MeshHash

-

Implementacja w języku Java

Maciej Lipski

Kacper Średnicki

Czerwiec 2023

Spis treści

1. Wprowadzenie

Algorytm MeshHash to funkcja skrótu przygotowana jako kandydat na algorytm SHA-3. Została zaprojektowana przez Björn'a Fay'a w 2008 roku. Kładzie ona nacisk przede wszystkim na bezpieczeństwo. Szybkość jest w niej elementem drugorzędnym, jednak jest ona zapewniona na poziomie funkcji SHA-2. W ramach projektu realizowanego na przedmiocie Bezpieczeństwo Danych w semestrze 2023L, przygotowano implementację algorytmu MeshHash w języku Java.

1.1. Ogólny opis działania algorytmu

Algorytm MeshHash działa w blokach. Jego stan wewnętrzny składa się z tzw. rur. Umożliwia to równoległe przetwarzanie bloków wiadomości. Każdy blok przetwarzany jest w P standardowych rundach (P to liczba rur, opis standardowych rund zawarto w 3.1). Po każdej rundzie zmienia się wewnętrzny stan algorytmu (opisany w 2.1)

Gdy wykonano P standardowych rund, następuje finalna runda bloku (opisana w 3.2), w której przetwarzany jest klucz oraz wewnętrzny stan algorytmu.

Kiedy wszystkie bloki zostaną przetworzone, wykonują się rundy finalne całego algorytmu (opisane w 3.3). Następnie obliczana jest wartość skrótu poprzez tzw. wyciskanie gąbki.

2. Implementacja MeshHash w języku Java

Implementację algorytmu w języku Java przygotowano na podstawie dokumentacji algorytmu [1] oraz referencyjnej implementacji w języku C. Problemem w implementacji algorytmu MeshHash w języku Java okazał się typ Word (dalej nazywany także „słowo"). W dokumentacji algorytmu jest on definiowany jako 64-bitowy typ danych o wartościach w zakresie od 0 do $2^{64} - 1$ [1]. W Javie nie ma takiego typu danych [2], w przeciwieństwie do języka C, gdzie istnieje typ unsigned long long spełniający założenia algorytmu. Mimo to, do reprezentacji typu Word, zdecydowano się na wykorzystanie typu long, który jest 8-bajtowym typem przechowującym wartości w zakresie od -2^{63} do $2^{63} - 1$. Jest to jedyna różnica między algorytmem opisanym w dokumentacji, a przygotowaną implementacją w języku Java.

2.1. Klasa MeshHash

W celu reprezentowania wewnętrznego stanu algorytmu przygotowano klasę MeshHash. Posiada ona następujące atrybuty:
— **int P** - liczba „rur" służących przetwarzaniu danych.
— **long[] pipes** - tablica przechowująca „rury", składająca się ze słów (reprezentowanych przez typ long).
— **int block_round_counter** - przechowuje ilość rund wykonanych na aktualnie przetwarzanym bloku.
— **int key_counter** - licznik użyć klucza.
— **long [] key** - klucz zapisany w postaci tablicy słów.
— **int key_length** - długość klucza w słowach.
— **byte [] message** - wiadomość w postaci tablicy bajtów.
— **int hash_bit_length** - długość skrótu w bitach.
— **long [] dataStream** - źródłowy strumień danych dla algorytmu.
— **Counter bit_counter** - obiekt klasy Counter (opisanej w 2.1.1), przechowujący ilość bitów wiadomości.
— **Counter block_counter** - obiekt klasy Counter, zliczający przetworzone bloki.
Wszystkie atrybuty klasy MeshHash są prywatne.

2.1.1. Klasa Counter

Klasa Counter reprezentuje liczniki, które składają się z czterech słów (**bit_counter, block_counter**). Jej atrybutem jest czteroelementowa tablica **long [] counterArray**. Wartość licznika to $\sum_{i=0}^{3} 2^{64i} * counterArray[i]$. Można ją uzyskać za pomocą metody *public long getValue()*. Klasa Counter posiada także metodę *public void increment()* do zwiększania wartości licznika o 1.

2.1.2. Inicjalizacja algorytmu

Inicjalizacja stanu wewnętrznego algorytmu ma miejsce w konstruktorze klasy MeshHash. Przyjmuje on jako argumenty wiadomość i klucz w postaci byte[], oraz długość skrótu w postaci int. Na początku następuje przypisanie argumentów do wewnętrznych zmiennych obiektu klasy MeshHash.

Następnie, z wykorzystaniem metody *computeNumberOfPipes(int hash_lenght)*, obliczana jest liczba „rur" - P. Liczba rur jest najmniejszą wartością całkowitą $\geqslant hash\_bit\_length/64 + 1$. Musi być ona także większa lub

równa od 4 i mniejsza lub równa od 256. Obliczenia te realizuje wspomniana metoda. Kolejnym krokiem jest inicjalizacja P-elementowej tablicy **long[] pipes**.

Po przetworzeniu „rur" algorytm przechodzi do inicjalizacji liczników. Każdy licznik, poza **bit_counter** jest zerowany. Wartość **bit_counter** ustawiana jest na liczbę bitów wiadomości (*message.length*8*). Dokonywane jest to poprzez inkrementowanie bit_counter odpowiednią ilość razy (2.1.1).

Klucz, podany jako byte [], jest przetwarzany do tablicy słów (long[] key). Dokonywane jest to z wykorzystaniem metody *convertBytesToWords(byte [] bytes)*, która wykonuje odpowiednie operacje bitowe. Zmiennej key_length jest przypisywana długość klucza w słowach.

Ostatnim krokiem inicjalizacji algorytmu jest przygotowanie strumienia danych do przetwarzania (**long [] dataStream**). Zajmuje się tym metoda *prepareDataStream()*. **DataStream** w ostatecznej postaci to wynik konkatenacji **key, message** (**message** przetwarzane jest na słowa) oraz r zer. R jest obliczane jako najmniejsza całkowita wielokrotność 64P większa od $(key\_length * 64 + bit\_counter.getValue() + r)$.

Po tych krokach algorytm jest gotowy do obliczania skrótu.

2.2. Klasa MathOperations

Klasa MathOperations jest klasą odpowiedzialną za obsługę działań matematycznych i bitowych wykonywanych w algorytmie MeshHash, których brakuje w bibliotece standardowej Javy. Posiada metody wykonujące potrzebne w tym celu operacje. W przypadku części z nich zastosowano tzw. overloading, aby umożliwić podawanie do nich różnych parametrów. Jeżeli jako parametr podawany jest String, jest on parsowany do typu long w systemie szesnastkowym. Metody klasy MathOperations to:

1. *public static long addModulo264(long a, String b)* to metoda wykonująca dodawanie dwóch liczb modulo 2^{64}. Wykorzystuje w tym celu operator logiczny "AND"oraz odpowiednią maskę. Zwraca wynik będący liczbą typu long.
2. *public static long multiplyModulo264(long a, String b)* to metoda wykonująca mnożenie dwóch liczb modulo 2^{64}. Podobnie jak metoda wyżej, wykorzystuje operator logiczny "AND"oraz maskę. Również zwraca wynik będący liczbą typu long.
3. *public static long rotRi(long word, int i)* to metoda wykonująca rotację bitową o i bitów modulo 64. Sprawdza czy liczba i mieści się w zakresie 0-63 wykonując na niej operację modulo. Dopiero wtedy następuje rotacja bitowa podanego jako parametr słowa o i bitów, która jest wykonywana z wykorzystaniem funkcji *rotateRight* oferowanej przez klasę Long z biblioteki standardowej Javy. Analogicznie do poprzednich metod, typ zwracany to long.

3. Obliczanie wartości skrótu

Po wykonaniu inicjalizacji, możliwe jest na rzecz obiektu klasy MeshHash wywołanie metody *private byte [] computeMeshHash()* (przedstawiona w kodzie 1). Metoda ta zwraca skrót w postaci tablicy bajtów. Wywoływanie są w niej pomocnicze metody odpowiadające poszczególnym etapom działania algorytmu MeshHash.

```
private byte[] computeMeshHash() {
    for (int i = 0; i < dataStream.length; i++) {
        long data = dataStream[i];
        while (block_round_counter < P) {
            normalRound(data, block_round_counter);
        }
        finalBlockRound();
    }
    final_rounds();
    byte[] hashValue = computeHashValue();
    return hashValue;
}
```

Rysunek 1. Metoda *computeMeshHash()* - główna metoda klasy MeshHash

W metodzie *computeHashValue* przetwarzane są wszystkie bloki **dataStream**. Każdy z nich jest przetwarzany przez P standardowych rund (opisanych w 2). Po wykonaniu P normalnych rund, wykonywana jest finalna runda bloku (tak jak w 3.2). Następnie wykonywane są finalne rundy całego algorytmu (opisane w 3.3) oraz metoda computeHashValue w celu obliczenia ostatecznej wartości funkcji skrótu (opisana w 3.4).

3.1. Standardowa runda

Standardowa runda (*normalRound()*) składa się przede wszystkim z metody *SBox()*. Na jej początku mają miejsce operację mające na celu przygotowanie danych wejściowych dla metody *SBox*. Są to operacje takie mnożenie i dodawanie modulo 2^{64}, XOR, rotRi (opisane w 2.2). wykonywane na „rurach" oraz odpowiednich stałych (widocznych w kodzie 2).

Po wykonaniu standardowej rundy inkrementowany jest licznik **block_round_counter**. Standardowa runda jest wykonywana P razy dla każdego bloku. W przygotowanej implementacji, metoda *normal_round)* przyjmuje jako argument słowo (dane typu long) oraz indeks rury. Stąd też standardowe rundy są zawsze wykonywane wewnątrz pętli P razy.

```
long sBoxInput = pipes[index] ^ (MathOperations.multiplyModulo64(index,
↪  "0101010101010101") ^ data);
sBoxInput = MathOperations.rotRi(sBoxInput, 37 * index);
pipes[index] = MathOperations.addModulo64(SBox(sBoxInput), pipes[(index
↪  + 1) % P]);
block_round_counter++;
```

Rysunek 2. Standardowa runda

3.1.1. SBox

SBox to zbiór podstawowych operacji matematycznych wykonywanych wielokrotnie w trakcie działania algorytmu. To na nich oparte jest jego bezpieczeństwo. SBox algorytmu MeshHash składa się z trzech operacji wykonywanych dwukrotnie (przedstawionych w kodzie 3): mnożenie modulo 2^{64} przez odpowiednią stałą, dodawanie modulo 2^{64} kolejnej stałej oraz operacji rotRi(37) (operacje opisane w 2.2).

```
    private long SBox(long input) {
    input = MathOperations.multiplyModulo264(input, "9e3779b97f4a7bb9");
    input = MathOperations.addModulo264(input, "5e2d58d8b3bcdef7");
    input = MathOperations.rotRi(input, 37);
    input = MathOperations.multiplyModulo264(input, "9e3779b97f4a7bb9");
    input = MathOperations.addModulo264(input, "5e2d58d8b3bcdef7");
    input = MathOperations.rotRi(input, 37);
    return input;
}
```

Rysunek 3. Zaimplementowany w języku Java SBox algorytmu MeshHash

3.2. Finalna runda dla bloku

Finalna runda dla bloku (*final_block_round*) wykonywana jest po P normalnych rundach. Składa się ona z dwóch czynności - przetworzenia licznika **block_counter** (opisane w 3.2.1) i przetworzenia klucza, jeśli został podany (opisane w 3.2.2).

3.2.1. Przetwarzanie block_counter

Na początku przetwarzania licznika **block_counter**, zerowany jest licznik **block_round_counter**. Następnie wykonywany jest P razy SBox dla danych uwzględniających rury oraz licznik block_counter (tak jak w kodzie 4). Po wykonaniu tych działań, licznik block_counter jest inkrementowany.

```
    private void processBlockCounter() {
    block_round_counter = 0;
    for (int i = 0; i < P; i++) {
        pipes[i] = SBox(pipes[i] ^ block_counter.counterArray[i % 4]);
    }
    block_counter.increment();
}
```

Rysunek 4. Przetwarzanie licznika block_counter

3.2.2. Przetwarzanie klucza

Klucz jest przetwarzany w dwóch krokach. Na początku przetwarzany jest w zagnieżdżonej pętli. Modyfikuje ona wartości rur z wykorzystaniem SBox dla danych będących wynikami operacji XOR między daną rurą, a elementem klucza. Licznik klucza jest aktualizowany, a następnie przetwarzana jest długość klucza (również z wykorzystaniem SBox). W tym kroku rury modyfikowane są podobnie, jednakże operacji XOR poddawane są poza daną rurą: *key_length* oraz stała wartość.

```
private void processKey() {
    for (int i = key_counter; i < key_length + key_counter; i += P) {
        for (int j = 0; j < P; j++) {
            pipes[j] = SBox(pipes[j] ^ key[(i + j) % key_length]);
        }
    }
    key_counter = (key_counter + 1) % key_length;
    for (int i = 0; i < P; i++) {
        pipes[i] = SBox(pipes[i] ^ key_length ^
        ↪  (Long.parseUnsignedLong("0101010101010101", 16) * i));
    }
}
```

Rysunek 5. Przetwarzanie klucza

3.3. Rundy finalne

Po przetworzeniu wszystkich bloków wykonywane są rundy finalne algorytmu. P razy wykonywane są działania SBox przetwarzające **bit_counter, hash_bit_length** wraz z odpowiednimi rurami i stałymi połączonymi działaniem XOR (tak jak w kodzie 6).

```
private void final_rounds() {
        for (int i = 0; i < bit_counter.counterArray.length; i++) {
            for (int j = 0; j < P; j++) {
                pipes[j] = SBox(pipes[j] ^ bit_counter.counterArray[i] ^
                ↪  (Long.parseUnsignedLong("0101010101010101", 16) * j));
            }
        }
        for (int i = 0; i < P; i++) {
            {
                pipes[i] = SBox(pipes[i] ^ hash_bit_length ^
                ↪  (Long.parseUnsignedLong("0101010101010101", 16) * i));
            }
        }
}
```

Rysunek 6. Finalne rundy

3.4. Obliczanie skrótu

Finalna wartość funkcji skrótu obliczana jest przez tzw. „wyciskanie gąbki” (z ang. squeezing the sponge). Dla każdego bajtu z wynikowej tablicy wykonywana jest standardowa runda z danymi wejściowymi równymi 0. Zgodnie z opisem z sekcji 3.1, wykonywana jest ona P razy dla każdego bajtu wynikowego skrótu. Następnie ma miejsce działanie XOR łączące „rury” oraz ich AND z odpowiednią stałą (widoczne w kodzie 7). Jeśli indeks danego bajtu w skrócie modulo P jest równy P-1, należy dla tego bajtu wykonać finalną rundę blokową (opisaną w 3.2).

```
private byte[] computeHashValue() {
    byte[] hashValue = new byte[hash_bit_length / 8];
    for (int i = 0; i < hashValue.length; i++) {
        for (int j = 0; j < P; j++) {
            normalRound(0, j);
        }
        byte temp = 0;
        for (int j = 0; j < P; j += 2)
            temp ^= pipes[j];
        hashValue[i] = (byte) (temp & 255);
        if (i % P == P - 1) finalBlockRound();
    }
    return hashValue;
}
```

Rysunek 7. Obliczanie skrótu

4. Sposób użycia przygotowanej implementacji

Dla użytkowników przygotowanej implementacji, przygotowano statyczną metodą publiczną *public static String computeMeshHash()*. Zwraca ona wartość skrótu w postaci hexadecymalnego Stringa. Każdy bajt skrótu odpowiada dwóm znakom w zwracanym Stringu. Bajty wypisywane są w kolejności od najmłodszego do najstarszego. Jako argument, metoda *computeMeshHash()* przyjmuje wiadomość (w postaci typu String), długość skrótu w bitach oraz opcjonalnie klucz (także jako String). W celu zachowania bezpieczeństwa, użytkownik nie ma dostępu do konstruktora klasy MeshHash, a także do metod używanych bezpośrednio przy obliczaniu skrótu.

4.1. Opis metody computeMeshHash

Na początku działania metody *computeMeshHash()* wiadomość i klucz są przetwarzane do tablicy bajtów, z wykorzystaniem metody *getBytes()* z klasy String. W przypadku niepodania klucza, jego wartość jest uzupełniania pustym String'iem. Następnie tworzony jest obiekt klasy MeshHash z przetworzonymi wcześniej argumentami. Wywoływana jest dla niego metoda *private byte[] computeMeshHash()* (opisana w 3.4), której wynik jest przypisywany do tablicy bajtów. Obliczona funkcja skrótu jest konwertowana do postaci opisanego wyżej hexadecymalnego String'a, z wykorzystaniem obiektu klasy StringBuilder.

4.2. Użycie jako PRG

Algorytm MeshHash może być używany także jako generator liczb pseudolosowych. W takim przypadku *hash_bit_length* musi być ustawione na 0, a liczba rur jawnie podana. Obliczenie wartości skrótu w *computeHashValue()* musi być natomiast powtórzone wielokrotnie, zależnie od preferencji.

5. Testy działania algorytmu

Podstawową różnicą między przygotowaną w języku Java implementacją algorytmu MeshHash, a referencyjną implementacją i dokumentacją algorytmu jest wykorzystanie typu ze znakiem (long) w celu reprezentacji „słów" (więcej na ten temat w 2). Jest to znacząca różnica, która wpływa na niezgodność skrótów produkowanych przez przygotowaną implementację z wektorami testowymi.

Pomimo tego, skróty są pomyślnie generowane. Cechują się bezkolizyjnością i powtarzalnością. Jak w każdej funkcji skrótu, przy niewielkich zmianach wiadomości, wynikowe skróty znacząco się różnią. Przykładowe wyniki działania funkcji widoczne są na rysunku 8.

```
Ala ma koty:
1a56c1562981ec886cee3526
Ala ma kota:
2ddd1ff3a6f440d1c4782c85
Ala ma koty:
1a56c1562981ec886cee3526
```

Rysunek 8. Przykładowy wynik działania funkcji skrótu (z kluczem "bdan")

6. Podsumowanie i wnioski

Głównym problemem implementacyjnym był wspomniany już w rozdziale 5 brak wymaganego w algorytmie typu zmiennej. Takiej natury problem nie wystąpiłby w przypadku języka programowania, zapewniającego wymagany w dokumentacji funkcji skrótu typ. Z tego względu, Java nie jest optymalnym wyborem do implementacji MeshHash. Mimo problemów implementacyjnych, udało się jednak skonstruować poprawnie działający algorytm, spełniający wytyczne z referencyjnej dokumentacji oraz podstawowe zasady działania bezpiecznej funkcji skrótu.

7. Bibliografia

[1] Björn Fay. *MeshHash*. Submission to NIST. 2008. URL: `http://ehash.iaik.tugraz.at/uploads/5/5a/Specification_DIN-A4.pdf`.

[2] Oracle. *The Java Tutorials - Primitive Data Types*. URL: `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.htmll` (term. wiz. 08. 05. 2023).