SEBASTIAN CZARNOTA

Tęczowe tablice – przyśpiesz atak brute-force

Stopień trudności

Tęczowe tablice pozwalają przyspieszyć ataki brute-force, eliminując potrzebę wykonywania w każdym kolejnym ataku tych samych obliczeń poprzez zapamiętani części kluczowych danych. Tablice znalezione w Internecie bywają jednak niekompletne lub płatne, dlatego zobaczmy jak zrobić je samemu!

Każdy doskonale wie, że przechowywanie haseł w postaci jawnej nie jest bezpieczne. Przyklejanie haseł do monitora, czy wkładanie pod klawiaturę jest zmorą osób odpowiedzialnych w firmach za bezpieczeństwo. Takie eksponowanie haseł jest wręcz proszeniem się o kłopoty.

W systemach komputerowych jest podobnie. Przechowywanie haseł w postaci jawnej jest niebezpieczne, zwłaszcza gdy hasła te przechowuje system operacyjny, który zajmuje się przydzielaniem praw użytkownikom.

Najprostszym sposobem rozwiązania tego problemu wydaje się zaszyfrowanie hasła za pomocą szyfru symetrycznego. Jednak pojawia się problem: kryptosystem symetryczny do szyfrowania wymaga... hasła, które również trzeba byłoby w sposób bezpieczny przechować w systemie operacyjnym. Nie rozwiązuje to jak widać problemu, ale go pogłębia.

Funkcje skrótu

Dlatego postanowiono podejść do problemu w inny sposób. Użyto funkcji skrótu. Funkcja skrótu (funkcja haszująca, ang. *hash function*) przyjmuje jako parametr dowolny ciąg binarny, a jako wynik swojego działania zwraca inny ciąg binarny o następujących cechach:

- ciąg wynikowy (nazywany skrótem) jest zależny od ciągu wejściowego (zwany również wiadomością) i jest on deterministyczny, tj. jeden ciąg wejściowy zawsze da taki sam ciąg wynikowy;
- ciąg wynikowy (skrót) wydaje się być ciągiem losowym – zależność między ciągiem wejściowym (wiadomością) i skrótem jest tak skomplikowana, że nie jesteśmy w stanie odtworzyć ciągu wejściowego na podstawie ciągu wynikowego – mówi się, że taka funkcja jest jednokierunkowa;
- skrót ma ustaloną długość w bitach (MD5 – 128 bitów, SHA-0 – 160 bitów, SHA-1 – warianty 224, 256, 384, 512 bitowe) (Rysunek 1).

Warto w tym miejscu wtrącić dygresję na temat nazwy funkcja *skrótu* (ang. *message digest*), ponieważ, jeśli *skracamy* na przykład hasło *tajne_haslo* to jego skrót MD5 wynosi *1b955 38f3035f4cba2189716fd96173c*, a więc jest dłuższy. Nazwa ma swoje uzasadnienie w tym, że funkcje skrótu projektowane były dla tworzenia krótkiego odpowiednika dużego dokumentu. Skrót przydatny jest w procesie cyfrowego podpisywania dokumentu, ponieważ działanie algorytmów podpisywania jest bardzo powolne, a czas rośnie wraz ze zwiększeniem objętości dokumentu, dlatego podpisuje się jedynie skrót tego dokumentu. Do przechowywania haseł wykorzystano istniejące już rozwiązanie zwane funkcją skrótu, a nazwa już pozostała.

Z ARTYKUŁU DOWIESZ SIĘ

do czego służą funkcje skrótu,

jak wykorzystać tęczowe tablice,

jak zaprojektować skuteczną tęczową tablicę.

CO POWINIENEŚ WIEDZIEĆ

mieć ogólne pojęcie o przechowywaniu i przesyłaniu haseł.

Czasem stosuje się określenie funkcja haszujące. Osoby zajmujące się tym zagadnieniem nie potrafią dojść do porozumienia, która nazwa jest poprawna, o czym można przeczytać w Wikipedii, w dyskusji do tematu *Funkcje skrótu*. Polskie normy definiują jednak pojęcie *funkcji skrótu*, dlatego będę się tym terminem posługiwał w dalszej części artykułu.

Cechą funkcji skrótu jest występowanie *kolizji* – czyli sytuacji, gdy skróty dwóch różnych wiadomości są identyczne. Cecha ta jest bezpośrednim następstwem ustalenia długości skrótu – ponieważ skoro można skracać (poddawać działaniu funkcji skrótu) wiadomość o dowolnym rozmiarze, do ciągu o z góry określonej długości, to muszą istnieć co najmniej dwie takie wiadomości, które dają identyczny skrót (działa tutaj twierdzenie zwane *Zasadą szufladkową Dirichleta*) (Rysunek 2).

W kryptologii występowanie kolizji jest cechą niepożądaną, ponieważ statystycznie znalezienie dowolnej wiadomości, która daje określony skrót jest bardziej prawdopodobne, niż znalezienie dokładnie tej wiadomości, której użyto do wygenerowania skrótu (dzieje się tak dlatego, gdyż istnieje więcej niż jeden ciąg, dający ten sam skrót). Oznacza to, że aby zalogować się do systemu, który przechowuje wyłącznie skróty haseł użytkowników, można użyć nie tylko hasła, które użytkownik sobie wybrał, ale również wszystkich innych haseł, których skrót jest identyczny. Atak kryptoanalityczny wykorzystujący tą cechę funkcji skrótu nazywa się poszukiwaniem przeciwobrazu (ang. *preimage attack*).

Kolizje w funkcjach skrótu będą zawsze występowały. Prostym sposobem na obniżenie prawdopodobieństwa ich wystąpienia jest wydłużenie skrótu. Zwiększa to ilość różnych skrótów, zmniejszając szansę na wystąpienie kolizji. Aktualnie następuje zmiana standardowej długości skrótu i zastępowanie starych, niebezpiecznych funkcji skrótu nowymi. Właśnie trwa konkurs na nowy standard funkcji skrótu SHA-3 organizowany przez amerykańską agencję federalną NIST (ang. *National Institute of Standards and Technology*).

Wykorzystanie funkcji skrótu

Spójrzmy teraz jak funkcje skrótu rozwiązują problem przechowywania hasła. Gdy użytkownik loguje się do systemu jest proszony o podanie swojego hasła. System operacyjny interpretuje hasło użytkownika jako ciąg binarny i poddaje go działaniu funkcji skrótu (np. MD5). Teraz to ten skrót jest porównywany z przechowywanym skrótem hasła w systemie. Jeśli oba skróty są takie same, następuje zalogowanie do systemu, jeśli nie, system odmówi użytkownikowi dostępu.

Znając już zjawisko kolizji, wiemy, że poprawne zalogowanie do systemu może wystąpić również wtedy, gdy użytkownik poda dowolne inne hasło, o takim samym skrócie. Wydaje się to niebezpieczne, ale pamiętajmy, że funkcja skrótu jest funkcją jednokierunkową i znalezienie hasła na podstawie skrótu jest obliczeniowo trudne. Trudne, ale nie niemożliwe.

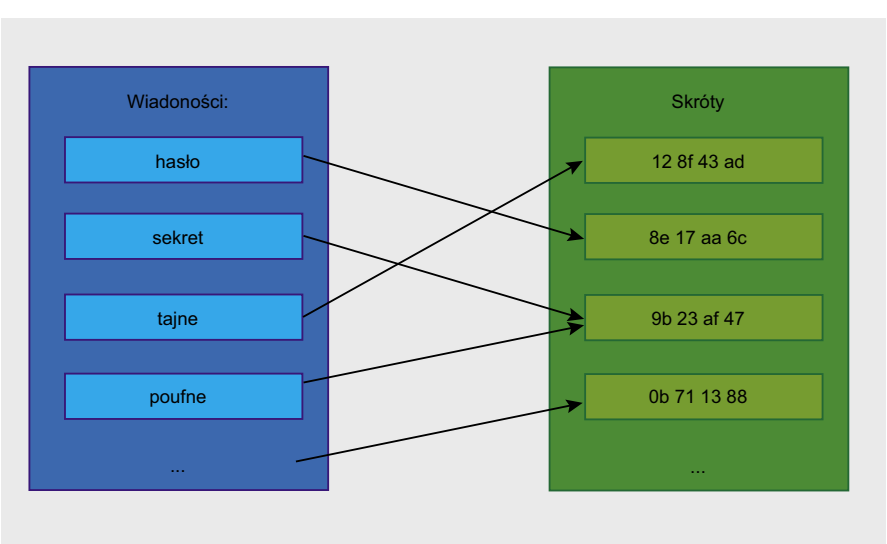
Zanim przejdziemy do omówienia ataków na funkcje skrótu, zauważmy jeszcze kilka interesujących faktów. Obliczenie skrótu wiadomości powinno być szybkie i wydajne, natomiast proces odwrotny musi być trudny obliczeniowo. Kryptologia określa procesy trudne obliczeniowo jako takie, których rozwiązanie, przy użyciu najlepszych znanych algorytmów i najlepszych komputerów na świecie, biorąc pod uwagę wzrost ich wydajności w czasie, zajmie tak niewyobrażalnie długi czas, że atak jest niepraktyczny. Niczym dziwnym jest, że złamanie pewnych algorytmów kryptograficznych szacuje się na miliony lat.

Trzeba jednak pamiętać, że przy obecnym postępie technicznym dostępna moc obliczeniowa ciągle wzrasta. Jak również stale ulepsza się i wynajduje nowe algorytmy oraz następuje ciągły rozwój matematyki wyższej, która ma niebanalny wpływ na wydajność ataków szyfry i funkcje skrótu. Dlatego tak ważne jest ich ciągłe udoskonalanie: zarówno mechanizmów, według których działają, jak i długości skrótu. Jest to o tyle ważne, że nawet najlepiej zaprojektowana funkcja skrótu zostanie szybko złamana, gdy jej skrót będzie zbyt krótki.

Klasyczne ataki na skróty haseł

W dalszej części artykułu będziemy rozpatrywać atak polegający na odzyskaniu hasła, z jego skrótu. Skróty haseł są używane w czasie logowania się zdalnie, w przypadku obecności specjalnego klienta. Użytkownik wtedy podaje swoje hasło w aplikacji klienckiej, która skraca to hasło i w takiej postaci przesyła do serwera (to jest znaczne uproszczenie, gdyż napastnik mógłby zrezygnować z użycia programu klienta, a tylko przechwycić i przesłać sam skrót – w rzeczywistości stosuje się bardziej zaawansowane protokoły, które uniemożliwiają takie nadużycie).

Hasła systemów operacyjnych również są przechowywane w postaci



Rysunek 1. *Idea działania funkcji skrótu*

skrótów. Jednak zdobycie pliku, który przechowuje skróty, często wiąże się z wtargnięciem do systemu ofiary, więc odzyskiwanie hasła już nie jest potrzebne. Wyjątkiem jest sytuacja, gdy administrator systemu popełni błąd (lub zostanie do niego sprowokowany) i uda się bez włamywania na konto administratora uzyskać ten plik. Metody dalej opisane mogę wtedy zostać wykorzystane do eskalacji uprawnień w systemie i to w bardzo bezpieczny sposób, bo offline.

Dwa najprostsze pomysły ataku na funkcje skrótu nasuwają się chyba każdemu:

- sprawdzanie wszystkich możliwych haseł – skracanie ich i testowanie, czy dają pożądany skrót;
- zbudowanie ogromnej tablicy wszystkich haseł i odpowiadających im skrótów, posortowanie według skrótów i wyszukiwanie w tablicy.

Są to metody pewne, czyli gdyby udało się je przeprowadzić, istnieje stuprocentowa szansa na poprawne znalezienie hasła. Jednak w rzeczywistości ich pełne przeprowadzenie wymaga potężnej mocy obliczeniowej w pierwszym przypadku i niewyobrażalnej przestrzeni dyskowej w drugim. Warto w tym miejscu podać pewne liczby, które uzmysłowią skalę problemu. 128-bitowa funkcja skrótu (to znaczy taka, której skrót ma 128-bitów, wiadomość może być dowolnej długości) posiada 2^128 różnych skrótów, daje to 6*10^23 petabajtów danych potrzebnych na przechowanie tych wszystkich skrótów. Jest to mniej więcej tyle petabajtów danych, ile wynosi szacowana liczba gwiazd we wszechświecie. A to tylko skróty! Drugie tyle zajmą hasła im odpowiadające, a czas potrzebny, do wygenerowania takiej tablicy będzie liczony w milionach lat.

Należy przy okazji sprostować, że są to wyliczenia teoretyczne, zakładające, że funkcja skrótu jest wyidealizowana w taki sposób, że nie da się jej budowy wewnętrznej użyć do znalezienia hasła, oraz że wykorzystuje wszystkie dostępne skróty. Rzeczywistość jest zupełnie inna. To właśnie analiza budowy funkcji skrótu (i algorytmów kryptograficznych w ogólności) daje najlepsze kryptoanalityczne efekty. Jeszcze lepsze efekty uzyskuje się, gdy znaleziony zostanie błąd w implementacji algorytmu w konkretnej realizacji programowej, ale na to nie należy liczyć, gdyż błąd może być naprawiony.

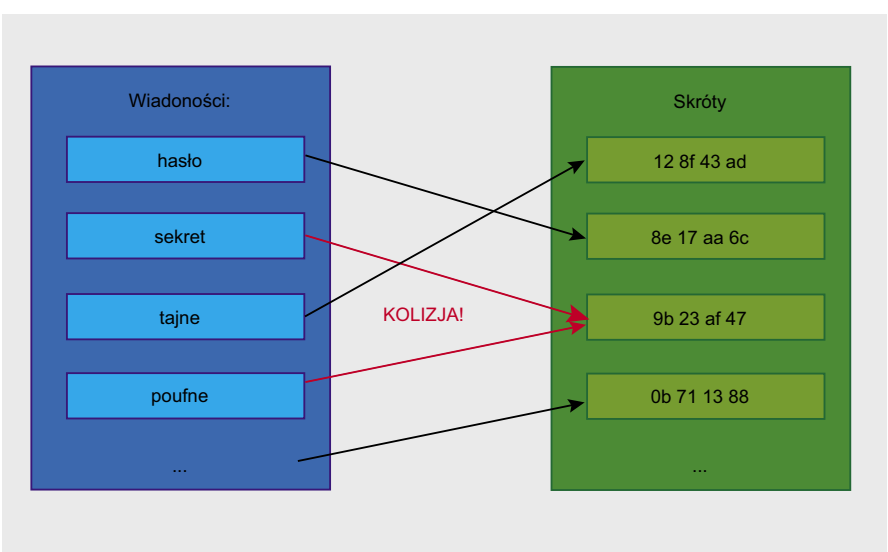
Kolejnym faktem jest, że nie musimy budować tablicy, dla wszystkich możliwych skrótów, ponieważ interesują nas jedynie hasła. Ilość wszystkich haseł do 10 znaków włącznie jest znacznie mniejsza, niż liczba wszystkich skrótów, dla funkcji 128-bitowej. Dokładne wyliczenia zależą od przyjętego zestawu znaków.

Tęczowe tablice

Tęczowe tablice to pomost między przytoczonymi przeze mnie dwoma metodami ataku. Łączy w sobie zaletę przechowywania wyników, umożliwiającą wygenerowanie tablicy raz i korzystanie z niej w każdym następnym ataku oraz zmniejsza objętość tablic. Następuje bardziej racjonalne wykorzystanie mocy obliczeniowej i pamięci.

Wyobraźmy sobie listę ciągów binarnych. Pierwszym elementem tego ciągu jest pewne hasło ze słownika haseł. Drugim elementem jest skrót tego hasła. Kolejnymi elementami są: pewne hasło, wygenerowane na podstawie poprzedzającego skrótu i skrót tego hasła. Dalej podobnie, hasło wygenerowane na podstawie poprzedzającego skrótu i jego skrót, itd. Należy sobie uzmysłowić, że hasła generowane na podstawie skrótów, to nie są hasła, które po skróceniu dają ten skrót, są to po prostu jakieś hasła zależne od skrótu (Rysunek 3).

Generowaniem hasła ze skrótu zajmuje się tak zwana funkcja redukująca. Może ona mieć dowolną postać, ale warto, by zapewnić, aby była w stanie wygenerować możliwie najwięcej różnych haseł. Najprostszą



Rysunek 2. *Kolizja – dwie wiadomości mają jednakowy skrót*

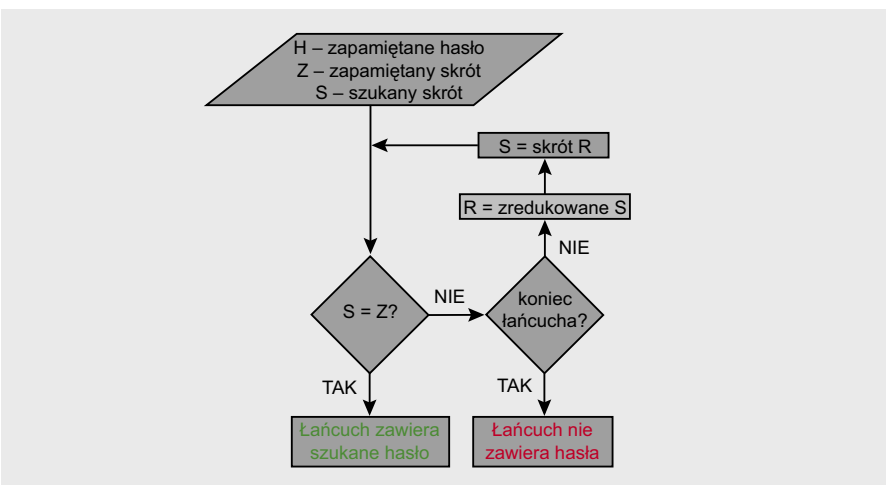
Rysunek 3. *Łańcuch – podstawowa jednostka przechowywania danych w tęczowej tablicy. H – funkcja skrótu, R – funkcja redukująca*

funkcją redukującą może być funkcja, która będzie binarną reprezentację skrótu zamieniać na znaki traktując je jak kod ASCII.

Zapisujemy sobie początkowe hasło takiego łańcucha i ostatni skrót. Pośrednie hasła i skróty nie będą zapisywane, bo jak się za chwilę okaże, nie są potrzebne. Przydatna będzie jedynie informacja o liczbie par w całym łańcuchu. Taka para: hasło i odległy skrót jest bardzo interesująca.

Wyobraźmy sobie, że chcemy znaleźć hasło odpowiadające skrótowi. Najpierw musimy się upewnić, że szukane hasło odpowiadające skrótowi znajduje się w naszym łańcuchu. Ponieważ nie zapisaliśmy żadnych wartości pośrednich, będziemy opierać się na zapamiętanym, odległym skrócie. Postępujemy zgodnie z algorytmem jak na Rysunku 4.

1. H – przyjmujemy za H szukany skrót,
2. sprawdzamy, czy H jest równy końcowemu skrótowi łańcucha,
3. jeśli nie, sprawdzamy, czy ilość skracań i redukcji jest większa niż długość łańcucha,
4. jeśli nie, to poddajemy nasz skrót H działaniu funkcji redukującej, otrzymujemy w ten sposób hasło.
5. otrzymane hasło skracamy otrzymując inny skrót, ten skrót zapamiętujemy w zmiennej H,
6. przechodzimy do kroku 2,
7. gdy H jest równy końcowemu skrótowi łańcucha – łańcuch posiada szukane hasło,
8. gdy ilość skracań i redukcji przekroczyła długość łańcucha – łańcuch nie posiada hasła.



Rysunek 4. *Schemat sprawdzania, czy łańcuch zawiera skrót. Należy pamiętać, że redukowanie i skracanie wykonujemy tylko tyle razy, ile wynosi długość łańcucha (zapamiętana w czasie tworzenia)*

Jeśli w kroku drugim okaże się, że któreś H jest równe końcowemu skrótowi, to mamy pewność, że w ciągu znajduje się hasło do naszego skrótu. Jeśli natomiast wykonamy krok 3 więcej razy niż wynosiła liczba par w całym łańcuchu, to zaprzestajemy poszukiwań, bo łańcuch hasła nie zawiera (Rysunek 5).

Gdy już wiemy, że łańcuch zawiera hasło, przystępujemy do jego odzyskiwania. W tym celu bierzemy z pary zapamiętanej dla ciągu hasło i poddajemy go cyklicznym skracaniom i redukcjom. Kiedy natrafimy na poszukiwany skrót, to poprzednie hasło jest hasłem, które szukaliśmy!

Pewnie niezrozumiałe jest teraz, dlaczego od razu nie można było cyklicznie skracać i redukować. Już śpieszę z wyjaśnieniem. Łańcuchów jest wiele, nawet bardzo wiele. Im więcej tym lepiej, gdyż mamy większe prawdopodobieństwo, że szukane hasło

jest w naszym zbiorze. Cała siła tęczowych tablic zawiera się właśnie w ilości łańcuchów i ich długości (Rysunek 6).

Długość łańcucha to nic innego jak ilość cyklicznych skracań i redukcji od pierwszego hasła do ostatniego skrótu. Im dłuższe łańcuchy, tym mniej miejsca zajmą tablice przy tej samej liczbie haseł. Jednocześnie im dłuższe tablice, tym więcej mocy obliczeniowej będzie potrzebne do wyszukiwania w tablicy. Należy więc znaleźć złoty środek.

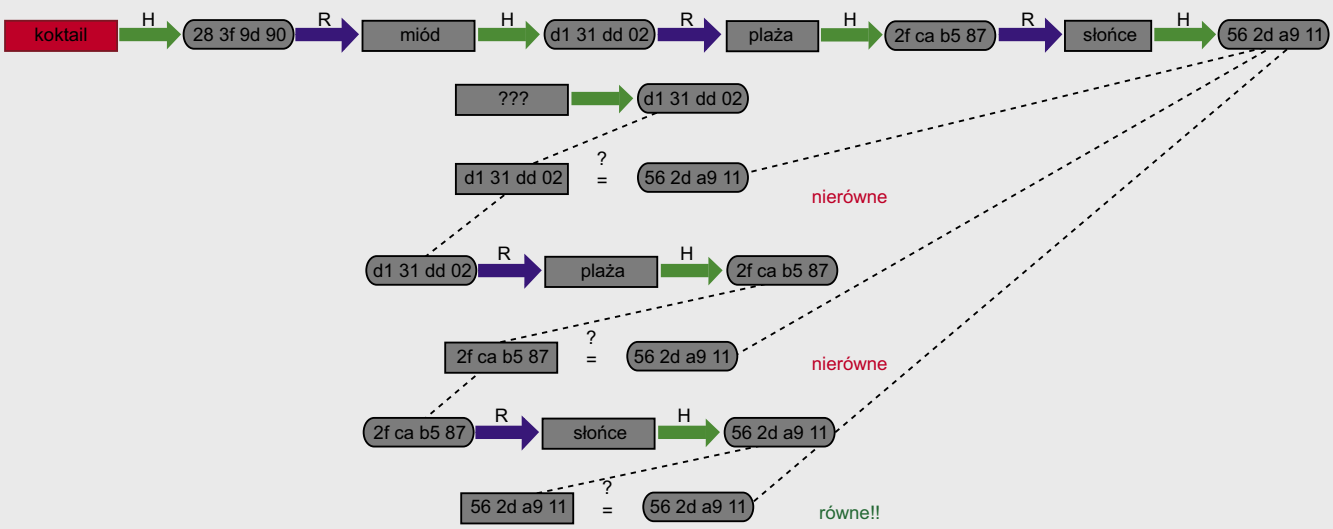
Kolizje i pętle w tęczowych tablicach

Wydaje się, że nasze tęczowe tablice są już gotowe do pracy. Zastanówmy się jednak, co by się stało, gdybyśmy obliczając pewien łańcuch natrafili na hasło, które już w innym łańcuchu zostało zawarte. Ponieważ funkcja redukująca jest jednakowa, wszystkie następujące później hasła i skróty powtórzyłyby się! Byłoby to ogromne marnotrawstwo miejsca na dysku i mnóstwo zbędnych obliczeń. Gdyby kolizje występowały często, sporo danych byłaby zdublowana i tylko niepotrzebnie spowalniałaby użycie tęczowych tablic (Rysunek 7).

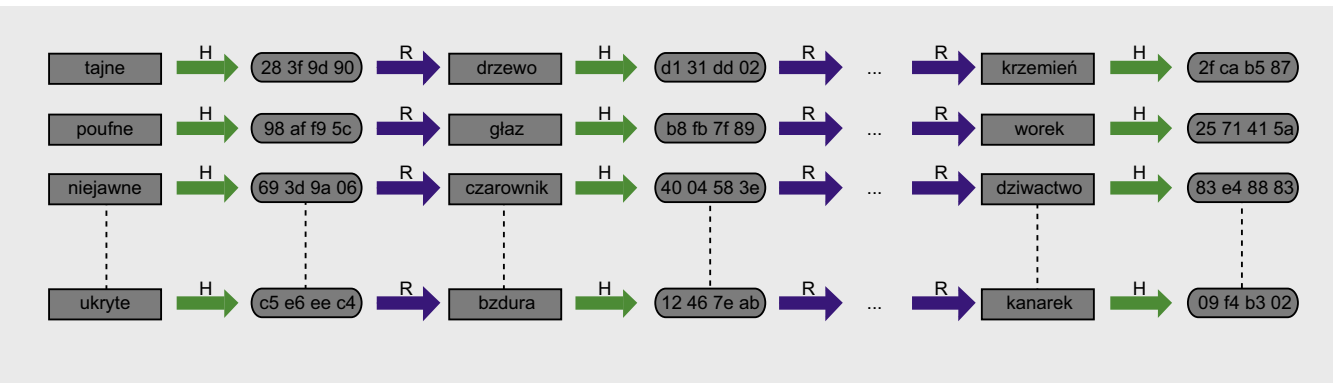
To ciekawe, że kolizje, które są zjawiskiem niepożądanym w funkcjach skrótu, utrudniają ataki metodą tęczowych tablic. Okazuje się, że funkcja skrótu o licznych kolizjach sparaliżowałaby działania tęczowej tablicy! Na domiar złego, gdyby w tym samym ciągu jakieś hasło wystąpiło dwa razy, to otrzymalibyśmy zapętlenie – w równych odstępach pojawiałyby się sekwencje tych samych skrótów i haseł (Rysunek 8).

Czy więc jesteśmy skazani na tak niedoskonałe narzędzie? Okazuje się, że poradzono sobie w bardzo sprytny sposób z tymi problemami, poprzez wprowadzenie wielu różnych funkcji redukujących. Różnych, czyli takich które dla takiego samego skrótu zwracają różne hasła. Przychodzi nam zapewne na myśl kilka sposobów ulokowania tych różnych funkcji redukujących. Trzy główne z nich to:

- użycie za każdym razem innej funkcji – bardzo nieefektywny sposób, gdyż potrzeba przechowywania ogromnej ilość funkcji redukujących przekracza wielokrotnie rozmiar samej tęczowej tablicy;



Rysunek 5. *Przykład wyszukiwania hasła w jednym łańcuchu. W tablicy wykonujemy te porównania z wieloma skrótami kończącymi łańcuchy*



Rysunek 6. *Tęczowa tablica – w poziomie widzimy łańcuchy, a w pionie kolejno od lewej każda para stanowi warstwę*

- użycie w każdym łańcuchu innej funkcji redukującej – patrząc na tablicę, jednakowe funkcje redukujące byłyby ułożone w poziomie;
- użycie w każdym łańcuchu, każdej funkcji redukującej dokładnie jeden raz – takie ułożenie da pionowe ułożenie jednakowych funkcji (ułożenie w warstwy).

Rozważmy dwa ostatnie sposoby. Gdyby każdy łańcuch posiadał inną funkcję redukującą, rozwiązany zostałby definitywnie problem kolizji między łańcuchami. Nawet gdyby natrafiono na takie same hasło, które występowało w innym łańcuchu, to kolejne hasła i skróty były różne, gdyż już w kolejnym kroku ze skrótu tego *kolizyjnego* hasła powstałyby różne skróty. Nie rozwiązuje to jednak problemu pętli, ponieważ w ramach jednego łańcucha funkcja jest jednakowa.

Podejście to ma jeszcze inne wady. Pierwszą jest konieczność wygenerowania funkcji redukującej dla każdego nowego łańcucha. Jest to z różnych względów niepraktyczne, gdyż należy wtedy wraz ze wzrostem liczby łańcuchów pamiętać coraz więcej funkcji redukujących. Natomiast gdy chcielibyśmy wyszukiwać w tablicy, każda ta funkcja musiałaby być wykonana na poszukiwanym skrócie, wynik ponownie skrócony i dopiero wówczas porównany z przechowywaną wartością. Powstaje więc duży narzut zarówno pamięciowy, jak i obliczeniowy, bo trzeba byłoby dla każdej warstwy tablicy wykonywać tyle redukcji i skróceń, ile jest wszystkich łańcuchów i każdy z wyników porównać z zapamiętanym skrótem.

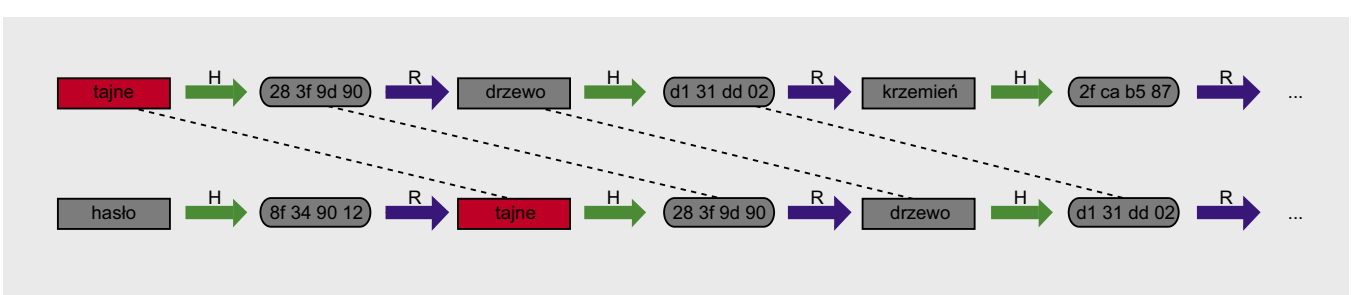
Dlatego w tęczowych tablicach stosuje się to drugie podejście. Używa się tylu funkcji redukujących, ile wynosi długość łańcucha. Każda redukcja w łańcuchu jest inna, jednak kolejność redukcji w każdym z nich jest jednakowa. Zauważmy teraz cechy takiego podejścia. Zjawisko pętli jest całkowicie wyeliminowane, gdyż każda funkcja redukująca jest inna i jest bardzo mała szansa, że choćby kilka haseł i skrótów się powtórzy. Nadal może dojść do kolizji, ale jedynie wtedy spowoduje ona powtórzenie wszystkich kolejnych haseł i skrótów, gdy identyczne hasło znajduje się na tej samej pozycji w łańcuchu. Nietrudno obliczyć, że szansa na taką kolizję zmniejsza się diametralnie wraz ze wzrostem długości łańcucha.

To ostatnie podejście do doboru funkcji redukujących ma szereg zalet:

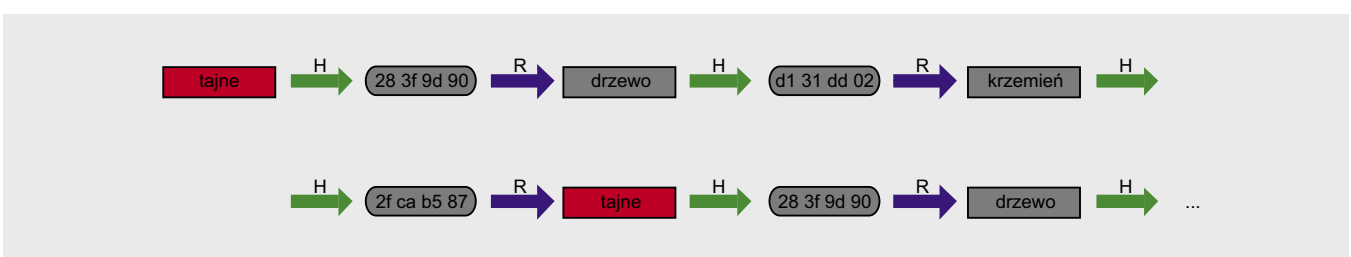
- W czasie przeszukiwania jednej warstwy, potrzeba wykonać jedną redukcję i jedno skracanie oraz wyszukanie ciągu w liście zapamiętanych skrótów. Zyskujemy gigantyczny wzrost szybkości tej operacji, natomiast operacja wyszukiwania może być bardzo szybka, gdy lista skrótów zostanie posortowana w porządku rosnącym czy malejącym;
- Proces generowania takich tablic można bardzo wydajnie zaimplementować na nowoczesnych procesorach, ponieważ nietrudno zauważyć, że wygenerowanie każdego łańcucha będzie wymagało identycznych operacji, a różnica będzie jedynie w danych początkowych. Istnieją specjalne rozkazy popularnych procesorów do pracy w trybie SIMD (ang. *Single Instruction Multiple Data*), z którym mamy tutaj do czynienia. Rozkazy te są częścią między innymi zestawu instrukcji SSE (ang. *Streaming SIMD Extensions*);
- Na początku tworzenia tęczowej tablicy tworzymy listę funkcji redukujących, następnie generujemy łańcuchy dbając tylko o różne hasła początkowe, co nie nastręcza większych problemów, gdyż są one zapamiętywane w tęczowej tablicy;
- Możemy rozszerzyć zbiór funkcji redukujących już po utworzeniu tęczowej tablicy i również wtedy skorzystać z zalet trybu SIMD.

O funkcjach redukujących

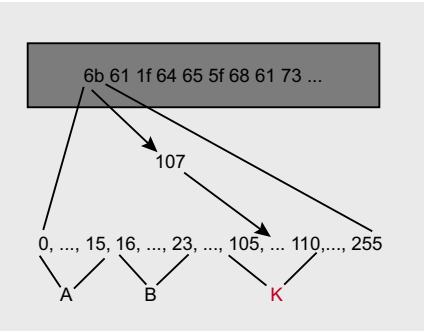
Funkcje redukujące są kluczowym elementem tęczowych tablic. Od ich jakości zależy, czy nasze tablice uwzględnią prawdopodobne hasła czy



Rysunek 7. *Przykład wystąpienia kolizji. Od haseł zaznaczonych na czerwono dalsze części łańcuchów są identyczne*



Rysunek 8. *Przykład pętli. Ciągłe, cykliczne występowanie kilku haseł i skrótów*



Rysunek 9. *Wykorzystanie zakresów do wyboru znaku, ze zmniejszonego zestawu znaków. 6B szesnastkowo to 107 dziesiętnie, 107 wpada do zakresu od 105 do 110, który odpowiada literze K. Bajt 6B zostanie zamieniony na literę K*

całkowicie losowe ciągi znaków, których właściwie nikt nie użyje jako hasło (nikt z wyjątkiem osób korzystających z generatorów pseudolosowych). To od jakości funkcji redukujących zależy jakość całej tablicy.

Zastanówmy się nad prostą funkcją redukującą – funkcja, która bierze pierwszych 10 znaków funkcji skrótu i zwraca je jako hasło.

```
427842A8A984FE86775D679D8061FECB →
                          427842A8A9
```

Czy to jest dobra funkcja redukująca? Zdecydowanie nie, ponieważ dla tej funkcji jest tak naprawdę niewiele haseł, które może wygenerować spośród wszystkich możliwych haseł. Poza tym hasła przez nią zwracane składają się tylko z 16 znaków (10 cyfr i 6 liter od A do F). Powinniśmy poszukać lepszej funkcji. Weźmy teraz funkcję, która traktuje bajty skrótu, jako kody ASCII.

```
427842A8A984FE86775D679D8061FECB →
                          BxB?????w]g??a??
```

Co tutaj się wydarzyło? Dlaczego tyle razy powtarza się znak zapytania, chociaż jego kod ASCII to 3F i nie występuje on w tym skrócie ani razu? Dzieje się tak dlatego, gdyż tabela kodów ASCII zawiera 256 elementów i składa się nie tylko z liter i cyfr, ale również z symboli i kodów sterujących, takich jak powrót karetki, nowa linia, czy sygnał dźwiękowy oraz ze znaków białych, takich jak spacja czy tabulacja. W tym ciągu reprezentowane są jako znaki zapytania. Wybrana funkcja redukująca może wygenerować każde możliwe hasło

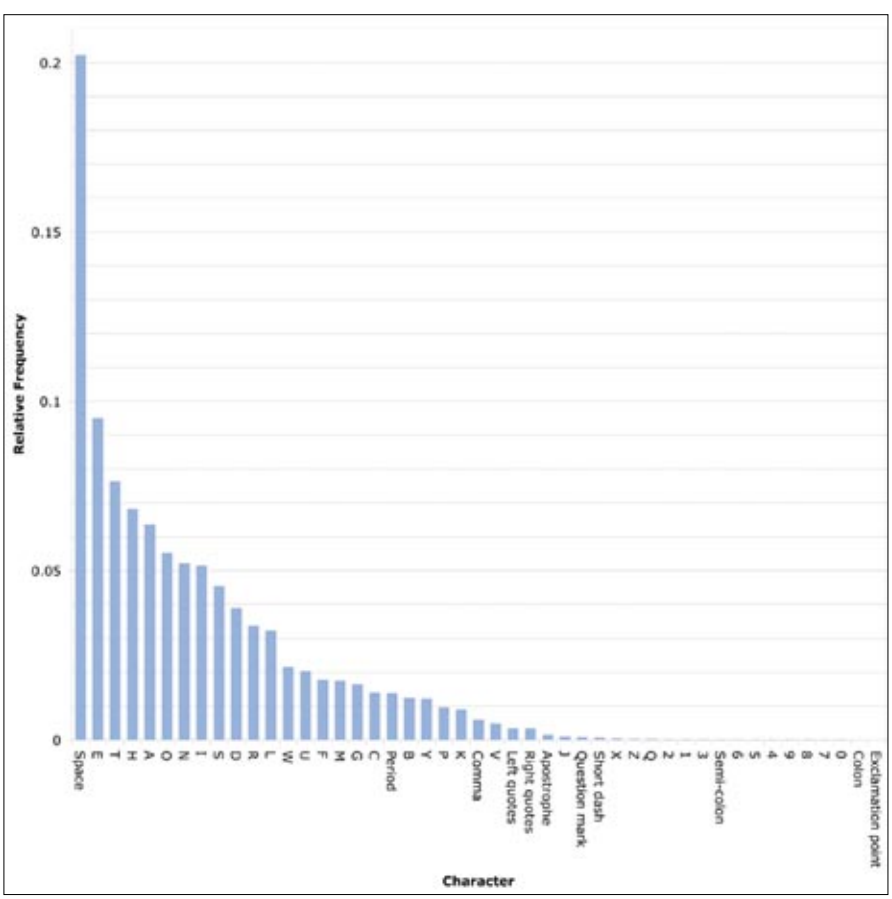
```
6b611f64655f6861731f6f5f64616a1f →
                          każde_hasło_daję
```

ale większość stanowić będą te, złożone właśnie ze znaków niedrukowalnych, gdyż jest ich po prostu więcej w tabeli ASCII. Warto tutaj wspomnieć, że wszystkie znaki ASCII można otrzymać poprzez kombinację [*ALT*]+ kod znaku w postaci dziesiętnej wpisany z klawiatury numerycznej. Nie potrzeba sięgać po żadne dodatkowe narzędzie.

Jednak jeśli nie potrzebujemy, aby nasza tablica skupiała się na tak wyszukanych hasłach, zaprezentowana powyżej funkcja redukująca jest zbyt *rozrzutna*, ponieważ równie często będzie sięgała po znaki alfanumeryczne, jak i po inne.

W popularnych programach łamiących hasła metodą pełnego przeglądu (ang. *brute force*), często mamy możliwość wybrania zestawu znaków, jakie mają być testowane. Każdy kto używał takiego programu i spędził z nim kilka chwil, z pewnością zauważył, że zwiększenie zestawu znaków wydłuża czas poszukiwania, to naturalne i w przypadku tęczowych tablic, też tak się dzieje. Dlatego należy umiejętnie wybrać zestaw znaków, który nasze funkcje redukujące będą używać, ponieważ od tego zależy skuteczność i szybkość działania tablicy.

Warto chwilę się zastanowić, jakich znaków sami używamy w hasłach i być może na tej podstawie wybrać interesujący nas zestaw. Zestawy najczęściej stosowane bazują na dużych i małych literach oraz cyfrach, bardziej rozszerzone zawierają dodatkowo symbole dostępne po naciśnięciu [*SHIFT*] i cyfry (!, @, #, …), spacja, nawiasy, polskie znaki diakrytyczne, a w skrajnych wypadkach udostępniają nawet kody ASCII o wysokich wartościach, dostępne przez kombinację [*ALT*]+ kod ASCII. Należy wybrać złoty środek między skutecznością a szybkością działania.



Rysunek 10. *Częstość występowania znaków w języku angielskim*

Jak zredukować skrót do hasła, które używa zmniejszony zestaw znaków?

Rozwiązań jest kilka. Można na przykład wybrać zakresy wartości bajtów,

odpowiadających określonym literom, np. gdy wartość bajtu wynosi od 0 do 15 można przyjąć, że zamieniamy wtedy taki bajt na *A*, wartości z zakresu 16 do 23 zamieniamy na *B* itd. Używanie takiej funkcji redukującej spowoduje, że podobne skróty będą redukowane do takich samych haseł. Nie jest to cecha niepożądana, jednak ze względów wydajnościowych może lepsze byłoby zastosowanie jakiegoś wzoru przekształcającego, niż tabeli z zakresami (Rysunek 9).

Innym pomysłem może być zastosowanie właśnie wzoru, do powyższego pomysłu. Możemy na przykład przeskalować cały zakres znaków ASCII (od 0 do 255) do naszego zmniejszonego zbioru, np. 70 znaków. W tym celu, dzielimy `256 : 70 = 3,7` i co `3,7` wartości bajta w skrócie wybieramy kolejny znak z naszego zbioru. Można wymyślić całkiem sporo ciekawych metod takiego rozszerzania, zachęcam do eksperymentowania!

Co jeszcze można ulepszyć?

Jesteśmy coraz bliżej stworzenia doskonałych funkcji redukujących, jednak nie wzięliśmy pod uwagę częstości występowania znaków. Oczywiste jest, że w języku polskim, czy angielskim znaki w tekstach występują nierównomiernie. W języku polskim litera *a* występuje najczęściej, natomiast w angielskim, litera *e*. Co prawda hasła nie zawsze składają się ze słów, jednak warto zastosować rozróżnienie występowania znaków, bo, zastanówmy się sami, jak często używamy w hasłach znaków w stylu !, @, #, $, (,),...? Jestem pewien, że większość użytkowników nie stosuje tych znaków bardzo często. Zdarza się nawet, że używamy słów z języka polskiego (Rysunek 10).

Świetnym podejściem do tego zagadnienia może być przeanalizowanie listy haseł jakiegoś działającego systemu z dużą liczbą użytkowników i zebranie potrzebnych danych statystycznych na temat znaków. Dane statystyczne występowania znaków dla języka można znaleźć w literaturze, a nawet łatwo samemu zebrać, natomiast listy haseł powinny być (i są) chronione przed dostępem osób nieupoważnionych, dlatego raczej nie mamy do nich dostępu.

Rozróżnianie częstotliwości występowania znaków można zrealizować wykorzystując zakresy wartości bajtów funkcji skrótu. Dla znaków o dużym prawdopodobieństwie zakres będzie proporcjonalnie większy.

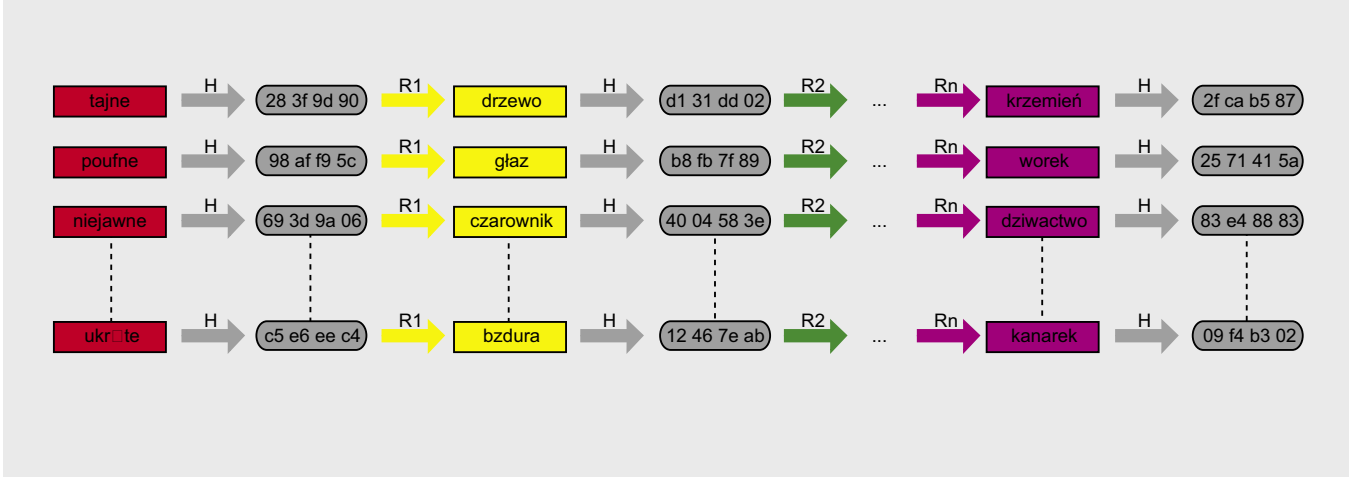
Jak otrzymywać różnej długości hasła?

Hasła często różnią się długościami. Funkcje redukujące muszą redukować skróty do haseł różnej długości. Możemy do tego podejść dwojako. Możemy ustalić, że jedna funkcja redukująca zawsze zwraca hasła o konkretnej długości albo uzależnić długość hasła od skrótu.

W pierwszym przypadku stworzymy sobie zbiory funkcji redukujących, które zawsze zwracają określonej długości hasła. Jest to podejście wystarczające, możemy wtedy nawet regulować ilość wygenerowanych haseł o danej długości poprzez zwiększenie funkcji w danym zbiorze. Warto rozważyć tutaj pewną uwagę projektową. Funkcje w łańcuchu powinny być uszeregowane w kolejności od tych, które generują najdłuższe hasła do tych najkrótszych. Dlaczego? Aby możliwie maksymalnie zmniejszyć szansę wystąpienia kolizji.

Rozważmy przykład, że wbrew tej zasadzie na początku umieściliśmy funkcję zwracającą hasła o długości jednego znaku. Mamy 1/n szansy (gdzie n to wielkość zbioru znaków) na powtórzenie się tego samego hasła, co przy 100 znakach w zbiorze daje aż 1%! Oznacza to, że średnio co setny łańcuch będzie identyczny, ponieważ kolizja wystąpi w tej samej warstwie, co spowoduje, że wszystkie kolejne skróty i hasła będą takie same. Jest to wysoce niepożądane zjawisko. Gdy natomiast na początku umieścimy funkcje generujące długie hasła, np. 10-znakowe, wtedy szansa wystąpienia tego samego hasła spada do 1/(n^10) co staje się szansą małą, nawet w ogromnych tablicach.

Drugie podejście do generowania różnych długości haseł to ustalenie, że pewne bajty skrótu będą odpowiadać długości wygenerowanego hasła. Przykładowo, pierwszy bajt może określać ile znaków chcemy otrzymać. Moim zdaniem takie podejście jedynie niepotrzebnie komplikuje sprawę i utrudnia analizę tęczowej tablicy, ponieważ nie możemy sprawdzić, ile jakiej długości haseł mamy, a jedynie statystycznie określić. Poza tym, aby zróżnicować ilość haseł o danych



Rysunek 11. *Tęczowa tablica – te same funkcje redukujące są tego samego koloru tworząc tęczę*

długościach trzeba byłoby zastosować znów podejście z różnymi zakresami wartości dla określonej długości.

Jak sprawić, aby funkcji redukujących było wiele?

Trzeba mieszać. Mieszać możemy kolejność zakresów, zakresy możemy rozbić na pojedyncze wartości i wymieszać te wartości.

Możemy na początku wymieszać skrót – wtedy wystarczy nam jedna funkcja redukująca (tzw. wewnętrzna) i wiele sposobów mieszania. Wtedy projektujemy jedną funkcję wewnętrzną o dobrych parametrach co do częstości liter w hasłach i zapamiętujemy różne klucze. Gdy dostajemy skrót do zredukowania to doklejamy do niego klucz i ponownie skracamy. Otrzymujemy wtedy wymieszany skrót i działamy na nim naszą dobrą funkcją wewnętrzną. Takie podejście sprawia, że różne funkcje redukujące to ta sama funkcja wewnętrzna z różnymi kluczami mieszającymi, więc zapamiętać musimy tylko funkcję wewnętrzną i różne klucze, co pozwala na zaoszczędzenie miejsca. Co więcej, tworzenie nowych funkcji redukujących jest proste, bo sprowadza się do wybierania nowych kluczy.

Sposobów i pomysłów jest wiele, sztuką jest wybrać optymalny i go stosować konsekwentnie w całej tablicy, dlatego ważne jest przemyślane zaprojektowanie kluczowych jej elementów.

Sól (w oku)

Tęczowe tablice pozwalają na znaczne przyspieszenie ataku na funkcje skrótu. Raz wygenerowane tablice można używać wielokrotnie i ciągle rozbudowywać o nowe łańcuchy zwiększając jej skuteczność. Wydawać by się mogło, że użycie tęczowych tablic obali zasadność stosowania funkcji skrótu, jednak tak się nie stało.

Istnieje jednak prosty sposób na utrudnienie pracy tęczowych tablic. Stosuje się tak zwaną sól. Polega to na zapamiętaniu obok skrótu hasła pewnego ciągu. Ciąg ten jest dopisywany do hasła i dopiero wtedy skracany. Metoda weryfikacji hasła polega wtedy na pobraniu ciągu od uwierzytelniającego się użytkownika, doklejenie soli do tego ciągu i skrócenie. Wtedy następuje porównanie z zapamiętanym skrótem. W efekcie uzyskujemy o wiele większy stopień, gdyż nie wystarczy już znalezienie dowolnego ciągu, którego skrót jest równy przechowywanemu. Dodatkowo trzeba zapewnić, by część tego ciągu była solą. Ilość możliwych ciągów pasujących jest diametralnie zmniejszona.

Czy to już koniec tęczowych tablic? Niezupełnie. Kolizje, których chcieliśmy ciągle unikać (i nadal chcemy ze względu na dublowanie tych samych danych w różnych łańcuchach) okazują się częściowo radzić sobie z tym problemem. Interesujące są dla nas tylko takie kolizje, które występują na różnych pozycjach w łańcuchach i dotyczą funkcji skrótu, czyli sytuacji, gdy z różnych haseł powstaje taki sam skrót.

Gdyby taka sytuacja występowała często, moglibyśmy rozszerzyć algorytm wyszukiwania haseł w tęczowej tablicy, aby nie zatrzymywał się po znalezieniu pierwszego pasującego skrótu, ale kontynuował poszukiwania dopóty, dopóki oprócz skrótu, zgadza się część ciągu zawierająca sól.

Są to rozważania czysto teoretyczne, gdyż rozmiar tablicy, która umożliwiałaby dużą skuteczność takich poszukiwań byłby przeogromny, jednak jest ciekawostką, którą należy mieć na uwadze, gdyż pewnego dnia, może się okazać możliwą do realizacji.

Skąd się wzięła nazwa?

Znając mechanizm rządzący tworzeniem tęczowych tablic warto wyjaśnić, skąd wzięła się ich nazwa. Otóż gdyby narysować tablicę, w której wierszach znajdowałyby się łańcuchy, a w kolumnach kolorami zaznaczono by funkcje redukujące, to powstanie tęcza. Bardzo długa tęcza (długość tęczy to ilość łańcuchów) o wielu barwach (każda barwa symbolizuje inną funkcję redukującą – warstwę) (Rysunek 11).

Podsumowanie

Tęczowe tablice są świetnym sposobem, na usprawnienie ataków na hasła przechowywane lub przesyłane, w postaci funkcji skrótu. W pełni pokazują swoje walory, gdy musimy atakować wiele różnych haseł, skróconych tą samą funkcją skrótu, ponieważ wtedy opłaca się wygenerować ogromne tablice, które można przechować i wykorzystać wyniki obliczeń w przyszłości.

Prostym sposobem na utrudnienie działania tęczowych tablic jest użycie soli, czyli połączenie hasła z pewnym ciągiem przed skróceniem. Ciąg ten zapamiętujemy obok skrótu hasła, aby się upewnić, że wpisano prawidłowe hasło, a nie wykorzystano kolizji występującej w funkcji skrótu. Skuteczność tej metody jest bardzo wysoka, a koszt stosowania niewielki, dlatego wszędzie gdzie to możliwe, należy stosować sól.

Inną dość oczywistą cechą tęczowych tablic, nie wspomnianą przeze mnie wyżej, jest zdolność do przeprowadzenia ataku offline. Offline, ponieważ gdy już zdobędziemy skrót interesującego nas hasła, nie potrzebujemy żadnej dodatkowej interakcji, a tylko dużą moc obliczeniową. Gdy już odzyskamy hasło, możemy go użyć zupełnie jak uprawniony użytkownik.

Tęczowe tablice są wręcz idealne do wykorzystania w modelu obliczeń rozproszonych. Każdy węzeł sieci rozproszonej posiada własny podzbiór łańcuchów i poszukiwanie wykonuje wyłącznie na nim.

Należy sobie jeszcze uświadomić, że tęczowe tablice nie są szybsze niż atak *brute force*. Etap generowania tablicy jest nawet wolniejszy, bo wymaga wykonania dla każdego skróconego hasła redukowania. Nie mamy też pewności, ile potrzebujemy łańcuchów, aby tablica zawierała wszystkie interesujące nas hasła.

Tęczowe tablice posiadają ograniczenia i nie na pewno nie są uniwersalną metodą łamania haseł, ale w przypadku powtarzalnych schematów ich zastosowanie daje doskonałe rezultaty.

Sebastian Czarnota
Autor jest studentem czwartego roku Wojskowej Akademii Technicznej na specjalności Kryptologia. Pracuje jako programista aplikacji GIS. Ponadto interesuje się zagadnieniami bezpieczeństwa kryptograficznego, algorytmami i szyframi.
Kontakt z autorem: *sebastian.czarnota@gmail.com*.

Jeśli nie możesz odczytać zawartości płyty CD, a nie jest ona uszkodzona mechanicznie, sprawdź ją na co najmniej dwóch napędach CD.

W razie problemów z płytą, proszę napisać pod adres:
cd@software.com.pl