RiffleScrambler – bezpieczna funkcja do przechowywania haseł *

Filip Zagórski[1]

Oktawave

Abstract. RiffleScrambler jest rodziną grafów skierowanych i odpowiadającą jej funkcją, która może m. in. służyć do do bezpiecznego przechowywania haseł.
Funkcja jest zaprojektowana w taki sposób, aby ataki na hasła z wykorzystaniem dedydkowanego sprzętu były jak nieopłacalne.
W niniejszej pracy opisujemy funkcję RiffleScrambler*, która jest zmodyfikowaną wersją RiffleScrambler – uproszczony jest setup systemu i dla większej liczby warstw wykonywanych jest mniej obliczeń.

Keywords: Memory hardness, password storing, Markov chains, mixing time.

1 Introduction

W pierwszych systemach komputerowych hasła przechowywane w sposób niezakodowany jako para $\langle user, password\rangle$, dostęp do pliku z hasłami był jedynie ograniczony uprawnieniami danego użytkownika. Dzięki temu, administratorzy mogli poznać hasła użytkowników. W latach 60-tych XX wieku zmieniono podejscie i hasła zaczęto szyfrować, choć de facto stało się to powszechną praktyką dopiero w latach 70-tych, gdy powstała funkcja `crypt`. Hasło przechowywano jako $\langle user, f_k(password)\rangle$, gdzie f była funkcją szyfrującą (np. DES), a k oznaczało klucz szyfrujący. Z uwagi na fakt, że klucz i tak należało przechowywać w danym systemie, rozwiązanie to nie podnosiło znacząco bezpieczeństwa. Dość szybko szyfrowanie zastąpiono funkcjami jednokierunkowymi i hasła przechowano jako parę $\langle user, h(password)\rangle$, gdzie h była bezkolizyjną funkcją haszującą. Rozwiązanie to choć uniemożliwia administratorowi na deszyfrowanie haseł, to podowuje, że dwaj użytkownicy mający to samo hasło mają przypisaną tą samą wartość.

Tak więc, zazwyczaj przechowuje się hasła w postaci $\langle user, h(pass, salt), salt\rangle$, gdzie *user* jest identyfikatorem użytkownika, *pass* jego hasłem, h jest

* Wyniki badań były finansowane przez Regionalny Program Operacyjny Województwa Mazowieckiego na lata 2014-2020, umowa RPMA.01.02.00-14-5767/16-02

funkcją haszującą, a *salt* jest tzw. solą. Taki sposób przechowywania haseł ma swoje zalety: (1) można łatwo zweryfikować poprawność podanego hasła, poprzez obliczenie h, (2) dzięki wykorzystaniu soli, nawet dla użytkowników o tych samych hasłach, przchowywana wartość jest różna. Przez wiele lat wydawało się, że taki sposób przechowywania haseł jest dobry, o ile tylko wykorzystana funkcja haszująca h jest bezkolizyjna.

Aby spowolnić atakujących, wprowadzono dwa dodatkowe zabezpieczenia:

pepper – hasła przechowywane są jako trójka $\langle user, f(password, salt, pepper), salt\rangle$, gdzie wartość pepper nie jest przechowywana – obliczenie hasła zostaje spowolnione przez współczynnik proporcjonalny do rozmiaru przestrzeni, z której wybrano (losową) wartość pepper.

garlic – powoduje spowolnienie obliczenia wartości funkcji poprzez jej składanie: $\left\langle user, f^{garlic}(password, salt, pepper), salt, garlic\right\rangle$.

Jednakże zarówno wykorzystanie pieprzu jak i soli ma również słabą stronę: tak samo zwalnia obliczenie funkcji u atakującego jak i na serwerze.

1.1 Memory hard functions

Większość wykorzystywanych w kryptografii funkcji jest projektowanych w ten sposób, aby jednocześnie uzyskać zarówno wysoki poziom bezpieczeństwa jak i najwyższą wydajność (w szczególności na różnych platformach sprzętowych). Wymaganie dotyczące wydajności najczęściej powoduje, że do obliczenia danej funkcji nie jest wymagane dużo pamięci. Okazuje się, że takie podejście może się obrócić przeciwko twórcom systemów zabezpieczeń. Osoby, które chcą złamać hasła wykorzystują dedykowane układy liczące (FPGA bądź ASIC), które umożliwiają im na tanie, równoległe obliczanie wartości funkcji i dzięki temu, dużo efektywniejsze łamanie haseł. Dobry porcesor jest w stanie obliczać SHA-256 z prędkością ok 1GH/s, dobra karta graficzna z prędkościami ok 30GH/s, natomiast dedykowany układ liczący (np. Antminer S9) osiąga 14 000GH/s. Jeżeli porównamy wydajność energetyczną, to przewaga układów ASIC w przeliczeniu na liczbę liczonych haszy wynosi ok. 1 000 000 (tj. policzenie hasza na Antminer S9, kosztuje ok milion razy taniej niż policzenie tego samego hasza na serwerze).

Chcąc zlikwidować asymetrię w kosztach weryfikacji hasła względem kosztów łamania haseł należy wykorzystać funkcje, które ograniczają przewagę atakujących. Te funkcje to *memory hard functions* (MHF, “funkcje pamięciożerne”). Pierwsze funkcje typu memory-hard zostały zaproponowane przez Percivala [Per], który zaproponował funkcję `scrypt`. Jego nowatorskie podejscie polegało na tym, że nie tylko funkcja liczyła się "wolniej" (efekt ten był osiągany już wcześniej dzięki parametrom: salt, pepper, garlic), ale przede wszystkim dlatego, że do jej obliczenia wymaganych jest dużo pamięci.

O obliczaniu funkcji F, możemy myśleć jak o ciągu odwołań do pamięci. Dowolna funkcja może być opisana jako skierowany acykliczny graf $G = G_F = \langle V, E\rangle$ (gdzie V odpowiada wierzchołkom grafu $V = \langle v_1, \ldots, v_M\rangle$, a krawędzie $E \subset V \times V$). Gdy do obliczania wartości v_i potrzebne są wartości przechowywane w $v_{i,1}, \ldots, v_{i,k}$ to są one rodzicami wierzchołka v_i (i odpowiadają im krawędzie skierowane $\langle v_{i,j}, v_i\rangle$.

Przykład 1 *Funkcja PBKDF2 [Bur00] jest zdefiniowana jako* $F(pass) = F_{pbkdf,h}(pass) = hash^{1024}(pass)$ *dla funkcji haszującej* h*, może być reprezentowana przez graf składający się z wierzchołków* $V = \{v_0, \ldots, v_N\}$ *dla* $N = 1024$ *i krawędzi* $E = \{\langle v_{i-1}, v_i\rangle : i \in \{1, \ldots, N\}\}$*, zależność między wartościami w wierzchołkach:* $v_i = h(v_{i-1})$*, a* $v_0 = pass$*. Natomiast wartość* $F(pass) = v_N$*.*

Funkcje MHF dzielą się na dwie grupy: te, dla których graf G_F zależy od obliczanego hasła (dMHF – data dependent MHF) oraz takie, dla których graf nie zależy od hasła (iMHF – data independent MHF). Dla iMHF dopuszczalne jest, aby graf zależał np. od soli, co więcej jest to wręcz porządane, bo utrudnia łamanie haseł. Funkcje dMHF (np. scrypt) są podatne na ataki typu side-channel.

Poziom bezpieczeństwa może być rozpatrywany w dwóch modelach: sekwencyjnym i równoległym. W modelu sekwencyjnym zakładamy, że adwersarz próbuje odwrócić F (tj. znaleźć pasujące hasło) poprzez wykonywanie obliczeń na jednoprocesorowej maszynie. Bezpieczeństwo w tym modelu odpowiada własnościom grafu G_F, w szczególności, gdy G_F jest tzw. superkoncentratorem (Definicja 3) to funkcja F jest memory-hard w modelu sekwencyjnym. Natomiast, gdy G_F jest grafem o własności *depth-robust* to F jest memory-hard w modelu równoległym.

1.2 Definicje

Złożoność sekwencyjną dla grafu G oznaczamy przez $\Pi_{st}(G)$ i odpowiada czasowi, który jest potrzebny do "obliczenia" grafu przemnożonej przez maksymalną liczbę komórek pamięci, wykorzystywaną przez najlepszy sekwencyjny algorytm obliczający F (G_F).

Złożoność równoległa jest definiowana jako suma komórek pamięci wykorzystywanych przez najlepszy algorytm równoległy obliczający G, oznaczamy ją przez $\Pi_{cc}^{||}(G)$. Definicje te zostały wprowadzone w ciągu prac [AS15,BCGS16,FLW,LT82].

Przykład 2 *Dla wspomnianej już (w Przykładzie 1, otrzymujemy następujące wartości:* $\Pi_{st}(PBKDF2) = n$ *a* $\Pi_{cc}^{||}(PBKDF2) = n$, *oznacza to, że PBKDF2 nie jest funkcją typu memory-hard, bo dla takowych oczekujemy:* $\Pi_{st}(mhf) = \Omega(n^2)$.

Konkurs na najlepszą funkcję do przechowywania haseł Password Hashing Competition [phc] został ogłoszony w 2013 roku. Zwycięzcą została funkcja `Argon2i` [BDK16], funcje `Catena` [FLW15], `Lyra2` [SAA$^+$], `yescript` [yes14] i `Makwa` [Por15] otrzymały wyróżnienia.

Niech DFG_n^λ oznacza graf wykorzystywany w Catenie (wersja Dragonfly), a BFG_n^λ niech oznacza graf wersji Butterfly (porównaj [ABP17,FLW15]), natomiast przez $\mathsf{BHG}_\sigma^\lambda$ oznaczmy graf obliczeń w Balloon Hashing [BCGS16].

Obliczenie wartości funkcji wymaga N komórek pamięci i jeżeli proces dysponuje taką ilością pamięci, to obliczenie wykonuje się w czasie T. Przedstawiamy wyniki, które pokazują trade-off (w modelu sekwencyjnym): tj. pokazujemy ile zajmie obliczenie wartości funkcji jeżeli dysponuje się jedynie $S << N$ komórkami pamięci. Dla modelu sekwencyjneg i grafów DFG_N^λ i $\mathsf{BHG}_\sigma^\lambda$ zachodzą następujące wyniki:

Lemat 1 *Każdy adwersarz wykorzystujący* $S \leq N/20$ *komórek pamięci, oblicza fukcję Catena (w wersji Dragonfly –* DFG_N^λ*) w czasie* T

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

Lemat 2 *Adwersarz dysponujący* $S \leq N/64$ *komórkami pamięci, oblicza funkcję BalloonHashing (*$\mathsf{BHG}_\sigma^\lambda$*) dla* $\delta = 7$ *i liczby rund* λ *w czasie nie mniejszym niż:*

$$T \geq \frac{(2^\lambda - 1)N^2}{32S}.$$

Analogiczne wyniki zachodzą też dla ataków równoległych, dla BFG_n^λ, DFG_λ^n and $\mathsf{BHG}_\sigma^\lambda$ zachodzą następujące trade-offy:

Twierdzenie 1 (Twierdzenie 7 w [ABP17])

- *Dla* $\lambda, n \in \mathbb{N}^+$ *takich, że* $n = 2^g(\lambda(2g-1)+1)$ *i dla pewnego* $g \in \mathbb{N}^+$ *zachodzi:*

$$\Pi_{cc}^{||}(\mathsf{BFG}_n^\lambda) = \Omega\left(\frac{n^{1.5}}{g\sqrt{g\lambda}}\right)$$

- *Jeżeli $\lambda, n \in \mathbb{N}^+$ są takie, że $k = n/(\lambda+1)$ jest potęgą 2, to:*

$$\Pi_{cc}^{||}(\mathsf{DFG}_n^{\lambda}) = \Omega\left(\frac{n^{1.5}}{\sqrt{\lambda}}\right)$$

- *Jeżeli $\tau, \sigma \in \mathbb{N}^+$ są takie, że $n = \sigma \cdot \tau$ to z dużym prawdopodobieństwem zachodzi:*

$$\Pi_{cc}^{||}(\mathsf{BHG}_\tau^{\sigma}) = \Omega\left(\frac{n^{1.5}}{\sqrt{\tau}}\right).$$

Wynik Alwena i Blocki'ego [AB16] pokazuje, że atakujący w modelu równoległym może, dla każdej funkcji typu iMHF (np. Argon2i, Balloon, Catena, *etc.*) oszczędzić na pamięci, zatem optimum nie wynosi $\Pi_{cc}^{||} = \Omega(n^2)$, ale $\Pi_{cc}^{||} = \Omega(n^2/\log n)$. Z drugiej strony, wydaje się, że w praktyce wynik ten nie będzie miał znaczenia z uwagi na fakt, że zakłada, że zużycie pamięci musi być duże ($>$ 1GB), podczas gdy funkcje MHF wykorzystują najczęściej co najwyżej kilkanaście megabajtów (np. 16MB).

1.3 Wkład

W tej pracy opisujemy (bazując na [GLZ18] w Rozdziale 3) RiffleScrambler nową rodzinę skierowanych grafów acyklicznych i związaną z nimi funkcję typu iMHF.

Następnie opisujemy, zmodyfikowaną wersję RiffleScrambler *, która charakteryzuje się łatwiejszym pre-procesingiem.

RiffleScrambler dla hasła x, soli s i parametrów bezpieczeństwa (liczby całkowite) g, λ, wykonuje następujące obliczenia:

1. tworzy permutację $\rho = \rho_g(s)$ dla $N = 2^g$ elementów, wykorzystując algorytm Riffle Shuffle: Algorytm 1),
2. tworzy graf $\mathsf{RSG}_\lambda^N = G_{\lambda,g,s} = G_{\lambda,g}(\rho)$,
3. pierwszy wierzchołek RSG_λ^N przechowuje hasło x, wartością funkcji jest wartość ostatniego obliczonego wierzchołka.

Zaproponowany nowy schemat RiffleScrambler* nie wymaga tworzenia i przechowywania permutacji ρ, kolejne warstwy grafu są tworzone "online". Konstrukcja zapewnia te podobne parametry bezpieczeństwa (w modelu sekwencyjnym) oraz większą wydajność. Funkcja wymaga jednak, aby parametr $\lambda > 2$.

Lemat 3 *Każdy algorytm wykorzystujący $S \leq N/20$ komórek pamięci, wymaga do obliczenia RiffleScrambler* czasu T, dla którego zachodzi*

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

gdzie $\mathsf{RSG}_{\lambda}^{N}$ *jest grafem obliczeń RiffleScrambler*.*

Powyższy rezultat pokazuje, że schemat RiffleScrambler gwarantuje ten sam poziom bezpieczeństwa co Catena i jednocześnie większy poziom bezpieczeństwa niż BalloonHashing (Lemat 2). Główną przewagą RiffleScrambler nad Ceteną jest to, że graf obliczeń zależy od soli. Dla ataku równoległego zachodzi następujący wynik.

Lemat 4 *Dla dodatnich liczb całkowitych* λ, g *niech* $n = 2^g(2\lambda g + 1)$, *wtedy*

$$\Pi_{cc}^{||}(\mathsf{RSG}_{\lambda}^{k}) = \Omega\left(\frac{n^{1.5}}{\sqrt{g\lambda}}\right).$$

Ponadto, warto zauważyć, że RiffleScrambler jest odporne na ataki typu cache-timing, z uwagi na fakt, że dostęp do pamięci jest niezależny od hasła.

2 Podstawowe pojęcia

Dla skierowanego grafu acyklicznego DAG (directed acyclic graph) $G = (V, E)$ na $n = |V|$ wierzchołkach, definiujemy jego stopień wejściowy jako $\delta = \max_{v \in V} indeg(v)$. Rodzicami wierzchołka $v \in V$ jest zbiór $\mathsf{parents}_G(v) = \{u \in V : (u, v) \in E\}$ (bezpośrednich poprzedników v).

Wierzchołek $u \in V$ nazywamy *źródłem* jeżeli nie posiada rodziców (jego stopień wejściowy wynosi 0), natomiast o $u \in V$ mówimy, że jest *ujściem* jeżeli nie jest rodzicem dla żadnego innego wierzchołka (ma stopień wyjściowy 0) Oznaczamy zbiór wszystkich ujść G przez $\mathsf{sinks}(G) = \{v \in V : outdegree(v) = 0\}$.

Mówimy, że skierowana ścieżka $p = (v_1, \ldots, v_t)$ jest długości t w G jeżeli $(\forall i) v_i \in V$, $(v_i, v_{i+1}) \in E$ i oznaczamy to przez $\mathsf{length}(p) = t$. *Głębokością* (średnicą) grafu G oznaczamy $d = \mathsf{depth}(G)$ i jest to długość najdłuższej skierowanej ścieżki G.

2.1 Pebbling game i złożoność

Główną techniką służącą do analizy funkcji iMHF jest nazywana *pebbling game*. Intuicyjnie, jest to gra, w której adwersarz, mając do dyspozycji graf skierowany kładzie i zdejmuje kolejno kamyki z wierzchołków grafu. Kamyk można położyć na wierzchołek tylko wtedy, gdy wszyscy rodzice wierzchołka mają położony kamyk (kamyk można też położyć na dowolny z wierzchołków-źródeł). Zdjąć kamyk można z dowolnego wierzchołka w każdym momencie. Celem gry jest takie układanie kamieni, aby na końcu gry kamyki leżały na wszystkich wierzchołkach-ujściach (wygrana gra). Złożoność pamięciowa danego grafu określana jest jako minimalna liczba kamyków, która jest potrzebna do wygrania gry.

Poniżej przedstawiamy bardziej formalną definicję [ABP17].

Definicja 1 (Równoległy/Sekwencyjny Graph Pebbling) *Niech $G = (V, E)$ będzie grafem skierowanym, a $T \subset V$ niech będzie zbiorem docelowym wierzhołków (celem gry jest położenie kamyków na wszystkich wierzchołkach z T). Konfiguracją (kamyków na G) w chwili i jest podzbiór $P_i \subset V$. Poprawnym, równoległym pebblowaniem T jest taki ciąg $P = (P_0, \ldots, P_t)$ konfiguracji na G, gdzie $P_0 = \emptyset$ i dla którego zachodzą warunki* 1 *i* 2*. Sekwencyjne pebblowanie musi spełniać dodatkowo warunek* 3*.*

1. *Dla każdego wierzchołka docelowego z T, istnieje taka konfiguracja, która zawiera dany wierzchołek.*

$$\forall x \in T \quad \exists z \leq t \quad : \quad x \in P_z.$$

2. *Kamyki mogą być położone wyłącznie na te wierzchołki, których rodzice posiadają kamyki w poprzednim kroku.*

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

3. *Co najwyżej jeden kamyk jest położony w jednym kroku.*

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

Przez $\mathcal{P}_{G,T}$ i $\mathcal{P}^{||}_{G,T}$ oznaczamy zbiory poprawnych ciągów konfiguracji (sekwencyjnych i równoległych) grafu G dla zbioru docelowego T.

Łatwo można zauważyć, że $\mathcal{P}_{G,T} \subset \mathcal{P}^{||}_{G,T}$. Gdy zbiór docelowy składa się wyłącznie z ujść grafu $T = \mathsf{sinks}(G)$, to w takim przypadku używamy oznaczeń $\mathcal{P}_G$ i $\mathcal{P}^{||}_G$.

Definicja 2 (Czasowa/Pamięciowa/Łączna Złożoność Pebblingu) *Złożoności czasowa (time - t), pamięciowa (space - s), pamięciowo-czasowa (space-time st) oraz złożoność łączna (cumulative cc) dla pebblowania $P = \{P_0, \ldots, P_t\} \in \mathcal{P}^{||}_G$ są zdefiniowane następująco:*

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

Dla $\alpha \in \{s, t, st, cc\}$ i zbioru docelowego $T \subset V$, złożoność sekwencyjna i równoległa pebblowania G są zdefiniowane jako

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

Gdy $T = \mathsf{sinks}(G)$, piszemy $\Pi_\alpha(G)$ oraz $\Pi_\alpha^{||}(G)$.

2.2 Ataki sekwencyjne

Definicja 3 (N-Superkoncentrator) *Skierowany graf acykliczny $G = \langle V, E\rangle$ ze zbiorem wierzchołków V i zbiorem krawędzi E, ograniczonym stopniem wejściowym, N wierzchołkami wejściowymi (źródłami) i N ujściami jest nazywany N-Superconcentratorem jeżeli dla każdego k takiego, że $1 \leq k \leq N$ i dla dowolnej pary podzbiorów $V_1 \subset V$ dla k wierzchołków źródłowych i $V_2 \subset V$ – k ujść, istnieje k wierzchołkowo-rozłącznych ścieżek łączących wierzchołki V_1 z wierzchołkami V_2.*

Przez połączenie (zestakowanie) λ (liczba całkowita) N-Superkoncentratora, otrzymujemy graf nazywany (N, λ)-Superkoncentratorem.

Definicja 4 ((N, λ)-Superkoncentrator) *Niech $G_i, i = 0, \ldots, \lambda-1$ będą N-Superkoncentratorami. Niech G będzie grafem uzyskanym poprzez połączenie ujść G_i ze źródłami $G_{i+1}, i = 0, \ldots, \lambda - 2$. Graf G nazywamy (N, λ)-superkoncentratorem.*

Theorem 1 (Dolne ograniczenie dla (N, λ)-superconkoncentratora [LT82]). *Gra pebbling gmae dla (N, λ)-superkoncentratora, używająca $S \leq N/20$ kamyków wymaga T rund, gdzie*

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

3 RiffleScrambler

Funkcja RiffleScrambler wykorzystuje następujące parametry:

- s – parametr salt wykorzystywana do generowania grafu G,
- g – parametr garlic, $G = \langle V, E\rangle$, *i.e.*, $V = V_0 \cup \ldots \cup V_{2\lambda g}$, $|V_i| = 2^g$,
- λ – parametr określający liczbę warstw grafu G.

Niech $HW(x)$ oznacza wagę Hamminga binarnego ciągu x (liczba jedynek), a $\bar{x}$ oznacza dopełnienie x (jeżeli na współrzędnej i jest bit b_i to $\bar{b}_i = 1 - b_i$ (tak więc $HW(\bar{x})$ odpowiada liczbie zer x).

Definicja 5 *Niech* $B = (b_0 \dots b_{n-1}) \in \{0,1\}^n$ *(binarne słowo długości* n*). Definiujemy rząd* $r_B(i)$ *bitu* i *w słowie* B *jako*

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

Definicja 6 (Riffle-Permutation) *Niech* $B = (b_0 \dots b_{n-1})$ *będzie binarnym słowem długości* n*. Permutacja* π *indukowana z* B *jest definiowana jako*

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

for all $0 \leq i \leq n-1$*.*

Przykład 3 *Gdy* $B = 11100100$*, to* $r_B(0) = 0$*,* $r_B(1) = 1$*,* $r_B(2) = 2$*,* $r_B(3) = 0$*,* $r_B(4) = 1$*,* $r_B(5) = 3$*,* $r_B(6) = 2$*,* $r_B(7) = 3$*. Pemutacja Riffle-Permutation indukowana przez* B *jest równa:* $\pi_B = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 4\,5\,6\,0\,1\,7\,2\,3 \end{pmatrix}$*. Graficzna reprezentacja znajduje się na rysunku 1.*

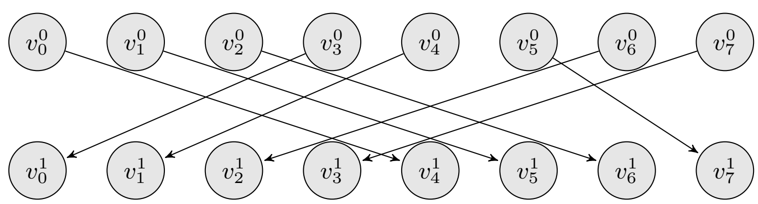


Fig. 1: Riffle-Permutation indukowana przez $B = 11100100$.

Definicja 7 (N-Single-Layer-Riffle-Graph) *Niech* $\mathcal{V} = \mathcal{V}^0 \cup \mathcal{V}^1$*, gdzie* $\mathcal{V}^i = \{v_0^i, \dots, v_{N-1}^i\}$ *i niech* B *będzie* N*-bitowym słowem. Niech* π_B *będzie Riffle-Permutation indukowaną przez* B*.Definiujemy* N*-Single-Layer-Riffle-Graph (dla parzystych* N*) jako graf z następującym zbiorem krawędzi* E*:*

- *1 krawędź:* $v_{N-1}^0 \to v_0^1$*,*
- N *krawędzi:* $v_i^0 \to v_{\pi_B(i)}^1$ *dla* $i = 0, \dots, N-1$*,*
- N *krawędzi:* $v_i^0 \to v_{\pi_{\bar{B}}(i)}^1$ *dla* $i = 0, \dots, N-1$*.*

Przykład 4 *Dla* B *i* π_B *takich jak w Przykładzie 3, graf* 8*-Single-Layer-Riffle-Graph jest zaprezentowany na Rysunku 2.*

Algorytm 3 generuje graf N-Double-Riffle-Graph, który jest zdefiniowany w sposób następujący. W dalszej części zakładamy, że $N = 2^g$.

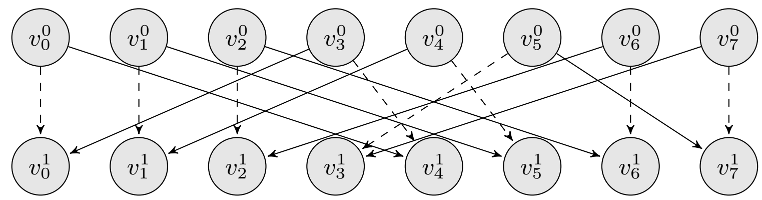


Fig. 2: Przykład 8-Single-Layer-Riffle-Graph (przedstawiony bez krawędzi (v_7^0, v_0^1)), indukowany przez $B = 11100100$. Permutacja π_B jest przedstawiona liniami ciągłymi, natomiast $\pi_{\bar{B}}$ liniami przerywanymi.

Definicja 8 (N-Double-Riffle-Graph) *Niech V oznacza zbiór wierzchołków, a E zbiór krawędzi grafu $G = (V, E)$. Niech $B_0, \ldots, B_{g-1}$ będzie ciągiem g słow binarnych długości 2^g każde. Wtedy graf N-Double-Riffle-Graph jest otrzymywany przez połączenie (zestakowanie) $2g$ grafów Single-Layer-Riffle-Graphs, dając graf składający się z $(2g+1)2^g$ wierzchołków*

- $\{v_0^0, \ldots, v_{2^g-1}^0\} \cup \ldots \cup \{v_0^{2g}, \ldots, v_{2^g-1}^{2g}\}$,

oraz następujących krawędzi:

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

oraz następujących krawędzi dla kolejnych g warstw – które są symetryczne względem numeru poziomu g (dla których wykorzystywane są permutacje odwrotne indukowane przez $B_j, j \in \{0, \ldots, g-1\}$):

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

Definicja 9 ((N, λ)-Double-Riffle-Graph) *Niech $G_i, i = 0, \ldots, \lambda-1$ będzie zbiorem λ grafów N-Double-Riffle-Graph. Wtedy (N, λ)-Double-Riffle-Graph jest zdefiniowany jako graf otrzymany przez zestakowanie (połaczenie) λ grafów N-Double-Riffle-Graphs w taki sposób, że ujścia G_i są wierzchołkami źródłami dla $G_{i+1}, i = 0, \ldots, \lambda-2$.*

Dla danej permutacji σ na $\{0, \ldots, 2^g-1\}$ elementach, niech $\mathbf{B}$ będzie macierzą binarną rozmiariu $2^g \times g$, której $j$ty wiersz jest binarną reprezentacją $\sigma(j), j = 0, \ldots, 2^g-1$. Przez B_i oznaczamy itą kolumnę $\mathbf{B}$, zatem

$\mathbf{B} = (B_0, \ldots, B_{2^g-1})$. Macierz taką określamy jako *binarną reprezentacją permutacji* σ. Macierz $\mathfrak{B} = (\mathfrak{B}_0, \ldots, \mathfrak{B}_{2^g-1})$ (wymiaru $2^g \times g$) uzyskujemy w następujący sposób: Niech $\mathfrak{B}_0 = B_0$, dla $i = 1, \ldots, 2^g - 1$ przypisujemy $\mathfrak{B}_i = \pi_{\mathfrak{B}^T_{i-1}}(B_i)$. Procedura ta $\mathsf{TraceTrajectories}$ jest przedstawiona jako Algorytm 2.

W przybliżeniu, procedura $\mathsf{RiffleScrambler}(x, s, g, \lambda)$ działa następująco.

- Dla danej soli s oblicza pseudolosową permutację σ (wykorzystując odwrócone w czasie Riffle Shuffle), niech $\mathbf{B}$ oznacza binarną reprezentację σ.
- Oblicza $\mathfrak{B} = \mathsf{TraceTrajectories}(\mathbf{B})$.
- Tworzy instancje grafu N-Double-Riffle-Graph ($2g + 1$ rzędów, każdy po 2^g wierzchołków) używając $\mathfrak{B}^T_0, \ldots, \mathfrak{B}^T_{g-1}$ jako słow binarnych (indukujących).
- Obliczając wartość funkcji dla x na grafie, przypisuje wartość w ostatnim rzędzie, czyli: $v_0^{2g+1}, \ldots, v_{2g+1}^{2^g-1}$.
- Ostatni rząd jest przepisywany do pierwszego: $v_i^0 = v_i^{2g+1}, i = 0, \ldots 2^g - 1$, a całe obliczenie jest powtarzane λ razy.
- Ostatecznie, wartość $v_{2^g-1}^{2g}$ jest zwracana.

Głowna procedura $\mathsf{RiffleScrambler}(x, s, g, \lambda)$ do obliczania hasła dla x, soli s i parametrów g, λ jest przedstawiona jako Algorytm refalg:init.

Przykład 5 *Przyklad grafu* $(8, 1)$*-Double-Riffle-Graph, który został uzyskany z permutacji:* $\sigma = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 5\,4\,6\,3\,2\,7\,0\,1 \end{pmatrix}$. *Jej reprezentacja binarna jest następująca:*

$$\mathbf{B} = (B_0, B_1, B_2), \quad \mathbf{B}^T = \begin{pmatrix} 1\,1\,1\,0\,0\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,0\,0 \\ 1\,0\,0\,1\,0\,1\,0\,1 \end{pmatrix}.$$

W ten sposób otrzymujemy trajektorie elementów i możemy z tego wyprowadzić słowa/permutacje dla każdej z warstw grafu::

- *Dla* $\mathfrak{B}_0 = B_0 = (11100100)^T$ *(z poprzednich przykładów) – otrzymane poprzez konkatenację fiperwszych cyfr elementów, otrzymujemy* $\pi_{B_0} = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 4\,5\,6\,0\,1\,7\,2\,3 \end{pmatrix}$.
- $\mathfrak{B}_1 = \pi_{\mathfrak{B}^T_0}(B_1) = \pi_{\mathfrak{B}^T_0}(11100100) = (11000011)^T$, *zatem* $\pi_{\mathfrak{B}_1} = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 4\,5\,0\,1\,2\,3\,6\,7 \end{pmatrix}$.
- $\mathfrak{B}_2 = \pi_{\mathfrak{B}^T_1}(B_2) = \pi_{\mathfrak{B}^T_1}(10010101) = (01011001)^T$, *czyli* $\pi_{B_2} = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 0\,4\,5\,1\,6\,2\,3\,7 \end{pmatrix}$.

- *i ostatecznie*

$$\mathfrak{B} = (\mathfrak{B}_0, \mathfrak{B}_1, \mathfrak{B}_2), \quad \mathfrak{B}^T = \begin{pmatrix} 1\,1\,1\,0\,0\,1\,0\,0 \\ 1\,1\,0\,0\,0\,0\,1\,1 \\ 0\,1\,0\,1\,1\,0\,0\,1 \end{pmatrix}.$$

Wynikowy graf jest przedstawiony na Rysunku 3.

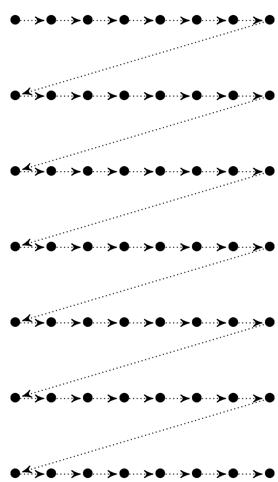
3.1 Pseudocodes

Algorithm 1 $\mathsf{RiffleShuffle}_H(n, s)$

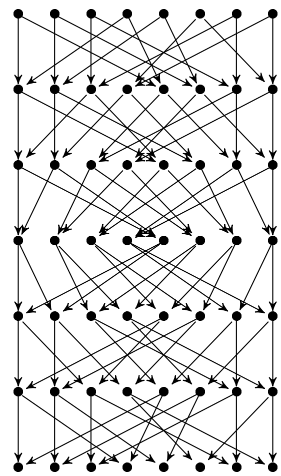
$\pi = \langle 1, \ldots, n \rangle$
for $i = 1$ to n **do**
  for $j = 1$ to $i - 1$ **do**
    $M[i, j] = 0$
  end for
end for
while $\exists_{1 \le i \le n} \exists_{1 \le j < i} M[i, j] = 0$ **do**
  $\mathcal{S}_0, \mathcal{S}_1 = \emptyset$
  for $w := 1$ to n **do**
    $b = H(s, w, r)_1$
    $\mathcal{S}_b = \mathcal{S}_b \uplus \pi[w]$
  end for
  for $i \in \mathcal{S}_0$ **do**
    for $j \in \mathcal{S}_1$ **do**
      $M[\max(i, j), \min(i, j)] = 1$
    end for
  end for
  $\pi = \mathcal{S}_0 \uplus \mathcal{S}_1$
end while
return π

Algorithm 2 $\mathsf{TraceTrajectories}(\mathbf{B})$

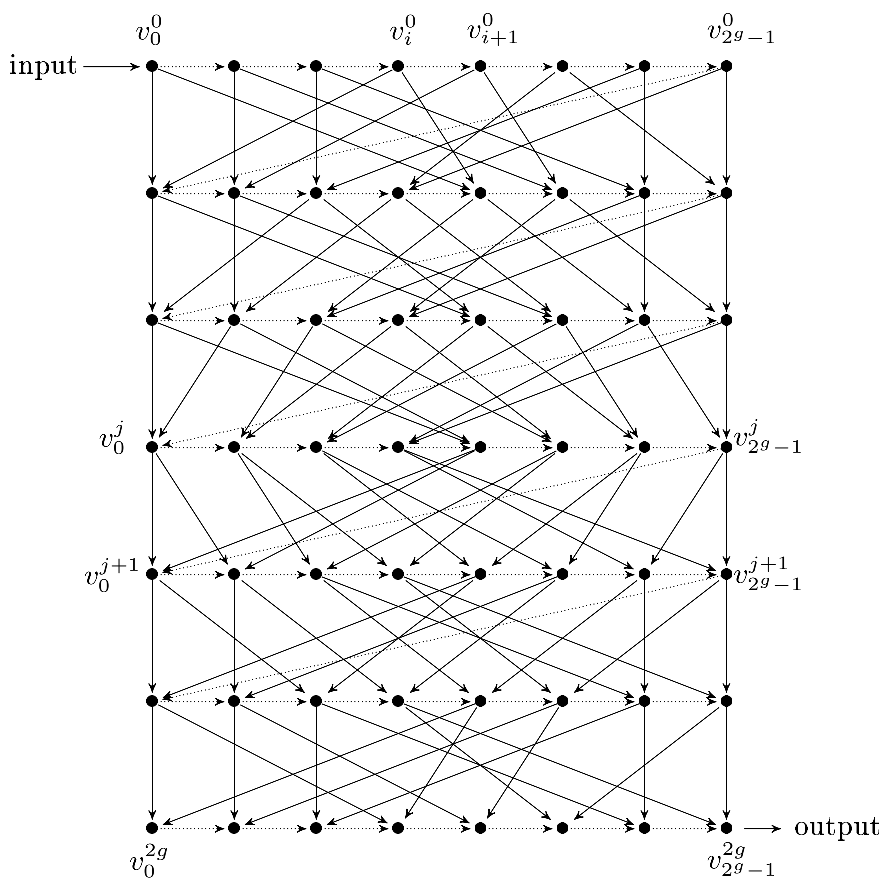
Require: : $\mathbf{B} = (B_0, \ldots, B_{g-1})$ {binary matrix of size $2^g \times g$ with columns $B_i, i = 0, \ldots, g-1$}
Ensure: : $\mathfrak{B}$ {binary matrix of size $2^g \times g$ with recalculated trajectories}
$\mathfrak{B}_0 = B_0$
for $i := 1$ to $g - 1$ **do**
  $\mathfrak{B}_i = \pi_{\mathfrak{B}^T_{i-1}}(B^T_i)$ *//Riffle-Permutation induced by* B^T_i
end for
return $\mathfrak{B} = (\mathfrak{B}_0, \ldots, \mathfrak{B}_{g-1})$



(a) warstwy sekwencyjne i łączące (niezależne od soli)



(b) Główna część GenGraph (zależna od soli)



(c) Wynikowy graf GenGraph

Fig. 3: Instancja grafu (8,1)-Double-Riffle-Graph

Algorithm 3 $\mathsf{GenGraph}_H(g, \sigma)$

$N = 2^g$
$V = \{v_i^j : i = 0, \dots, N-1; j = 0, \dots, 2g\}$
$E = \{v_i^j \to v_{i+1}^j : i = 0, \dots, N-2; j = 0, \dots, 2g\}$
$E = E \cup \{v_{n-1}^j \to v_0^{j+1} : j = 0, \dots, 2g-1\}$
Let $\mathbf{B}$ be a binary representation of σ
Calculate $\mathfrak{B} = (\mathfrak{B}_0, \dots, \mathfrak{B}_{g-1}) =$ TraceTrajectories($\mathbf{B}$). Let $\mathfrak{B}_{2g+1-m} = \mathfrak{B}_m, m = 0, \dots, g-2$. Let $\mathfrak{B}_j = \mathfrak{B}_{j,0}\mathfrak{B}_{j,1} \dots \mathfrak{B}_{j,2^g-1}$
for $i = 0$ to $2g$ **do**
end for
for $j = 0$ to $2g-1$ **do**
    for $i = 0$ to $2^g - 1$ **do**
        $E = E \cup \{v_i^j \to v_{\pi_{\mathfrak{B}_{j,i}}}^{j+1}\} \cup \{v_i^j \to v_{\pi_{\bar{\mathfrak{B}}_{j,i}}}^{j+1}\}$
    end for
end for
return π

Algorithm 4 $\mathsf{RiffleScrambler}(n, x, s, g, \lambda)$

Require: s {Salt}, g {Garlic}, x {Value to Hash}, λ {Depth}, H {Hash Function}
Ensure: x {Password Hash}
$\sigma = \mathsf{RiffleShuffle}_H(2^g, s)$
$G = (V, E) = \mathsf{GenGraph}(g, \sigma)$
$v_0^0 \leftarrow H(x)$
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**
            $v_i^{j+1} := 0$
            for all $v \to v_i^{j+1} \in E$ **do**
                $v_i^{j+1} := H(v_i^{j+1}, v)$
            end for
        end for
    end for
    for $i = 0$ to $2^g - 1$ **do**
        $v_i^0 = v_i^{2g+1}$
    end for
end for
$x := v_{n-1}^{2g}$
return x

4 Podsumowanie

Przedstawiono nową funkcję typu memory-hard, która może być wykorzystywana do bepiecznego przechowywania haseł. RiffleScrambler ma lepszy time-memory trade-off niż Argon2i i Balloon Hashing, gdy rozmiar pamięci n rośnie, a liczba rund r jest stała (tak jak Catena-DBG).

	BHG_7	BHG_3	Argon2i	Catena BFG	RiffleScrambler
Serwer - czas T (for S = N)	$8\lambda N$	$4\lambda N$	$2\lambda N$	$4\lambda N$	$3\lambda N$
Atakujący - czas T ($S \leq \frac{N}{64}$)	$T \geq \frac{2^{\lambda}-1}{32S} N^2$	$T \geq \frac{\lambda N^2}{32S}$	$T \geq \frac{N^2}{1536S}$	$T \geq (\frac{\lambda N}{64S})^{\lambda} N$	$T \geq (\frac{\lambda N}{64S})^{\lambda} N$
Atakujący - czas T ($\frac{N}{64} \leq S \leq \frac{N}{20}$)	nieznane				
Graf zależny od soli	tak	tak	tak	nie	tak

Fig. 4: Porównanie wydajności i bezpieczeństwa BalloonHashing (BHG_3 to BalloonHashing dla $\delta = 3$, a BHG_7 oznacza graf BHG dla $\delta = 7$), Argon2i, Catena (Butterfly graph) oraz RiffleScrambler (RSG).

5 Kody źródłowe

Listing 1.1 Kod źródłowy RiffleShuffle.h – plik nagłówkowy

RiffleShuffle.h

```
class RiffleShuffle {
    private:
        uint8_t salt;
        int pepper;
        int N;
        int Nhalf;
        int lambda;
        const char* algorithm;
        SHA256_CTX digest;
        const char hexTable[16]  = {'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f

    public:
        void init(int, const char[]);
        string result;
        string scramble(std::string , std::string , int);
        string sha256(const std::string);
        bool isSet(std::string, int);
        string saltedChoices(int, int, std::string);
        string getHash();

    private: char hashString(uint8_t);
};
```

References

AB16. Joël Alwen and Jeremiah Blocki. Efficiently Computing Data-Independent Memory-Hard Functions. pages 241–271. 2016.

ABP17. Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-Robust Graphs and Their Cumulative Memory Complexity. In *EUROCRYPT 2017*, pages 3–32. Springer, Cham, apr 2017.

AD86. David Aldous and Persi Diaconis. Shuffling cards and stopping times. *American Mathematical Monthly*, 93(5):333–348, 1986.

AD87. David Aldous and Persi Diaconis. Strong Uniform Times and Finite Random Walks. *Advances in Applied Mathematics*, 97:69–97, 1987.

AS15. Joël Alwen and Vladimir Serbinenko. High Parallel Complexity Graphs and Memory-Hard Functions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing - STOC '15*, pages 595–603, New York, New York, USA, 2015. ACM Press.

BCGS16. Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks. pages 220–248. Springer, Berlin, Heidelberg, dec 2016.

BDK16. Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, mar 2016.

Listing 1.2 Metoda inicjalizująca obliczenia

RiffleShuffle::init

```
void RiffleShuffle::init(int N, const char * algorithm) {
    if (N % 2 != 0)
        N++;

    this->N = N;
    this->Nhalf = (int) N/2;

    this->lambda = 2 * (int) ceil(log2(N));
    this->pepper = 1;

    this->algorithm = "SHA-256";
    string result = "";
    SHA256_Init(&this->digest);
}
```


Bra17. William F. Bradley. Superconcentration on a Pair of Butterflies. jan 2017.

Bur00. Kaliski Burt. PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, 2000.

FLW. Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A Memory-Consuming Password-Scrambling Framework.

FLW15. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. 2015.

GLZ18. Karol Gotfryd, Paweł Lorek, and Filip Zagórski. RiffleScrambler – A Memory-Hard Password Storing Function. pages 309–328. Springer, Cham, sep 2018.

LT82. Thomas Lengauer and Robert E. Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *Journal of the ACM*, 29(4):1087–1130, oct 1982.

Per. Colin Percival. STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS.

phc. Password Hashing Competition.

Por15. Thomas Pornin. The MAKWA Password Hashing Function Specifications v1.1. 2015.

SAA^{+}. Marcos A Simplicio, Leonardo C Almeida, Ewerton R Andrade, Paulo C F Dos Santos, and Paulo S L M Barreto. Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs.

yes14. yescrypt -a Password Hashing Competition submission. Technical report, 2014.

Listing 1.3 Metoda obliczająca hash zależny od: parametru bezpieczeństwa λ (layer), soli (salt) i numeru rundy (round).

saltedChoices

```
string RiffleShuffle::saltedChoices(int layer, int round, string salt) {
    int numberOfBlocks = (int) (ceil(this->N/256));

    string choices;

    for (int i=0; i <= numberOfBlocks; i++) {
        string saltChoices;
        saltChoices = "RSfree:" + to_string(this->N) + ":" +
            to_string(this->pepper) + ":" +  to_string(layer) + ":" +
            to_string(round) + ":" + salt + ":" +
            to_string(numberOfBlocks) + ":" + to_string(i);
        choices = sha256(saltChoices);
    }

    return choices;
}
```

Listing 1.4 Implementacja *RiffleSchuffle** – uproszczonej wersji RiffleSchuffle

RiffleShuffle*

```
string RiffleShuffle::scramble(string password, string salt, int pepper) {
    if (this->pepper < pepper)        this->pepper = pepper;
    this->pepper += 2;
    vector <string> bufferOld(this->N);
    vector <string> bufferNew(this->N);
    //initialization of the buffer:
    string initVal = password + ":" + salt;
    bufferOld[0] = sha256(initVal);
    for (int i = 1; i < this->N; i++) {
        string stringToHash;
        stringToHash =  "RSfreeInit:" + to_string(this->N) + ":" +  to_string(this->pepper) +  ":0
        bufferOld[i] = sha256(stringToHash);
    }
    vector <int> leftParents(this->N);
    vector <int> rightParents(this->N);
    //actual RiffleScrambler computation
    //outer loop
    for (int layer = 0; layer < this->pepper; layer++) {
        //inner loop
        for (int round = 0; round < this->lambda; round++) {
            int leftCounter = 0;
            string choices = saltedChoices(layer, round, salt);
            //first pass to learn how many zeros we have
            for (int i = 0; i < this->N; i++) {
                if (!isSet(choices, i))
                    leftCounter++;
            }
            int noOfZeros = leftCounter;
            int noOfOnes = this->N - leftCounter;
            leftCounter = 0;
            int rightCounter = 0;
            //buffer loop
            for (int i = 0; i < this->N; i++) {
                if (!isSet(choices, i)) {
                    leftParents[leftCounter] = i;
                    rightParents[noOfOnes + leftCounter] = i;
                    leftCounter++;
                } else {
                    leftParents[noOfZeros + rightCounter] = i;
                    rightParents[rightCounter] = i;
                    rightCounter++;
                }
            }
            //compute values at the next level:
            for (int i = 0; i < this->N; i++) {
                string toNewHash;
                toNewHash = bufferOld[leftParents[i]] + bufferOld[rightParents[i]];
                bufferNew[i] = sha256(toNewHash);
            }
            for (int i = 0; i < this->N; i++) {
                bufferOld[i] = bufferNew[i];
            }
        }
    }
    this->result = bufferNew[((this->N)-1)];
    return bufferNew[((this->N)-1)];
}
```