Duży projekt
Funkcja skrótu JH
-
BDAN 2023L

Stanisław Ciszkiewicz, Jakub Kusznier

Politechnika Warszawska, Cyberbezpieczeństwo

12 kwietnia 2024

Spis treści

1. Cel zadania projektowego

Celem dużego projektu z przedmiotu BDAN było zaimplementowanie algorytmu kryptograficznego w wybranym przez zespół języku programowania (innym niż C). Każdy zespół mógł wybrać kategorię, z której chciałby otrzymać zadanie. Nasz zespół wybrał kategorię funkcji skrótu i zdecydował się na implementację w języku Java. Po poinformowaniu prowadzącego o preferencji zespołu otrzymaliśmy wiadomość zwrotną z propozycją zaimplementoania funkcji skrótu JH. Po przeanalizowaniu specyfiki samego algorytmu zdecydowaliśmy się na przyjęcie zadania i przystąpiliśmy do jego realizacji. Praca nad tym projektem miała na celu przybliżenie tematyki algorytmów kryptograficznych oraz zastosowania tradycyjnych języków programowania w cyberbezpieczeństwie.

2. Wstęp

JH brała udział w konkursie na nową funkcję skrótu w konkursie NIST. Funkcja trafiła do grona finalistów konkursu, jednak nie wygrała, przez co jest mniej publikacji na jej temat niż tych związanych z zwycięzcą konkursu. Mała ilość innych źródeł na temat JH sprawiła, że do wykonania zadania wykorzystalismy tylko przesłaną dokumentację. Samo JH jest algorytmem blokowym, w którym kolejne kroki zależą od poprzednich, a to wszystko wykonuje sie w przeciagu 42 rund potrzebnych do wygenerowania skrótu. Funkcja JH ma 4 implementacje pozwalające generować skróty różnych długości; JH-224, JH-256, JH-384, JH-512, kolejno oznaczających wygenerowany skrót długości 224, 256, 384, 512 bitów.

3. Opis algorytmu

[1] W poniższym rozdziale opiszemy kolejne kroki działania algorytmu funkcji skrótu JH. Ponadto do każdego omówionego fragmentu załączymy kod w języku java, implementujący omówioną wcześniej funkcjonalność.

3.1. Przygotowanie

Przed przystąpieniem do obliczania funkcji skrótu musimy przeprowadzić kilka kroków przygotowywujących naszą wiadomość do dalszego procesu.

3.1.1. Zamiana wiadomości na ciąg binarny

Po wpisaniu wiadomości M do zaszyfrowania przez użytkownika, musimy ją zamienić na ciąg bitów (1 i 0) - jest to kluczowe, gdyż logika algorytmu opiera się na liczbach binarnych. Na potrzebę implementacji w javie ciąg bitów zapisaliśmy w zmiennej typu String. Ważne w tym procesie są wszystkie znaki, także 0 na początku wiadomości (w tym celu sprawdzaliśmy długość każdego bitu i dodawaliśmy 0, aż długość paczki osiągnie 8 - java pomija w tym procesie początkowe 0).

```
Scanner scanner = new Scanner(System.in);
String text = scanner.next(); //getting the message from the user
byte[] btext = text.getBytes(); //chaniging text to bytes
String inBinaryText = "";
String result = "";
for (byte b : btext){
    result = Integer.toBinaryString(b);
    while (result.length() % 8 != 0){
        result = "0" + result;
    }
    inBinaryText += result; //creating binary String of our message
}
```

3.1.2. Padding

Kolejnym krokiem, który należy wykonać jest dodanie paddingu do wiadomości. W algorytmie JH Padding konstruuje się w następujący sposób:

1. Sprawdzamy długość pierwotnego ciągu bitów i oznaczamy ją zmienną l.
2. Na końcu ciągu dodajemy bit o wartości 1.
3. Dodajemy $896 - 1 - lmod512$ zer na końcu wiadomości
4. Na ostatnich 128 bitach zapisujemy binarną reprezentację liczby l.
   Tym samym otrzymujemy wiadomość równą wielokrotności liczby 512.

```
int l = inBinaryText.length();
inBinaryText += "1";
//adding padding
//counting how many "0" is needed to add
int amount_0 = 896 - 1 - (l)%512;

StringBuilder sb = new StringBuilder();

sb.append(inBinaryText);
String zeros = "0".repeat(amount_0);
sb.append(zeros);
String l_in_binary = Integer.toBinaryString(l);
zeros = "0".repeat(128-l_in_binary.length());
sb.append(zeros);
sb.append(l_in_binary);
```

3.1.3. Podział wiadomości na bloki

Nastepnie musimy podzielić wiadomość na n 512 bitowych części. Każdą paczkę bitów zapisujemy w kolejnych komórkach tablicy, zaczynając od komórki o indeksie 1. Wynika to z faktu, iż w komórce o indeksie 0 znajdzie się ciąg 512 zer konieczny do wyliczenia wektorów inicjalizujących (omówione zostaną w kolejnych rozdziałach).

```
String[] savedInBlocks = new String[1 + sb.length()/512];
        savedInBlocks[0] = "0".repeat(512);
        for (int i = 1; i <= sb.length()/512; i++) {
            savedInBlocks[i] = sb.substring((i - 1)*512, (i)*512);
        }
```

3.1.4. Funkcja $\mathbf{hH_1}$

Poza modyfikacją samej wiadomości musieliśmy zaimplementować funkcję, które przydzieli wartość początkową H^{-1} w zależności od trybu funkcji skrótu wybranego przez użytkownika. Zgodnie z informacjami zawartymi we wstępie nasz algorytm może wygenerować skróty: 224, 256, 384 i 512 bitowe. Funkcja ta przypisuje zmiennej h_1 zapis binarny odpowiedniej liczby (tj. 224, 256, 384, 512) na pierwszych 16 bitach, a następnie uzupełnia blok 1008 zerami. Tym samym otrzymujemy początkową wartość H^{-1}, która zostanie użyta podczas pierwszego obrotu funkcji F.

```
public static String hH_1(int number){
        String h_1 = "";
        switch (number){
            case 224:
                h_1 = h_1 + "0000000011100000" + "0".repeat(1008);
                break;
            case 256:
                h_1 = h_1 + "0000000100000000" + "0".repeat(1008);
                break;
            case 384:
                h_1 = h_1 + "0000000110000000" + "0".repeat(1008);
                break;
            case 512:
                h_1 = h_1 + "0000001000000000" + "0".repeat(1008);
                break;
```

```
            default:
                System.exit(0);
        }
        return h_1;
    }
```

3.2. Funckja F

Po przygotowaniu wiadomości możemy przejść do procesu obliczania skrótu. Funkcja F jako argumenty przyjmuje wartość H^{-1} obliczoną przez funkcję hH_1 (patrz poprzedni rozdział), tablicę podzielonej na bloki wiadomości oraz jej długość. W funkcji f znajduje się główna pętla for, która wykonywana jest n razy (n - długośc tablicy tableOfDividedA). Funckja F jest "mózgiem algorytmu", w której wnętrzu przeprowadzane są wszystkie obliczenia mające na celu wyliczenie wartości skrótu.

Poniżej prezentujemy równanie, które obrazuje działanie funkcji F:

$$H^{(i)} = F_d(H^{(i-1)}, M^{(i)})$$

Zgodnie z tym równaniem, z każdym przejściem pętli for znajdującej się wewnątrz funkcji F przystępujemy do obliczenia nowego H^i.

W skład funkcji F wchodzą 3 główne kroki:

1. $A^j = H^{(i-1),j} \oplus M^{(i),j}$ dla $0 \leqslant$ j $\leqslant$ 511;
   $A^j = H^{(i-1),j}$ dla 512 $\leqslant$ j $\leqslant$ 1023;
2. B = B_8(A);
3. $H^{(i),j} = B^j$ dla 0 $\leqslant$ j $\leqslant$ 511;
   $H^{(i),j} = B^j \oplus H^{(i),j-512}$ dla 512 $\leqslant$ j $\leqslant$ 1023;

3.2.1. Etap 1

W pierwszym kroku zmiennej A przypisujemy odpowiednią kombinację bitów zgodną z opisem powyżej. W naszym programie zmienną A zapisaliśmy małą literą - wynika to ze specyfiki nazewnictwa występującej w javie.

```
public static String f_F(String h_1, String[] savedInBlocks, int n){
        String h = h_1;
        String a = "";
        String b = "";
        for (int i = 0; i < n; i++) {

            //XORing; first stage of f_F
            for (int j = 0; j < 512; j++) {
                if (savedInBlocks[i].charAt(j) == h.charAt(j)) {
                    a = a + "0";
                }
                else {
                    a = a + "1";
                }
            }
            a = a + h.substring(512, 1023);
```

3.2.2. Etap 2

W tym etapie funkcji F następuje wywołanie funkcji E, której argumentem staje się obliczona w poprzednim etapie zmienna a. Efekt pracy tej metody zapisujemy pod zmienną B (w naszym programie b). Dokładne działanie metody E zostanie omówione w kolejnym rozdziale.

```
b = e_E(a);
```

3.2.3. Etap 3

Ostatnim krokiem jest obliczenie nowego H na podstawie wartości M^i i B - zgodnie z powyższym wzorem. Gdy wartość i osiągnie długość tablicy podzielonych fragmentów wiadomości (n), następuje wyjście z metody - zwracana jest właśnie wartość H^n. W innym przypadku funkcja wraca do kroku pierwszego i wykonuje te same operacje obliczając kolejne wartości H.

```
            h = "";
            for (int k = 0; k < 512; k++) {
                h += b.charAt(i);
            }
            for (int j = 512; j < 1024; j++) {
                if (b.charAt(j) == savedInBlocks[i].charAt(j-512)) {
                    h = h + "0";
                }
                else {
                    h = h + "1";
                }
            }
            a = "";
            b = "";
        }
        return h;
    }
```

3.3. Funkcja E

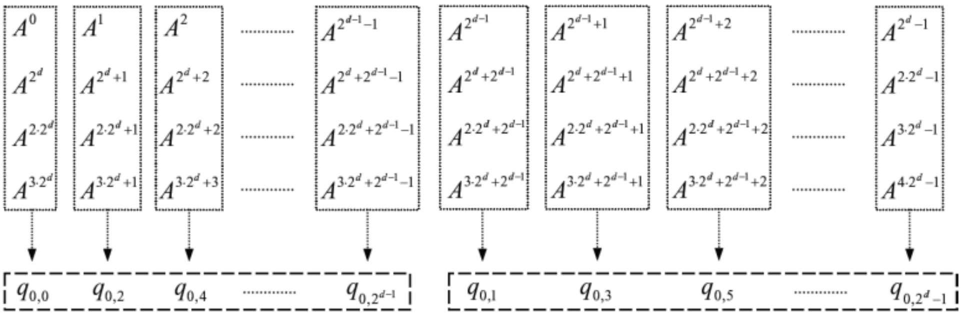
W tym podrozdziale omówione zostanie działanie metody E, wywoływanej w kroku 2 metody F. Funkcja E przyjmuje na wejściu zmiennąż typu String a, która zawiera 1024 bity (opracowane w kroku 1 funkcji F). Metoda E składa sie z trzech etapów:
1. Dzielenie 1024 bitowego Stringa na 256 paczek 4-bitowych (według podanego schematu).
2. Wywołanie funkcji rundowej R, która wykonywana jest 42 razy.
3. Degrupowanie paczek 4-bitowych.

Poniżej omówione zostaną kolejne etapy tej metody.

```
    public static String e_E(String a){
        String dividedA[] = new String[256];
        dividedA = grouping(a);
        dividedA=R(dividedA,roundConstant);
        char[] q42;
        String b = "";
        q42 = deGrouping(dividedA);
        for (int i = 0; i < q42.length; i++) {
            b += q42[i];
        }
        return b;
    }
```

3.3.1. Grouping

Pierwszym etapem metody E jest podział znaków zmiennej a na grupy (każda po 4 znaki). Grupy były tworzone według schematu przedstawionego w dokumentacji wstawiane były do tablicy Stringów tableOfDividedA (256 komórek).



Rys. 1. Grupowanie w funkcji E[1]

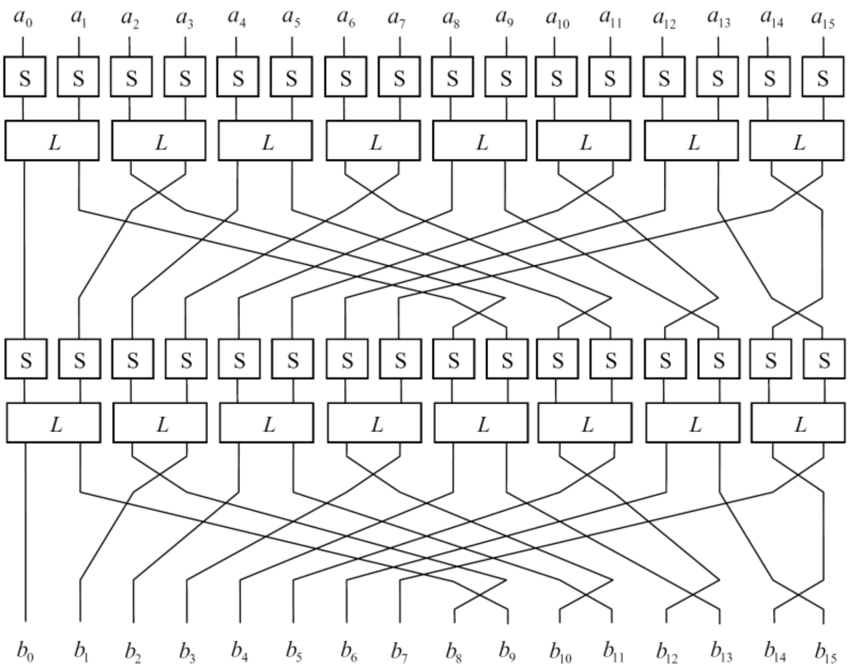
Zgodnie ze schematem dokonaliśmy podziału bitów, wprowadzając zmienną pomocniczą temp, której wartość na końcu wstawiliśmy do tablicy tableOfDividedA (którą ostatecznie zwracamy z metody).

```
public static String[] grouping(String a){
    String[] tableOFdividedA = new String[256];
    String[] temp = new String[256];
    String group="";
    for (int i = 0; i < 128; i++)
    {
        group="";
        group += a.charAt(i);
        group += a.charAt(256 + i);
        group += a.charAt(512 + i);
        group += a.charAt(768 + i);
        temp[2*i]=group;
        group="";
        group += a.charAt(i+128);
        group += a.charAt(384 + i);
        group += a.charAt(640 + i);
        group += a.charAt(896 + i);
        temp[2 * i + 1] = group;
    }
    tableOFdividedA = temp;
    return tableOFdividedA;
}
```

3.3.2. Funkcja rundowa R

Funkcja rundowa jest mózgiem działania całego algorytmu. To w niej zachodzą wszystkie najważniejsze operacje na bitach, które sprawiają, że złamanie funkcji skrótu staje się praktycznie niemożliwe. Każda runda metody R posiada dedykowany klucz rundowy z tablicy round_constant (zapisany w postaci hex, metodą countingRoundedKey zmieniamy obecny klucz na ciąg 256 bitów). Po wybraniu i obliczeniu klucza przystępujemy do trzech najważniejszych etapów:

- sboxes - zamiana paczek 4-bitowych na inne (według schematu, który opiszemy w dalszej częsci dokumentu)
- linearTransformation - transformacja, która parami przekształca paczki bitów
- permutacje - permutacje paczek 4-bitowych. Opisana w algorytmie permutacja jest złożeniem trzech permutacji: π_d, P'_d, ϕ_d.



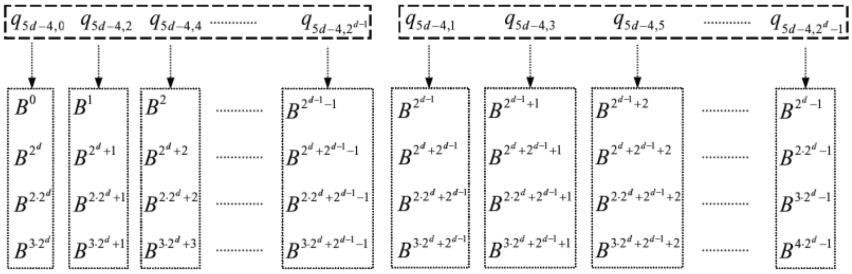
Rys. 2. Zbliżenie na dwie rundy metody R[1]

Dogłębny opis metod wywoływanych we wnętrzu metody R zostanie przedstawiony w dalszej części dokumentu.

```
public static String[] R(String [] dividedA, int [][] roundConstant){
    String roundKeyinBinary = "";
    for (int i = 0; i < 42; i++) {
        roundKeyinBinary = countingRoundedkey(roundConstant, i);
        dividedA = sBOXES(dividedA, roundKeyinBinary);
        dividedA = linearTransformation(dividedA);
        dividedA = pi_od_D(dividedA);
        dividedA = p_prim_d(dividedA);
        dividedA = fi_d(dividedA);
    }
    return dividedA;
}
```

3.3.3. DeGrouping

Ostatnim etapem metody E jest degroupowanie, tzn. ponowne przetasowanie bitów i ustawienie je kolejności zgodnej z dokumentacją. Wykonywane jest ono w funkcji degrouping, która przyjmuje na wejściu 256 - bitową tablicę 4-bitowych paczek i dokonuje zamiany). Połączenie paczek w jednego długiego 1024 bitowego Stringa następuje już w ciele funkcji E.



Rys. 3. Degroupowanie w funkcji E[1]

Według powyższego schematu przeprowadziliśmy degroupowanie, które opisuje poniższa metoda przyjmująca i zwracająca tablicę 256 Stringów (4 bitowych paczek):

```
public static char[] deGrouping(String[] a){
    String[] tableOfDividedA = a;
    char[] temp = new char[1024];

    for (int i = 0; i < 128; i++)
    {

        temp[i] = tableOfDividedA[2 * i].charAt(0);
        temp[i + 256] = tableOfDividedA[2 * i].charAt(1);
        temp[i + 512] = tableOfDividedA[2 * i].charAt(2);
        temp[i + 768] = tableOfDividedA[2 * i].charAt(3);
        temp[i + 128] = tableOfDividedA[2 * i + 1].charAt(0);
        temp[i + 384] = tableOfDividedA[2 * i + 1].charAt(1);
        temp[i + 640] = tableOfDividedA[2 * i + 1].charAt(2);
        temp[i + 896] = tableOfDividedA[2 * i + 1].charAt(3);
    }
    return temp;
}
```

3.4. Funkcja R

W tym rozdziale omówimy kolejne metody znajdujące się we wnętrzu metody R. Metoda R wywoływana jest we wnętrzu metody E i odpowiada za najważniejsze operacje algorytmu. Funkcja R przyjmuje tablicę kluczy rundowych oraz tablicę 256 paczek 4-bitowych, wykonuje 42 rundy i na koniec zwraca tablicę podzielonych A.

```
public static String[] R(String [] dividedA, int [][] roundConstant){
    String roundKeyinBinary = "";
    for (int i = 0; i < 42; i++) {
        roundKeyinBinary = countingRoundedkey(roundConstant, i);
        dividedA = sBOXES(dividedA, roundKeyinBinary);
        dividedA = linearTransformation(dividedA);
        dividedA = pi_od_D(dividedA);
        dividedA = p_prim_d(dividedA);
        dividedA = fi_d(dividedA);


    }
    return dividedA;
}
```

3.4.1. sBoxes

Metoda sBoxes odpowiada za pierwszą zamianę 256 grup 4-bitowych. Do metody tej tafia klucz rundowy (256 bitów) obliczony dla danej rundy oraz wcześniej wspomniana tablica dividedA. Istota działania sboxów (substitution box) jest zamiana określonego bitu na inny według danych zawartych w tablicy. W naszym algorytmie mamy 2 sboxy, których zawartość została przedstawiona w dokumentacji algorytmu.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_0(x)$	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
$S_1(x)$	3	12	6	13	5	7	1	9	15	2	0	4	11	10	14	8

Rys. 4. Tablica sboxów[1]

W ciele metody brane są kolejne paczki (od i=0 do i=255), a następnie sprawdzany jest i-ty bit klucza rundowego. W zależności od tego czy na i-tym bicie znajduje się 0 lub 1 trafiamy do sboxów s0 lub s1. We wnętrzu sboxów zachodzi zamiana bitów według schematu:
— zamieniany jest ciąg bitów znajdujący się w i-tej grupie na liczbę całkowitą (zmienna temp)
— następnie po wejściu do konkretnego sboxa sprawdzamy jaka wartość znajduje się pod indeksem temp w tablicy danego sboxa
— pobieramy wartość spod tego indeksu i zamieniamy zapisanego inta na ciąg bitów (dodając odpowiednią liczbę zer na początku, tak aby w każdej paczce znajdowały się 4 bity).
— obliczoną wartośc zapisuejmy do zmiennej tableOfDividedA, którą zwracamy z tej metody.

```
public static String[] sBOXES(String[] tableOFdividedA, String roundKeyinBinary){
        int[] sBOX_0 = {9, 0, 4, 11, 13, 12, 3, 15, 1, 10, 2, 6, 7, 5, 8, 14};
        int[] sBOX_1 = {3, 12, 6, 13, 5, 7, 1, 9, 15, 2, 0, 4, 11, 10, 14, 8};
        for (int i = 0; i < 256; i++) {
            //changing from binary to decimal
            int temp = Integer.parseInt(tableOFdividedA[i], 2);

            if(roundKeyinBinary.charAt(i) == '0'){
                tableOFdividedA[i] = Integer.toBinaryString(sBOX_0[temp]);
                while (tableOFdividedA[i].length() % 4 != 0){
                    tableOFdividedA[i] = "0" + tableOFdividedA[i];
                }
            }else{
                tableOFdividedA[i] = Integer.toBinaryString(sBOX_1[temp]);
                while (tableOFdividedA[i].length() % 4 != 0){
                    tableOFdividedA[i] = "0" + tableOFdividedA[i];
                }
            }
        }
        return tableOFdividedA;
    }
```

3.4.2. Transformacja liniowa

Po wyjściu z sboxów tablica dividedA trafia do metody linearTransforamtion. W jej wnętrzu zachodzi zamiana grup 4-bitowych poprzez xorowanie wybranych bitów poprzednich wartości. W liniowej transformacji xorowane są odpowiednie bity starych wartości parami (o indeksach 2i i 2i+1). Nowe grupy tworzymy w zmiennych c i d, a na sam koniec nadpisujemy wartość tablicy. Z metody zwracana jest tablica 256 grup 4-bitowych.

$$
\begin{aligned}
D^0 &= B^0 \oplus A^1\,; & D^1 &= B^1 \oplus A^2\,;\\
D^2 &= B^2 \oplus A^3 \oplus A^0\,; & D^3 &= B^3 \oplus A^0\,;\\
C^0 &= A^0 \oplus D^1\,; & C^1 &= A^1 \oplus D^2\,;\\
C^2 &= A^2 \oplus D^3 \oplus D^0\,; & C^3 &= A^3 \oplus D^0\,.
\end{aligned}
$$

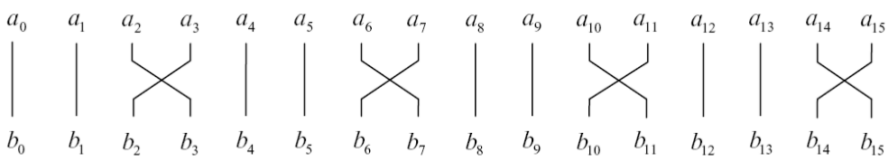
Rys. 5. Opis liniowej transformacji[1]

```
public static String[] linearTransformation(String[] tableOFdividedA){
        String a = "";
        String b = "";
        String c = "";
        String d = "";
        for (int i = 0; i < 128; i++) {
            c = "";
            d = "";
            a = tableOFdividedA[2*i];
            b = tableOFdividedA[2*i+1];
            //this is B(0) XORed with A(1)
            d += XORing(b.charAt(0), a.charAt(1));
            d += XORing(b.charAt(1), a.charAt(2));
            d += XORing((XORing(b.charAt(2), a.charAt(3)).charAt(0)), a.charAt(0));
            d += XORing(b.charAt(3), a.charAt(0));
            c += XORing(a.charAt(0), d.charAt(1));
            c += XORing(a.charAt(1), d.charAt(2));
            c += XORing((XORing(a.charAt(2), d.charAt(3)).charAt(0)), d.charAt(0));
            c += XORing(a.charAt(3), d.charAt(0));
            tableOFdividedA[2*i] = c;
            tableOFdividedA[2*i+1] = d;
        }

        return tableOFdividedA;
    }
```

3.4.3. Permutacja π_d

Bloki po przejściu liniowej transformacji trafiają do bloku permutacji. Tak jak wcześniej zaznaczyliśmy, w naszym algorytmie następuje złożenie trzech różnych permutacji - na potrzeby przejrzystej implementacji wykonywaliśmy każdą z nich po kolei. Pierwszą permutacją jest permutacja π_d. Jej charakterystytka zakłada zamianę komórek o indeksach, których reszta z dzielenia daje resztę 2 lub 3. Pozostałe grupy (indeksy podzielone przez 4 dają resztę 0 i 1) pozostają bez zmian. Funckja przyjmuje i po przetasowaniu zwraca tablicę 256 4-bitowych Stringów.



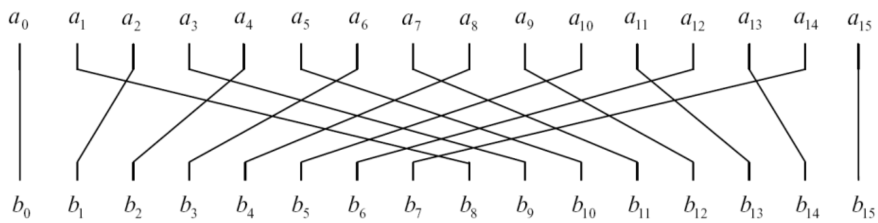
Rys. 6. Permutacja π_d[1]

```java
public static String[] pi_od_D(String[] tableOFdividedA){

        String[] temp = new String[256];
        for (int i = 0; i < 256; i++) {
            if(i%4==0){
                temp[i] = tableOFdividedA[i];
            } else if(i%4==1){
                temp[i] = tableOFdividedA[i];
            } else if(i%4==2){
                temp[i] = tableOFdividedA[i+1];
            } else if (i%4==3) {
                temp[i] = tableOFdividedA[i-1];
            }
        }
        tableOFdividedA = temp;
        return tableOFdividedA;
    }
```

3.4.4. Permutacja P'_d

Kolejna metoda odpowiada za zamianę grup bitów w taki sposób, aby grupy znajdujące się na parzystych indeksach wejściowej tablicy ustawione zostały w porządku rosnącym na początku tablicy(indeksy od 0 do 127), natomiast grupy o indeksach nieparzystych trafiają na druga połowę nowo powstałej tablicy(indeksy od 128 do 255). Po przejściu metody zwracana jest nowa wartość tablicy dividedA.



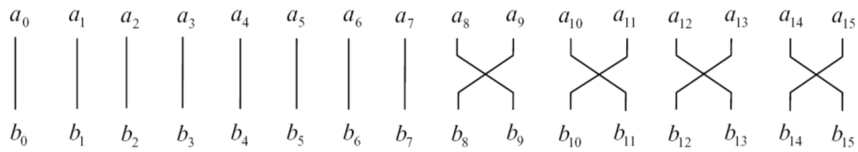
Rys. 7. Permutacja $P'_d d$[1]

```java
    public static String[] p_prim_d(String[] tableOFdividedA){

        String[] temp = new String[256];
        for (int i = 0; i < 128; i += 1)
        {
            temp[i] = tableOFdividedA[2 * i];
            temp[i + 128] = tableOFdividedA[2 * i + 1];
        }
        tableOFdividedA = temp;
        return tableOFdividedA;
    }
```

3.4.5. Permutacja ϕ_d

Ostatnia permutacja odpowiada za zamianę grup bitów jedynie w drugiej połowie tablicy (od indeksu 128 do 255). Tym samym początkowe grupy pozostają bez zmian, natomiast w drugiej połowie parami zamieniają się grupy znajudjące się obok siebie. Metoda zwraca nam tablicę dividedA, a sama funkcja R przechodzi do kolejnego obiegu pętli (aż osiągnie 42 powtórzenia).



Rys. 8. Permutacja ϕ_d

```
public static String[] fi_d(String[] tableOFdividedA){

    String[] temp = new String[256];
    for (int i = 0; i < 128; i++) {
        temp[i] = tableOFdividedA[i];
    }
    for (int i = 128; i < 256; i++) {
        if(i%2==0){
            temp[i] = tableOFdividedA[i+1];
        } else {
            temp[i] = tableOFdividedA[i-1];
        }
    }
    tableOFdividedA = temp;
    return tableOFdividedA;
}
```

3.5. Metoda obliczająca ostateczną wartość skrótu

Po wykonaniu wszystkich obejść funkcji F dostajemy ostateczną wersję 1024 bitowej zmiennej h. W tym moemncie naszym zadaniem jest przygotowanie ostatecznego skrótu według opisu zawartego w dokumentacji. Zgodnie z nim odcinamy określoną liczbę bitów z końca ciągu h adekwatną do wybranej funkcji skrótu (np. dla JH 256 ucinamy 256 ostatnich bitów). W tej samej funkcji uzyskany przez nas sktrót zamieniamy na postać hex. Na tym etapie uzyskujemy szukany skrót (wypisywany jest w metodzie main).

```
public static String cut_cut_cut(String h, int number){
    String bin = h.substring(h.length()-number);
    BigInteger decimal = new BigInteger(bin, 2);
    String hex = decimal.toString(16);
    return hex;
}
```

4. Testy

Przeprowadziliśmy testy dla napisanego kodu. W celu ich wykonanania musieliśmy uruchomić kod napisany przez autora algorytmu w języku C. Dodaliśmy do niego następującą funkcję main:

```c
int main() {
    int hashbitlen = 256; // Długość wynikowego skrótu (256-bitowy w tym przykładzie)
    BitSequence data[] = "Dane wejściowe"; // Dane wejściowe
    DataLength databitlen = strlen( Str: (char*)data) * 8; // Długość danych wejściowych w bitach
    BitSequence hashval[hashbitlen/8]; // Bufor na wynikowy skrót

    HashReturn result = Hash(hashbitlen, data, databitlen, hashval);

    if (result == SUCCESS) {
        printf( format: "Skrót obliczony poprawnie:\n");
        for (int i = 0; i < hashbitlen/8; i++) {
            printf( format: "%02x", hashval[i]); // Wyświetlanie wynikowego skrótu w formacie szesnastkowym
        }
        printf( format: "\n");
    } else {
        printf( format: "Błąd: Nieprawidłowa długość skrótu.\n");
    }
```

Rys. 9. Funkcja main dopisana do kodu w języku C

Przeprowadzilismy testy dla napisanego kodu. Testy jednostkowe wyglądają następująco:

```java
@Test
    public void hash2() {
        String text = "";
        String[] bl = JH.blocks(text);
        int number = 224;
        String h = JH.f_F(CopiedJH.hH_1(number), bl, bl.length);
        String hash = JH.cut_cut_cut(h, number);
        assertEquals("2c99df889b019309051c60fecc2bd285a774940e43175b76b2626630", hash);
    }
```

Wartości skrótów dla poszczególnych długości oraz wyrazów bralismy z wikipedii (jednak znajdowało sie na niej niewiele skrótów - były skróty dla pustych wyrazów oraz 2 zdań) oraz wygenerowaliśmy w udostępnionym nam w kodzie w języku C. Łącznie przeprowadzilismy 16 testów, które potwierdziły poprawność działania napisanego przez nas kodu. Rezultaty testów jednostkowych widać na zrzucie ekranu załączonym poniżej.[2]

Rys. 10. Wynik przeprowadzonych testów w javie

5. Podsumowanie

Efektem projektu było zaimplementowanie algorytmu JH w języku java. Mimo, iż jak mogłoby się wydawać, że wykonanie zadania z dostępem do dokumentacji będzie prostym zadaniem, podczas pracy napotkaliśmy wiele kłopotów. Jednym z nich okazała się interpretacja tego, co w dokumencie zapisał autor (dokument był dość chaotyczny), natomiast drugim i zdecydowanie bardziej pracochłonnym było zdebugowanie całego kodu i znalezienie miejsca, w którym program zawiera błąd - specyfika algorytmu powoduje, iż jeden mały błąd w kodzie powoduje efekt lawinowy. Praca nad tym projektem pokazała nam kolejne połączenie informatyki i języków programowania z cyberbezpieczeństwem i przybliżyła nam tematykę algorytmów kryptograficznych. Co więcej, mimo słabszych momentów podczas pracy udało nam się zaimplementować w pełni działający algorytm, który zwraca poprawne wartości skrótu dla kolejnych wektorów testowych.

[1] *Dokumentacja udostępniona przez prowadzącego.*
[2] Wikipedia contributors. *JH (hash function) — Wikipedia, The Free Encyclopedia.* `https://en.wikipedia.org/wiki/JH_(hash_function)`. Dostęp 12.06.2023. 2023.