Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Memory-hard Functions

Konrad Świerczyński
Nr indeksu: 229818

Promotor
dr Filip Zagórski

Politechnika
Wrocławska
Wrocław 2019

Spis treści

Wstęp

Niniejsza praca swoim zakresem obejmuje kryptograficzne funkcje do przechowywania haseł typu *memory-hard* zapewniającą wysokie bezpieczeństwo przechowywania haseł. Celem pracy jest analiza bezpieczeństwa oraz implementacja funkcji tego typu, jaką jest RiffleScrambler [12]. Praca przedstawia także definicje opisu jakości takich funkcji oraz przedstawia porównanie RiffleScrambler do obecnych rozwiązań.

W rozdziale 2 przedstawiono motywację, formalne definicje oraz twierdzenia potrzebne po przeprowadzenia analizy bezpieczeństwa oraz porównywania funkcji *memory-hard*. Rozdział 3 przedstawia opis algorytmów, analizę bezpieczeństwa i porównanie do istniejących rozwiązań. Rozdział 4 opisuje wykonane wykonane prace implementacyjne oraz dokumentację interfejsu, a w rozdziale 5 znajduje się podsumowanie ukończonych prac oraz uzyskanych wyników.

Analiza problemu

2.1 Wprowadzenie

Początkowo, w latach 60. XX wieku, hasła przechowywane były w postaci jawnej. W latach siedemdziesiątych zaczęto stosować funkcję `crypt`, początkowo opartą na symulacji mechanicznej maszyny szyfrującej M-209, a w kolejnych wersjach opartą na szyfrowaniu DES oraz opartą na funkcji jednokierunkowej. Metoda z szyfrowanie pozwalała odszyfrować hasła osobie znającej klucz, a oby dwie metody pozwalały znaleźć użytkowników z takim samym hasłem, co umożliwiało tworzenie bazy haseł i odpowiadającym im wynikom. Wprowadzono więc parametr nazwany solą (ang. *salt*), który był losowo wybierany dla każdego hasła, a więc z dużym prawdopodobieństwem takie same hasła były przechowywane jako inne wartości. Ponieważ hasła nie są idealnie losowe, wystarczy obliczać funkcje tylko dla najczęściej używanych ciągów znaków, zamiast tak jak w ataku *brute force* dla wszystkich. Gdy adwersarz zdobył plik z przechowywanym hashami, mógł dla każdego hasha próbować znaleźć odpowiadające mu hasło obliczając funkcję dla wszystkich pozycji w słowniku najczęściej występujących haseł.

Aby zapobiec tego rodzaju atakom, zaczęto stosować funkcje hashujące na tyle ciężkie do obliczenia, aby atak słownikowy był jak najbardziej niepraktyczny. Ponieważ funkcja hashująca musi być obliczana podczas każdego uwierzytelniania w celu sprawdzenia poprawności hasła, nie może być ona zbyt ciężka do obliczenia dla aplikacji uwierzytelniającej. Z drugiej strony, gdy krotka (*login*, *hash*, *salt*) wycieknie, adwersarz przeprowadzający atak słownikowy obliczając funkcję hashującą dla każdej wartości ze słownik, co trzeba uczynić jak najbardziej kosztowne. Dobra funkcja do przechowywania haseł powinna być tak samo kosztowna do obliczenia dla aplikacji uwierzytelniającej jak dla adwersarza.

W tym celu zaczęto stosować funkcje, które obliczają wiele razy kryptograficzną funkcję skrótu. Przykładem takiej funkcji jest PBKDF2 [15] (ang. *Pssowrd-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.

Takie podejście nie gwarantuje zabezpieczenia przed adwersarzem używającym sepcjalizowany układ scalony (ang. *ASIC - Application-Specyfic Integrated Circut*). Układy takie są znacznie bardziej wydajne pod względem szybkości obliczania funkcji skrótu takich jak SHA256 czy MD5, niż tradycyjne architektury, a więc zwiększenie bezpieczeństwa za pomocą parametrów obciąży aplikację uwierzytelniającą w dużo większym stopniu, niż adwersarza. Dla przykładu Antminer S9 [**?**] potrafi w jednej sekundzie obliczyć 1.4×10^{13} wyników (dla funkcji double SHA256), podczas gdy obecnie dostępne karty graficzne są w stanie liczyć taką funkcję z prędkością 3×10^{10} wyników na sekundę, a procesory z prędkością około 10^9 wyników na sekundę [12].

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

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, ang. *directed acyclic graph*).

Funkcje takie możemy podzielić na funkcje z dostępem do pamięci zależnym od danych dMHF(ang. *data-dependent MHF*) oraz z dostępem do pamięci niezależnym od danych iMHF(ang. *data-independent MHF*)). Ponieważ dostęp do pamięci oparty na podanym haśle może pozostawiać sposobność przeprowadzenia ataków, na przykład opartych na kolejności dostępu do pamięci, w pracy omówione są jedynie funkcje typu iMHF.

Istnieje już wiele funkcji typu *memory-hard*. Jednymi z najbardziej popularnych, a co za tym idzie, naj-

lepiej zbadanych są Argon2 [9], Catena [11] oraz Balloon Hashing [10].

2.2 Notacja

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 $b \leqslant c$.

Skierowany graf acykliczny (ang. *directed acyclic graph*, DAG) $G = (V,E)$ jest rozmiaru n jeżeli $|V| = n$. Wierzchołek $v \in V$ ma stopień wchodzący δ równy największemu stopniu wchodzącemu wśród jego wierzchołków $\delta = \mathbf{indeg}(v)$, jeżeli istnieje δ wchodzących krawędzi $\delta = |(V \times v) \cap E|$. Graf G ma stopień wchodzący $\delta = \mathbf{indeg}(G) = \max_{v \in V} \mathbf{indeg}(v)$. 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 $\mathbf{parents}_G(v) = \{u \in V : (u,v) \in E\}$. Uogólniając, zbiór przodków v oznaczany jest jako $\mathbf{ancestors}_G(v) = \bigcup_{i \geqslant 1} \mathbf{parents}^i_G(v)$, przyjmując $\mathbf{parents}^{i+1}_G(v) = \mathbf{parents}_G(\mathbf{parents}^i_G(v)))$. Jeżeli wybór grafu G wynika z kontekstu, będziemy oznaczać te zbiory jako **parents** oraz **ancestors**.

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

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

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.

2.3 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).*
$$\forall x \in T \quad \exists x \leqslant t \ : \ x \in P_x$$
2. *Oetykietować wierzchołek można tylko wtedy, gdy wszyscy jego rodzice są oetykietowani w poprzednim kroku.*
$$\forall i \in [t] \ : \ x \in (P_i \setminus P_{i-1}) \ \Rightarrow \ \mathbf{parents}(x) \subset P_{i-1}$$
3. *W każdym kroku można oetykietować co najwyżej jeden wierzchołek.*
$$\forall i \in [t] \ : \ |P_i \setminus P_{i-1}| \leqslant 1$$

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

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

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

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

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

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

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

Kiedy $T = \mathbf{sinks}(G)$, piszemy $\Pi^{\|}_\alpha(G)$ oraz $\Pi_\alpha(G)$.

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

W tej pracy głównie rozważane jest badanie złożoności Π_{st}, oraz $\Pi^{\|}_{cc}$, 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

$$\forall G_n \in \mathbb{G}_n \ : \ \Pi^{\|}_{cc}(G_n) \leqslant \Pi_{st}(G_n) \leqslant n^2.$$

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

$$n(n-1)/2 \leqslant \Pi^{\|}_{cc}(K_n) \leqslant \Pi_{st}(K_n) \leqslant n^2$$

Graf K_n maksymalizuje złożoności etykietowania, a $\Pi^{\|}_{cc}$ jest rożne tylko o stałą od Π_{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

$$\Pi^{\|}_{cc}(Q_n) = \Pi_{st}(Q_n) = n.$$

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.

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

Lemat 2.1 (Ograniczenie dolne dla (N, λ)-Superkoncentratora) *[12, Lemat 1] Etykietowanie (N, λ)-Superkoncentratora używając $S \leqslant N/20$ etykiet, wymaga T kroków, gdzie*

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

2.5 Grafy depth-robust

Alwen i Blocki w swojej pracy [8] pokazali, że istnieje zależność między złożonością etykietowania, a własnością depth-robustness. Na podstawie tej własności 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*

$$\forall S \subset V \ |S| \leqslant e \Rightarrow \mathbf{depth}(G - S) \geqslant d.$$

Twierdzenie 2.1 *[8, 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-roubust to nazywany jest (e,d)-reducible.*

Twierdzenie 2.2 *[8, 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 $g = \sqrt{\frac{dn}{g}}$ upraszcza się to do $\Pi_{cc}^{\parallel}(G) = O\left(n(\sqrt{dn\delta} + e)\right)$.

2.6 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 $p_1, \ldots, p_z$ kończące się w L, gdzie każda jest długości co najmniej g.*

Definicja 2.8 (Graf rozproszony, ang. *dispersed*) *Niech $g, k \in \mathbb{N}$ i $g \geqslant k$. DAG G jest nazywany (g, k)-dispersed 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 $\forall j \in [\lfloor k/g \rfloor]$ 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 (ϵ, g, g)-dispersed.*

Definicja 2.9 *Acykliczny graf skierowany $G = (V, E)$ nazywany jest (λ,ϵ,g,k)-dispersed 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 (ϵ,g,k)-dispersed.*

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

Twierdzenie 2.3 *[8, Twierdzenie 6] Niech $G \in \mathbb{D}_{\epsilon,g}^{\lambda,k}$.*

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

2.7 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 $(v_1, \ldots, v_n)$ oraz dzieląc go na rzędy $L_j = \{v_{jk+1}, \ldots, v_{jk+k}\}$, dla $j \in \{0, \ldots, N\}$, pozostałe krawędzie łączą wierzchołki z niższego rzędu L_j jedynie z wierzchołkami z wyższych rzędów L_i, $i \in \{j + 1, \ldots, N\}$.*

Lemat 2.2 *[7, Lemat 4.2] Niech G będzie N rzędowym grafem, wtedy dla dowolnego $t \in \mathbb{N}^+$, G jest $(n/t, Nt + t - N - 1)$-reducible.*

Riffle Scrambler

RiffleScrambler [12] 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 Balloon Hashing. Oznacza, to, że dla każda sól odpowiada (z dużym prawdopodobieństwem) innemu grafowi, co zwiększa odporność na ataki równoległe. Dla Cateny są tylko dwa predefiniowane grayf *bit-reversal* i *double-butterfly*. Jednocześnie RiffleScrambler zapewnia lepszą wydajność przy obliczaniu niż Balloon 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 $V = V_0 \cup V_1 \cup \cdots \cup V_{2\lambda g}$, gdzie $|V_i| = 2^g$,
- λ - 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 pamiętanie 2^g komórek, gdzie każda przechowuje wynik kryptograficznej funkcji skrótu. Wartość 2^g będziemy oznaczać jako N. Parametr λ 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^λ. 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 $\overline{x}$ oznacza negację wyrazu x, zatem $HW(\overline{x})$ oznacza liczbę zer w wyrazie x.

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

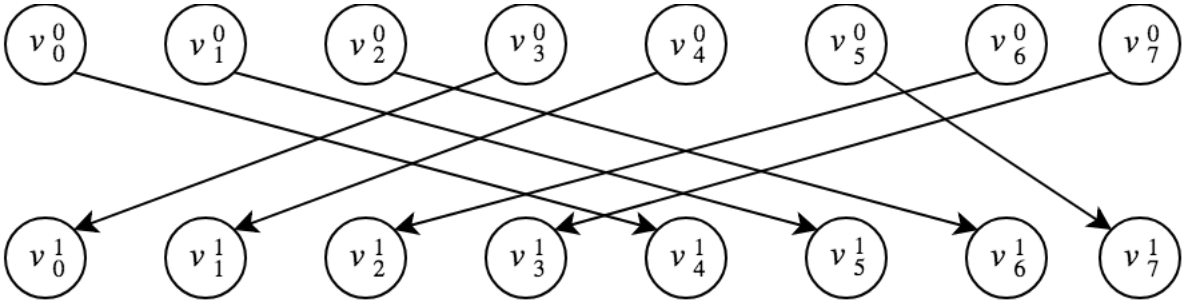
$$r_B(i) = |\{j < i : b_j = b_i\}|.$$

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

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

dla każdego $0 \leqslant i \leqslant n-1$.

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-Premutation indukowaną przez* B $\pi_B = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 0 & 1 & 7 & 2 & 3 \end{smallmatrix}\right)$. *Ilustracja tego przykładu widoczna jest na rysunku 3.1.*

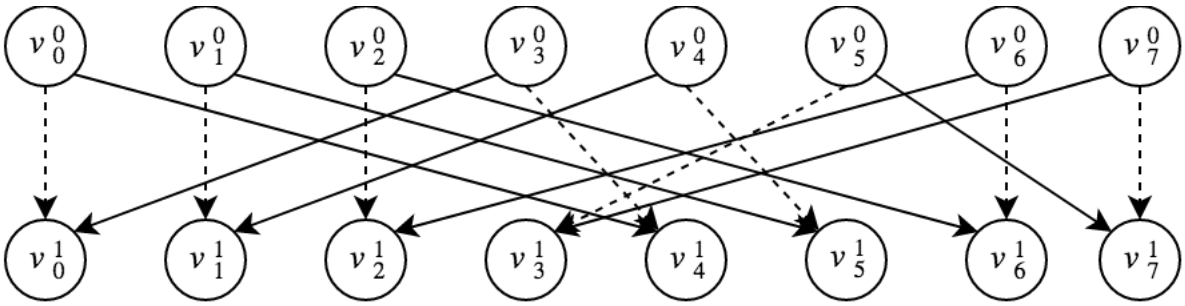


Rysunek 3.1: Graf utworzony z Riffle-Permutation indukowanej przez $B = 11100100$.

Definicja 3.3 (N-Single-Layer-Riffle-Graph) *Niech* $V = V^0 \cup V^1$, *gdzie* $V^i = \{v_0^i, \ldots, v_{N-1}^i\}$, *niech* B *będzie słowem binarnym długości* N. *Niech* π_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:* $v_{N-1}^0 \to v_0^1$,
- $2(N-1)$ *krawędzi:* $v_{i-1}^j \to v_i^j$, *dla* $i \in [N-1]$ *oraz* $j \in \{0,1\}$,
- N *krawędzi:* $v_i^0 \to v_{\pi_B(i)}^1$, *dla* $i \in \{0, \ldots, N-1\}$,
- N *krawędzi:* $v_i^0 \to v_{\pi_{\overline{B}}(i)}^1$, *dla* $i \in \{0, \ldots, N-1\}$.

Przykład 3.2 *Kontynuując z danymi z przykładu 3.1,* $\pi_{\overline{B}} = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 4 & 5 & 3 & 6 & 7 \end{smallmatrix}\right)$. *8-Single-Layer-Riffle-Graph ukazany jest na rysunku 3.2.*



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 π_B oznaczone są linią ciągłą, a krawędzie dla permutacji $\pi_{\overline{B}}$ 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* $B_0, \ldots, B_{g-1}$ *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* $\{v_0^0, \ldots, v_{2^g-1}^0\} \cup \cdots \cup \{v_0^{2g}, \ldots, v_{2^g-1}^{2g}\}$, *oraz następujące krawędzie:*

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

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

- $g2^g$ *krawędzi:* $v_i^{2g-j} \to v_{\pi_{B_j}^{-1}(i)}^{2g-j+1}$, *dla* $i \in \{0,\ldots,2^g-1\}$ i $j \in [g]$,
- $g2^g$ *krawędzi:* $v_i^{2g-j} \to v_{\pi_{\overline{B}_j}^{-1}(i)}^{2g-j+1}$, *dla* $i \in \{0,\ldots,2^g-1\}$ i $j \in [g]$.

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

3.1.3 Śledzenie trajektorii

Graf jest generowany za pomocą permutacji pseudolosowej σ. 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 $\mathfrak{B} = (\mathfrak{B}_0,\ldots,\mathfrak{B}_{g-1})$ 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, ale 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 jet 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 [12].

3.2.2 Algorytm generowania permutacji

Razem z prezentacją RiffleScrambler zaproponowany został algorytm generowania permutacji [12, Algorym 1], który sprawdzał warunek końca w czasie $O(n^2)$ oraz potrzebował $O(n^2)$ komórek pamięci, gdzie n oznacza wielkość permutacji.

Pseudokod 3.1 przedstawia algorytm sprawdzający warunek końca w czasie $O(n)$ korzystając z $O(nlogn)$ komórek pamięci.

Algorytm opiera się na sortowaniu pozycyjnym radix sort [6], 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, λ) można z grubsza przedstawić następująco.

- Dla podanej soli s obliczana jest pseudolosowa permutacja σ (używając algorytmu **InverseRiffleShuffle**).
- Dla permutacji σ tworzona jest macierz $\mathfrak{B} =$ **TraceTrajectories**(B).
- Dla wyrazów binarnych $\mathfrak{B}_0, \ldots, \mathfrak{B}_{g-1}$ generowany jest graf $G =$ **GenGraph**(g, σ) (pseudokod 3.2), który jest N-Double-Riffle-Graph. Przypomnijmy, $N = 2^g$.
- Na grafie G zainicjalizowanym wartością pwd, oblicze są wartości w ostatnim rzędzie $(v_0^{2g+1}, \ldots, v_{2^g-1}^{2g+1})$.
- Wartości z ostatniego rzędu przepisywane są do pierwszego, $v_i^0 = v_i^{2g+1}$ dla $i \in \{0, \ldots, 2^g - 1\}$, a następnie znów oblicza się wartość ostatnich rzędów. Powtarzane jest λ razy.
- Wartością końcową jest wartość w ostatnim wierzchołku, czyli w $v_{2^g-1}^{2g}$.

Procedura **RiffleShuffle** przedstawiona jest w pseudokodzie 3.3.

Przykład 3.3 (Generowanie (8,1)-Double-Riffle-Graph) *Niech $g = 3$, $N = 2^g = 8$ i $\lambda = 1$. Niech otrzymaną permutacją σ będzie $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 4 & 6 & 3 & 2 & 7 & 0 & 1 \end{pmatrix}$. Binarną 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 $\pi_{\mathfrak{B}_0}, \pi_{\mathfrak{B}_1}, \pi_{\mathfrak{B}_2}$:

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

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

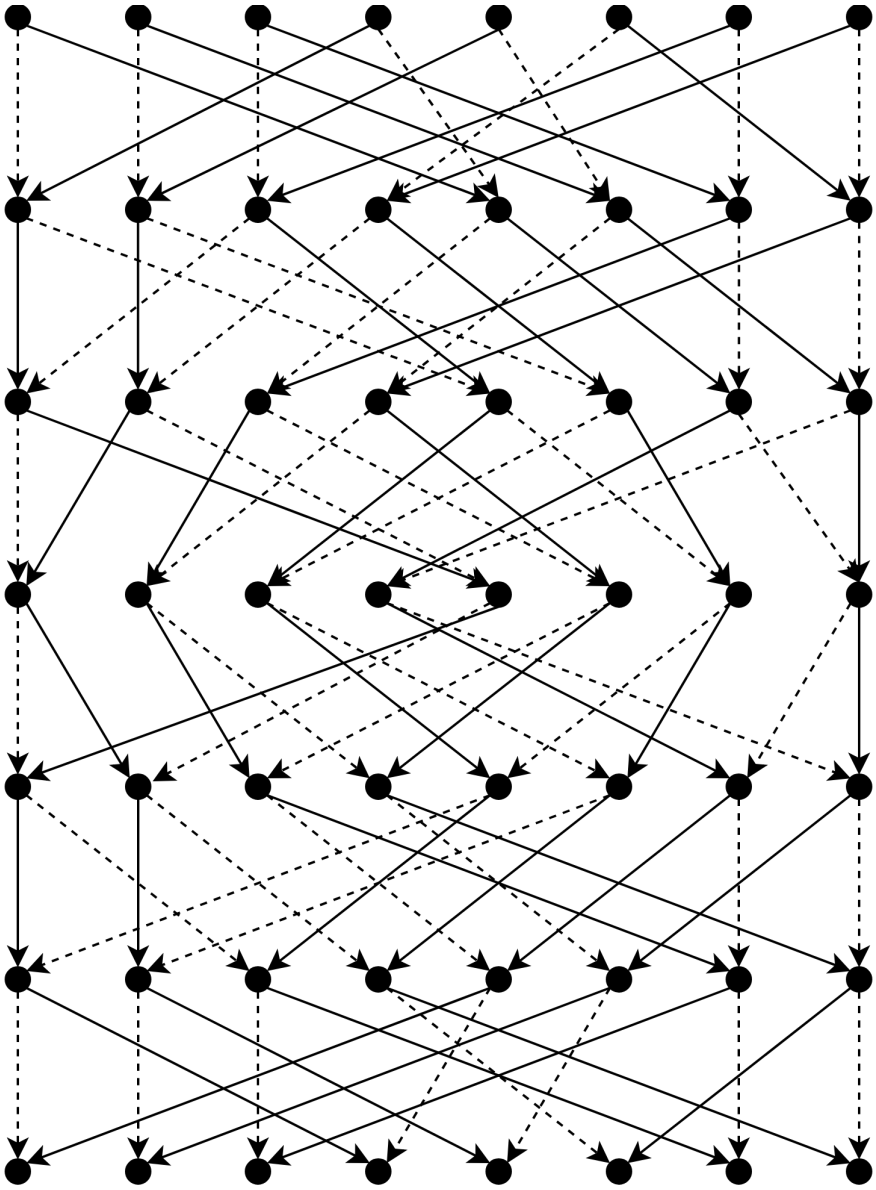
$$\pi_{\mathfrak{B}_2} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & 1 & 5 & 2 & 6 & 3 & 7 \end{pmatrix}.$$

Wygenerowany graf z krawędziami zależnymi od permutacji ukazany jest na rysunku 3.3. Dodając do niego krawędzie niezależne od permutacji, otrzymujemy pełny graf (8,1)-Double-Riffle-Graph pokazany na rysunku 3.4.

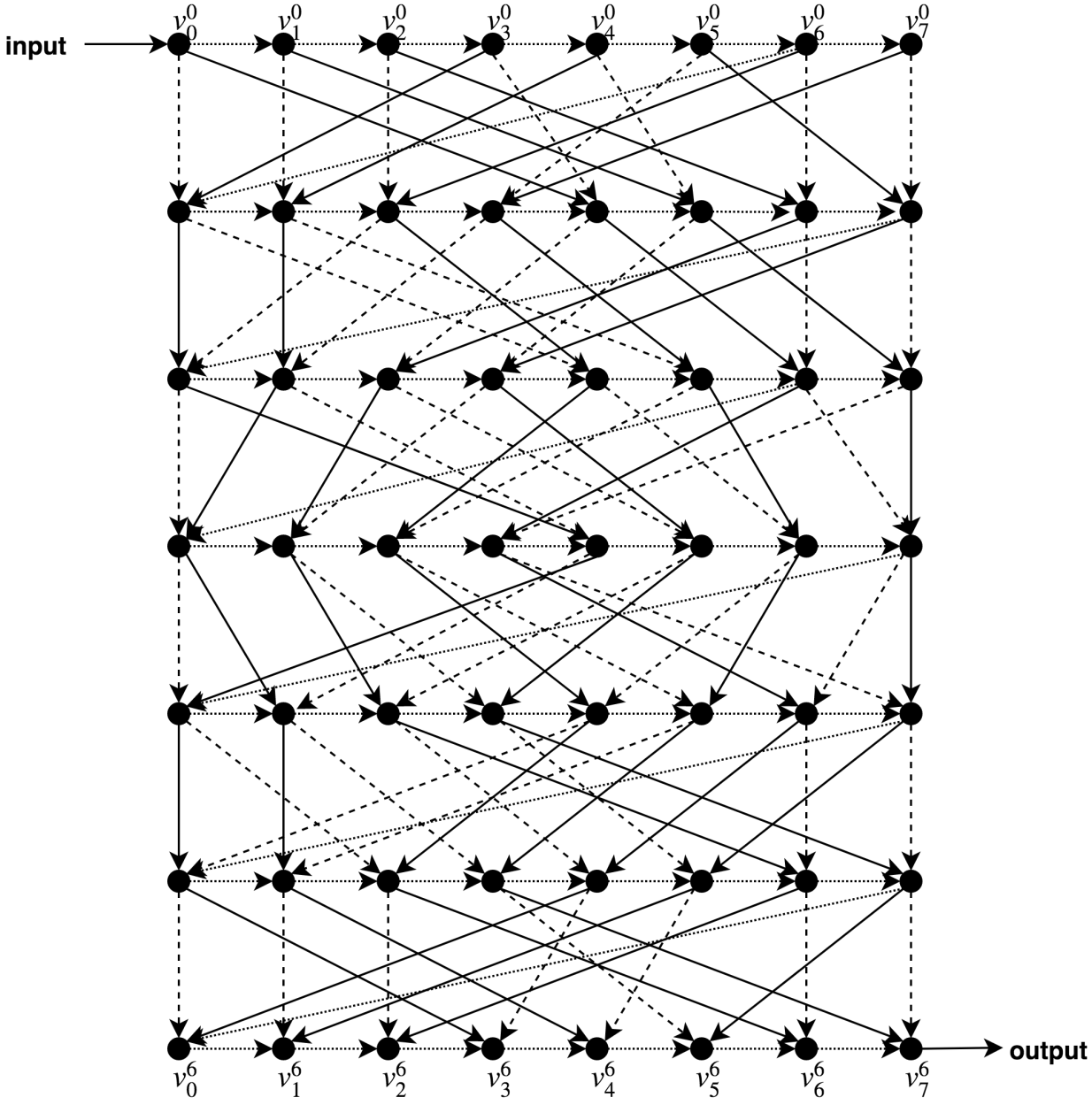
3.4 Właściwości

Twierdzenie 3.1 *[12, Twierdzenie 5] Niech σ będzie permutacją $N = 2^g$ elementów. Niech G będzie N-Double-Riffle-Graph wygenerowanym dla $\mathfrak{B} =$ **TraceTrajectories**(σ). Wtedy G jest $N-$Superkoncentratorem.*

Twierdzenie 3.2 *RSG_λ^N jest N rzędowym grafem.*



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 - wielkość premutacji do wygenerowania, getNextBit - generator bitów losowych
Output: σ - permtacja wielkości N

/* *perm* zawiera krotki będące początkowym indeksem oraz wyrazem binarnym. $perm[i] = (id, bin)$ dla $i \in \{0,\ldots,N-1\}$ */

$perm = ((0,\text{“”}),(1,\text{“”}),\ldots,(N-1,\text{“”}))$;
while $\exists_{i\in\{0,\ldots,N-2\}} perm[i][1] == perm[i+1][1]$ **do**
  $numberOfOnes = 0$;
  for $i = 0$ **to** $N-1$ **do**
    $newBit = bitGenerator()$;
    $perm[i][1] = perm[i][1] + newBit$;
    if $newBit ==$ “1“ **then**
      $numberOfOnes = numberOfOnes + 1$;
    end if
  end for
  $lastIndexOfOnes = 0$;
  $lastIndexOfZeros = numOfOnes$;
  for $i = 0$ **to** $N-1$ **do**
    if $lastBit(permutation[i][1]) ==$ “1“ **then**
      $tmp[lastIndexOfOnes] = permutation[i]$;
      $lastIndexOfOnes = lastIndexOfOnes + 1$;
    else
      $tmp[lastIndexOfZeros] = permutation[i]$;
      $lastIndexOfZeros = lastIndexOfZeros + 1$;
    end if
  end for
  $perm = tmp$;
end while
$\sigma = (perm[0].id, perm[1].id, \ldots, perm[N-1].id)$;

Algorithm 3.2: GenGraph

Input: g - parametr wielkości grafu, σ - permutacja wielkości 2^g
Output: G - graf na 2^g wierzchołkach z krawędziami wygenerowanymi na podstawie permutacji

$N = 2^g$;
$V = \{v_i^j : i \in \{0,\ldots,N-1\}; j \in \{0,\ldots,2g\}\}$;
$E = \{v_i^j \to v_{i+1}^j : i \in \{0,\ldots,N-2\}; j \in \{0,\ldots,2g\}\}$;
$E = E \cup \{v_{N-1}^j \to v_0^{j+1} : j \in \{0,\ldots,2g-1\}\}$;
$\mathfrak{B} = (\mathfrak{B}_0,\ldots,\mathfrak{B}_{g-1}) =$ **TraceTrajectories**(σ) ;
for $m = 0$ **to** $g-2$ **do**
  $\mathfrak{B}_{2g+1-m} = \mathfrak{B}_m$;
end for
/* Bity w słowach binarnych oznaczamy dodając indeks dolny. $\mathfrak{B}_j = \mathfrak{B}_{j,0}\mathfrak{B}_{j,1}\ldots\mathfrak{B}_{j,2^g-1}$ */
for $j = 0$ **to** $2g-1$ **do**
  for $i = 0$ **to** $2^g - 1$ **do**
    $E = E \cup \{v_i^j \to v_{\pi_{\mathfrak{B}_{j,i}}}^{i+1}\} \cup \{v_i^j \to v_{\pi_{\overline{\mathfrak{B}}_{j,i}}}^{i+1}\}$;
  end for
end for
$G = (V,E)$;

Algorithm 3.3: RiffleScrambler

Input: s - sól, g - parametr wielkości grafu, pwd - hasło, λ - ilość warstw w grafie, H - kryptograficzna funckja sktóru
Output: pwd - hash podanego hasła

$\sigma = \textbf{RiffleShuffle}(2^g, pseudoRandomBitGenerator(s,H))$;
$G = (V,E) = \textbf{GenGraph}(g, \sigma)$;
$v_0^0 = H(X)$;
for $i = 1$ **to** $2^g - 1$ **do**
  $v_i^0 = H(v_{i-1}^0)$;
end for
for $r = 1$ **to** λ **do**
  for $j = 0$ **to** $2g$ **do**
    for $i = 0$ **to** $2^g - 1$ **do**
      $v_i^{j+1} = 0$;
      forall $v \to v_i^{j+1} \in E$ **do**
        $v_i^{j+1} = H(v_i^{j+1}, v)$;
      end forall
    end for
  end for
  for $i = 0$ **to** $2^g - 1$ **do**
    $v_i^0 = v_i^{2g+1}$;
  end for
end for
$hash = v_{2^g-1}^{2g}$;

Dowód. Z definicji RSG_λ^N jest (N,λ)-Double-Riffle-Graph dla $N = 2^g$, a liczba wszystkich wierzchołków wynosi $n = (2g+1)2^g$. Z definicji 3.4 zawiera on ścieżkę przechodzącą przez n wierzchołków $(v_1, \ldots, v_n)$ oraz można rozłożyć graf na $N+1$ rzędów $V = V^0 \cup \ldots V^{2g}$, a krawędzie łączą jedynie wierzchołki w jednym rzędzie, lub łączą wierzchołki z rzędu V^i z wierzchołkami z rzędu V^{i+1} dla $i \in \{0, \ldots, 2g-1\}$. Zatem graf jest N rzędowy. □

3.5 Ograniczenia złożoności RSG

Twierdzenie 3.3 *Niech $\lambda, n \in \mathbb{N}^+$ takie, że $n = \overline{n}(2\lambda c + 1)$, gdzie $c \in \mathbb{N}$ i $\overline{n} = 2^c$. Wtedy dla $g = \lfloor \sqrt{\overline{n}} \rfloor$ $RSG_\lambda^{\overline{n}} \in \mathbb{D}_{1,g}^{\lambda,\overline{n}}$ oraz $\Pi_{cc}^{\|}(RSG_\lambda^{\overline{n}}) = \Omega\left(\frac{n^{1.5}}{c\sqrt{c\lambda}}\right)$.*

Dowód. Niech $G = RSG_\lambda^{\overline{n}}$, niech $G_1, G_2, \ldots, G_\lambda$ będą podgrafami G opisanymi w definicji 3.5. Pokażemy, że każdy G_i jest (g, $\overline{n}$)-dispresed dla $g = \lfloor \sqrt{\overline{n}} \rfloor$.

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

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

Z twierdzenia 3.1 mówiącym, że RSG jest N-Superkoncentratorem 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 zbiory S są oddzielone między sobą o $g-2c$ wierzchołków. Możemy dodać na początek ścieżki p ścieżkę $(<0, v-(g-2c-1)>, <0, v-(g-2c-2)>, \ldots, <0, v>)$. 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 $RSG_\lambda^{\overline{n}} \in \mathbb{D}_{1,g}^{\lambda,\overline{n}}$. Pozostaje obliczyć górne ograniczenie używając twierdzenia 2.3.

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

Twierdzenie 3.4 *Niech $\lambda, g \in \mathbb{N}^+$, $N = 2^g$, $n = N(2\lambda g+1)$, wtedy*

$$\Pi_{cc}^{\|}(RSG_\lambda^{\overline{n}}) = O\left(n^{1.667}\right)$$

Dowód. Kożystając, z twierdzenia 3.2, RSG_λ^N jest N rzędowym grafem. Z lematu 2.2 wynika więc, że RSG_λ^N jest $(n/t,\ N+t-N-1)$-reducible dla dowolnego $t \geqslant 1$. Z lematu 2.2 wynika, że $\Pi_{cc}^{\|}(RSG_\lambda^{\overline{n}}) = O\left(n\left(\sqrt{(Nt+t-N-1)n\delta}+\frac{n}{t}\right)\right)$. Aby dostać najdokładniejsze (najmniejsze) ograniczenie górne trzeba zminimalizować $n\left(\sqrt{(Nt+t-N-1)n\delta}+\frac{n}{t}\right)$. Zanim jednak przejdziemy do minimalizowania, uprośćmy nieco to wyrażenie

$$n\left(\sqrt{(Nt+t-N-1)n\delta}+\frac{n}{t}\right) \leqslant n\left(\sqrt{2Ntn\delta}+\frac{n}{t}\right).$$

Teraz możemy znaleźć minimum względem naszego parametru t.

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

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

$$\frac{\sqrt{2\delta N n^3}}{2t} - \frac{n^2}{t^2} = 0$$

$$t = \frac{2n^2}{\sqrt{2\delta N n^3}} = \tilde{O}\left(n^{\frac{1}{3}}\right)$$

Zatem podstawiając t minimalizujące ograniczenie górne do wzoru z twierdzenia 2.2 otrzymujemy

$$\Pi_{cc}^{\|}(RSG_\lambda^{\overline{n}}) = \tilde{O}\left(n\left(\sqrt{(Nn^{\frac{1}{3}}+n^{\frac{1}{3}}-N-1)n\delta}+\frac{n}{n^{\frac{1}{3}}}\right)\right) = \tilde{O}\left(n^{1.667}\right). \quad \square$$

3.6 Porówanie

MHF można porównywać za pomocą złożoności całkowitej, która mówi o koszcie równoległego obliczania funkcji. Im ta złożoność jest wyższa, tym bardziej czas obliczeń jest zdominowany przez koszt dostępu do pamięci, a więc funkcja jest bezpieczniejsza. Porównanie przedstawione jest w tabeli 3.1.

Algorytm	Ograniczenie dolne	Ograniczenie górne
Argon2i-B	$\tilde{\Omega}(n^{1.667})$	$\tilde{O}(n^{1.8})$
Catena Butterfly	$\tilde{\Omega}(n^{1.5})$	$\tilde{O}(n^{1.625})$
RiffleScrambler	$\tilde{\Omega}(n^{1.5})$	$\tilde{O}(n^{1.667})$

Tablica 3.1: [8, Tablica 1] Porównianie złożoności całkowitej dla Argon2i, Catena (dla grafu Butterfly) oraz RiddleScrambler (Twierdzenia 3.3, 3.4).

Ważnym aspektem jest również czas obliczeń, czyli w ilu krokach można obliczyć daną funkcję. Ponieważ koszt obliczania funkcji ma pochodzić głównie z kosztu dostępu do pamięci, to czas obliczeń może zostać zminimalizowany. Jednocześnie przy próbie obliczenia funkcji z mniejszą, niż wymaganą ilością pamięci, czas obliczeń powinien rosnąć jak najszybciej, tak aby wymuszać użycie założonej ilości pamięci. Tablica 3.2 zawiera porównanie właściwości czasu obliczeń (etykietowania) T w zależności od dostępnej pamięci S (ilości etykiet).

	BHF_7	BHG_3	Argon2i	Catena BFG	RiffleScrambler
Czas obliczeń T dla S = N	$8\lambda N$	$4\lambda N$	$2\lambda N$	$4\lambda N$	$3\lambda N$
Czas obliczeń T dla $S \leqslant \frac{N}{64}$	$T \geqslant \frac{2^{\lambda}-1}{32S}N^2$	$T \geqslant \frac{\lambda N^2}{32S}$	$T \geqslant \frac{N^2}{1536S}$	$T \geqslant \left(\frac{\lambda N^2}{64S}\right)^{\lambda}$	$T \geqslant \left(\frac{\lambda N^2}{64S}\right)^{\lambda}$
Czas obliczeń T dla $\frac{N}{64} \leqslant S \leqslant \frac{N}{20}$	nieznany				
Graf zależny od soli	tak	tak	tak	nie	tak

Tablica 3.2: [12, Tablica 4] Porównianie bezpieczeństwa i wydajności dla Balloon Hashing(BHF_7 dla grafu o stopniu wejściowym 7, BHF_3 dla grafu o stopniu wejściowym 3), Argon2i, Catena (dla grafu Butterfly) oraz RiffleScrambler.

Implementacja

Implementacja **RiffleScrambler** została napisana tak, aby spełniała wymagania konkursu na funkcję do przechowywania haseł [5, 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,
- 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 [19], yescryt [17] oraz Makwa [18] 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 [13].

W implementacji używana jest biblioteka OpenSSL [3]. Z podanej biblioteki użyto interfejsu EVP [4], 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 płycie 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)[1] 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 [14] - 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 pwdlen_bytes Długość hasła w bajtach
5  * @param salt Wskaźnik na sól
6  * @param saltlen_bytes Długość soli w bajtach
7  * @param garlic Paremetr "g" oznaczający koszt pamięci
8  * @param depth Parametr "lambda" oznaczający koszt czasowy
9  * @param hash_func Nazwa kryptograficznej funkcji skrótu do użycia wewnątrz
10 * @return Hash hasła jako ciąg znaków w systemie szesnastkowym
11 */
12 std::string riffle_scrambler(const void *const pwd, const size_t pwdlen_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 Paremetr "g" oznaczający koszt pamięci
21 * @param depth Parametr "lambda" oznaczający koszt czasowy
22 * @param pwd Wskaźnik na hasło
23 * @param pwdlen_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 wewnątrz
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 pwdlen_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 Paremetr "g" oznaczający koszt pamięci
38 * @param depth Parametr "lambda" oznaczający koszt czasowy
39 * @param pwd Wskaźnik na hasło
40 * @param pwdlen_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 pwdlen_bytes,
48 const void *salt, const size_t saltlen_bytes, const std::string hash_func="sha256");
49
50
51 /**
52 * Weryfikacja hasha 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 przeciwny 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 pwdlen_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 hasha hasła ma zostać zapisane
11 * @return RIFFLE_SCRAMBLER_OK jeśli obliczenia zakończyły się pomyślnie, kod błędu w przeciwnym przypadk
12 */
13 int riffle_scrambler(void *hash, const size_t hashlen, const void *pwd, const size_t pwdlen_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 3.1.1.

- 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 λ, 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 wystarczające 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` (λ), 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.

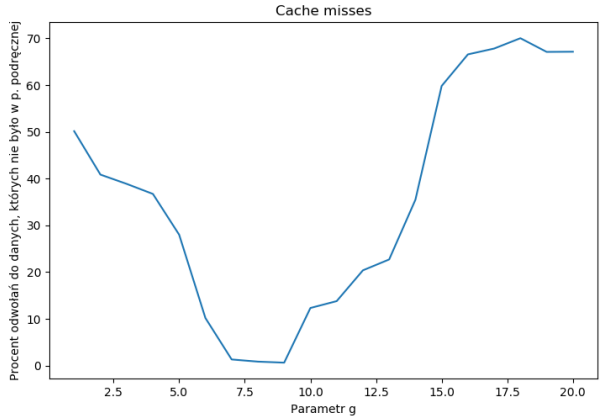
Dobrze odwzorowuje to wykres 4.1a. Oprócz sytuacji dla $g < 7$, którą ciężko wyjaśnić, a więc biorąc pod uwagę wartości dla $g \geqslant 7$, liczba odwołań do danych, które nie znajdowały się w pamięci podręcznej (ang. *chache misses*) zaczyna rosnąć od pewnego momentu. Pierwszy wzrost ma miejsce dla $g = 10$, 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.

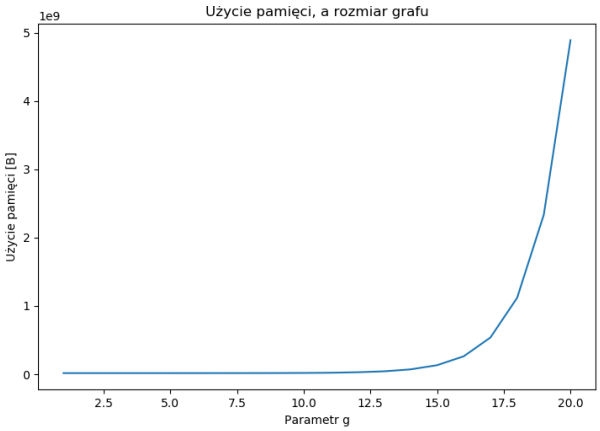
Co za tym idzie, obliczając graf z taką samą liczbą wierzchołków, ale dla różnych parametrów g (wyrównując ilość wierzchołków za pomocą parametru λ), czas obliczeń takich funkcji powinien być wyższy dla większych wartości g. Tą zależność wydaje się potwierdzać wykres 4.1f.

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

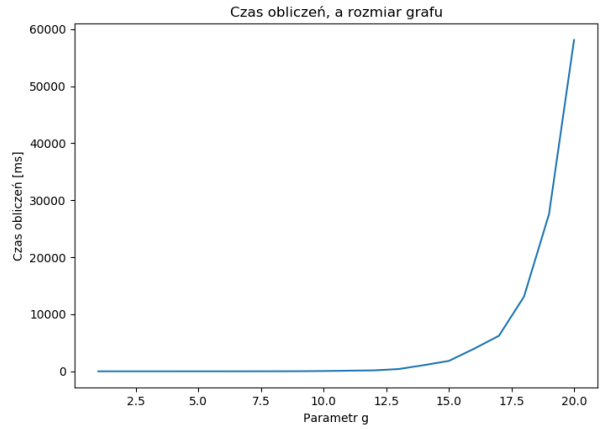
Wykres 4.1c przedstawia wpływ na czas obliczeń **RiffleScrambler** parametru g, który jest wykładniczy, oraz parametry λ, który jest liniowy.



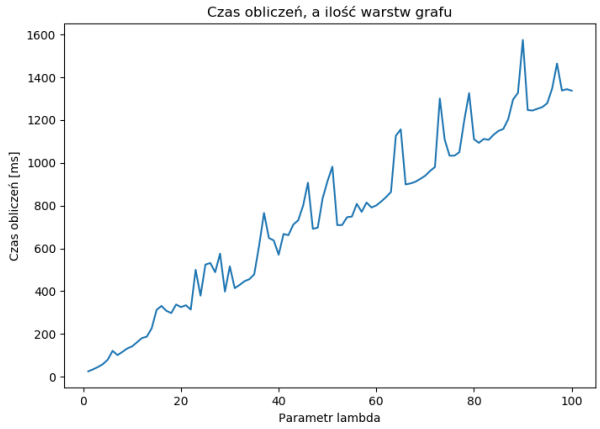
(a) Wykresy liczby odczytów spoza pamięci podręcznej.



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

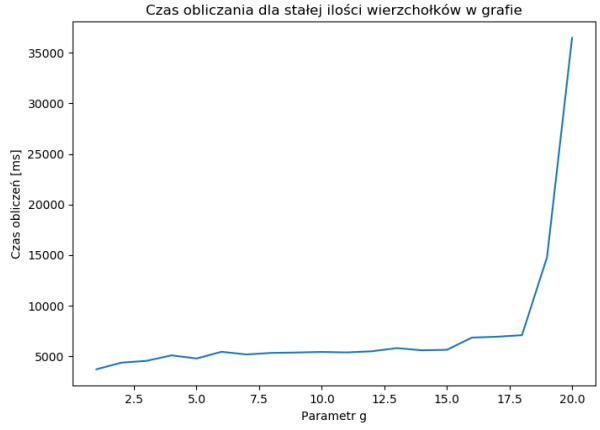


(c) Wykres czasu obliczania **RiffleScrambler** w zależności od parametru g.



(d) Wykres czasu obliczania **RiffleScrambler** w zależności od parametru λ.

(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.

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:
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) {

        // stowrzenie 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 paramertach, 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) {
        // stowrzenie 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 [2].
Testowane jest zewnętrzny interfejs oraz funkcje wewnętrzne.
Przykład uruchomienia testów.

```
$ ./catch_test
```

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

Wymagania - cmake - make -openssl - kompilator

4.9 Kompilowanie i użycie

4.9.1 Biblioteka

Kody biblioteki `riffle` znajdują się w katalogi `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.

4.9.2 Testy

4.9.3 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.

Wydajność czasowa jest na dobrym poziomie, jednak wydajność 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] Argon2 - implementation. https://github.com/P-H-C/phc-winner-argon2.

[2] Catch2 - a modern, c++-native, header-only, test framework for unit-tests. https://github.com/catchorg/Catch2.

[3] Openssl - cryptography and ssl/tls toolkit. https://www.openssl.org.

[4] Openssl evp - a high level interface to openssl cryptographic functions. https://wiki.openssl.org/index.php/EVP.

[5] The password hashing competition. https://password-hashing.net/cfh.html.

[6] Radix sort - a non-comparative stable algorith. https://en.wikipedia.org/wiki/Radix˙sort.

[7] J. Alwen, J. Blocki. Efficiently computing data-independent memory-hard functions. *Annual Cryptology Conference*, strony 241–271. Springer, 2016.

[8] 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.

[9] A. Biryukov, D. Dinu, D. Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, strony 292–302. IEEE, 2016.

[10] D. Boneh, H. Corrigan-Gibbs, S. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. *Advances in Cryptology–ASIACRYPT 2016*, strony 220–248. Springer, 2016.

[11] C. Forler, S. Lucks, J. Wenzel. Catena: A memory-consuming password-scrambling framework. Raport instytutowy, Citeseer, 2013.

[12] 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.

[13] Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, Gru. 2017.

[14] S. Josefsson. The base16, base32, and base64 data encodings. Raport instytutowy, 2006.

[15] B. Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. Raport instytutowy, 2000.

[16] C. Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, strony 1–16, 2009.

[17] 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.

[18] T. Pornin. The makwa password hashing function, 2015.

[19] M. A. Simplício 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

Dołączona do pracy płyta CD zawiera kod implementacji wraz z testami, skryptami do analizy wydajności oraz programem do testowania funkcji z lini poleceń. Struktura katologów na płycie CD.

- `RiffleScrambler` - katalog zawierający implementację funkcji **RiffleScrambler**, implementacja podzielona jest na pliki nagłówkowe znajdujące się w podkatalogu `include` oraz pliki źródłowe znajdujące się w podkatalogu `src`.
- `test` - w tym katalogu znajdują się testy jednostkowe oraz biblioteka `catch.hpp` używana do przeprowadzania testów.
- `benchmark` - katalog zawierający kod testów wydajnościowych razem ze skryptem `run_benchmarks.sh` służącym do uruchamiania testów. W katalogu znajdują się również podkatalogi `results` zawierający pliki z wynikami testów oraz podkatalog `plots`, w kórym znajduje się skrypt `plot_results.py` służący do przedstawiania wyników w formie wykresów.
- `rs` - katalog zwierający program do uruchamiania funkcji **RiffleScrambler** z linii poleceń.

W każdym z wymienionych katalogów znajdują się dodatkowo pliki `CMakeLists.txt` pozwalające na łatwą kompilację źródeł.