

WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI  
POLITECHNIKA WROCŁAWSKA

# MEMORY-HARD FUNCTIONS

KONRAD ŚWIERCZYŃSKI  
NR INDEKSU: 229818

Promotor  
dr Filip Zagórski



Politechnika  
Wrocławska  
WROCŁAW 2019



# Spis treści

<b>1</b>	<b>Wprowadzenie</b>	<b>1</b>
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	Etykietowanie Grafu . . . . .	3
2.2	Superkoncentrator . . . . .	4
2.3	Grafy Depth-Robust . . . . .	5
2.4	Grafy rozproszone . . . . .	5
2.5	Grafy rzędowe . . . . .	6
2.6	Ograniczenia Złożoności RSG . . . . .	6
<b>3</b>	<b>Riffle Scrambler</b>	<b>9</b>
3.1	Budowa grafu . . . . .	9
3.1.1	Parametry . . . . .	9
3.1.2	Tworzenie grafu na podstawie permutacji . . . . .	9
3.1.3	Śledzenie trajektorii . . . . .	11
3.2	Tworzenie permutacji . . . . .	11
3.2.1	Talia kart jako permutacja . . . . .	11
3.2.2	Algorytm generowania permutacji . . . . .	12
3.3	Algorytm . . . . .	12
<b>4</b>	<b>Implementacja</b>	<b>17</b>
4.1	Wymagania konkursu . . . . .	17
4.2	Opis technologii . . . . .	17
4.3	Omówienie kodów źródłowych . . . . .	17
4.3.1	Wysokopoziomowy interfejs . . . . .	18
4.3.2	Niskopoziomowy interfejs . . . . .	19
4.4	Parametry . . . . .	19
4.5	Uwidacznianie się własności <i>memory-hard</i> . . . . .	20
4.6	<b>RiffleScrambler</b> z wiersza poleceń . . . . .	20
4.7	Przykład użycia . . . . .	22
4.8	Testy . . . . .	23
<b>5</b>	<b>Instalacja i wdrożenie</b>	<b>25</b>
5.1	Biblioteka . . . . .	25
5.2	Testy . . . . .	25
5.3	Pomiary wydajności . . . . .	25
5.4	<b>RiffleScrambler</b> z wiersza poleceń . . . . .	25
<b>6</b>	<b>Podsumowanie</b>	<b>27</b>
	<b>Bibliografia</b>	<b>29</b>
<b>A</b>	<b>Zawartość płyty CD</b>	<b>31</b>



# Wprowadzenie

Moderately hard functions. Funkcje umiarkowanie ciężkie do obliczenia mają wiele zastosowań takich jak dowody pracy (ang. *proofs of work*), funkcje wyprowadzenia klucza oraz password hashing. Przy przechowywaniu hasel ważne jest, aby zminimalizować skutki wycieknienia pliku z hasłami. Zamiast przechowywać krotki (*login, password*) tekstem jawnym, dodaje się losową sól i przechowuje w postaci (*login, f(password, salt), salt*), gdzie  $f$  jest moderately hard function. Oznacza to, że funkcja ta musi być obliczana podczas każdego uwierzytelniania w celu sprawdzenia poprawności hasła. Nie może być ona zatem zbyt ciężka do obliczenia dla aplikacji uwierzytelniającej. Z drugiej strony, gdy krotka (*login, y, salt*) wycieknie, adversarz może przeprowadzać atak słownikowy obliczając funkcję  $f$  przy każdej próbie, co powinno być kosztowne. W tym celu zaczęto stosować funkcje, które obliczają wiele razy kryptograficzną funkcję skrótu. Popularnym przykładem takiej funkcji jest PBKDF2 (ang. *Password-Based Key Derivation Function 2*), dla której zalecanym parametrem bezpieczeństwa w 2000 roku było 1024 iteracji, a już w 2005 zaczęto zalecać 4096 iteracji, z powodu wzrostu wydajności CPU. Niestety takie podejście nie gwarantuje zabezpieczenia przed adversarzem używającym specjalizowany układ scalony (ang. *ASIC - Application-Specific Integrated Circuit*). Układy takie są znacznie bardziej wydajne pod względem szybkości obliczania funkcji skrótu takich jak SHA256 czy MD5 niż tradycyjne architektury.

– Porównanie Antminer - GPU - CPU –

Zauważono jednak, że na różnych architekturach koszt dostępu do pamięci jest dużo bardziej zrównoważony niż koszt obliczeń. [Percival [16]] Zaproponowano więc memory-hard functions (MHF), które wywołują podczas obliczania wiele kosztownych czasowo odwołań do pamięci.

scrypt - pierwsza taka funkcja

O MHF można myśleć jako o pewnej kolejności dostępu do komórek pamięci. Odwołania następują do już wcześniej obliczonych wartości w komórkach. Zatem kolejność tą można opisać jako acykliczny graf skierowany (DAG).

.....



# Preliminaries

W dalszej części używana będzie następująca notacja. Zbiory  $\mathbb{N} = \{0, 1, 2, \dots\}$ ,  $\mathbb{N}^+ = \{1, 2, \dots\}$ .  $[c] := \{1, 2, \dots, c\}$  oraz  $[b, c] = \{b, b+1, \dots, c\}$ , gdzie  $b, c \in \mathbb{N}$  oraz  $b \leq c$ .

Skierowany graf acykliczny (ang. *directed acyclic graph*, DAG)  $G = (V, E)$  jest rozmiaru  $n$  jeżeli  $|V| = n$ . Wierzchołek  $v \in V$  ma stopień wchodzący  $\delta$  równy największemu stopniowi wchodzącemu wśród jego wierzchołków  $\delta = \text{indeg}(v)$ , jeżeli istnieje  $\delta$  wchodzących krawędzi  $\delta = |(V \times v) \cap E|$ . Graf  $G$  ma stopień wchodzący  $\delta = \text{indeg}(G) = \max_{v \in V} \text{indeg}(v)$ . Wierzchołki o stopniu wchodzącym 0 nazywane są źródłami, a wierzchołki bez krawędzi wychodzących nazywane są ujściami.

Zbiór rodziców wierzchołka  $v \in V$  oznaczany jest jako  $\text{parents}_G(v) = \{u \in V : (u, v) \in E\}$ . Uogólniając, zbiór przodków  $v$  oznaczany jest jako  $\text{ancestors}_G(v) = \bigcup_{i \geq 1} \text{parents}_G^i(v)$ , przyjmując  $\text{parents}_G^{i+1}(v) = \text{parents}_G(\text{parents}_G^i(v))$ . Jeżeli wybór grafu  $G$  wynika z kontekstu, będziemy oznaczać te zbiory jako  $\text{parents}$  oraz  $\text{ancestors}$ .

Zbiór wszystkich ujść w grafie  $G$  oznaczany jest jako  $\text{sinks}(G) = \{v \in V : \nexists (v, u) \in E\}$ . DAG  $G$ , który jest spójny, a tylko takie będą rozważane w dalszej części pracy, zachowuje równość  $\text{ancestors}(\text{sinks}(G)) = V$ .

Dla skierowanej ścieżki  $p = (v_1, v_2, \dots, v_z)$  w  $G$ , jej długość jest równa ilości wierzchołków przez które przechodzi  $\text{length}(p) := z$ . Mając DAG  $G$ , oznaczamy długość jego najdłuższej ścieżki jako  $\text{depth}(G)$ .

Mając podzbiór wierzchołków grafu  $S \subset V$ , poprzez  $G - S$  oznaczać będziemy DAG otrzymany z  $G$  poprzez usunięcie wierzchołków z  $S$  oraz krawędzi wychodzących lub wchodzących do wierzchołków z  $S$ .

## 2.1 Etykietowanie Grafu

**Definicja 2.1** (*Równoległe/Sekwencyjne Etykietowanie Grafu*). Niech  $G = (V, E)$  będzie grafem skierowanym grafem acyklicznym i niech  $T \subset V$  będzie zbiorem wierzchołków do oetykietowania.  $T$  będzie nazywane celem. Stanem etykietowania  $G$  jest zbiór  $P_i \subset V$ . Poprawnym etykietowaniem równoległym jest ciąg  $P = (P_0, \dots, P_t)$  stanów etykietowania  $G$ , gdzie  $P_0 = \emptyset$  oraz gdzie spełnione są warunki 1 oraz 2 poniżej. Etykietowanie sekwencyjne musi dodatkowo spełniać warunek 3.

1. Każdy wierzchołek z celu jest w pewnej konfiguracji oetykietowany (nie koniecznie wszystkie jednocześnie).

$$\forall x \in T \exists x \leq t : x \in P_x$$

2. Oetykietować wierzchołek można tylko wtedy, gdy wszyscy jego rodzice są oetykietowani w poprzednim kroku.

$$\forall i \in [t] : x \in (P_i \setminus P_{i-1}) \Rightarrow \text{parents}(x) \subset P_{i-1}$$

3. W każdym kroku można oetykietować co najwyżej jeden wierzchołek.

$$\forall i \in [t] : |P_i \setminus P_{i-1}| \leq 1$$

Zbiory poprawnych etykietowań sekwencyjnych i równoległych grafu  $G$  z celem  $T$  oznaczamy odpowiednio jako  $\mathcal{P}_{G,T}$  oraz  $\mathcal{P}_{G,T}^{\parallel}$ . Etykietowania najbardziej interesujących przypadków, gdy  $T = \text{sinks}(G)$ , oznaczamy  $\mathcal{P}_G$  oraz  $\mathcal{P}_G^{\parallel}$ .

Można zauważyć, że  $\mathcal{P}_{G,T} \subset \mathcal{P}_{G,T}^{\parallel}$ .



**Definicja 2.2** *Złożoność czasową (ang. time,  $t$ ), pamięciową (ang. space,  $s$ ), pamięciowo-czasową (ang. space-time,  $st$ ) oraz łączna (ang. cumulative,  $cc$ ) etykietowania  $P = (P_0, \dots, P_t) \in \mathcal{P}_G^\parallel$  są zdefiniowane jako*

$$\Pi_t(P) = t, \Pi_s(P) = \max_{y \in [t]} |P_y|, \Pi_{st}(P) = \Pi_t(P) * \Pi_s(P), \Pi_{cc}(P) = \sum_{i \in [t]} |P_i|$$

*Dla  $\alpha \in \{s, t, st, cc\}$  oraz celu  $T \subset V$ , złożoności sekwencyjnego oraz równoległego etykietowania grafu  $G$  definiujemy jako*

$$\Pi_\alpha(G, T) = \min_{P \in \mathcal{P}_{G,T}} \Pi_\alpha(P)$$

$$\Pi_\alpha^\parallel(G, T) = \min_{P \in \mathcal{P}_{G,T}^\parallel} \Pi_\alpha(P)$$

*Kiedy  $T = \text{sinks}(G)$ , piszemy  $\Pi_\alpha^\parallel(G)$  oraz  $\Pi_\alpha(G)$ .*

Ponieważ  $\mathcal{P}_{G,T} \subset \mathcal{P}_{G,T}^\parallel$ , dla dowolnej złożoności etykietowania  $\alpha \in \{s, t, st, cc\}$  oraz dowolnego grafu  $G$  złożoność etykietowania równoległe jest nie większa, niż złożoność etykietowania sekwencyjnego  $\Pi_\alpha(G) \geq \Pi_\alpha^\parallel(G)$ , a złożoność łączna jest nie większa, niż czasowo-pamięciowa  $\Pi_{st}(G) \geq \Pi_{cc}(G)$  i  $\Pi_{st}^\parallel(G) \geq \Pi_{cc}^\parallel(G)$ .

W tej pracy głównie rozważane jest badanie złożoności  $\Pi_{st}$ , oraz  $\Pi_{cc}^\parallel$ , ponieważ ukazują one kolejno koszt przeprowadzania etykietowania na jednordzeniowej maszynie (np. procesor x86) oraz koszt etykietowania na wyspecjalizowanym układzie.

Aby zobaczyć jakie wartości mogą przyjąć przedstawione złożoności, rozważmy graf rozmiaru  $n$ . Każdy graf rozmiaru  $n$  może zostać oetykietowany w  $n$  krokach, ponieważ ma tylko  $n$  wierzchołków. Każdy stan etykietowania nie może również zawierać więcej niż  $n$  elementów. Zatem górne ograniczenie możemy przedstawić następująco

$$\forall G_n \in \mathbb{G}_n : \Pi_{cc}^\parallel(G_n) \leq \Pi_{st}(G_n) \leq n^2.$$

Zobaczmy jak wyglądają złożoności dla grafu pełnego  $K_n = (V = [n], E = \{(i, j) : 1 \leq i < j \leq n\})$  oraz dla  $Q_n = (V = [n], E = \{(i, i+1) : 1 \leq i < n\})$ .

$$n(n-1)/2 \leq \Pi_{cc}^\parallel(K_n) \leq \Pi_{st}(K_n) \leq n^2$$

Graf  $K_n$  maksymalizuje złożoności etykietowania, a  $\Pi_{cc}^\parallel$  jest różne tylko o stałą od  $\Pi_{st}$ . Co oznacza, że koszt obliczania funkcji opartej na takim grafie byłby zdominowany przez koszt dostępu do pamięci, a wyspecjalizowane układy nie dały by dużej przewagi nad tradycyjnym procesorem. Jest to bardzo pożądane dla MHF, jednak ze względu na bardzo wysoki stopień grafu  $K_n$ , nie jest on przydatny przy konstruowaniu takich funkcji. Stopień wchodzący w grafie  $Q_n$  jest równy 1, jednak złożoność etykietowania jest bardzo niska

$$\Pi_{cc}^\parallel(Q_n) = \Pi_{st}(Q_n) \leq n.$$

Oznacza to, że koszt obliczania funkcji opartej na takim grafie (taką funkcją jest PBKDF2) nie jest zależny w dużej mierze od kosztu dostępu do pamięci nawet dla dużego  $n$ .

## 2.2 Superkoncentrator

Superkoncentrator jest grafem, w którym moc zbioru przodków dla wierzchołków szybko rośnie wraz z numerem pokolenia. Oznacza to, że zbiór kolejnych wierzchołków, będzie posiadał liczny zbiór rodziców, co czyni superkoncentrator bardzo przydatnym do konstrukcji MHF.

**Definicja 2.3** (*N-Superkoncentrator*). *Skierowany graf acykliczny  $G = (V, E)$  o ustalonym stopniu wchodzącym,  $N$  wejściach i  $N$  wyjściach nazywany jest N-Superkoncentratorem, gdy dla każdego  $k \in [N]$  oraz dla każdej pary podzbiorów  $V_1 \subset V$  k wejść i  $V_2 \subset V$  k wyjść istnieje  $k$  wierzchołkowo-rozłącznych ścieżek łączących wierzchołki ze zbioru  $V_1$  z wierzchołkami w  $V_2$ .*



**Definicja 2.4** ( $(N, \lambda)$ -Superkoncentrator). Niech  $G_i, i = 0, \dots, \lambda - 1$  będą  $N$ -Superkoncentratorami. Niech graf  $G$  będzie połączeniem wyjść  $G_i$  do odpowiadających wejść w  $G_{i+1}$  dla  $i = 0, \dots, \lambda - 2$ . Graf  $G$  jest nazywany  $(N, \lambda)$ -Superkoncentratorem.

**Lemat 2.1** [8, Lemat 1] (Ograniczenie dolne dla  $(N, \lambda)$ -Superkoncentratora). Etykietowanie  $(N, \lambda)$ -Superkoncentratora używając  $S \leq N/20$  etykiet, wymaga  $T$  kroków, gdzie

$$T \geq N \left( \frac{\lambda N}{64S} \right)^\lambda.$$

## 2.3 Grafy Depth-Robust

Alwen i Blocki w swojej pracy [7] pokazali, że istnieje zależność między złożonością etykietowania, a depth-robustness. Na podstawie tej właściwości potrafimy określić dolne i górne ograniczenie  $\Pi_{cc}^\parallel$ .

**Definicja 2.5** (Depth-Robustness). Dla  $n \in \mathbb{N}$  oraz  $e, d \in [n]$  DAG  $G = (V, E)$  jest  $(e, d)$ -depth-robust, jeżeli

$$\forall S \subset V \ |S| \leq e \Rightarrow \text{depth}(G - S) \geq d.$$

**Twierdzenie 2.1** [7, Twierdzenie 4] Niech DAG  $G$  będzie  $(e, d)$ -depth-robust, wtedy  $\Pi_{cc}^\parallel > ed$ .

**Definicja 2.6** Jeżeli graf  $G$  nie jest  $(e, d)$ -depth-robust to nazywany jest  $(e, d)$ -reducible.

**Twierdzenie 2.2** [7, Twierdzenie 10] Niech  $G \in \mathbb{G}_{n, \delta}$  taki, że  $G$  jest  $(e, d)$ -reducible. Wtedy

$$\Pi_{cc}^\parallel(G) = O \left( \min_{g \in [d, n]} \left\{ n \left( \frac{dn}{g} + \delta g + e \right) \right\} \right)$$

biorąc  $g = \sqrt{\frac{dn}{\delta}}$  upraszcza się to do  $\Pi_{cc}^\parallel(G) = O \left( n(\sqrt{dn\delta} + e) \right)$ .

## 2.4 Grafy rozproszone

**Definicja 2.7** (Zależność, ang. dependency). Niech  $G = (V, E)$  będzie acyklicznym grafem skierowanym. Niech  $L \subseteq V$ . Mówimy, że  $L$  ma  $(z, g)$ -dependency jeżeli istnieją wierzchołkowo rozłączne ścieżki  $p_1, \dots, p_z$  kończące się w  $L$ , gdzie każda jest długości co najmniej  $g$ .

**Definicja 2.8** (Graf rozproszony, ang. dispresed). Niech  $g, k \in \mathbb{N}$  i  $g \geq k$ . DAG  $G$  jest nazywany  $(g, k)$ -dispresed jeżeli istnieje uporządkowanie jego wierzchołków takie, że następujące warunki są spełnione. Niech  $[k]$  oznacza ostatnie  $k$  wierzchołków o uporządkowaniu  $G$  i niech  $L_j = [jg, (j+1)g - 1]$  będzie  $j$ -tym podprzedziałem. Wtedy  $\forall j \in [k/g]$  przedział  $L_j$  ma  $(g, g)$ -dependency. W ogólności, jeżeli dla  $\epsilon \in (0, 1]$  każdy przedział  $L_j$  ma tylko  $(\epsilon g, g)$ -dependency, graf  $G$  nazywany jest  $(\epsilon, g, g)$ -dispresed.

**Definicja 2.9** Acykliczny graf skierowany  $G = (V, E)$  nazywany jest  $(\lambda, \epsilon, g, k)$ -dispresed jeżeli istnieje  $\lambda \in \mathbb{N}^+$  rozłącznych podzbiorów wierzchołków  $\{L_i \subseteq V\}$ , każdy o rozmiarze  $k$  oraz spełnione są następujące warunki.

1. Dla każdego  $L_i$  istnieje ścieżka przechodząca przez wszystkie wierzchołki  $L_i$ .
2. Dla ustalonego porządku topologicznego  $G$ . Dla każdego  $i \in [\lambda]$  niech  $G_i$  będzie podgrafem  $G$ , zawierającym wszystkie wierzchołki z  $G$ , aż do ostatniego wierzchołka z  $L_i$ .  $G_i$  jest  $(\epsilon, g, k)$ -dispresed.

Zbiór grafów, które są  $(\lambda, \epsilon, g, k)$ -dispresed oznaczamy jako  $\mathbb{D}_{\epsilon, g}^{\lambda, k}$ .

**Twierdzenie 2.3** Niech  $G \in \mathbb{D}_{\epsilon, g}^{\lambda, k}$ .

$$\Pi_{cc}^\parallel(G) \geq \epsilon \lambda g \left( \frac{k}{2} - g \right)$$



## 2.5 Grafy rzędowe

**Definicja 2.10** (Graf  $N$  rzędowy). Niech  $n, N \in \mathbb{N}^+$  takie, że  $N + 1$  dzieli  $n$  oraz niech  $k = n/(N + 1)$ . Mówimy, że graf  $G$  jest  $N$  rzędowy, jeżeli  $G$  zawiera ścieżkę przechodzącą przez  $n$  wierzchołków  $(v_1, \dots, v_n)$  oraz dzieląc go na warstwy  $L_j = \{v_{jk+1}, \dots, v_{j(k+1)}\}$ , dla  $j \in \{0, \dots, N\}$ , pozostałe krawędzie łączą wierzchołki z niższej warstwy  $L_j$  jedynie z wierzchołkami z wyższych warstw  $L_i$ ,  $i \in \{j + 1, \dots, N\}$ .

**Lemat 2.2** [6, Lemat 4.2] Niech  $G$  będzie  $N$  rzędowym grafem, wtedy dla dowolnego  $t \in \mathbb{N}^+$ ,  $G$  jest  $(n/t, Nt)$ -reducible.

## 2.6 Ograniczenia Złożoności RSG

**Twierdzenie 2.4** Niech  $\lambda, n \in \mathbb{N}^+$  takie, że  $n = \bar{n}(2\lambda c + 1)$ , gdzie  $c \in \mathbb{N}$  i  $\bar{n} = 2^c$ . Wtedy dla  $g = \lfloor \sqrt{\bar{n}} \rfloor$   $RSG_{\lambda}^{\bar{n}} \in \mathbb{D}_{1,g}^{\lambda, \bar{n}}$  oraz  $\Pi_{cc}^{\parallel}(RSG_{\lambda}^{\bar{n}}) = \Omega\left(\frac{n^{1.5}}{c\sqrt{c\lambda}}\right)$ .

**Dowód.** Niech  $G = RSG_{\lambda}^{\bar{n}}$ , niech  $G_1, G_2, \dots, G_{\lambda}$  będą podgrafami  $G$  opisanymi w DEF[...]. Pokażemy, że każdy  $G_i$  jest  $(g, \bar{n})$ -dispersed dla  $g = \lfloor \sqrt{\bar{n}} \rfloor$ .

Wybermy  $i \in [\lambda]$  niech  $L_1$  będzie ostatnimi  $\bar{n}$  wierzchołkami w porządku topologicznym grafu  $G_i$ . Oznaczamy wierzchołki zbioru  $L_1$  poprzez  $1 \times [\bar{n}]$ , gdzie druga pozycja odpowiada kolejności wierzchołka w porządku topologicznym. Niech  $\bar{g} = \lfloor \bar{n}/g \rfloor$ , dla każdego  $j \in [\bar{g}]$   $L_{1,j} = \{< 1, jg + x > : x \in [0, g - 1]\}$ . Pokażemy, że wszystkie  $L_{1,j}$  mają  $(g, g)$ -dependency.

Niech  $L_0$  będzie  $\bar{n}$  pierwszymi wierzchołkami  $G_i$ , które oznaczamy  $0 \times [\bar{n}]$  (ponownie druga pozycja odpowiada porządkowi topograficznemu). Zauważmy, że dla  $n > 1$  i  $g = \lfloor \sqrt{\bar{n}} \rfloor$  prawdą jest, że  $g(g - 2c + 1) \leq n$ . Zatem zbiór  $S = \{< 0, i(g - 2c + 1) > : i \in [g]\}$  jest całkowicie zawarty w  $L_0$ .

Z własności RSG [Superconcentrator- ....] wynika, że skoro zbiory  $S$  oraz  $L_{1,j}$  mają po  $g$  wierzchołków, to istnieje  $g$  wierzchołkowo-rozłącznych ścieżek o długości  $2c$  między wierzchołkami tych zbiorów. Zatem  $L_{1,j}$  ma  $(g, 2c)$ -dependency.

Rozszerzmy to do  $(g, g)$ -dependency. Niech ścieżka  $p$  zaczynająca się w wierzchołku  $< 0, v > \in S$  będzie ścieżką w  $(g, 2c)$ -dependency  $L_{1,j}$ . Zauważmy, że istnieje ścieżka przechodząca przez wszystkie wierzchołki  $L_0$  oraz, że wierzchołki zbioru  $S$  są oddzielone między sobą o  $g - 2c$  wierzchołków. Możemy dodać na początek ścieżki  $p$  ścieżkę  $< 0, v - (g - 2c - 1) >, < 0, v - (g - 2c - 2) >, \dots, < 0, v >$ . Otrzymujemy w ten sposób ścieżkę  $p_+$  o długości  $2c + g - 2c = g$ . Ponieważ każda para ścieżek  $p \neq q$  w  $(g, 2c)$ -dependency  $L_{1,j}$  jest wierzchołkowo-rozłączna, to w szczególności zaczynać się muszą w różnych wierzchołkach  $< 0, v_p > \neq < 0, v_q >$ . Ponieważ wierzchołki w  $S$  są od siebie oddalone o  $g - 2c$  wierzchołków, zatem ścieżki  $p^+$  i  $q^+$  nadal pozostają rozłączne. Rozszerzając w ten sposób wszystkie ścieżki z  $(g, 2c)$ -dependency otrzymujemy ścieżki wierzchołkowo-rozłączne długości  $g$ . Z tego wynika, że  $L_{1,j}$  ma  $(g, g)$ -dependency, co dowodzi, że  $RSG_{\lambda}^{\bar{n}} \in \mathbb{D}_{1,g}^{\lambda, \bar{n}}$ . Pozostaje obliczyć górne ograniczenie używając [8, Twierdzenie 6].

$$\Pi_{cc}^{\parallel}(RSG_{\lambda}^{\bar{n}}) = \lambda g \left( \frac{\bar{n}}{2} - g \right) \geq \lambda \lfloor \sqrt{\bar{n}} \rfloor \left( \frac{\bar{n}}{2} - \lfloor \sqrt{\bar{n}} \rfloor \right) = \lambda \sqrt{\bar{n}} \left( \frac{\bar{n}}{2} - \sqrt{\bar{n}} \right) - O(\bar{n}) = \Omega(\lambda \bar{n}) = \Omega\left(\frac{n^{1.5}}{c\sqrt{c\lambda}}\right) \quad \square$$

**Twierdzenie 2.5** Niech  $\lambda, g \in \mathbb{N}^+$ ,  $N = 2^g$ ,  $n = N(2\lambda g + 1)$ , wtedy

$$\Pi_{cc}^{\parallel}(RSG_{\lambda}^{\bar{n}}) = O\left(n^{1.6}\right)$$

**Dowód.** Kozystając z [...],  $RSG_{\lambda}^N$  jest  $N$  rzędowym grafem. Z [6, Lemat 4.2] wynika więc, że  $RSG_{\lambda}^N$  jest  $(n/t, \lambda + t - \lambda - 1)$ -reducible dla dowolnego  $t \geq 1$ . Z twierdzenia [7, Twierdzenie 10] wynika, że  $\Pi_{cc}^{\parallel}(RSG_{\lambda}^{\bar{n}}) = O\left(n\left(\sqrt{(\lambda t + t - \lambda - 1)n\delta + \frac{n}{t}}\right)\right)$ . Aby dostać najdokładniejsze (najmniejsze) ograniczenie górne trzeba zminimalizować  $n\left(\sqrt{(\lambda t + t - \lambda - 1)n\delta + \frac{n}{t}}\right)$ . Zanim jednak przejdziemy do minimalizowania, uproścmy nieco to wyrażenie

$$n\left(\sqrt{(\lambda t + t - \lambda - 1)n\delta + \frac{n}{t}}\right) \leq n\left(\sqrt{2\lambda t n\delta + \frac{n}{t}}\right).$$

Teraz możemy znaleźć minimum względem naszego parametru  $t$ .

$$\frac{\partial}{\partial t} n \left( \sqrt{(2\lambda t)n\delta} + \frac{n}{t} \right) = \frac{\sqrt{2\delta\lambda n^3}}{2t} - \frac{n^2}{t^2}$$

Minimum znajduje się w punkcie, gdzie pochodna ma wartość zero.

$$\begin{aligned} \frac{\sqrt{2\delta\lambda n^3}}{2t} - \frac{n^2}{t^2} &= 0 \\ t &= \frac{2n^2}{\sqrt{2\delta\lambda n^3}} = O\left(n^{\frac{1}{3}}\right) \end{aligned}$$

Zatem podstawiając  $t$  minimalizujące ograniczenie górne do wzoru z [7, Twierdzenie 10] otrzymujemy

$$\Pi_{cc}^{\parallel}(RSG_{\lambda}^{\bar{n}}) = O\left(n \left( \sqrt{(\lambda n^{\frac{1}{3}} + n^{\frac{1}{3}} - \lambda - 1)n\delta} + \frac{n}{n^{\frac{1}{3}}} \right)\right) = O\left(n^{1.\bar{6}}\right)$$

□



# Riffle Scrambler

RiffleScrambler [8] jest nową rodziną acyklicznych grafów skierowanych (nazywaną RSG), której odpowiada funkcja *memory-hard* z dostępem do pamięci niezależnym od hasła (iMHF). W funkcji tej, podobnie jak w Catenie, kolejność obliczeń zdefiniowana jest za pomocą grafu. Przewagą funkcji RiffleScrambler, jest to, że graf generowany jest na podstawie soli, tak jak w funkcji Ballon Hashing. Oznacza, to, że dla każda sól odpowiada (z dużym prawdopodobieństwem) innemu grafowi, co zwiększa odporność na ataki równoległe. Dla Cateny są dwa predefiniowane grafy *bit-reversal* i *double-butterfly*. Jednocześnie RiffleScrambler zapewnia lepszą wydajność przy obliczaniu niż Ballon Hashing, ponieważ ma dużo mniejszy stopień wchodzący grafu, który jest równy 3. Ponieważ jest superkoncentratorem, osiąga kompromis pamięć-czas oraz dolne ograniczenie złożoności etykietowania równoległego takie same jak Catena.

## 3.1 Budowa grafu

### 3.1.1 Parametry

Funkcja **RiffleScrambler** używa następujących parametrów:

- $s$  - sól, używana do wygenerowania grafu  $G$ ,
- $g$  - ilość pamięci potrzebnej do obliczeń, dla  $G = (V, E)$  zbiór wierzchołków można przedstawić jako  $V = V_0 \cup V_1 \cup \dots \cup V_{2\lambda g}$ , gdzie  $|V_i| = 2^g$ ,
- $\lambda$  - liczba warstw grafu  $G$ , może być postrzegana jako liczba iteracji.

Sól używana jest do generowania liczb pseudolosowych, potrzebnych do zbudowania grafu. Parametr  $g$  określa ilość pamięci, jaką trzeba będzie wykorzystać podczas obliczania funkcji. Podczas obliczeń potrzebne jest  $2^g$  komórek, gdzie każda przechowuje wynik kryptograficznej funkcji skrótu. Wartość  $2^g$  będziemy oznaczać jako  $N$ . Parametr  $\lambda$  definiuje ile warstw będzie miał końcowy graf, co bezpośrednio wpływa na czas obliczania funkcji. Graf dla zadanych parametrów wpływających na jego rozmiar oznaczany jest  $RSG_N^\lambda$ . Sól nie ma wpływu na rozmiar grafu, ani jego właściwości, używana jest do generowania części krawędzi w grafie.

### 3.1.2 Tworzenie grafu na podstawie permutacji

Niech  $HW(x)$  (ang. *Hamming weight*) oznacza ilość jedynek w wyrazie binarnym  $x$ . Niech  $\bar{x}$  oznacza negację wyrazu  $x$ , zatem  $HW(\bar{x})$  oznacza liczbę zer w wyrazie  $x$ .

**Definicja 3.1** Niech  $B = (b_0 \dots b_{n-1}) \in \{0, 1\}^n$  będzie wyrazem binarnym o długości  $n$ . Definiujemy rangę  $r_B(i)$   $i$ -tego bitu w  $B$  jako

$$r_B(i) = |\{j < i : b_j = b_i\}|.$$

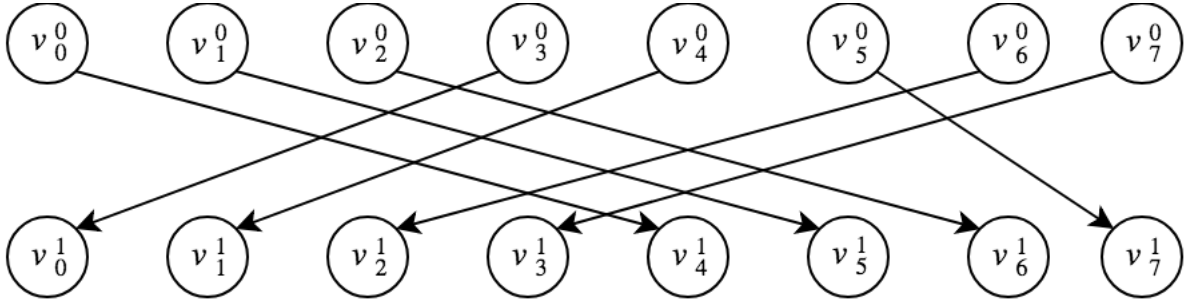
**Definicja 3.2** (*Riffle-Permutation*). Niech  $B = (b_0 \dots b_{n-1})$  będzie wyrazem binarnym o długości  $n$ . Permutacja  $\pi$  indukowana przez  $B$  zdefiniowana jest następująco

$$\pi_B(i) = \begin{cases} r_B(i), & \text{if } b_i = 0 \\ r_B(i) + HW(\bar{B}), & \text{if } b_i = 1 \end{cases}$$

dla każdego  $0 \leq i \leq n - 1$ .



**Przykład 3.1** Niech  $B = 11100100$ , wtedy  $r_B(0) = 0$ ,  $r_B(1) = 1$ ,  $r_B(2) = 2$ ,  $r_B(3) = 0$ ,  $r_B(4) = 1$ ,  $r_B(5) = 3$ ,  $r_B(6) = 2$ ,  $r_B(7) = 3$ . Mając rangi dla wszystkich pozycji, można utworzyć Riffle-Permutation indukowaną przez  $B$   $\pi_B = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 0 & 1 & 7 & 2 & 3 \end{pmatrix}$ . Ilustracja tego przykładu widoczna poniżej. *TODO link*

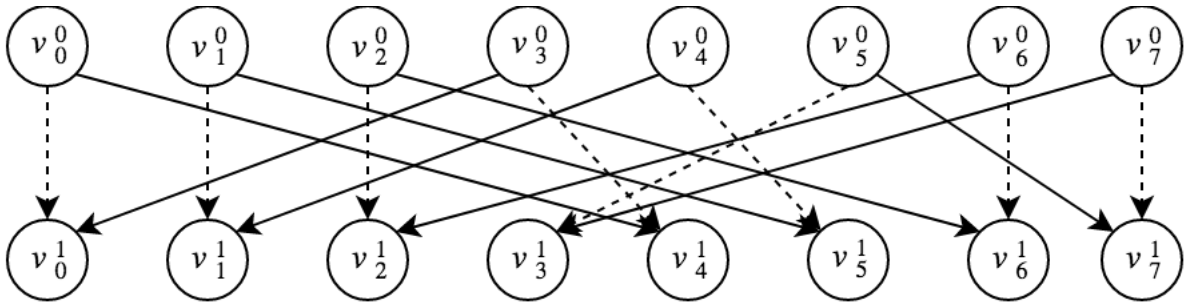


Rysunek 3.1: Graf utworzony z Riffle-Permutation indukowanej przez  $B = 11100100$ .

**Definicja 3.3** (*N-Single-Layer-Riffle-Graph*). Niech  $V = V^0 \cup V^1$ , gdzie  $V^i = \{v_0^i, \dots, v_{N-1}^i\}$ , niech  $B$  będzie słowem binarnym długości  $N$ . Niech  $\pi_B$  będzie Riffle-Permutaion indukowaną przez  $B$ . Graf *N-Single-Layer-Riffle-Graph* (dla parzystego  $N$ ) zdefiniowany jest jako graf na wierzchołkach  $V$  z następującymi krawędziami w zbiorze  $E$ :

- jedna krawędź:  $v_{N-1}^0 \rightarrow v_0^1$ ,
- $2(N-1)$  krawędzi:  $v_{i-1}^j \rightarrow v_i^j$ , dla  $i \in [N-1]$  oraz  $j \in \{0, 1\}$ ,
- $N$  krawędzi:  $v_i^0 \rightarrow v_{\pi_B(i)}^1$ , dla  $i \in \{0, \dots, N-1\}$ ,
- $N$  krawędzi:  $v_i^0 \rightarrow v_{\pi_{\bar{B}}(i)}^1$ , dla  $i \in \{0, \dots, N-1\}$ .

**Przykład 3.2** Kontynuując z danymi z poprzedniego przykładu *TODO* dodać link,  $\pi_{\bar{B}} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 4 & 5 & 3 & 6 & 7 \end{pmatrix}$ . *8-Single-Layer-Riffle-Graph* ukazany jest na Rysunku *TODO*.



Rysunek 3.2: 8-Single-Layer-Riffle-Graph dla  $B = 11100100$  (krawędź  $(v_7^0, v_0^1)$  oraz krawędzie poziome zostały pominięte). Krawędzie dla permutacji  $\pi_B$  oznaczone są linią ciągłą, a krawędzie dla permutacji  $\pi_{\bar{B}}$  oznaczone są linią przerywaną.

**Definicja 3.4** (*N-Double-Riffle-Graph*). Niech  $V$  oznacza zbiór wierzchołków, a  $E$  zbiór krawędzi grafu  $G = (V, E)$ . Niech  $B_0, \dots, B_{g-1}$  będą wyrazami binarnymi o długości  $N = 2^g$  każdy. *N-Double-riffle-Graph* jest otrzymywany poprzez ułożenie w stos  $2g$  grafów, które spełniają warunki *N-Single-Layer-Riffle-Graph*. Otrzymany tak graf ma  $(2g+1)2^g$  wierzchołków  $\{v_0^0, \dots, v_{2^g-1}^0\} \cup \dots \cup \{v_0^{2^g}, \dots, v_{2^g-1}^{2^g}\}$ , oraz następujące krawędzie:

- $(2g+1)2^g$  krawędzi:  $v_{i-1}^j \rightarrow v_i^j$  dla  $i \in [2^g - 1]$  i  $j \in \{0, 1, \dots, 2^g\}$ ,
- $2g$  krawędzi:  $v_{2^g-1}^j \rightarrow v_0^{j+1}$  dla  $j \in \{0, \dots, 2^g - 1\}$ ,
- $g2^g$  krawędzi:  $v_i^{j-1} \rightarrow v_{\pi_{B_j}(i)}^j$ , dla  $i \in \{0, \dots, 2^g - 1\}$  i  $j \in [g]$ ,
- $g2^g$  krawędzi:  $v_i^{j-1} \rightarrow v_{\pi_{\bar{B}_j}(i)}^j$ , dla  $i \in \{0, \dots, 2^g - 1\}$  i  $j \in [g]$ ,

oraz dla dolnych  $g$  warstw, które są symetryczne względem warstwy  $g$ :

- $g2^g$  krawędzi:  $v_i^{2^g-j} \rightarrow v_{\pi_{B_j}^{-1}(i)}^{2^g-j+1}$ , dla  $i \in \{0, \dots, 2^g - 1\}$  i  $j \in [g]$ ,
- $g2^g$  krawędzi:  $v_i^{2^g-j} \rightarrow v_{\pi_{\bar{B}_j}^{-1}(i)}^{2^g-j+1}$ , dla  $i \in \{0, \dots, 2^g - 1\}$  i  $j \in [g]$ ,

**Definicja 3.5** ( $(N, \lambda)$ -Double-Riffle-Graph). Niech  $G_i$ ,  $i \in \{0, 1, \dots, \lambda - 1\}$  będą  $N$ -Double-Riffle-Graph. Graf  $(N, \lambda)$ -Double-Riffle-Graph jest skonstruowany poprzez złączenie wyjść grafu  $G_i$  do odpowiadających wejść grafu  $G_{i+1}$ ,  $i \in \{0, 1, \dots, \lambda - 2\}$ .

### 3.1.3 Śledzenie trajektorii

Graf jest generowany za pomocą permutacji pseudolosowej  $\sigma$ . Ponieważ do generowania grafu potrzebne jest  $g$  słów binarnych o długości  $2^g$ , a permutacje zawierają  $2^g$  elementów, gdzie każdy ma maksymalnie  $g$  bitów znaczących, trzeba przekształcić permutację tak, aby otrzymać pożądane dane. Ta procedura nazwana jest śledzeniem trajektorii (ang. *trace trajectories*). Niech  $B$  będzie macierzą binarną o rozmiarze  $2^g \times g$ , gdzie  $j$ -ta kolumna jest binarną postacią  $\sigma(j) \in [2^g - 1]$ . Macierz  $\mathfrak{B} = (\mathfrak{B}_0, \dots, \mathfrak{B}_{g-1})$  oznaczać będzie transpozycję macierzy  $B$ , a więc macierz o potrzebnym do generacji grafu rozmiarze  $g \times 2^g$ .

## 3.2 Tworzenie permutacji

### 3.2.1 Talia kart jako permutacja

Początkową częścią funkcji **RiffleScrambler** jest generowanie permutacji na podstawie soli. Do utworzenia takiej permutacji używany jest algorytm **InverseRiffleShuffle**, który imituje tasowanie kart do gry w odwrotnej kolejności.

Podczas tasowania talia kart dzielona jest na dwie części. Podział odbywa się poprzez wybranie karty w środku talii, karty które są przed wybraną kartą tworzą pierwszą część, a pozostałe tworzą drugą część. Następnie te dwie części są ze sobą łączone w jeden stos poprzez losowe umieszczanie karty z góry pierwszej lub drugiej części na stosie. Można zauważyć, że kolejność kart wśród stosu z którego pochodzą nie zmienia się, lecz między kolejne karty z jednego stosu mogą wejść karty z drugiego.

W celu wygenerowania permutacji  $N$  elementów, myślimy o tych elementach jako o kartach, a o permutacji, jako o pewnej ich kolejności. Krok odwróconego sortowania wygląda następująco.

Dla każdej karty w talii losowany jest jeden bit, 0 lub 1. Wszystkie karty, dla których wylosowano 1 wyciągamy, zachowując ich kolejność i układamy na stos. Następnie umieszczamy ten stos na stosie kart, dla których wylosowano zera.

Dla każdego takiego kroku, dla każdej karty zapisywana jest informacja jaki bit został dla niej wylosowany. Zatem po  $n$  krokach, każda karta ma przypisany ciąg binarny długości  $n$ . Kończymy, kiedy talia kart jest dobrze posortowana, czyli kiedy każdej karcie przypisano unikalny ciąg binarny. Nowa kolejność elementów oznacza wygenerowaną permutację.

Można zauważyć, że liczba kroków tasowania  $N$  kart na pewno nie będzie mniejsza od  $\log N$ , bo potrzebujemy  $\log N$  bitów, aby każdej karcie przyporządkować inny ciąg binarny.

Algorytm **InverseRiffleShuffle** będzie mógł się zakończyć, kiedy każdy element będzie posiadał różny ciąg binarny. Średnio oznacza to  $2 \log N$  kroków [8].



### 3.2.2 Algorytm generowania permutacji

Razem z prezentacją RiffleScrambler zaproponowany został algorytm generowania permutacji [8, Algorytm 1], który sprawdzał warunek końca w czasie  $O(n^2)$  oraz potrzebował  $O(n^2)$  komórek pamięci, gdzie  $n$  oznacza wielkość permutacji.

Pseudokod [TODO...] przedstawia algorytm sprawdzający warunek końca w czasie  $O(n)$  korzystając z  $O(n \log n)$  komórek pamięci.

Algorytm opiera się na sortowaniu pozycyjnym radix sort [5], gdzie jako pozycje sortowanych elementów podawane są losowane bity. Permutowane elementy, trzymane są w tablicy, która jest sortowana po każdej iteracji dokładania kolejnego bitu. Dzięki temu sprawdzenie warunku końca odbywa się na posortowanej tablicy, którą wystarczy przejść sprawdzając czy każde dwa sąsiednie elementy są różne, co wykonywane jest w czasie liniowym.

## 3.3 Algorytm

Procedurę **RiffleScrambler**( $pwd, s, g, \lambda$ ) można z grubsza przedstawić następująco.

- Dla podanej soli  $s$  obliczana jest pseudolosowa permutacja  $\sigma$  (używając algorytmu **InverseRiffleShuffle**).
- Dla permutacji  $\sigma$  tworzona jest macierz  $\mathfrak{B} = \mathbf{TraceTrajectories}(B)$ .
- Dla wyrazów binarnych  $\mathfrak{B}_0, \dots, \mathfrak{B}_{g-1}$  generowany jest graf  $G$ , który jest N-Double-Riffle-Graph. Przypomnijmy,  $N = 2^g$ .
- Na grafie  $G$  zainicjalizowanym wartością  $pwd$ , oblicze są wartości w ostatnim rzędzie  $(v_0^{2g+1}, \dots, v_{2^g-1}^{2g+1})$ .
- Wartości z ostatniego rzędu przepisywane są do pierwszego,  $v_i^0 = v_1^{2g+1}$  dla  $i \in \{0, \dots, 2^g - 1\}$ , a następnie znów oblicza się wartości ostatnich rzędów. Powtarzane jest  $\lambda$  razy.
- Wartością końcową jest wartość w ostatnim wierzchołku, czyli w  $v_{2^g-1}^{2g}$ .

**Przykład 3.3** (*Generacja (8,1)-Double-Riffle-Graph*). Skoro  $N = 8$ , to  $g = 3$ , ponieważ  $N = 2^g$ .  $\lambda = 1$ .

Niech otrzymaną permutacją  $\sigma$  będzie  $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 4 & 6 & 3 & 2 & 7 & 0 & 1 \end{pmatrix}$ . Zatem binarna postać permutacji  $B = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$ .

Przeprowadzając śledzenie trajektorii, czyli transponując  $B$  otrzymujemy  $\mathfrak{B} = (\mathfrak{B}_0, \mathfrak{B}_1, \mathfrak{B}_2)$ ,  $\mathfrak{B} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}^T$ .

Teraz należy obliczyć permutacje  $\pi_{\mathfrak{B}_0}, \pi_{\mathfrak{B}_1}, \pi_{\mathfrak{B}_2}$ :

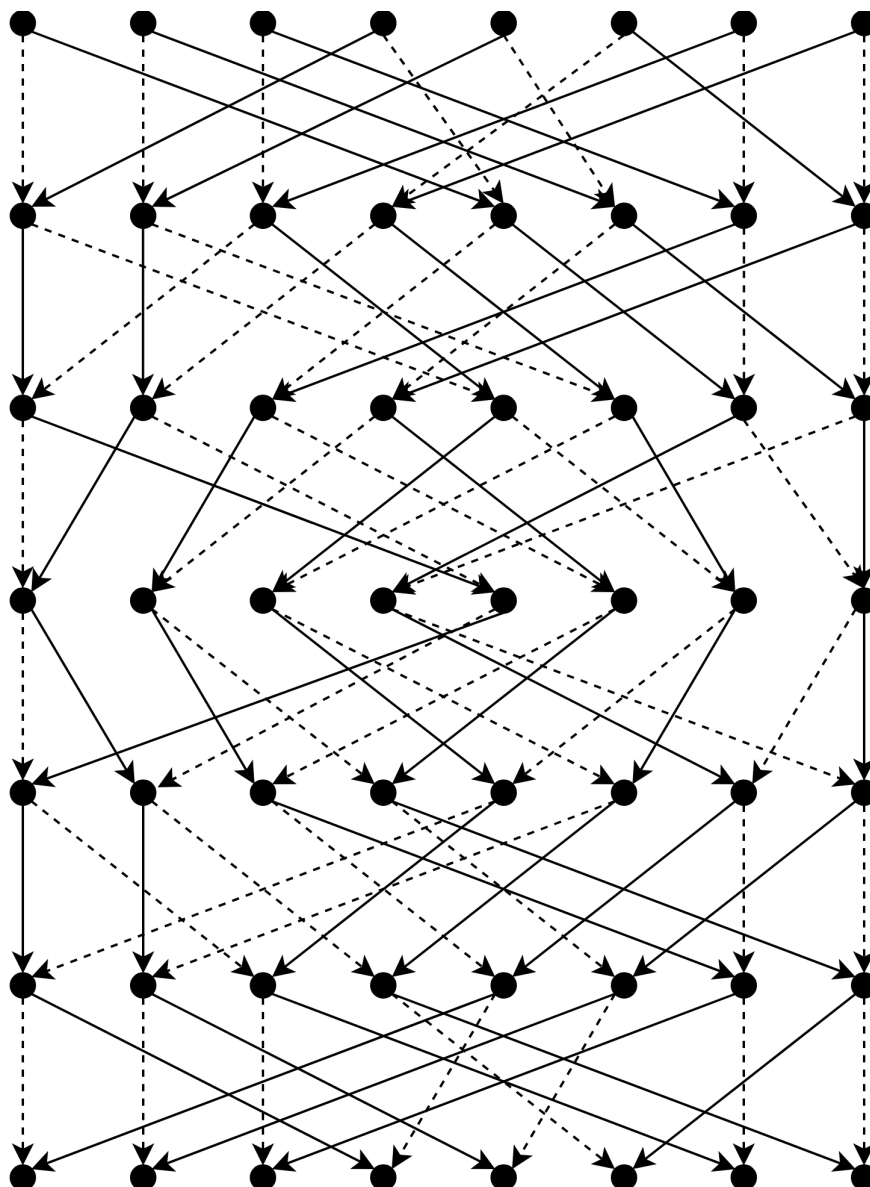
$$\pi_{\mathfrak{B}_0} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 0 & 1 & 7 & 2 & 3 \end{pmatrix},$$

$$\pi_{\mathfrak{B}_1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 4 & 5 & 6 & 7 & 2 & 3 \end{pmatrix},$$

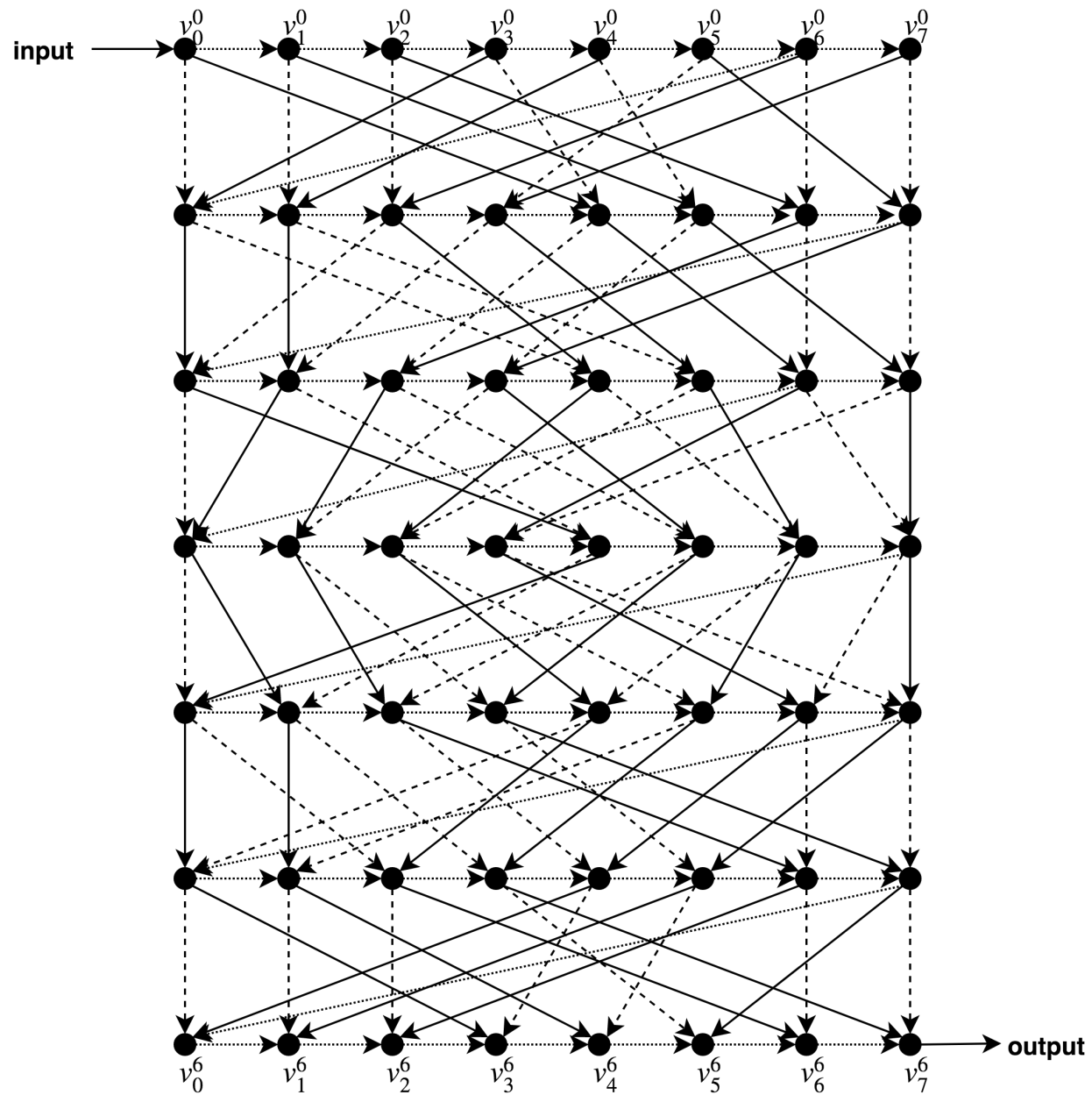
$$\pi_{\mathfrak{B}_2} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & 1 & 5 & 2 & 6 & 3 & 7 \end{pmatrix}.$$

Wygenerowany graf z krawędziami zależnymi od permutacji ukazany jest na ... Dodając do nie go krawędzie nie zależne od permutacji, otrzymujemy pełny graf (8,1)-Double-Riffle-Graph.





Rysunek 3.3:  $(8,1)$ -Double-Riffle-Graph (krawędzie niezależne od permutacji, czyli przekątne oraz krawędzie poziome, zostały pominięte). Krawędzie dla permutacji oznaczone są linią ciągłą, a krawędzie dla permutacji negacji oznaczone są linią przerywaną.



Rysunek 3.4: (8,1)-Double-Riffle-Graph ze wszystkimi krawędziami oraz oznaczeniem wejścia i wyjścia. Krawędzie dla permutacji oznaczone są linią ciągłą, krawędzie dla permutacji negacji oznaczone są linią przerywaną, a krawędzie nie zależne od permutacji oznaczone są liniami kropkowanymi

---

**Algorithm 3.1: RiffleShuffle**

---

**Input:**  $N$ , bitGenerator**Output:**  $\sigma$ 

```
/* perm zawiera krotki będące początkowym indeksem oraz wyrazem binarnym. perm[i] = (id, bin) dla
   i ∈ {0, ..., N - 1} */
1 perm = ((0, ""), (1, ""), ..., (N - 1, "")) ;
2 while ∃i ∈ {0, ..., N - 2} perm[i][1] == perm[i + 1][1] do
3   numberOfOnes = 0 ;
4   for i = 0 to N - 1 do
5     newBit = bitGenerator() ;
6     perm[i][1] = perm[i][1] + newBit ;
7     if newBit == "1" then
8       numberOfOnes = numberOfOnes + 1 ;
9     end if
10  end for
11  lastIndexOfOnes = 0 ;
12  lastIndexOfZeros = numOfOnes ;
13  for i = 0 to N - 1 do
14    if lastBit(permutation[i][1]) == "1" then
15      tmp[lastIndexOfOnes] = permutation[i] ;
16      lastIndexOfOnes = lastIndexOfOnes + 1 ;
17    else
18      tmp[lastIndexOfZeros] = permutation[i] ;
19      lastIndexOfZeros = lastIndexOfZeros + 1 ;
20    end if
21  end for
22  perm = tmp ;
23 end while
24 σ = (perm[0].id, perm[1].id, ..., perm[N - 1].id) ;
25 return σ ;
```

---

---

**Algorithm 3.2: GenGraph**

---

**Input:**  $g, \sigma$ **Output:**  $G$ 

```
1 N = 2g ;
2 V = {vij : i ∈ {0, ..., N - 1}; j ∈ {0, ..., 2g}} ;
3 E = {vij → vi+1j : i ∈ {0, ..., N - 2}; j ∈ {0, ..., 2g}} ;
4 E = E ∪ {vN-1j → v0j+1 : j ∈ {0, ..., 2g - 1}} ;
5 B = (B0, ..., Bg-1) = TraceTrajectories(σ) ;
6 for m = 0 to g - 2 do
7   B2g+1-m = Bm ;
8 end for
/* Bity w słowach binarnych oznaczamy dodając indeks dolny. Bj = Bj,0Bj,1...Bj,2g-1 */
9 for j = 0 to 2g - 1 do
10  for i = 0 to 2g - 1 do
11    E = E ∪ {vij → vπBj,ii+1} ∪ {vij → vπBj,ii+1} ;
12  end for
13 end for
14 return G = (V, E) ;
```

---




---

**Algorithm 3.3: RiffleScrambler**


---

**Input:**  $s, g, \text{pwd}, \lambda, H$  - funkcja skrótu

**Output:** pwd hash

```

1  $\sigma = \text{RiffleShuffle}(2^g, \text{pseudoRandomBitGenerator}(s, H))$  ;
2  $G = (V, E) = \text{GenGraph}(g, \sigma)$  ;
3  $v_0^0 = H(X)$  ;
4 for  $i = 1$  to  $2^g - 1$  do
5    $v_i^0 = H(v_{i-1}^0)$  ;
6 end for
7 for  $r = 1$  to  $\lambda$  do
8   for  $j = 0$  to  $2g$  do
9     for  $i = 0$  to  $2^g - 1$  do
10       $v_i^{j+1} = 0$  ;
11      forall  $v \rightarrow v_i^{j+1} \in E$  do
12         $v_i^{j+1} = H(v_i^{j+1}, v)$  ;
13      end forall
14    end for
15  end for
16  for  $i = 0$  to  $2^g - 1$  do
17     $v_i^0 = v_i^{2g+1}$  ;
18  end for
19 end for
20  $\text{hash} = v_{2^g-1}^{2g}$  ;
21 return  $\text{hash}$  ;

```

---

# Implementacja

Implementacja **RiffleScrambler** została napisana tak, aby spełniała wymagania konkursu na funkcję do przechowywania haseł [4, PHC, ang *Password Hashing Competition*], który rozgrywał się w latach 2013 - 2015.

## 4.1 Wymagania konkursu

Wymagania konkursu PHC odnoszące się do implementacji:

- implementacja powinna być napisana w języku C(++) w taki sposób, aby była przenośna,
- do implementacji powinny być dołączone instrukcje kompilacji (np. Makefile),
- interfejs powinien być dostępny do użycia w języku C, oraz powinien zapewniać funkcję z podaną poniżej sygnaturą 1,
- implementacja może używać biblioteki OpenSSL,
- do implementacji powinny zostać dołączone testy.

```
// Interfejs zaproponowany w ogłoszeniu konkursu na funkcję do przechowywania haseł
int PHS(void *out, size_t outlen, const void *in, size_t inlen,
        const void *salt, size_t saltlen, % unsigned int t_cost, unsigned int m_cost);
```

Na zwycięzce konkursu wybrano Argon2, a Catena, Lyra2 [13], yescrypt [11] oraz Makwa [12] otrzymały specjalne wyróżnienie.

## 4.2 Opis technologii

Początkowa implementacja napisana została w języku Python 3.7, jednak ze względu na niską wydajność oraz nie spełnianie wymagań PHC napisana została ostatecznie w języku C++17, standard ISO/IEC 14882:2017 [9].

W implementacji używana jest biblioteka OpenSSL [2]. Z podanej biblioteki użyto interfejsu EVP [3] który dostarcza podstawowe kryptograficzne funkcje skrótu takie jak SHA2, SHA3, blake2s czy ripemd160.

## 4.3 Omówienie kodów źródłowych

Omówiony zostanie interfejs funkcji **RiffleScrambler**, który spełnia wymogi PHC oraz posiada dodatkowy wysokopoziomowy interfejs pozwalający na wygodne tworzenie aplikacji uwierzytelniającej.

Kompletne kody źródłowe znajdują się do płyce CD dołączonej do niniejszej pracy. (patrz Dodatek A).



### 4.3.1 Wysokopoziomowy interfejs

Wysokopoziomowy interfejs upraszcza użycie funkcji **RiffleScrambler** podczas pisania programów na wyższym abstrakcji takich jak na przykład aplikacje uwierzytelniające. Podobnie jak w implementacji zwycięzcy konkursu PHC (argon2) w udostępnionych funkcjach znajdują się:

- funkcja zwracająca wynik jako tekst ze znakami w systemie heksadecymalnym - linia 1,
- funkcja zwracająca wynik jako wektor znaków w systemie heksadecymalnym - linia 18,
- funkcja zwracająca zakodowany wynik wraz z użytą solą oraz parametrami w celu łatwego przechowywania i łatwej weryfikacji, sól oraz hasło zakodowane są za pomocą base64 [10] - linia 34,
- funkcja weryfikująca poprawność podanego hasła dla zakodowanego wyniku - linia 51.

Każda z podanych funkcji ma również sygnaturę, gdzie hasło i sól podawane są za pomocą typu `std::string`.

```

1  /**
2   * Hashowanie hasła za pomocą RiffleScrambler
3   * @param pwd Wskaźnik na hasło
4   * @param pwrlen_bytes Długość hasła w bajtach
5   * @param salt Wskaźnik na sól
6   * @param saltlen_bytes Długość soli w bajtach
7   * @param garlic Parametr "g" oznaczający koszt pamięci
8   * @param depth Parametr "lambda" oznaczający koszt czasowy
9   * @param hash_func Nazwa kryptograficznej funkcji skrótu do użycia we wnętrzu
10  * @return Hash hasła jako ciąg znaków w systemie szesnastkowym
11  */
12  std::string riffle_scrambler(const void *const pwd, const size_t pwrlen_bytes,
13  const void *const salt, const size_t saltlen_bytes,
14  const uint64_t garlic, const uint64_t depth,
15  const std::string hash_func="sha256");
16
17
18  /**
19   * Hashowanie hasła za pomocą RiffleScrambler
20   * @param garlic Parametr "g" oznaczający koszt pamięci
21   * @param depth Parametr "lambda" oznaczający koszt czasowy
22   * @param pwd Wskaźnik na hasło
23   * @param pwrlen_bytes Długość hasła w bajtach
24   * @param salt Wskaźnik na sól
25   * @param saltlen_bytes Długość soli w bajtach
26   * @param hash_func Nazwa kryptograficznej funkcji skrótu do użycia we wnętrzu
27   * @return Hash hasła jako wektor znaków w systemie szesnastkowym
28  */
29  std::vector<unsigned char> riffle_scrambler_hash_raw(const uint64_t garlic,
30  const uint64_t depth, const void *pwd, const size_t pwrlen_bytes,
31  const void *salt, const size_t saltlen_bytes, const std::string hash_func="sha256");
32
33
34  /**
35   * Hashowanie hasła za pomocą RiffleScrambler
36   * kodowanie wyniku wraz z parametrami w jeden ciąg znaków
37   * @param garlic Parametr "g" oznaczający koszt pamięci
38   * @param depth Parametr "lambda" oznaczający koszt czasowy
39   * @param pwd Wskaźnik na hasło
40   * @param pwrlen_bytes Długość hasła w bajtach

```

```
41  * @param salt Wskaźnik na sól
42  * @param saltlen_bytes Długość soli w bajtach
43  * @param hash_func Nazwa kryptograficznej funkcji skrótu do użycia wewnątrz
44  * @return Zakodowane parametry funkcji wraz z hashem hasła i solą
45  */
46  std::string riffle_scrambler_encoded(const uint64_t garlic, const uint64_t depth,
47  const void *pwd, const size_t pwrlen_bytes,
48  const void *salt, const size_t saltlen_bytes, const std::string hash_func="sha256");
49
50
51  /**
52  * Weryfikacja hasła z hashem zakodowanym hashem dla zakodowanych parametrów i soli
53  * @param encoded Zakodowane parametry wraz z hashem i solą
54  * @param pwd Wskaźnik na hasło
55  * @param pwd_len Długość hasła w bajtach
56  * @return true jeśli hashe są równe, false w przeciwnym przypadku
57  */
58  bool riffle_scrambler_verify(const std::string encoded, const void *pwd, const size_t pwd_len);
```

### 4.3.2 Niskopoziomowy interfejs

Implementacja jest zgodna z interfejsem zaproponowanym przez PHC.

```
1  /**
2  * Hashowanie hasła za pomocą RiffleScrambler, wynik zwracany jest do zmiennej @hash
3  * @param pwd Wskaźnik na hasło
4  * @param pwrlen_bytes Rozmiar hasła w bajtach
5  * @param salt Wskaźnika na sól
6  * @param saltlen_bytes Rozmiar soli w bajtach
7  * @param t_cost Parametr "lambda" oznaczający ilość iteracji
8  * @param m_cost Parametr "g" oznaczający koszt rozmiar używanej pamięci
9  * @param hash Bufor do którego będzie zapisany hash hasła
10  * @param hashlen Rozmiar bufora, oznacza ile bajtów z hasła hasła ma zostać zapisane
11  * @return RIFFLE_SCRAMBLER_OK jeśli obliczenia zakończyły się pomyślnie, kod błędu w przeciwnym przypadku
12  */
13  int riffle_scrambler(void *hash, const size_t hashlen, const void *pwd, const size_t pwrlen_bytes,
14  const void *salt, const size_t saltlen_bytes, const uint64_t t_cost, const uint64_t m_cost);
```

## 4.4 Parametry

Funkcja przyjmuje parametry opisane w poprzednim rozdziale [Parametry].

- Parametr **garlic** oznacza  $g$  z poprzedniego rozdziału, czyli długość rzędu, która wynosi  $N = 2^g$ . Definiuje to rozmiar pamięci, która jest potrzebny do wykonania obliczeń. Ponieważ w pamięci trzymane są wartości z dwóch rzędów grafu w jednej chwili, czyli dwóch rzędów po  $2^g$  elementów, gdzie każda trzyma wynik wewnętrznej funkcji haszującej. Zatem zapotrzebowanie na pamięć wynosi  $2^{g+1} \times m$ , gdzie  $m$  oznacza rozmiar wyniku funkcji haszującej (np. dla SHA256 jest to 256 bitów, tak jak wskazuje nazwa). Na ogólne zużycie pamięci przez program w obecnej wersji wpływa też graf, który jest generowany na początku i trzymany przez czas działania programu. Przyjmuje wartość od 0 do  $2^{24} - 1$ .
- Parametr **depth** oznacza ilość warstw grafu i w poprzednim rozdziale oznaczony był jako  $\lambda$ , przyjmuje wartość całkowitą od 1 do  $2^{64} - 1$ .



- `pwd` oznacza hasło. Może być to wskaźnik na dane o długości od 0 do  $2^{32} - 1$  bitów. Nie jest sprawdzana jakość hasła. Zapewnienie, że podane hasło jest wystarczająco długie lub spełnia pewne warunki leży po stronie programu używającego funkcję **RiffleScrambler**.
- Parametr `salt` dostarcza do programu sól kryptograficzną o długości od 0 do  $2^{32} - 1$  bitów. Powinna być ona losowa oraz inna dla każdego z przechowywanych haseł, zalecaną długością soli jest co najmniej 16 bajtów [?].
- Parametr `hash_func` określa funkcję hashującą, która ma być użyta wewnątrz **RiffleScrambler**. Dostępne są następujące funkcje: sha224, sha256, sha384, sha512, ripemd160, blake2s256, blake2s512 oraz do celów testowych sha1, md5, mdc2.

Ważny jest dobór parametrów `garlic` ( $g$ ) oraz `depth` ( $\lambda$ ), ponieważ to od nich zależy bezpieczeństwo przechowywanych haseł. Zalecanymi parametrami, podobnie jak dla Argon2, jest `garlic` = 12 oraz `depth` = 4 zapewniające dostateczne bezpieczeństwo oraz racjonalny czas obliczeń. Oznacza to użycie pamięci wynoszące 16 MiB, co jest wyższą wartością niż rozmiar pamięci podręcznej na poziomie L-1 oraz L-2 dla większości procesorów, a zazwyczaj nawet większa od całkowitej pamięci podręcznej, co wymusza kosztowne sięganie do pamięci RAM.

## 4.5 Uwidacznianie się własności *memory-hard*

Implementacja została zbadana pod kontem wydajności w zależności od podawanych parametrów. Na podstawie wyników można zaobserwować, jak uwidacznia się własność *memory-hard*.

Na wykresie 4.1b widać, że zużycie pamięci rośnie wykładniczo od wartości parametru  $g$ . Ponieważ zapotrzebowanie na pamięć rośnie, procentowo mniej elementów mieści się w pamięci podręcznej.

Dobrze odwzorowuje to wykres 4.1a. Oprócz sytuacji dla  $g < 7$ , którą ciężko wyjaśnić, a więc biorąc pod uwagę wartości dla  $g \geq 7$ , liczba odwołań do danych, które nie znajdowały się w pamięci podręcznej (ang. *cache misses*) zaczyna rosnąć od pewnego momentu. Pierwszy wzrost ma miejsce dla  $g = 10$ , co odpowiada zapotrzebowaniu na pamięć ok. 1 MiB. Biorąc pod uwagę, że pamięć podręczna obsługuje również inne procesy, w tym program monitorujący, można wnioskować, że liczba odczytów z pamięci RAM zaczyna być procentowo coraz większa od momentu, gdy zapotrzebowanie na pamięć dorównuje ilości pamięci podręcznej (procesor na maszynie testowej wyposażony był w 3 MiB pamięci podręcznej).

Kontynuując to rozważanie, skoro liczba wolnych odczytów zaczyna przeważać dla dużych  $g$ , to czas obliczenia pojedynczego wierzchołka w grafie powinien się zwiększać. Tak właśnie się dzieje, co obrazuje wykres 4.1e, który ilustruje czas obliczania pojedynczego wierzchołka w zależności od parametru  $g$ .

Co za tym idzie, obliczając graf z taką samą liczbą wierzchołków, ale dla różnych parametrów  $g$  (wyównując ilość wierzchołków za pomocą parametru  $\lambda$ ), czas obliczeń takiej funkcji powinien być wyższy dla większych wartości  $g$ . Tą zależność wydaje się potwierdzać wykres 4.1f.

Oznacza to, że koszt odwoływania się do pamięci ma wpływ na czas obliczeń, co ilustruje własność *memory-hard*.

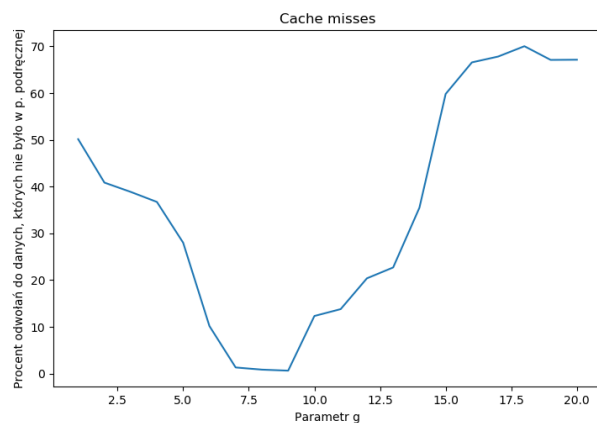
Wykres 4.1c przedstawia wpływ na czas obliczeń **RiffleScrambler** parametru  $g$ , który jest wykładniczy, oraz parametry  $\lambda$ , który jest liniowy.

## 4.6 RiffleScrambler z wiersza poleceń

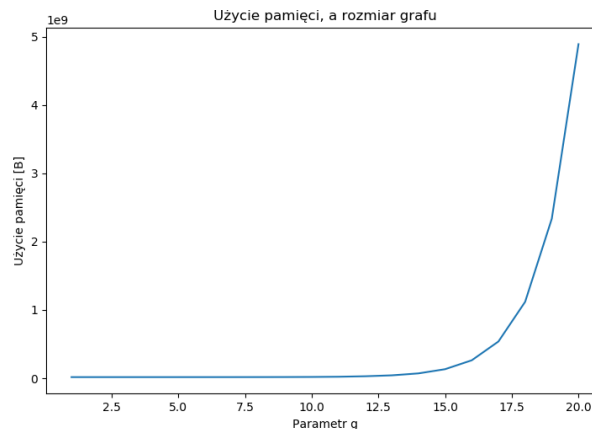
Do implementacji dołączony został też program `rs` uruchamiany z wiersza poleceń umożliwiający testowanie działania funkcji **RiffleScrambler** w łatwy sposób. Aby wyświetlić instrukcje, jak używać tego programu należy wykonać polecenie `./rs -h` lub `./rs --help`.

```
$ ./rs --help
Password hashing memory-hard function
Usage:
```

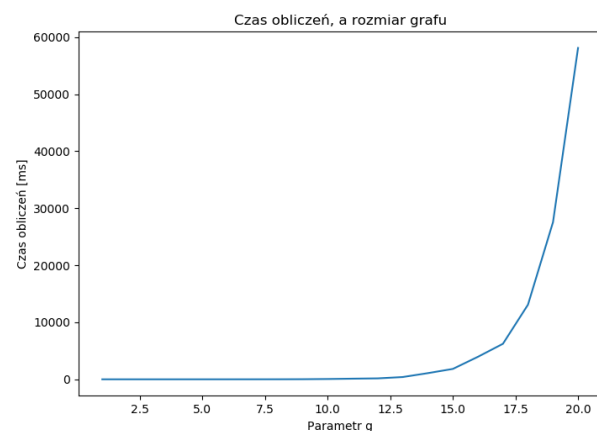




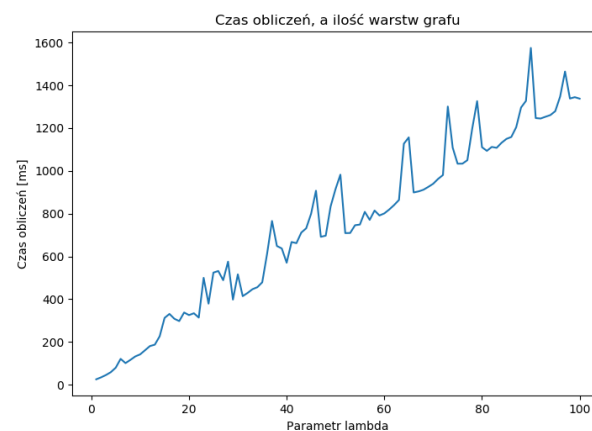
(a) Wykresy liczby odczytów spoza pamięci podręcznej.



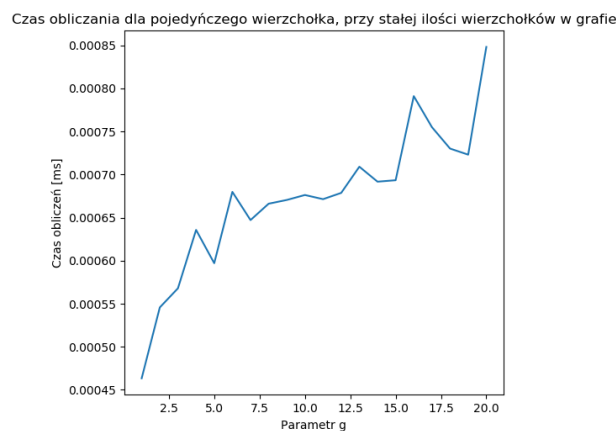
(b) Wykresy zużycia pamięci przez **RiffleScrambler** w zależności od parametru  $g$ .



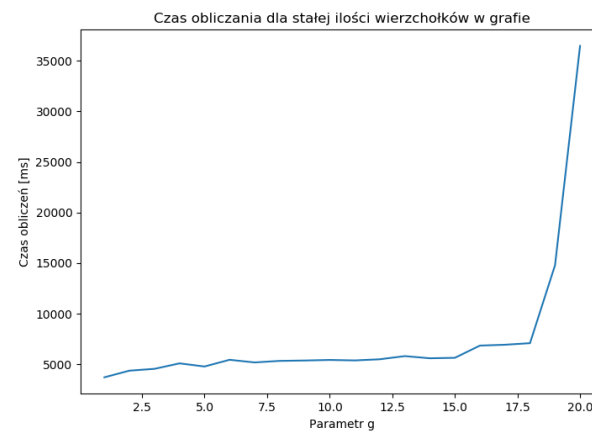
(c) Wykres czasu obliczania **RiffleScrambler** w zależności od parametru  $g$ .



(d) Wykres czasu obliczania **RiffleScrambler** w zależności od parametru  $\lambda$ .



(e) Wykres czasu obliczania **RiffleScrambler** w zależności od parametru  $g$  dla stałej liczby wierzchołków w grafie.



(f) Średni czas obliczania jednego wierzchołka w zależności od parametru  $g$  dla stałej liczby wierzchołków w grafie.



RiffleScrambler Password is read from stdin [OPTION...] [optional args]

```
-s, --salt arg      Salt for the given password
-w, --width arg     Width of the graph (default: 12)
-d, --depth arg     Number of stacks of the graph (default: 2)
-f, --func arg      Internal hash function (default: sha256)
-h, --help          Print help
```

Program przyjmuje parametry do uruchomienia funkcji **RiffleScrambler** w argumentach wywołania, natomiast hasło przyjmuje na standardowe wejście, ponieważ większość wierszy poleceń zapisuje historię wywołań programów i hasło podane jako argument wywołania mogłoby zostać zapisane w postaci jawnej.

Przykład użycia programu do obliczenia funkcji dla hasła `password`, soli `somesalt`, parametru `g 16`, parametru `λ 2` używając `sha256` jako funkcji wewnętrznej.

```
$ echo -n "password" | ./rs somesalt -w 16 -d 2 -f sha224
Graph width:      16
Graph depth:      2
Hash:             166c88a5edfba7228fbf49a4bcd4cb2f01cc5fcdc850bf78c317a5c8
Encoded:          $g=16$d=2$s=c29tZXNhbHQ=$f=sha224$h=MTY2Yzg4YTVlZGZiYTcyMjhmYmY0(...)
```

## 4.7 Przykład użycia

Wysokopoziomowy interfejs umożliwia uwierzytelnianie w wygodny sposób. Zaprezentowany zostanie przykład użycia biblioteki `riffle` w aplikacji internetowej podłączonej do bazy danych SQL. Aplikacja ta pozwala na zakładanie kont dla nowych użytkowników, oraz po zalogowaniu się na konto przez użytkownika, udostępnia użytkownikowi pewne zasoby. Biblioteka `riffle` zostanie wykorzystana przy tworzeniu nowego konta, w celu obliczenia wartości funkcji **RiffleScrambler** dla hasła użytkownika, którą można bezpiecznie trzymać w bazie danych oraz później podczas uwierzytelniania w celu weryfikacji poprawności hasła.

Tabela użytkowników w bazie danych została utworzona następująco.

```
1 CREATE TABLE Users (
2   UserID int,
3   Login varchar(255),
4   HahsEncoded varchar(255),
5   Name varchar(255),
6 );

// importowanie interfejsu z biblioteki
#include <riffle/riffle_scrambler.h>

/**
 * Funkcja zapisująca użytkownika w bazie danych
 * @param login Login użytkownika
 * @param password Hasło użytkownika (w postaci jawnej)
 * @param name Imie użytkownika
 * @param address Adres użytkownika
 */
void create_account(const std::string &login, const std::string &password,
                  const std::string &name, const std::string &address) {

    // stworzenie połączenia do bazy danych SQL
    const auto database_adapter = get_sql_database_adapter();
```

```
// generowana jest losowa sól
const std::string salt = generate_random_salt();
// hasło użytkownika jest hashowane dla podanych parametrów oraz soli
const std::string hash_encoded = riffle_scrambler_encoded(12, 4, password, salt);

// jeden, bezpieczny do przechowywania
// zawierający informacje o parametach, soli oraz hashu tekst
database_adapter.add_user(login, hash_encoded, name, address);
}

/**
 * Funkcja do autoryzacji użytkownika na podstawie podanego hasła
 * sprawdza, czy podane hasło jest takie samo, jak hasło podane przy tworzeniu konta,
 * które zapisane jest w bazie danych
 * @param login Login użytkownika
 * @param password Hasło podane przez użytkownika
 * @return true, jeśli podane przez użytkownika hasło jest poprawne,
 * false w przeciwnym przypadku
 */
bool is_password_valid(const std::string &login, const std::string &password) {
    // stworzenie połączenia do bazy danych SQL
    const auto database_adapter = get_sql_database_adapter();

    // pobranie wiersza z danymi użytkownika z bazy danych
    const auto user = database_adapter.get_user_by_login(login);

    // sprawdzenie czy wynik funkcji riffle_scrambler dla podanego przez użytkownika hasła
    // zgadza się z poprawnym hasłem dla podanego loginu
    return riffle_scrambler_verify(user.hash_encoded, password);
}
```

To, co zasługuje na docenienie podczas używania biblioteki `riffle`, to konieczność poświęcenia tylko jednej kolumny w tabeli, aby uzyskać możliwość uwierzytelniania użytkownika za pomocą hasła. Nie trzeba przechowywać soli, ani parametrów funkcji, ponieważ zawarte są one w zakodowanej wartości. Oznacza to, że można zmienić dowolne parametry funkcji używane podczas tworzenia nowych kont oraz uwierzytelniać użytkowników z hasłami zapisanymi wcześniej bez konieczności rozróżniania z jakimi parametrami jaki użytkownik zakładał konto.

## 4.8 Testy

Do implementacji zostały dołączone testy jednostkowe pisane przy wykorzystaniu biblioteki Catch2 [1].

Testowane jest zewnętrzny interfejs oraz funkcje wewnętrzne.

Przykład uruchomienia testów.

```
$ ./catch_test
```

---

```
All tests passed (740 assertions in 23 test cases)
```



# Instalacja i wdrożenie

Wymagania - cmake - make - openssl - kompilator

## 5.1 Biblioteka

Kody biblioteki `riffle` znajdują się w katalogu `RiffleScrambler` zawierającym plik `CMakeLists.txt`. W celu skompilowania biblioteki należy uruchomić polecenie `$ cmake CMakeLists.txt` w katalogu `RiffleScrambler`, które wygeneruje plik `Makefile`. Następnie należy wykonać polecenie `$ make`, które skompiluje kod biblioteki.

## 5.2 Testy

## 5.3 Pomiary wydajności

## 5.4 RiffleScrambler z wiersza poleceń



# Podsumowanie

W części teoretycznej niniejszej pracy udało się udowodnić dolne i górne ograniczenie na złożoność etykietowania grafu, ograniczenie dolne wynosi  $\Omega(n^{1.5})$  i jest równe ograniczeniu górnemu Catena Dragonfly oraz Butterfly. Ograniczenie górne wynosi  $O(n^{1.667})$ , czyli jest wyższe, niż ograniczenie dla algorytmu Catena wynoszące  $O(n^{1.625})$ . Jednak wnioskując po podobieństwie tych grafów, bardzo prawdopodobne jest to, że istnieje lepsze ograniczenie górne, niż pokazane. Dlatego jest to możliwy kierunek rozwinięcia pracy.

W części praktycznej udało się zrealizować założenie o zachowaniu wymagań konkursu na funkcje do przechowywania haseł, ponadto zaproponowano nowy, wydajniejszy algorytm do generowania permutacji, który sprawdza poprawność permutacji w czasie liniowym, zamiast kwadratowego oraz cechuje się niższą złożonością pamięciową. Stworzono również wysokopoziomowy interfejs, który jest dużą zaletą patrząc na aplikacje uwierzytelniające pisane w językach wysokopoziomowych.

Wydażność czasowa jest na dobrym poziomie, jednak wydażność pamięciowa mogłaby zostać poprawiona poprzez zmianę generowania struktury grafu, co okazało się podczas analizy pamięciowej gotowej implementacji. Zamiast generować od razu cały graf, który zajmuje stosunkowo dużo pamięci, można by trzymać jedynie obliczone obecnie używane krawędzie, a kolejne generować dopiero, gdy zajdzie potrzeba, co zmniejszyło by użycie pamięci.





# Bibliografia

- [1] Catch2 - a modern, c++-native, header-only, test framework for unit-tests. <https://github.com/catchorg/Catch2>.
- [2] Openssl - cryptography and ssl/tls toolkit. <https://www.openssl.org>.
- [3] Openssl evp - a high level interface to openssl cryptographic functions. <https://wiki.openssl.org/index.php/EVP>.
- [4] The password hashing competition (phc). <https://password-hashing.net/cfh.html>.
- [5] Radix sort - a non-comparative stable algorithm. [https://en.wikipedia.org/wiki/Radix'sort](https://en.wikipedia.org/wiki/Radix%27sort).
- [6] J. Alwen, J. Blocki. Efficiently computing data-independent memory-hard functions. *Annual Cryptology Conference*, strony 241–271. Springer, 2016.
- [7] J. Alwen, J. Blocki, K. Pietrzak. Depth-robust graphs and their cumulative memory complexity. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, strony 3–32. Springer, 2017.
- [8] K. Gotfryd, P. Lorek, F. Zagórski. Rifflescrambler—a memory-hard password storing function. *European Symposium on Research in Computer Security*, strony 309–328. Springer, 2018.
- [9] Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, Gru. 2017.
- [10] S. Josefsson. The base16, base32, and base64 data encodings. Raport instytutowy, 2006.
- [11] A. Peslyak. Yescrypt-a password hashing competition submission. *Password Hashing Competition. v0 edn.(March 2014)* <https://password-hashing.net/submissions/specs/yescrypt-v0.pdf>, 2014.
- [12] T. Pornin. The makwa password hashing function, 2015.
- [13] M. A. Simplicio Jr, L. C. Almeida, E. R. Andrade, P. C. dos Santos, P. S. Barreto. Lyra2: Password hashing scheme with improved security against time-memory trade-offs. *IACR Cryptology ePrint Archive*, 2015:136, 2015.



# Zawartość płyty CD

W tym rozdziale należy krótko omówić zawartość dołączonej płyty CD.

