótu	2
2	Grøstl - założenia	2
3	Grøstl - w jaki sposób działa	2
3.1	Funkcja kompresji	3
3.2	Transformacja wyjściowa	3
3.3	Permutacja P i Q	3
3.3.1	Mapowanie sekwencji bajtów do macierzy	4
3.3.2	AddRoundConstant	4
3.3.3	SubBytes	5
3.3.4	ShiftBytes	5
3.3.5	MixBytes	6
3.4	Funkcja Omega	6
3.5	IV - wektor inicjujący	6
3.6	Padding	7
4	Grøstl - implementacja	7
4.1	Padding	7
4.2	Podzielenie wiadomości na bloki	8
4.3	Permutacja P i Q	8
4.3.1	Mapowanie macierzy	9
4.3.2	addRoundConstant P i Q	9
4.3.3	SubBytes	10
4.3.4	ShiftBytes P i Q	10
4.3.5	MixBytes	11
4.4	Wywołanie funkcji kompresji	12
4.5	Funkcja kompresji	12
4.6	Funkcja Omega	12
4.7	Forma Hexadecymalna	13
4.8	wywoływanie funkcji haszującej	13
5	Testy	13
6	Tryby użytkowania dla Grøstl	14
7	Bezpieczeństwo Groestla	14
8	Wnioski	14
9	Bibliografia	15

1 Czym jest funkcja skrótu

Funkcja skrótu - funkcja, która przyporządkowuje dowolnie dużej liczbie znaków ciąg bitów o stałej długości. Najważniejszą cechą algorytmu funkcji skrótu jest to, że proces przypisywania skrótu jest nieodwracalny w praktycznym czasie. Oznacza to, że z otrzymanego skrótu nie jesteśmy w stanie odczytać ciągu znaków, któremu ten skrót został przypisany. Funkcje skrótu mają dużo zastosowań np: weryfikacja sygnatur dużych zbiorów danych, sumy kontrolne, optymalizacja dostępu do struktur danych w programach komputerowych. Najbardziej znanym zastosowaniem takich funkcji jest hashowanie haseł, czyli przetrzymywanie hasła w postaci skrótu przydzielonego przez funkcję skrótu. Jest to o wiele bezpieczniejsze, niż trzymanie haseł w bazach danych w formie jawnej.

Aby funkcja skrótu była uznawana za bezpieczną do zastosowań kryptograficznych powinna ona spełniać kombinację następujących kryteriów, w zależności od zastosowania:

- Odporność na kolizje.
- Odporność na sytuację, w której skrót jest taki sam jak podana wartość do funkcji skrótu.
- Jednokierunkowość - brak możliwości wywnioskowania wiadomości wejściowej na podstawie wartości skrótu.

2 Grøstl - założenia

Grøstl to kryptograficzna funkcja skrótu, która została finalistą konkursu organizowanym przez NIST "SHA-3". Swoją nazwę zawdzięcza austriackiemu daniu, które w Stanach Zjednoczonych jest znane pod nazwą "Hash". Grøstl to iteracyjna funkcja, w której funkcja kompresji zbudowana jest z dwóch stałych, dużych permutacji. Funkcja jest całkiem odmienna od innych z rodziny SHA i w działaniu bardziej przypomina algorytm AES. Dwie permutacje zostały zaprojektowane przy pomocy strategii szerokiej ścieżki, która pozwala na stwierdzenie, że funkcja jest bezpieczna przed różnymi atakami kryptoanalitycznymi.

Grøstl jest bitowo zorientowaną siecią permutacji podstawieniowych, która zapożycza niektóre rozwiązania z AES. Na przykład S-boxy są identyczne do tych, które używa AES. W podobny sposób zostały także zaprojektowane warstwy dyfuzyjne. Grøstl jest tak zwaną konstrukcją szerokopasmową, w której rozmiar stanu wewnętrznego jest znacznie większy niż rozmiar wyjścia. Skutkuje to tym, że wszystkie znane, ogólne ataki na funkcję skrótu są znacznie trudniejsze.

3 Grøstl - w jaki sposób działa

Grøstl iteruje funkcje kompresji w następujący sposób. Najpierw do wiadomości M dodawany jest padding, a następnie dzielona jest na l -bitowe bloki wiadomości m_1, \dots, m_t , gdzie każdy blok jest procesowany sekwencyjnie. Początkowa l -bitowa wartość $h_0 = iv$ jest definiowana i następnie bloki są przetwarzane w następujący sposób:

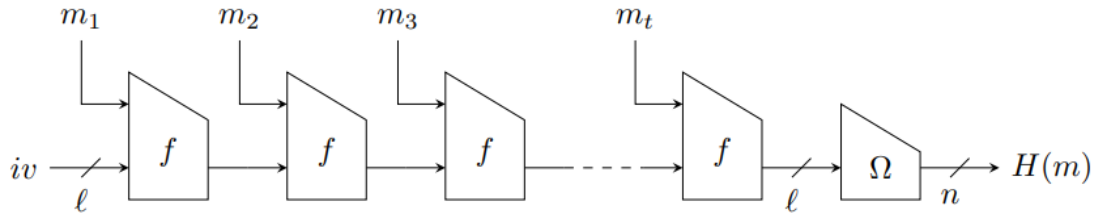
$$h_i \leftarrow f(h_{i-1}, m_i) \text{ dla } i = 1, \dots, t$$

Stąd funkcja f mapuje dwa wejścia o długości l każdy do jednego l -bitowego wyjścia. Dla wariatu Grøstl 256 l jest zdefiniowane jako 512 bitów, natomiast dla wariatu 512 l wynosi 1024.

Po tym jak przetwarzanie ostatniego bloku zostanie zakończone, wyjście $H(M)$ jest obliczane w następujący sposób

$$H(M) = \Omega(h_t),$$

gdzie Ω to transformacja wyjściowa. Rozmiar wyjścia Ω ma długość n bitów.

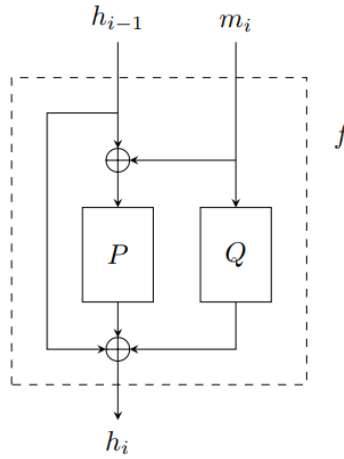


Rysunek 1: Funkcja haszująca Grøstl

3.1 Funkcja kompresji

Funkcja kompresji bazuje na dwóch l -bitowych permutacjach P i Q . Zdefiniowana jest w następujący sposób:

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h$$



Rysunek 2: Funkcja kompresji

3.2 Transformacja wyjściowa

Niech $trunc_n(x)$ będzie operacją, która odrzuca wszystkie bity oprócz n końcowych bitów x . Wyjście Ω wygląda w sposób następujący:

$$\Omega(x) = trunc_n(P(x) \oplus x)$$

3.3 Permutacja P i Q

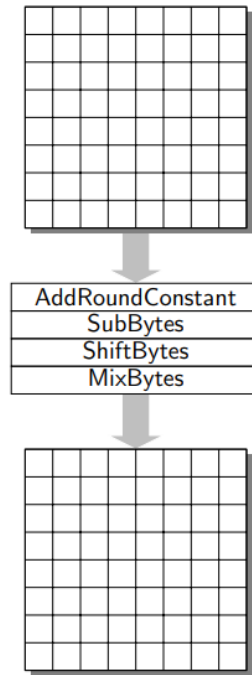
Permutacje zostały tak zaprojektowane, że składają się z określonej liczby rund $R = 10$, które składają się z czterech rund transformacyjnych:

- AddRoundConstant
- SubBytes
- ShiftBytes
- MixBytes

Każda runda R składa się z tych czterech rund transformacyjnych w następującej kolejności:

$$R = MixBytes \circ ShiftBytes \circ SubBytes \circ AddRoundConstant$$

Transformacje operują na macierzy bajtów A , która ma 8 wierszy i 8 kolumn.



Rysunek 3: Jedna runda permutacji P i Q

3.3.1 Mapowanie sekwencji bajtów do macierzy

Mapowanie odbywa się po przez wpisanie do kolumn macierzy kolejnych bajtów sekwencji. Stąd 64-bajtowa sekwencja 00 01 02 ... 3f zostanie zmapowana w następujący sposób:

$$\begin{bmatrix} 00 & 08 & 10 & 18 & 20 & 28 & 30 & 38 \\ 01 & 09 & 11 & 19 & 21 & 29 & 31 & 39 \\ 02 & 0a & 12 & 1a & 22 & 2a & 32 & 3a \\ 03 & 0b & 13 & 1b & 23 & 2b & 33 & 3b \\ 04 & 0c & 14 & 1c & 24 & 2c & 34 & 3c \\ 05 & 0d & 15 & 1d & 25 & 2d & 35 & 3d \\ 06 & 0e & 16 & 1e & 26 & 2e & 36 & 3e \\ 07 & 0f & 17 & 1f & 27 & 2f & 37 & 3f \end{bmatrix}.$$

3.3.2 AddRoundConstant

Transformacja AddRoundConstant dodaje stałe zależne od numeru rundy do macierzy A . Przez dodawanie mamy na myśli operację XOR. Wygląda to w następujący sposób:

$$A \leftarrow A \oplus C[i],$$

gdzie $C[i]$ jest stałą użytą w rundzie i . Permutacje P i Q mają różne stałe:

$$P_{512} : C[i] = \begin{bmatrix} 00 \oplus i & 10 \oplus i & 20 \oplus i & 30 \oplus i & 40 \oplus i & 50 \oplus i & 60 \oplus i & 70 \oplus i \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}$$

Rysunek 4: Macierz stałych dla permutacji P

$$Q_{512} : C[i] = \begin{bmatrix} ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff \oplus i & ef \oplus i & df \oplus i & cf \oplus i & bf \oplus i & af \oplus i & 9f \oplus i & 8f \oplus i \end{bmatrix}$$

Rysunek 5: Macierz stałych dla permutacji Q

Gdzie i to 8-bitowa reprezentacja numeru rundy, a wszystkie pozostałe wartości są zapisane w notacji heksadecymalnej.

3.3.3 SubBytes

Transformacja SubBytes podstawia za każdy bajt w macierzy A inny wzięty z s-boxa S . Więc jeśli $a_{i,j}$ to element z i -tego rzędu i j -tej kolumny macierzy A , to wtedy transformacja wykonuje następującą operację:

$$a_{i,j} \leftarrow S(a_{i,j})$$

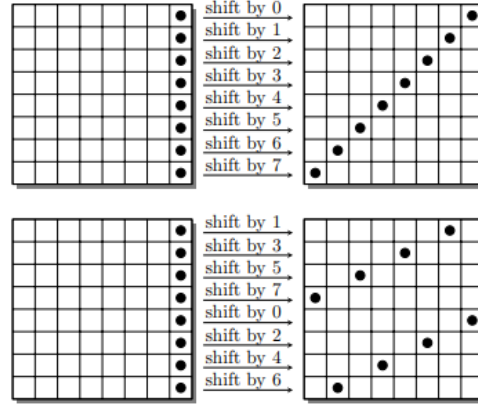
Aby określić jaki element z S-boxa odpowiada naszemu bajtowi należy wykonać operację $a_{i,j} \wedge 0f$ aby określić wiersz elementu zastępczego i $a_{i,j} \wedge f0$ aby określić kolumnę elementu zastępczego, gdzie \wedge jest rozumiane jako operacja AND.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Rysunek 6: S-box użyty w transformacji SubBytes

3.3.4 ShiftBytes

Transformacja ShiftBytes cyklicznie przesuwają bajty w rzędach w lewą stronę o pewną liczbę. Wektor $\sigma = [\sigma_0, \sigma_1, \dots, \sigma_7]$, który określa liczbę przesunięć w danym rzędzie, gdzie σ to liczba przesunięć, a indeks dolny do numer rzędu. Dla P używamy wektora $\sigma = [0, 1, 2, 3, 4, 5, 6, 7]$, a dla Q używamy $\sigma = [1, 3, 5, 7, 0, 2, 4, 6]$.



Rysunek 7: Sposób w jaki ShitBytes przesuwa bajty

3.3.5 MixBytes

Transformacja MixBytes mnoży każdą kolumnę macierzy A przez macierz 8×8 B w ciele \mathbb{F}_{256} . Stąd cała transformacja może być zapisana:

$$A \leftarrow A \times B$$

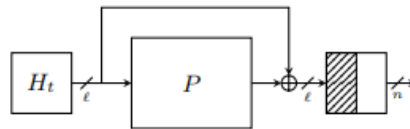
Macierz B wygląda w następujący sposób:

$$B = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}.$$

Rysunek 8: Macierz B

3.4 Funkcja Omega

Celem funkcji omega jest zwrócenie hashu byte array o długości 256 bitów, z racji tego, że funkcja operuje na tablicach 512 bitowych to musi on uciąć połowę bitów. Aby to zrobić wykonuje permutację P na otrzymanym bloku, a następnie wykonuje operację XOR na wyniku permutacji oraz bloku danych, który został do niej podany. Z wyniku ostatniej operacji zwraca ostatnie 256 bitów i jest to nasz hash. Wizualizuje to poniższa grafika :



Rysunek 9: Funkcja Omega B

3.5 IV - wektor inicjujący

Wektor inicjujący ma długość 64 bajtów i jego wartość wynosi: 00 .. 00 01 00

3.6 Padding

Funkcja paddingowa przyjmuje ciąg znaków x od długości N bitów i zwraca $x^* = pad(x)$, który ma długość, która jest wielokrotnością l .

Funkcja wykonuje następujące operacje. Na początku dołącza do końca ciągu '1'. Następnie dodaje $w = -N - 65 \bmod(l)$ zer i na końcu dodaje 64-bitową reprezentację liczby $\frac{N+w+65}{l}$. Ta liczba także wskazuje na ile bloków będzie podzielony ciąg x

4 Grøstl - implementacja

Do naszej implementacji użyliśmy języka programowania JAVA. Było to podyktowane tym, że pracujemy z tym językiem już drugi semestr i czuliśmy się w nim komfortowo.

4.1 Padding

Metoda odpowiadająca za padding przyjmuje obiekt typu String z wiadomością oraz rozmiar bloku. Zwraca natomiast tablicę bajtów. Do zaimplementowania zasad paddingu użyliśmy klasy StringBuilder, ponieważ jest ona szybka i wygodna w użytkowaniu

```
public static byte[] padMessage(String message, int blockSize) {
    StringBuilder sb = new StringBuilder();
    byte[] messageBytes = message.getBytes(StandardCharsets.US_ASCII);
    int messageLength = messageBytes.length * 8;
    // Convert message length to bits

    for (byte b : messageBytes) {
        for (int i = 7; i >= 0; i--) {
            int bit = (b >> i) & 0x01;
            sb.append(bit);
        }
    }

    // Calculate the number of bits needed for padding
    int paddingBits = (-messageLength - 65) % blockSize;
    if (paddingBits < 0) {
        paddingBits += blockSize;
    }

    // Calculate the number of blocks in the padded message
    int numBlocks = (messageLength + paddingBits + 65) / blockSize;

    // Calculate the length of the padded message in bytes
    int paddedLength = numBlocks * blockSize / 8;

    // Append '1' bit to the message
    sb.append("1");

    // Append '0' bits
    for (int i = 0; i < paddingBits; i++) {
        sb.append("0");
    }

    // Append length representation
    long lengthValue = numBlocks;
    String binaryRepresentation = Long.toBinaryString(lengthValue);
    int numZerosToAdd = 64 - binaryRepresentation.length();
    StringBuilder sb2 = new StringBuilder();
```

```

    for (int i = 0; i < numZerosToAdd; i++) {
        sb2.append("0"); // Dodawanie zer
    }
    sb2.append(binaryRepresentation);
    String bitString = sb2.toString();

    sb.append(bitString);

    String paddedMessageString = sb.toString();
    byte[] paddedMessage = new byte[paddedMessageString.length() / 8];

    for (int i = 0; i < paddedMessageString.length(); i += 8) {
        String byteString = paddedMessageString.substring(i, i + 8);
        byte byteValue = (byte) Integer.parseInt(byteString, 2);
        paddedMessage[i / 8] = byteValue;
    }
    return paddedMessage;
}

```

4.2 Podzielenie wiadomości na bloki

Metoda przyjmuje tablice bajtów, i rozmiar bloku. Zwraca natomiast ArrayListe tablic bajtów.

```

public static ArrayList<byte[]> divideMessage(byte[] byteArray, int subarraySize) {
    ArrayList<byte[]> dividedArray = new ArrayList<>();

    int totalSubarrays = ((byteArray.length*8)/ subarraySize) ;

    for (int i = 0; i < totalSubarrays; i++) {
        int startIndex = i * 64;
        int endIndex = (i+1) * 64 ;

        byte[] subarray = new byte[64];

        subarray=Arrays.copyOfRange(byteArray, startIndex, endIndex);

        dividedArray.add(subarray);
    }

    return dividedArray;
}

```

4.3 Permutacja P i Q

Obie metody przyjmują dany blok wiadomości zapisany w tablicy bajtów i rozmiar bloku. Następnie wywołuje kolejne metody pomocnicze permutacji, a na koniec zwraca tablicę bajtów

```

public static byte[] PermutationP(byte[] message, int blocksize){

    byte [][] matrixA=mixBytes(shiftLeftP(subBytes(
        roundConstantsP(mapMessageToMatrixA(message,512)))) ,8);

    return mapMatrixToByteArray(matrixA, blocksize);
}

```



```

public static byte[] PermutationQ(byte[] message, int blocksize){

    byte [][] matrixA=mixBytes(shiftLeftQ(subBytes(
roundConstantsQ(mapMessageToMatrixA(message,512)))) ,8);

    return mapMatrixToByteArray(matrixA , blocksize );
}

```

4.3.1 Mapowanie macierzy

Metoda przyjmuje tablicę bajtów i rozmiar bloku, a zwraca dwuwymiarową tablicę bajtów.

```

public static byte[][] mapMessageToMatrixA(byte[] message, int blockSize) {
    // Initialize matrix A
    byte[][] matrixA = new byte[8][8];

    int messageIndex = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            matrixA[j][i] = message[messageIndex];
            messageIndex++;
        }
    }

    return matrixA;
}

```

Idąc tą samą logiką należy uwzględnić mapowanie macierzy do byte array :

```

public static byte[] mapMatrixToByteArray(byte[][] matrix, int blocksize){
    byte[] message = new byte[64];
    int messageIndex = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            message[messageIndex]=matrix[j][i] ;
            messageIndex++;
        }
    }
    return message;
}

```

4.3.2 addRoundConstant P i Q

Obie metody są odpowiedzialne za XOR'owanie macierzy. Przyjmują dwuwymiarową tablicę bajtów i potem ją zwracają.

```

public static byte[][] roundConstantsP(byte[][] matrixA){
    int[][] matrixP = {
        {0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0}
    }
}

```

```

    };
    for (int j = 0; j < matrixA[0].length; j++) {
        matrixA[0][j] ^= CycleP ^ matrixP[0][j];
    }
    CycleP++;
    return matrixA;
}

public static byte[][] roundConstantsQ(byte[][] matrixA){
    int[][] matrixQ = {
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xEF, 0xDF, 0xCF, 0xBF, 0xAF, 0x9F, 0x8F}
    };
    for (int i = 0; i < matrixA[0].length; i++) {
        for (int j = 0; j < matrixQ[0].length; j++) {
            if(i==7) {
                matrixA[7][j] ^= CycleQ ^ (byte)matrixQ[i][j];
            }
            else matrixA[i][j] ^= (byte)matrixQ[i][j];
        }
    }
    CycleQ++;
    return matrixA;
}

```

4.3.3 SubBytes

Metoda pobierająca dwuwymiarową tablicę bajtów i też taką tablicę zwracająca. Podstawia do macierzy odpowiednie wartości z S-Boxa.

```

public static byte[][] subBytes(byte[][] matrix){
    for(int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            int row = (matrix[i][j] & 0x0f);
            int column = (matrix[i][j] & 0xf0)/16;
            matrix[i][j]=sBox[column][row];
        }
    }
    return matrix;
}

```

4.3.4 ShiftBytes P i Q

Metody przesuwają w odpowiedni sposób rzędy macierzy. Metody przyjmują dwuwymiarową tablicę bajtów i ją zwracają.

```

public static byte[][] shiftLeftP(byte[][] matrix){
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < i; j++) {
            byte firstElement = matrix[i][0];

```

```

        System.arraycopy(matrix[i], 1, matrix[i], 0, matrix[i].length - 1);
        matrix[i][matrix.length - 1] = firstElement;
    }
}
return matrix;
}

public static byte[][] shiftLeftQ(byte[][] matrix){
    int[] permutationVector = {1,3,5,7,0,2,4,6};
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < permutationVector[i]; j++) {
            byte firstElement = matrix[i][0];
            System.arraycopy(matrix[i], 1, matrix[i], 0, matrix[i].length - 1);
            matrix[i][matrix.length - 1] = firstElement;
        }
    }
    return matrix;
}
}

```

4.3.5 MixBytes

Przy tworzeniu tej metody inspirowaliśmy się implementacją Grøstla w języku C. Metoda przyjmuje dwuwymiarową tablicę bajtów oraz liczbę kolumn tej tablicy. Następnie dokonuje mnożenia kolumn używając multiplikatorów. Na koniec zwraca dwuwymiarową tablicę bajtów.

```

public static byte[][] mixBytes( byte[][] msg, int columns )
//mixBytes zaczerpnięte z języka C i przepisane na Java
{
    byte temp[] = new byte[8];
    for( int i = 0; i < columns; i++ )
    {
        for (int j = 0; j < 8; j++)
        {
            temp[j] = ( byte )(mul2(msg[(j + 0) % 8][i])
            ^ mul2(msg[(j + 1) % 8][i]) ^
            mul3(msg[(j + 2) % 8][i]) ^ mul4(msg[(j + 3) % 8][i]) ^
            mul5(msg[(j + 4) % 8][i]) ^ mul3(msg[(j + 5) % 8][i]) ^
            mul5(msg[(j + 6) % 8][i]) ^ mul7(msg[(j + 7) % 8][i]));
        }
        for( int j = 0; j < 8; j++ ) {
            msg[j][i] = temp[j];
        }
    }
    return msg;
}

public static byte mul1( byte b ) { return b ;}
public static byte mul2( byte b )
{ return ( byte )((0 != (b>>>7))?((b)<<1)^0x1b:((b)<<1)); }
public static byte mul3( byte b ) { return ( byte )(mul2(b) ^ mul1(b)); }
public static byte mul4( byte b ) { return ( byte )(mul2( mul2( b ))); }
public static byte mul5( byte b ) { return ( byte )(mul4(b) ^ mul1(b)); }
public static byte mul6( byte b ) { return ( byte )(mul4(b) ^ mul2(b)); }
public static byte mul7( byte b ) { return ( byte )(mul4(b) ^ mul2(b) ^ mul1(b)); }
}

```

4.4 Wywołanie funkcji kompresji

Za pierwszym razem wywoływana jest funkcja z wektorem inicjującym, a następnie już wywoływana jest z poprzednim blokiem.

```
public static void functionH(ArrayList<byte[]> message, int blocksize){
    if (init==0){//
        Hi =functionF (message.get(0),initialVector ,blocksize );
        init++;
    }
    for (int i = 1; i < message.size(); i++) {
        CycleP = 0;
        CycleQ = 0;
        byte[] hiFunction = Hi;
        Hi = functionF(message.get(i), hiFunction , blocksize );
    }
}
```

4.5 Funkcja kompresji

```
public static byte[] functionF (byte[] message,byte[] iv, int blocksize){
    byte[] hi = new byte[64];

    for (int i = 0; i < message.length; i++) {
        hi[i] = (byte) (message[i] ^ iv[i]);
    }
    for(int i = 0; i<10; i++) {
        message = PermutationQ(message, blocksize);
        hi = PermutationP(hi, blocksize);
    }
    for (int i = 0; i < message.length; i++) {
        hi[i] = (byte) ((message[i] ^ hi[i]) ^ iv[i]);
    }

    return hi;
}
```

4.6 Funkcja Omega

Warto dodać, że na początku należy ustawić zmienną statyczną, odpowiadającą za wykonanie metody Round-ConstantsP na 0 aby było poprawne działanie. Na początku wykonuje się permutacja p i zapisywany jej jej wynik do nowej tablicy. Następnie xorowany jest wynik permutacji z blokiem, który pobiera funkcja. Na sam koniec zwraca drugą połowę tablicy.

```
public static byte[] functionOmega (byte[] message){
    CycleP=0;
    byte[] outputofP = message;
    for (int i = 0; i<10; i++) {
        outputofP = PermutationP(outputofP,512);
    }
    for (int i = 0; i < message.length; i++) {
        outputofP[i] = (byte) (message[i] ^ outputofP[i]);
    }

    return Arrays.copyOfRange(outputofP, outputofP.length/2, outputofP.length);
}
```

4.7 Forma Hexadecymalna

Wszystkie operacje wykonujemy na byte, jednak sam wynik hashu należy przedstawić w postaci hexadecymalnej, dlatego też potrzebowaliśmy funkcji na zamianę formy

```
public static String byteToHex(byte num) {
    char[] hexDigits = new char[2];
    hexDigits[0] = Character.forDigit((num >> 4) & 0xF, 16);
    hexDigits[1] = Character.forDigit((num & 0xF), 16);
    return new String(hexDigits);
}
public static String encodeHexString(byte[] byteArray) {
    StringBuffer hexStringBuffer = new StringBuffer();
    for (int i = 0; i < byteArray.length; i++) {
        hexStringBuffer.append(byteToHex(byteArray[i]));
    }
    return hexStringBuffer.toString();
}
```

4.8 wywoływanie funkcji haszującej

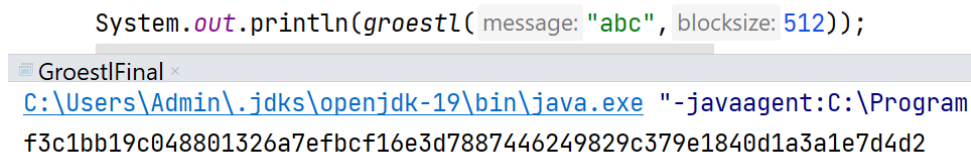
```
public static String groestl(String message, int blocksize){
    functionH(divideMessage(padMessage(message, blocksize), blocksize), blocksize);
    return encodeHexString(functionOmega(Hi));
}
```

5 Testy

Twórcy na stronie poświęconej tej funkcji haszującej udostępnili paczkę z różnymi testami wektrowymi. Były one bardzo przydadne w szczególności przy debugowaniu kodu, ponieważ niektóre skróty miały rozpisane jak wyglądają macierze krok po kroku w trakcie różnych transformacji.

Dla wersji Gröstl-256, którą implementowaliśmy były dostępne z wspomnianej wcześniej paczki były dostępne dwa testy. Dla wiadomości jedno blokowej oraz dla wiadomości dwu blokowej.

- Krótszą wiadomością była wiadomość "abc", której skrót wygląda tak:
"f3c1bb19c048801326a7efbcf16e3d7887446249829c379e1840d1a3a1e7d4d2"



```
System.out.println(groestl(message: "abc", blocksize: 512));
GroestlFinal
C:\Users\Admin\.jdk\openjdk-19\bin\java.exe "-javaagent:C:\Program
f3c1bb19c048801326a7efbcf16e3d7887446249829c379e1840d1a3a1e7d4d2"
```

Rysunek 10: Wynik programu dla wiadomości "abc"

- Druga wiadomość miała treść: "abcdcbdecdefdefghfghighijhijkijklklmklmnlmnopnopq", natomiast jej skrót:
"22c23b160e561f80924d44f2cc5974cd5a1d36f69324211861e63b9b6cb7974c"

```
GroestlFinal -
C:\Users\Admin\jdk\openjdk-19\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.2\lib\idea
22c23b160e561f80924d44f2cc5974cd5a1d36f69324211861e63b9b6cb7974c
```

Rysunek 11: Wynik programu dla wiadomości "abc"

6 Tryby użytkowania dla Grøstl

Grøstl może być używany w „trybie losowym”, np. jako kod uwierzytelniający wiadomość. Taki tryb obejmuje dodatkowe wejście, którym może być klucz, sól, wartość losowania itp. Twórcy uważają, że Grøstl jest bezpieczny, gdy jest używany w istniejących trybach randomizacji wykorzystujących skrót funkcje, ale proponują również dedykowany tryb MAC dla Grøstl.

7 Bezpieczeństwo Groestla

Attack type	Claimed complexity	Best known attack
Collision	$2^{n/2}$	$2^{n/2}$
d -collision	$\lg(d) \cdot 2^{n/2}$	$(d!)^{1/d} \cdot 2^{n(d-1)/d}$
Preimage	2^n	2^n
Second preimage	2^{n-k}	2^n

Rysunek 12: Bezpieczeństwo Groestla

Dla groestl-256 $n = 256$

2^k - liczba bloków pierwszego PreImage'a dla drugiego preImage'a

Atak typu "collision" odnosi się do sytuacji, w której dwa różne wejścia do danej funkcji lub systemu generują ten sam wynik, znany jako kolizja. W kontekście funkcji haszującej, atak typu "collision" oznacza znalezienie dwóch różnych danych wejściowych, które mają tę samą wartość skrótu (hash).

Atak PreImage (czasami nazywany atakiem "first preimage") jest rodzajem ataku kryptograficznego, który polega na odwróceniu funkcji haszującej. Celem ataku PreImage jest znalezienie takiego wejścia do funkcji haszującej, które generuje określoną wartość skrótu (hashu). Innymi słowy, atakujący próbuje odwrócić funkcję haszującą, aby znaleźć odpowiadające wejście dla danego wyjścia (wartości skrótu).

W przypadku ataku Second PreImage, atakujący posiada już pewne dane wejściowe (oryginalne dane), dla których znane jest odpowiadające im wyjście (skrót). Celem ataku jest znalezienie innego wejścia, które generuje ten sam skrót, ale jest różne od pierwotnego wejścia. Innymi słowy, atakujący próbuje znaleźć drugie wejście, które ma taką samą wartość skrótu jak już istniejące wejście.

8 Wnioski

Praca przy implementacji była wymagająca, jednak sprawiała dużo satysfakcji. Dokumentacja była zrobiona bardzo dobrze, było trzeba jednak przeczytać ją bardzo dokładnie, gdyż niektóre rzeczy były poruszane jednokrotnie i pominięcie małego szczegółu było kosztowne. W implementacjach były podane sposoby na przyspieszenie funkcji, jednak dotyczyły one języka C. Cały kod, który napisaliśmy opierał się na grafikach umieszczonych w dokumentacji. Wyjątkiem tu jest metoda `mixbytes`, która wykonuje mnożenie macierzy w Ciele Z256. Zaczerpniecie logiki wykorzystanej w języku C w celu implementacji multiplekserów do wykonania tych operacji było bardzo pomocne.

9 Bibliografia

[1]Oficjalna dokumentacja Grøstl'a, 13.06.2023