

Střední průmyslová škola Třebíč

Maturitní práce

Hashtesting

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2024/2025 Kamil Franek

Zadání práce

ABSTRAKT

Maturitní práce na téma hashování, jeho využití a typy hashovacích funkcí. Zabývá se problematikou spojenou s hashováním a vysvětlení použití hashování v IT. Tento dokument popisuje použitou technologii, praktiky a vytváření samotného programu a všeho okolo. Výsledný program disponuje základními i rozšířenými funkcemi práce s hashem a soubory pro zvýšení efektivity práce. Dále disponuje okénkem pro informace, vysvětlení rozdílů mezi hashovacími algoritmy, rozdíl mezi použitím a nepoužitím soli a dalších informací za účelem zvýšení chápání daného téma pro uživatele.

KLÍČOVÁ SLOVA

Maturitní práce, Výuka, Hash, Kybernetický útok, Integrita dat

ABSTRACT

\*abstrakt anglicky\*

KEYWORDS

\*klíčová slova anglicky\*

PODĚKOVÁNÍ

Děkuji Ing. Ladislavu Havlátu za cenné připomínky a rady, které mi poskytl při vypracování maturitní práce.

V Třebíči dne 17. března 2025 podpis autora

PROHLÁŠENÍ

Prohlašuji, že jsem tuto práci vypracoval/a samostatně a uvedl/a v ní všechny prameny, literaturu a ostatní zdroje, které jsem použil/a.

V Třebíči dne 17. března 2025

podpis autora

Obsah

[Úvod 7](#_Toc192746425)

[1 Teorie hashovaní 8](#_Toc192746426)

[1.1 Použití Hashů 8](#_Toc192746427)

[1.2 Použité Hashe 8](#_Toc192746428)

[1.2.1 MD5 8](#_Toc192746429)

[1.2.2 SHA1/SHA256/SHA512 9](#_Toc192746430)

[1.2.3 RIPEMD160 10](#_Toc192746431)

[1.2.4 CRC32 10](#_Toc192746432)

[1.3 Sůl, Pepř a jejich používání 11](#_Toc192746433)

[1.4 Kybernetické útoky na hashe 12](#_Toc192746434)

[1.4.1 Pravděpodobnost kolize (Narozeninový paradox) 13](#_Toc192746435)

[1.5 Útoky 15](#_Toc192746436)

[1.5.1 Duhové tabulky a slovníkové útoky 15](#_Toc192746437)

[1.5.2 Útok hrubou silou 17](#_Toc192746438)

[1.5.3 Pass the Hash 17](#_Toc192746439)

[2 Použité programy a technologie 19](#_Toc192746440)

[2.1 C# a .NET 19](#_Toc192746441)

[2.2 Visual Studio 2022 19](#_Toc192746442)

[2.3 Visual Code 19](#_Toc192746443)

[2.4 Git/Github a Github Desktop 20](#_Toc192746444)

[2.5 Freelo.io 20](#_Toc192746445)

[2.6 Wayback Machine 21](#_Toc192746446)

[2.7 Online hashers 22](#_Toc192746447)

[3 Praktická část 23](#_Toc192746448)

[3.1 Rozdělení 23](#_Toc192746449)

[3.2 Základní hashovací metody 23](#_Toc192746450)

[3.3 Multi-Hashování 25](#_Toc192746451)

[3.4 Postupné Hashování 26](#_Toc192746452)

[3.5 Práce se solí a pepřem 26](#_Toc192746453)

[3.6 Kontrolní součet 29](#_Toc192746454)

[3.7 Hledání kolize 30](#_Toc192746455)

[3.8 MultiThreading 33](#_Toc192746456)

[3.9 Výpočet prolomení hesla 33](#_Toc192746457)

[3.10 Slovníkový útok 33](#_Toc192746458)

[3.11 Útok duhovou tabulkou 33](#_Toc192746459)

[3.12 Útok hrubou silou 33](#_Toc192746460)

[4 Funkce Programu a UI 34](#_Toc192746461)

[4.1 Hlavní Formulář 34](#_Toc192746462)

[4.2 Nastavení 34](#_Toc192746463)

[4.3 UI 37](#_Toc192746464)

[4.4 Formátování výstupu 37](#_Toc192746465)

[4.5 Práce se soubory 37](#_Toc192746466)

[4.6 Jazyk 37](#_Toc192746467)

[4.7 Motivy 37](#_Toc192746468)

[5 Statistiky 38](#_Toc192746469)

[Závěr 39](#_Toc192746470)

[Seznam použitých zdrojů 40](#_Toc192746471)

[Seznam použitých symbolů a zkratek 41](#_Toc192746472)

[Seznam obrázků 42](#_Toc192746473)

[Seznam tabulek 43](#_Toc192746474)

[Seznam příloh 44](#_Toc192746475)

Úvod

Cílem této ročníkové práce a programu je zjednodušení práce s hashy, ukázka rozdílů mezi hashovacími funkcemi, používání soli a pepře, ukázka postupného hashování pomocí mezikroků, silné a slabé stránky hashů, kde a proč se používají. Dále program obsahuje test prolomení hesla pomocí útoku hrubou silou, slovníkovým útokem a útokem s duhovými tabulkami. V dokumentu jsou popsány použité programy a technologie, jak jsou použité a proč jsou použité. V praktické části je popsána celá cesta dělání programu, hlavní problémy, trable a vysvětlení fungování celého programu s ukázkami samotného kódu, testování a různé obrázky z pracovního postupu. V závěru jsou popsány moje pocity z práce na projektu a spokojenost s finální verzí programu.

# Teorie hashovaní

Hashování je matematický algoritmus pro převod dat do předem určitého dlouhého výstupu podle algoritmu tzv. hashovací funkce. Hashe mají několik výtečných vlastností: vstupní data můžou být jakkoliv dlouhá, minimální změna v datech znamená velký rozdíl ve výstupech, s větší výstupní délkou se exponenciálně zmenšuje šance na stejnost výstupních hodnot při jiném vstupu a ta nejdůležitější, nedá se získat z výstupních dat vstupní data (bez použití kybernetických útoků), znamenaje, že proces je jednosměrný. Díky tomu se hash bere jako unikátní otisk vstupních dat. [1]

## Použití Hashů

Hashe se používání k uschování důležitých informací (například hesel), kde pro bezpečnost nechceme dostat vstupní data zpátky, dělání kontroly a integrity dat (kontrolní součet), vytváření a ověřování elektronického podpisu (třeba pro bankovnictví nebo email ověření), hledání škodlivého malwaru antivirovým programem, k hledání úseků DNA sekvencí atd. [1]

## Použité Hashe

Existuje spoustu hashovacích funkcí a každá má svoje výhody, nevýhody a využití pro jiné účely. Tady je informace pro hashovací funkce, které jsou použity v programu.

### MD5

MD5 (Message-Digest Algorithm) pochází z rodiny „Message-Digest“ neboli algoritmus na strávení zprávy. Předchůdci MD5 jsou hashovací funkce MD2 a MD4, všechny tři vytvořeny a vydány Ronaldem Rivestem. MD2 byl vydán v roce 1989, MD4 jakožto pokračovatel v 1990 a MD5 jakožto vylepšená verze MD5 v roce 1991. MD5, na rozdíl od svých předchůdců, je docela složitý algoritmus na rozlousknutí. Používá 4 kola, místo 3 kol jako MD4, a pomocí matematických operací s maticema vypočítá výstup. Délka výstupu hashe je 128 bitů. I přes jeho používání v tehdější a dnešní době se v MD5 našla řada chyb, které by mohly být při ukládání hesel závažné. MD5 je totiž poměrně náchylný na takzvaný brute force attack, česky útok hrubou silou. [1][2]

\*Příklad 1. kola ze 4 u MD5

### SHA1/SHA256/SHA512

SHA, Secure-Hash Algorithm neboli bezpečná hashovací funkce je další velice známý a používaná hashovací funkce. SHA se bere za nástupce MD5 s větší bezpečností a delším výstupem (SHA1 – 160 bitů. SHA256 – 256 bitů, SHA512 – 512 bitů). SHA1 byla první verze SHA vydaná v roce 1995, nepočítaje SHA-0, což byla rychle zapomenutá „před“ verze SHA-1. SHA je rodina vytvořena a zveřejněna americkým ústavem pro technologické standardy (National Institute of Standards and Technology [NIST]). [3][4]

V roce 2005 byl na SHA1 nalezen možný útok a proto v roce 2010 vyšla skupina SHA-2, což je skupina několika hashovacích funkcí, u kterých se mění délka výstupu (pro nás důležité SHA 256 a SHA512, dále se nachází SHA-224 a SHA-384. Čísla na konci znamenají délku výstupu v bitech). Skupina SHA-2 se dodnes považují za bezpečné hashovací algorithmy pro integritu dat a ukládání hesel. [5]

Funkce SHA-256 se využívá ve virtuálních měnách jako třeba Bitcoin. Hlavní premise takzvaných těžiček je najít vstup zahashovaného textu pomocí SHA-256 funkce. [6]

„Google oznámil, že se mu [podařilo prolomit bezpečnost](https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html) hašovací funkce SHA-1. Od [první publikované slabiny](https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html) po úspěšný útok tak uběhlo deset let. Dva roky trvalo vědcům z CWI Institute in Amsterdam a společnosti Google, než dokončili práci na slabině a přinesli důkaz toho, že kolizní funkce existuje a významně urychluje útok.“ [7] Přikládám URL adresu projektu SHAttered: <https://shattered.io/>

### RIPEMD160

RipeMD160 je hashovací funkce, která měla za účel nahradit MD4 a MD5, stejně jako SHA1. Hlavní rozdíl je, že RipeMD160 byla vyvinuta v EU jakožto součást projektu RIPE (RACE Integrity Primitives Evaluation, 1988-1992). RipeMD160 byla vytvořena Hansem Dobbertinem, Antoonem Bosselaersem a Bartem Preneelem. Spolu s RipeMD128, RipeMD256 a RipeMD320 byly vydány v roce 1996. Všechny tyto verze vstávají z originální RipeMD hashovací funkce, která byla vydána roku 1992. Dnes všechny tyto verze RipeMD nejsou doporučovány používat, i když v lepších verzích RipeMD (vydány v 1996) nebyly nalezeny žádné kolize. [8]

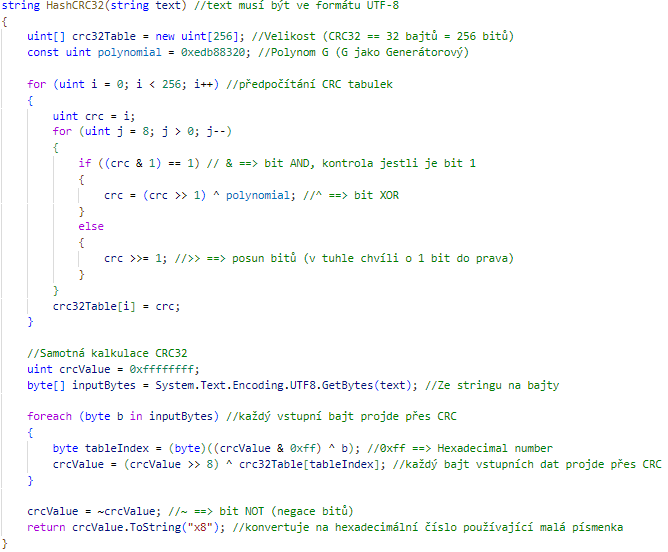
V originálním RipeMD z roku 1992 byla nalezena kolize v roce 2004. Pro maximální bezpečnost se může používat Hash160, což je kombinace SHA256 a RipeMD160, kde vstup prvně projde SHA256 a výsledek poté projde přes RipeMD160. Hash160 se používá u Bitcoinu jako identifikátor pro úschovu peněženky. [9] <https://www.hash160.com/>

### CRC32

CRC32 je hashovací funkce z rodiny cyclic redundancy check neboli cyklická redundantní součet (CRC) je hashovací funkce zaměřená na kontrolní součet a integritu dat. Kvůli tomuto využití jsou všechny CRC velice rychlé. Rodina CRC byla zveřejněna W. Wesley Petersonem v 1961. Nejznámější a nejpoužívanější funkce z rodiny CRC jsou CRC32, která se používá v „ISO 3309 (HDLC), ANSI X3.66 (ADCCP), FIPS PUB 71, FED-STD-1003, ITU-T V.42, ISO/IEC/IEEE 802-3 (Ethernet), SATA, MPEG-2, PKZIP, Gzip, Bzip2, POSIX cksum, PNG, ZMODEM atd.“ [10], a CRC16, které se používá v Bluetooth, SD, X.25, v různých mobilních sítích, USB a mnoho dalších. [11]

Wesley Petersonovi byla za návrh CRC udělena cena „Japan Prize“ v roce 1999. [12] Co se týče hashování důležitých dat, klíčů a bezpečnostních prvků nemá CRC mnoho využití, ale jakožto funkce pro kontrolní součet jich má nespočet.

#### Výpočet CRC

Základ celého výpočtu CRC jsou polynomy. Každý jeden bit v určité délce může být zapsán do polynomu. Příklad (1101) může být zapsán jako . poslední bit reprezentuje 1, druhý bit x, třetí atd. CRC32 má až a CRC16 až . Tento výpočet pro CRC32 je v mém kódu reprezentován druhým cyklem for (uint j = 8; j > 0; j--); Výpočet se prodeve tolikrát, kolik je výstupní délka v bitech (pro CRC32 to je 256, pro CRC16 to je rovno 128). Náš vytvořený polynom je poté porovnán operací XOR (Exclusive OR, které vrací 0, pokud jsou obě hodnoty stejné a 1, pokud jsou jiné).

Další část je porovnávání vstupních bajtů s bajty v tabulce. Index tabulky je zjištěn pomocí předběžného výstupu CRC (pro první projití je do hodnoty CRC nastavena nejvyšší hodnota), operací AND (AND vrací log. 1 pouze pokud jsou oba bity log. 1) a pomocí operace XOR se vstupem. Dále se na CRC výstupu posune 8 bitů do prava, tímto se 8 bitů vpravo efektivně ztratí a provede se XOR operace s tabulkovou hodnotou na indexu předchozího výpočtu. Na závěr se celý výstup ještě neguje bit po bitu (operace NOT). [10][11]

## Sůl, Pepř a jejich používání

Hashovací sůl a pepř jsou další vrstvou pro bezpečnost hashování. Sůl je náhodně vygenerována před generováním a dává se před samotnými daty. Může mít jakoukoliv délku, záleží na správci, který bude hodnotu ukládat. Díky soli se chráníme před takzvanými rainbow tables útoky a útoky hrubou silou.

Pepř je podobný jako sůl, jenže je většinou krátký, dává se na konec dat místo před data a nikde se neukládá. To znamená, že musíme provést všechny možné kombinace hashů a porovnávat výsledné hashe abychom zjistili shodu. Jediná nevýhoda pepře je, že musíme hashovací a porovnávací proces dělat několikrát, což výrazně zvýší prodlevu. V praxi se používají sůl i pepř pro maximální ochranu hesel. [13]

Příklad: Máme heslo “TestingPassword123”. Díky soli se před heslo vygeneruje sůl “a0\_X”, která je někde v tabulce uložena. Před heslo se vygeneruje pepř o délce jednoho ASCII znaku. To znamená že při každém pokusu o přihlášení se před heslo dá sůl a za heslo se postupně zkouší “000”, poté “001”, “010” a tak dale, dokud se nevyzkouší všechny kombinace pepře. Pokud žádný z těchto pepřových kombinací nevýjde, heslo je zadáno špatně.

Všechny hashe jsou šifrované v hashovací funkci MD5.  
(bez soli) TestingPassword123 == 1d898af5dbe7c9e07fc473e248f623a1  
(se solí) a0\_XTestingPassword123 == 5a6c1e14762baf73406b7267a8afae88  
(sůl I pepř) a0\_XTestingPassword123C == d7d0d822ce9faea482a0c5ae372d0ed0

## Kybernetické útoky na hashe

Hashe dokážou zpracovat jakékoliv množství dat a vrátit jenom určitou délku, to ovšem znamená menší problémy. Různé vstupní data mohou vracet stejnou hodnotu hashe, což v případě, že používáme hashe pro ukládání hesel znamená velký bezpečnostní problém. Šance kdy se něco takového může stát je závislá na délce výstupního hashe, proto se doporučuje používat delší a bezpečnější hashe pro ukládání důležitých dat (jako třeba hesel), jako třeba SHA-256 či SHA-512.

Pravděpodobnost si můžeme sami vypočítat pomocí jednoduchého zvorečku. Počet všech kombinací u hashovací funkce je rovna . To znamená že CRC32 má šanci 1 ku (skoro 4.3 miliardy kombinací) neboli 0.0000000233%. To se může zdát jako velice malá šance, jenomže tohle je šance jenom mezi 2 hashemy. Když započítáme šanci každého s každým (použití takzvaného narozeninového paradox), když hledáme čistě jenom kolize, počet všech kombinací se nám sníží na .

„Narozeninový paradox nám říká, že pokud máme v místnosti 23 lidí, existuje přibližně padesátiprocentní šance, že se dva z nich narodili ve stejný den. Toto je velice důležitý fakt, který nám dává dolní ohraničení pro délku hashe produkovaného dobrou kryptografickou hashovací funkcí.“ „Pokud budeme mít například 40-bitovou zprávu, abychom našli kolizi s pravděpodobností 0.5, potřebujeme pouze 2^20 náhodných hashů.“ [14, strana 2]

|  |  |  |  |
| --- | --- | --- | --- |
| Hashovací funkce | Velikost hashe  (bit) | Počet možných kombinací (zaokrouhleno) | 50% šance při hledání kolize (zaokrouhleno) |
| MD5 | 128 | 3,40e+38 | 18 446 744 073 709 551 616 |
| SHA-1 | 160 | 1,46e+48 | 1 208 925 819 614 629 174 706 176 |
| SHA256 | 256 | 1,15e+77 | 3,40e+38 |
| SHA512 | 512 | 1,346e+154 | 1,15e+77 |
| CRC32 | 32 | 4 294 967 296 | 65 536 |

### Pravděpodobnost kolize (Narozeninový paradox)

https://www.youtube.com/watch?v=yQ1pGhMRLKI

Pokud ovšem hashujeme více než dva vstupy, například při hledání kolize, pravděpodobnost kolize vzrůstá. Při **více projetí** se totiž pravděpodobnost kolize řídí tzv. „narozeninovým paradoxem“. Pravděpodobnost, že mezi vygenerovanými hashy alespoň jednou nastane kolize lze vypočítat podle vzorce pro kombinatoriku:

Tento vzorec je velice přesný, ovšem pro počítání skoro až nereálný, jelikož při N = jako to je u RipeMD-160 naprosto vyloučeno. Naštěstí si můžeme vzorec poupravit do mnohem lepší formy.

Tato upravená verze není až tak přesná, ale dá se mnohem snadněji vypočítat. [15, strana 11 a 12]  
k = počet vygenerovaných hashů  
e = eulerovo číslo (přibližně 2.718)  
N = počet možných hashů ()  
P ≥ zaokrouhlená šance na kolizi (mezi 0 a 1 / 0% a 100%)

Jak můžeme vidět, hledání kolizí je exponenciální. Díky narozeninovému paradoxu se nám drasticky zmenšuje počet projití, které je potřeba na najití jakékoliv jedné kolize. Pokud chceme najít pravděpodobnost kolize v určité šanci, třeba na 23 % (tedy p=0.23), můžeme si poupravit naši rovnici a získat vzorec   
(ln je přirozený logaritmus neboli logaritmus o základu e) ze které po dosazení vznikne , n ≐ 47 382. Po projití 47 382 hashů máme 23% šanci na naleznutí kolize.

Tento způsob výpočtu má dva problémy. Problém jedna je, že , jelikož bychom se dostali k dělení 0 v jmenovateli, ==> , proto musíme dosadit co nejbližší číslo menší než 1, což je 0.99 periodický. Další problém je zaokrouhlování, protože pomocí jedné metody () nám vychází, že 50 % na najití kolize je zaokrouhleně 65 tisíc, jenže pomocí našeho upraveného vzorečku pro p nám vychází zaokrouhleně 74 tisíc (tento útvar jde vidět v grafu).

Tady dokazuji, že exponenciální rovnice je pro všechny hashovací funkce stejná. Konce grafů ovšem neukazují 100% úspěšnost, jelikož 100% úspěšnost lze získat pouze zkoušením všech možných kombinací.

## Útoky

Jak už bylo zmíněno, na hashe a hlavně hesla existuje pár specifických útoků. Většinou jsou útoky zamýšlené na získání hesla či nějakého klíče či certifikátu, jako to používají cookies (sušenky) na webových prohlížečích. [16] Mezi nejrozšířenější jsou duhové tabulky a slovníkové útoky, útok hrubou silou a pass the hash.

### Duhové tabulky a slovníkové útoky

Slovníkový útok a útok pomocí duhové tabulky můžou na první pohled vypadat jako jeden a ten samý útok, jenže tady je jeden hlavní rozdíl. Oba útoky jsou nějaká databáze potenciálních hesel, jediný rozdíl je, že duhová tabulka vyhledává už zahešovaný výstup, nikoliv jeho vstup. Vstup si je schopná odvodit z nalezeného hashe. Naopak slovníkový útok se snaží najít počátek hashe.

Slovníkový útok se použije v případě, kdy nemáme přístup k databázi se zahešovanými hesly. Nádherný příklad je přihlášení. Zkoušíme všechna možná hesla ze seznamu, dokud nás jedno nedostane dovnitř. Pokud nenajdeme žádnou shodu, můžeme začít útok hrubou silou. Výhoda slovníkového útoku je, že nezáleží na použití soli, jediné, na čem záleží je velikost slovníku a čas. Jeden nádherný slovník má ve svém základě Kali Linux s názvem rockyou.txt, který obsahuje přes 12 milionů často používaných hesel.

Útok s duhovými tabulkami použijeme, když jsme se nějakým způsobem dostali k už zaheshovanému heslu. Díky délce hashe můžeme docela dobře zjistit použitou funkci. Každá funkce má svoji vlastní tabulku. Místo abychom jako u slovníkového útoku hledali počátek, hledáme radši shodu ve finálním hashy. Jakmile ho najdeme, tak s ním máme i originální vstup. Největší obtíže tomuto stylu útoku dělá používání soli a pepře při hashování vstupu. Jelikož skladování jak vstupu, tak výstupu může být u nějakých hashovacích funkcí náročné na místo (SHA512 má 512 bitů na heslo bez vstupu => pro 10 mil. hesel nám jenom hash zabírá 640 MB).

|  |  |  |
| --- | --- | --- |
| **Vlastnosti** | **Slovníkový útok** | **Útok přes duhové tabulky** |
| Rychlost hashování | Záleží na použité funkci | Předhashováno |
| Náročnost na uložiště | Malá (akorát vstup) | Velká (vstup i výstup) |
| Použitelnost | Platí na všechny funkce | Pro specifickou funkci |
| Rychlost | Pomalá (hashování) | Rychlá (hledání shody) |
| Efektivita | Záleží akorát na délce vstupu | Samostatně bezbranný proti solení/pepře |

### Útok hrubou silou

Útok hrubou silou (brute force) je nejzákladnější útok možný. Zkoušení hesel, dokud se nějakému nezadaří. Dal by se chápat, jako poslední možnost, když selžou všechny ostatní útoky. Fungování je jednoduché, začneme od nejmenšího možného znaku a postupně přidáváme, dokud se to jednoho dne nepovede. Záleží na délce hesla a na použitých znacích. Heslo může být 8 míst dlouhé, ale pokud víme, že používá pouze číslice, tak máme velikou výhodu. Vzorec pro vypočítání všech možných kombinací je . [17] V našem případě to je milionů možných kombinací. To možná zní jako hodně, ale když jedna grafická karta RTX 4090 dokáže zvládnout 200 tisíc hashů za sekundu BCryptu[[1]](#footnote-1) a 300 milionů hashů NTLM [[2]](#footnote-2) [17], tak 100 milionu najednou nezní tak krásně.

Při použití všech doporučených znaků pro heslo, což je malá písmena (26 znaků), velká písmena (26 znaků), číslice (10 znaků) a speciální znaky (33 znaků\*), tak máme dohromady 95 možných použitelných znaků. Minimální doporučená délka je 7 znaků, takže počet kombinací je , což je skoro 70 bilión kombinací, což může jedné RTX 4090 trvat při použití MD5 necelých 5 dní. [19, řádek 41] Můžeme zvýšit velikost hesla, ale útočník může zvýšit počet grafik. Naštěstí 15 znaků dlouhé heslo pořád nezvládne ani 10 vysoce výkonných grafických karet zvládnout za skoro 300 miliard let. [17]

### Pass the Hash

Hlavní důvod proč se hashování používá pro hesla je zabránit tomu, aby si někdo při komunikaci mohl jen tak vzít packet a přečíst si naše heslo. Když by někdo odposlouchával zaheshované heslo, tak musí pomocí dalšího útoku (například slovníkového) zjistit originální vstup. To, jak už určitě víme trvá spousty a spousty času a je díky tomu mnohem efektivnější dostat heslo jiným způsobem (třeba sociální inženýrství). Když ale na server přichází pouze zahashované heslo, a ne doopravdy heslo, proč bychom nemohli na daný server prostě poslat náš hash. O tomto je přesně pass the hash útok. Získat hash oběti je mnohem jednodušší než získat její heslo. Nejúčinnější je tento útok pro NTLM. Originální útok je od Paula Ashtona a byl zveřejněn roku 1997. Dokonce existuje i „toolkit“ pro zkoušení tohoto útoku přímo ve Windows či Kali Linuxu (Windows: <https://github.com/byt3bl33d3r/pth-toolkit>) (Kali: [https://www.kali.org/tools/passing-the-hash/).](https://www.kali.org/tools/passing-the-hash/).%20) [21] [22]

# Použité programy a technologie

Každý správný projekt potřebuje použití několika programů či stránek pro zlepšení práce na projektu. Každý použitý program či stránka jsou popsané k čemu slouží, teorii, popřípadě vysvětlení jak fungují a proč jsem je použil.

## C# a .NET

„Jazyk C# je multiplatformní jazyk pro obecné účely, který vývojářům umožňuje produktivní práci při psaní vysoce výkonného kódu. S miliony vývojářů je jazyk C# nejoblíbenějším jazykem .NET. Jazyk C# má širokou podporu v ekosystému a všech úlohách .NET. Na základě objektově orientovaných principů zahrnuje mnoho funkcí z jiných paradigmat, nikoli z nejméně funkčního programování. Funkce nízké úrovně podporují scénáře vysoké efektivity bez psaní nebezpečného kódu. Většina modulů runtime a knihoven .NET je napsaná v jazyce C# a pokroky v jazyce C# často využívají všechny vývojáře .NET.“ [23]

Programovací jazyk je základ všeho a jelikož jsem chtěl mít jistotu, tak jsem si vybral možnost, kterou velice dobře znám. Visual Studio, .NET framework pro Windows a s tím i spojený jazyk C#.

## Visual Studio 2022

Hlavní důvod vybrání si Visual Studia a .NET frameworku je jednoduchost k přidání knihoven a manipulace s nimi. Hashování je provedeno přes „System.Security.Cryptography“ knihovnu, která je v základu s .NET frameworkem. Tuto knihovnu používám pro všechny hashe použité v programu, až na CRC32. Dále poskytuje spousty UI elementů a práci s formulářem, jednoduché přidání unit-testů, chybové hlášení, krokování programu, statistika využití CPU a RAM a mnoho dalšího.

## Visual Code

Visual Code je univerzální editor kódu. Je nejlepší s použitím jazyka Python či HMTL a PHP, ale díky jeho univerzálnosti není žádný problém dělat v C#. Jediné, co stačí je doinstalovat si rozšíření, a to vyžaduje jedno zmáčknutí tlačítka. Při používání Visual Studia není potřeba používat Visual Code, ale líbí se mi jednoduchost programu. Použil jsem program při vytváření dodatečných skriptů (jako například hasher.cs nebo settings.cs), které jsem poté naimportoval do Visual Studia.

## Git/Github a Github Desktop

„Git je verzovací systém, pomocí kterého ukládáte své projekty a veškeré jejich verze. Je to distribuovaný systém správy verzí. To znamená, že k celému kódu i jeho historii se vývojář dostane z jakéhokoliv počítače.[[3]](#footnote-3) Většina operací, které se s kódem provádějí, se dějí lokálně na disku pomocí příkazového řádku. Pokud ale chcete Git sdílet s kolegy, probíhá spolupráce téměř vždy přes centrální server nebo úložiště.“ [24]

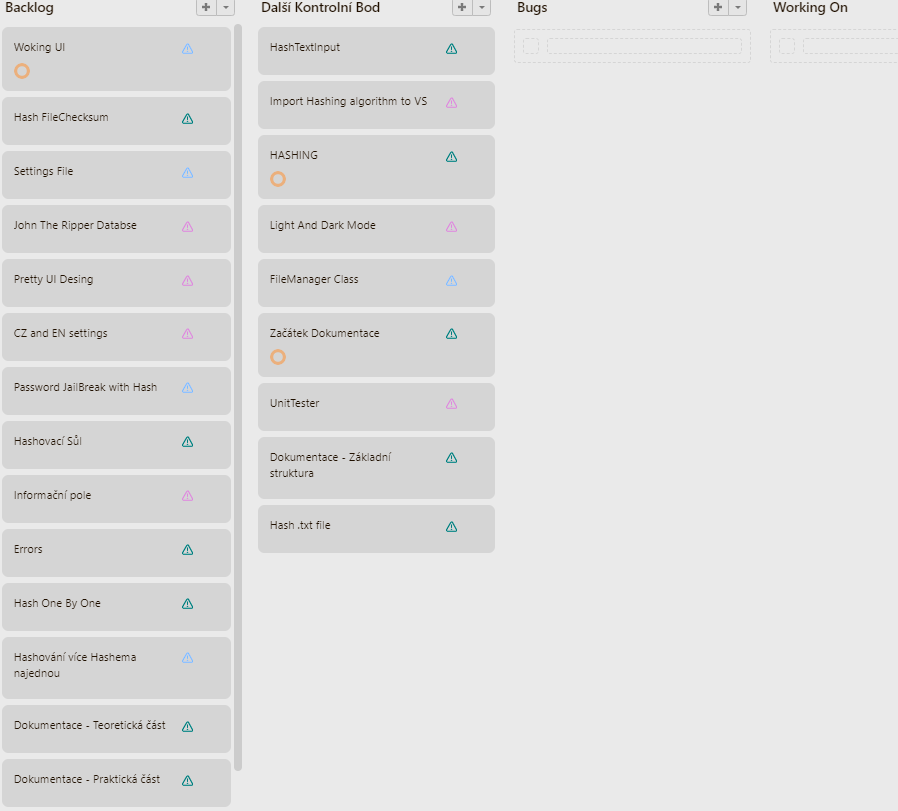
Github je webová stránka, kde se dají procházet a ukládat různé Git projekty. Github nabízí spousty služeb, článků a dokonce nabízí i předplatné pro ještě více funkcí. Jedna z jejich nejnovějších zaměření je Copilot, což je umělá inteligence (AI), která má dělat programování o něco jednodušší. Nejdůležitější věc je, že většina hlavních funkcí je zadarmo, a to i s uložištěm (byť trochu malým).

„Díky tomuto systému je spolupráce na projektu bezpečnější a jednodušší. Možná i to je důvod, proč **Git používá více než 87 % vývojářů**.“ [24]  
Github desktop je počítačová aplikace, která funguje stejně jako příkazový řádek v Git, akorát místo CLI[[4]](#endnote-1) to je GUI[[5]](#endnote-2) a dělá to Git uživatelsky přívětivé.

Existuje i test-driven development (TDD) (Vývoj řízený testy v češtině), který spočívá na systému právě unit-testů. Před prací na programu si prvně připravíme unit-testy, které nám říkají, jak se má program chovat, a podle těchto testů pak píšeme program. [26]

## Freelo.io

„Freelo je [online](https://cs.wikipedia.org/wiki/Online) aplikace s cílem usnadnit [řízení projektů](https://cs.wikipedia.org/wiki/%C5%98%C3%ADzen%C3%AD_projekt%C5%AF) a zvýšit efektivitu podnikatelů a firem. Řadí se mezi [SaaS](https://cs.wikipedia.org/wiki/SaaS)[[6]](#endnote-3) aplikace a v červnu 2021 překročilo hranici 50 000 uživatelů.“ „Mezi hlavní funkce Freela patří Projekty, To-Do listy, Diskuse, Kalendář a Finance. Jejich cílem je usnadnění finanční správy projektu.“ [27]

Freelo je založeno na agilním projektovém řízení Kanban[[7]](#footnote-4). Kanban je založen na systému výroby Just In Time[[8]](#footnote-5) (JIT), která byla vymyšlena v japonské automobilové firmě Toyota. Zakladatel je Taiičiho Óno. Kanban zjednodušuje práci několika lidí a její management. Díky Kanbanu jde krásně vidět co kdo dělá, co bude dělat a co se musí ještě udělat. V mém případě Freelo používám jako to-do list[[9]](#endnote-4). [29]  
Příklad mého použítí Freela na začátku projektu.

## Wayback Machine

Wayback Machine je digitální archiv, který je součástí Internet Archive. Internet Archive je nezisková organizace, původem z USA, která se snaží zachovat co nejvíce z internetu. Od webových stránek, po videa, audia, obrázky, programy a další. V dnešní době mají uloženo přes 960 miliard stránek v rozmezí 1996 až konce 2024. [31] Nejlepší je, že pro uživatele je všechno k podívání zadarmo bez potřeby účtu. U webových stránek stačí mít pouze URL[[10]](#endnote-5) adresu. Záznamů může být několik a člověk si může vybrat. Odkaz na Internet Archive <https://archive.org> a Wayback Machine <https://web.archive.org>

Já osobně jsem používal Wayback Machine při dělání citací, protože většina internetových stránek nemají veřejný datum vzniku. Když je internetová stránka uložena v Internet Archive, znamená to, že v té době už existovala. Není to nejpřesnější datum, ale je to lepší než nic.

## Online hashers

Online hashers jsou funkční algoritmy pro různé hashe, většinou ve formě stránky, které jsem primárně používal k porovnávání vygenerovaných hashů pro unit testy. Existuje spousty programů a stránek, já jsem používal <https://www.browserling.com/tools/all-hashes>, protože tam jsou všechny hashovací funkce, které používám v programu.

# Hashovací třídy

Pod tuto kapitolu patří popsání a vysvětlení fungování programu, ukázky kódu a další věci společné s programem. Celý program, zdrojový kód, písemná práce a další jsou veřejně dostupné na službě GitHub pod názvem HashTester.

## Rozdělení

Kvůli velikosti programu, přehlednosti a dodržování programátorských konvencí obsahuje program několik tříd a formulářů[[11]](#footnote-6), které jsou vysvětleny při jejich použití. Třída hasher v sobě obsahuje několik skupin funkcí, které se používají v celém programu. Proto je na třídu hasher odkazováno ve vícero částech práce.

## Základní hashovací metody

Hlavní funkce programu jsou hashovací funkce, proto jsem si vytvořil svůj vlastní skript na jednoduchou práci s hashy. Tento skript se jmenuje hasher. Hasher používá public enumerátor „Hashing Algorithm“, díky kterému se vybírá, kterou hashovací funkci použít. V základu je na výběr MD5, SHA1, SHA256, SHA512, RipeMD-160 a CRC32.

Základní metoda skriptu je Hash, která má 4 přetížení. Tyto přetížení jsou pro vstupní text a výstupní text, vstupní text a výstupní bajty, vstupní bajty a výstupní text a nakonec vstupní bajty a výstupní bajty. Text je podporován v kódování UTF-8.

public string Hash(string text, HashingAlgorithm algorithm)

{

switch (algorithm)

{

case HashingAlgorithm.MD5: return HashMD5(text);

case HashingAlgorithm.SHA1: return HashSHA1(text);

case HashingAlgorithm.SHA256: return HashSHA256(text);

case HashingAlgorithm.SHA512: return HashSHA512(text);

case HashingAlgorithm.RIPEMD160: return HashRIPEMD160(text);

case HashingAlgorithm.CRC32: return HashCRC32(text);

default: return null;

}

}

V metodě Hash se rozhoduje která hashovací funkce se má použít.

string HashMD5(string text)

{

using (MD5 md5 = MD5.Create())

{

byte[] hashBytes = md5.ComputeHash(Encoding.UTF8.GetBytes(text));

return BitConverter.ToString(hashBytes).Replace  
 ("-", "").ToLowerInvariant(); //Z AA-BB-CC udělá aabbcc

}  
}

Pro příklad HashMD5 je privátní metoda, která se stará o samotné hashování. Třída MD5 a další jsou součástí oboru názvů System.Security.Cryptography, který je součástí .NET Frameworku. Všechny metody mají také 4 přetížení, stejně jako metoda Hash.

Jediná výjimka je metoda pro CRC32, protože se nenachází v oboru. Naštěstí je na internetu dostupná.

string HashCRC32(string text)

{

if (crc32Table == null) // Check if the table is initialized

{

CRC32Table();

}

//Main algorithm

uint crcValue = 0xffffffff;

byte[] inputBytes = System.Text.Encoding.UTF8.GetBytes(text);

foreach (byte b in inputBytes)

{

byte tableIndex = (byte)((crcValue & 0xff) ^ b);   
 //0xff ==> Hexadecimal number

crcValue = (crcValue >> 8) ^ crc32Table[tableIndex];   
 //>> ==> bit shift - each byte of input data goes through CRC table

}

crcValue = ~crcValue; //~ ==> bit NOT

return crcValue.ToString("x8"); //converts to a hexadecimal number  
 using lowercase letters

}

CRC32 funguje na předpřipravené tabulce, která je vždy stejná. Pro optimalizaci si prvně kontroluji, zda už není tabulka vytvořena. Jestli není, tak vytvořím pomocí metody CRC32Table. Tato optimalizace velice zrychluje už tak rychlou hashovací funkci.

void CRC32Table()

{

crc32Table = new uint[256]; //Size (CRC32 == 32 bytes/256 bits)

const uint polynomial = 0xedb88320; //Polynom G (G as Generated)

for (uint i = 0; i < 256; i++) //CRC table precalculation

{

uint crc = i;

for (uint j = 8; j > 0; j--)

{

if ((crc & 1) == 1) crc = (crc >> 1) ^ polynomial;

//& ==> bit AND

//^ ==> bit XOR  
 //>> ==> bit shift to right

else crc >>= 1;

}

crc32Table[i] = crc;

}

}

Pro generování tabulky se používá polynom, což je předurčena konstanta. Ostatní CRC funkce jako například CRC16 bude mít polynom jiný.

## Multi-Hashování

Multi-hashování, neboli hashování vstupu vícero funkcemi najednou, je krásný příklad použití třídy Hasher. ProcessingHash se nachází v hlavním formuláři a slouží jako takový mezikrok. Zadá se do metody text v UTF-8 či bajty, jeden či více hashovacích funkcí a dobrovolná komponenta listbox. Metoda se ujistí o správné zakomponování soli a pepře, zpracování a výstupu podle nastavení uživatele.

//Pro vícero funkcí najednou

public void ProcessingHash(string[] originalText, Hasher.HashingAlgorithm[] algorithm, ListBox listbox)

//Pro vstup z .txt souboru

public void ProcessingHashTXTInput(Hasher.HashingAlgorithm[] algorithm, ListBox listbox)

//Základ v hlavním formuláři

public void ProcessingHash(string[] originalText, Hasher.HashingAlgorithm algorithm, bool askForSaltPepper)

Tady je několik příkladů přetížení metody ProcessingHash a ProcessingHashTXTInput, kde místo přímého vstupu se pomocí komponenty OpenFileDialog přečte text ze souboru podle řádků a ty se následně zpracují.

if (checkBoxMD5.Checked || checkBoxSHA1.Checked || checkBoxSHA256.Checked || checkBoxSHA512.Checked || checkBoxRipeMD160.Checked || checkBoxCRC32.Checked)

{

List<HashingAlgorithm> algorithm = new List<HashingAlgorithm>();

if (checkBoxMD5.Checked) algorithm.Add(HashingAlgorithm.MD5);

if (checkBoxSHA1.Checked) algorithm.Add(HashingAlgorithm.SHA1);

if (checkBoxSHA256.Checked) algorithm.Add(HashingAlgorithm.SHA256);

if (checkBoxSHA512.Checked) algorithm.Add(HashingAlgorithm.SHA512);

if (checkBoxRipeMD160.Checked)  
 algorithm.Add(HashingAlgorithm.RIPEMD160);

if (checkBoxCRC32.Checked) algorithm.Add(HashingAlgorithm.CRC32);  
 mainForm.ProcessingHash(textHashSimple.Lines, algorithm.ToArray(),  
 listBoxLog); //processes all the stuff  
 }  
 else

{

MessageBox.Show(Languages.Translate(223), Languages.Translate(10025), MessageBoxButtons.OK, MessageBoxIcon.Error);

}

Díky ProcessingHash metodě je implementace tohoto kódu velice jednoduchá a přehledná.

## Postupné Hashování

Postupné hashování není až tak použitelné v reálném životě, ale krásně se na něm ukazuje krása hashování, kde jenom malá změna na vstupu dokáže kompletně změnit výstup. Protože má pouze výuková použití, nedá se nastavit sůl či pepř.

Ve třídě Hasher jsou celé 2 metody, jedna bez soli a pepře, druhá s. Jenže druhou metodu nikde v kódu nepoužívám.

public string[] GradualHashing(string text, HashingAlgorithm algorithm)

{

string[] gradualHashing = new string[text.Length];

string textCurrentlyHashing = "";

for (int i = 0; i < text.Length; i++)

{

textCurrentlyHashing += text[i].ToString();

gradualHashing[i] = Hash(textCurrentlyHashing, algorithm);

}

return gradualHashing;

}

Kód je naprosto jednoduchý. Začneme prvním znakem textu a v každém projití cyklu se přidá jeden znak, dokud nemáme plnou délku textu. To se poté vrátí do formuláře s postupným hashováním, který výsledek zobrazí podle volby uživatele.

## Práce se solí a pepřem

Sůl a pepř je jedna z nejdůležitějších bezpečnostních opatření hashe, jelikož dělají hash imunní vůči útoku s duhovými tabulkami. Nejznámější použití je při práci s hesly.

V regionu „SaltAndPepperLogic“ ve třídě Hasher se nachází 4 hlavní metody. Metoda IsUsingSaltAndPepper, SaveSalt, LoadSalt a CheckPepper. Metoda IsUsingSaltAndPepper má několik přetížení.

public bool IsUsingSaltAndPepper(bool useSalt, bool usePepper, out string salt, out string pepper, out string hashID)

{

hashID = "";

salt = "";

pepper = "";

if (useSalt || usePepper)

{

using (SaltAndPepperSetup saltAndPepperQuestion =   
 new SaltAndPepperSetup(useSalt, usePepper))

{

// Show dialog and handle result

saltAndPepperQuestion.StartPosition =  
 FormStartPosition.CenterScreen;

if (saltAndPepperQuestion.ShowDialog() == DialogResult.OK)

{

saltAndPepperQuestion.GetSaltPepperInformation(

out bool generateSalt,

out int saltLength,

out string ownSalt,

out bool generatePepper,

out int pepperLength,

out string ownPepper,

out hashID

);

//Salt

if (generateSalt)

{

salt = GenerateSalt(saltLength);

}

else if (!string.IsNullOrEmpty(ownSalt))

{

salt = ownSalt;

}

//Pepper

(…)

SaveSalt(hashID, salt, pepper.Length);

return true; //Everything is fine

}

return false; //Dialog Canceled

}

}

return false; //Do not use Salt/Pepper

}

Každá metoda, která hashuje a uživatel povolil použití soli či pepře, má tuto metodu. Tato metoda nastavuje formulář SaltAndPepperQuestion, kde si uživatel zvolí buď vlastní sůl/pepř, nebo náhodně generovanou o nějaké délce. Tyto hodnoty se poté vrací zpátky, kde jsou metodou IsUsingSaltAndPepper zpracovány a uloženy.

public void SaveSalt(string hashID, string salt, int pepperLength)

{

string path = Path.Combine  
 (Settings.DirectoryToHashData, hashID + ".txt");

using (StreamWriter writer = new StreamWriter(path))

{

if (!String.IsNullOrEmpty(salt))   
 writer.WriteLine("salt==" + salt);

if (pepperLength > 0)   
 writer.WriteLine("pepperLength==" + pepperLength);

}

}

Metoda SaveSalt ukládá informace o soli a délce pepře do souboru identifikovaného pomocí hashID. Tento identifikátor je buď automaticky generován, nebo zadán uživatelem. Slouží k propojení konkrétního hashe s odpovídajícími parametry soli a pepře, aby bylo možné při ověřování nebo generování hashe použít správné údaje. Metoda LoadSalt pak získává tyto údaje pomocí hashID. Všechny tyto údaje se nachází v podsložce HashData.

Sůl je ukládaná jako text, ovšem pepř je uložen jako délka, nikoliv text. Ukládání pepře jako délku zajistí další bezpečnost. Kdyby se totiž někdo dostal k soli, mohl by započít útok pomocí duhových tabulek, který je vysoce efektivní. Díky neznámosti pepře se z útoku stává útok hrubou silou, který je velice neefektivní.

public bool CheckPepper(string originalText, string hashedText, int length, HashingAlgorithm algorithm, out string pepper)

{

pepper = "";

//Generate usable ASCII

List<char> usableChars = new List<char>();

for (int i = 0; i <= 255; i++)

{

usableChars.Add((char)i);

}

long totalCombinations = (long)Math.Pow(usableChars.Count, length);

for (long i = 0; i < totalCombinations; i++) // Finding Pepper

{

StringBuilder pepperTestBuilder = new StringBuilder();

long tempIndex = i;

for (int j = 0; j < length; j++) // Build the next pepper

{

pepperTestBuilder.Insert(0, usableChars[(int)  
 (tempIndex % usableChars.Count)]);

tempIndex /= usableChars.Count;

}

string pepperTest = pepperTestBuilder.ToString();

if (Hash(originalText + pepperTest, algorithm) == hashedText)

{

pepper = pepperTest;

return true; // Found match

}

}

return false;

}

Hledání pepře je poměrně těžké, ale je to výhodné pro maximální bezpečnost. Musíme projít všechny možné kombinace a postupně porovnávat vygenerovaný hash s cílovým hashem. Pokud nalezneme shodu, znamená to, že jsme našli odpovídající vstupní text. Pokud žádná kombinace neodpovídá, originální text a hash se neshodují.

Já používám pro pepř znaky UTF-8 neboli prvních 256 znaků Ascii, dokonce i prvních 32, které jsou netisknutelném, jelikož všude v programu je možnost vypisovaní v hexadecimální soustavě.

Pro rychlost programu a případné formátování používám třídu StringBuilder. Funguje stejně jako datový typ string, akorát s vlastnostmi a metodami (příklad Insert v kódu).

## Kontrolní součet

Kontrolní součet je jedna z dalších použití hashovacích funkcí. Nejčastěji se pro kontrolní součet používá hashovací funkce CRC32.

Ve třídě Hasher je celá jedna statická metoda pro kontrolní součet jménem FileChecksum.

public static string FileChecksum(string filename, HashingAlgorithm algorithm)

{

try

{

switch (algorithm)

{

case HashingAlgorithm.MD5:

{

using (MD5 md5 = MD5.Create())

using (FileStream stream = File.OpenRead(filename))

{

byte[] hash = md5.ComputeHash(stream);

return BitConverter.ToString(hash).

Replace("-", "").ToLower();

}

}

case HashingAlgorithm.SHA1:

(…)

case HashingAlgorithm.SHA256:

(…)

case HashingAlgorithm.SHA512:

(…)

case HashingAlgorithm.RIPEMD160:

(…)

case HashingAlgorithm.CRC32:

(…)

default: return "error";

}

}

catch(Exception ex)

{

(…)

}

}

Všechny funkce pracují na stejném principu jako MD5. Každá hashovací funkce v .NET framework má metodu ComputeHash, do které se dá dosadit FileStream a metoda si sama po kouscích vypracuje hash. Tato metoda čtení a zpracování kousek po kousku nezatěžuje RAM počítače a dají se takhle číst i obrovské soubory. Výstup se zapíše do proměné hash v bajtech a pomocí BitConverter.ToString je zapsaná jako hexadecimální číslo.

Správnost kontrolního součtu jsem si ověřil pomocí aplikace WinRAR, která vypisuje i kontrolní součet souboru v CRC32.

## Hledání kolize

Hledání kolizí nemá svoji vlastní třídu, místo toho se všechny metody nacházejí ve formuláři HashingCollisionForm, kde jsou taky použity. Samotné hledání kolize není složité, jediné věci, co jsou potřeba jsou generátor náhodných znaků a list. Horší bylo udělat hledání koliz, které dokáže použít všechny vlákna CPU.

private bool GenerateCollision(int threadNumber, Hasher.HashingAlgorithm algorithm, int length, long maxAttempts, bool useAttemps, bool saveLog, bool useHexForOutput, out string collision1, out string collision2)

{

collision1 = "";

collision2 = "";

List<string> hashedList = new List<string>();

List<string> textList = new List<string>();

Random random = new Random((int)(DateTime.Now.Ticks ^ threadNumber));

while (!foundCollision && !stopHashing && !attemptsRanOut)

{

Interlocked.Increment(ref attempts);

string randomText = GenerateRandomString(random, length);

string hashedValue = hasher.Hash(randomText, algorithm);

if (hashedList.Contains(hashedValue))

{

int collisionIndex = hashedList.IndexOf(hashedValue);

collision1 = textList[collisionIndex];

collision2 = randomText;

if (collision1 != collision2)

{

foundCollision = true;

(…)

return true;

}

}

else

{

hashedList.Add(hashedValue);

textList.Add(randomText);

}

if (maxAttempts > 0 &&

Interlocked.Read(ref attempts) >= maxAttempts)

{

attemptsRanOut = true;

return false;

}

}

Tohle je metoda GenerateCollision, která je základ celého algoritmu. V hlavičce metody jsou parametry jako threadNumber/threadID, která slouží pro multithreading, neboli použití více vláken najednou.

Random random = new Random((int)(DateTime.Now.Ticks \* threadNumber));

ThreadNumber se používá při generaci náhodného stringu a pomocí násobení se stávajícím časem zajistíme, že každé vlákno má jiné náhodné číslo a to při každém spuštění.

private string GenerateRandomString(Random random, int length)

{

char[] result = new char[length];

for (int i = 0; i < length; i++)

{

result[i] = (char)random.Next(33, 256);

}

return new string(result);

}

Metoda GenerateRandomString nám zajišťuje náhodný string podle délky, které máme hledat. Tato délka je na začátku algoritmu zadána uživatelem. String se generuje v rozmezí od 33 do 255, aby se při převádění na bajty použili všechny možnosti.

Metoda taky obsahuje několik vymožeností pro uživatele, jako je třeba maximální počet pokusů, výstup záznamu do listboxu, výstup v hexadecimálním zápisu a výstup kolize do souboru. Příklad vygenerovaného souboru.

Algorithm=CRC32

<STRING>

ÈÀÔ÷,+¾ÙÞs

+\_·^hLÍ2é

<HEX>

C8-C0-D4-F7-2C-2B-BE-D9-DE-73

2B-5F-80-B7-5E-68-4C-CD-32-E9

<HASH>

hash1: bff7a3dd

hash2: bff7a3dd

Další část kolizí je metoda CheckCollision, která vezme uživatelovi vstupy a zkontroluje, jestli se po zahešování rovnají či nikoliv. Vstup může být i přes soubor. Program obsahuje několik předpřipravených souborů, jako je například.

Algorithm=MD5

<STRING>

TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak

TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak

//Source: <https://x.com/realhashbreaker/status/1770161965006008570?mx=2>

Ve složce Collisions je také \_collisionInfo.txt, který popisuje vytváření vlastních souborů.

using (OpenFileDialog soubor = new OpenFileDialog())

{

soubor.DefaultExt = ".txt";

soubor.Filter = "Text files (\*.txt)|\*.txt|All files (\*.\*)|\*.\*";

soubor.InitialDirectory = Settings.DirectoryPathToCollisions;

if (soubor.ShowDialog() == DialogResult.OK)

{

using (StreamReader reader = new StreamReader(soubor.FileName))

{

(…)

while (!reader.EndOfStream && !gotInformation)

{

string line = reader.ReadLine();

//Ingoruje // a null

if (!line.StartsWith("//") && !String.IsNullOrEmpty(line))

{

if (line.StartsWith("Algorithm="))

{

//odmaže Algorithm=

string nextLine = line.Remove(0, 10);

Console.WriteLine(nextLine);

switch (nextLine)

{

case "MD5": algorithmTemp =

Hasher.HashingAlgorithm.MD5; break;

(…)

}

}

switch (line)

{

case "<STRING>":

{

format = CollisionDetectionFormat.STRING;

textCollision01 = reader.ReadLine();

textCollision02 = reader.ReadLine();

gotInformation = true;

break;

}

(…)

}

}

}

if (…) CheckCollision(algorithmTemp, textCollision01,

textCollision02, format);

(…)

}

}

}

Ukázka vstupu přes soubor. Pokud se soubor vybere, přečte se první řádek s hashovací funkcí a poté se vyhledává <string>, <hex> nebo <bin>, které představují vstupní formát. Přetížení CheckCollision si poté samo převede podle formátu, což je vlastní enumerátor.

## Multithreading[[12]](#footnote-7)

Multithreading je používání více vláken v programu najednou. Většina programů používají pouze jedno vlákno, jenže použití více vláken má několik výhod. Zabrání zablokování uživatelského rozhraní a dovolí použití celé síly procesoru. [33]

### Problémy s Multithreadingem

Použití několika vláken má ovšem i svoje problémy. Hlavní problém je komunikace mezi vlákny, protože vzniká šance na kolizi a případnou ztrátu dat. Pokud jedno vlákno čte hodnotu proměnné a jiné vlákno ji mezitím změní, může dojít ke ztrátě dat. Krásný příklad je x++. Jedno vlákno udělá x++, což znamená jak přečtení, tak zápis do paměti. Díky tomu bude vlákno 1 mít hodnotu proměnné x 13 a vlákno 2 hodnotu 12, což znamená ztrátu dat. Čím více vláken používáme, tím horší může být tento problém. Proto existují způsoby, jak správně pracovat s více vlákny. [33]

volatile bool foundCollision = false;

Základ je u proměnné, kde očekáváme, že bude použita mezi vlákny dáme volatile. Bohužel samotné volatile nezabrání problémům.

### Async a Await

private async void buttonGenerateCollision\_Click

{

(…)

List<Task> allTasks = new List<Task>();

(…)

allTasks.Add(Task.Run(() =>

CollisionThread(i, algorithm, maxAttempts, rngTextLenght,

false, checkBoxUseHex.Checked)));

(…)

await Task.WhenAll(allTasks);

(…)

}

Tady je krásný příklad použití více vláken správně. Metoda je async (asynchronní), což znamená že se nebude blokovat hlavní UI vlákno, tím pádem je uživatelské rozhraní pořád aktivní a reaguje na vstupy. S async je spojeno await, které čeká na provedení programu také bez blokování hlavního vlákna. Použití Task může být nahrazeno třídou Thread, ale třída Task je nadstavba třídy Thread s více metodami a jednodušším používáním. Task.Run započne operaci na novém vlákně. [34]

#### Lambda výraz

Lambda výraz () => nahrazuje potřebu dělat novou metodu a umožňuje předávání proměnných v hlavičce.

Příklad je nahrazení použití metody:

void Prace() { Console.WriteLine("Test"); }

Task.Run(Prace);

Se dá nahradit lambda výrazem:

Task.Run(() => { Console.WriteLine("Test"); });

### Interlocked.Increment

Další důležitá třída je Interlocked, specificky metoda Interlocked.Increment(ref), která zaručuje správné zpracování problému x++ mezi vlákny. Problém je když příjde více požadavků najednou, protože vlákno musí počkat než se akce dokončí. [35]

Interlocked.Increment(ref attempts); //attemps++

Příklad jednoduchého použití metody.

### Invoke

Pro vykonání nějaké akce ze sekundárního jádra se používá třída Invoke, která řekne hlavnímu vláknu (také známe jako UI vlákno) ať vykoná nějaký kód. Pokud se totiž pokusíme nějak poupravit UI komponentu ze sekundární vlákna, vyhodí se výjimka InvalidOperationException. [38]

this.Invoke((Action)(() => label.Text = "test"));

Příklad použití třídy Invoke.

### CancellationTokenSource

Poslední důležitá třída je CancellationTokenSource pro ukončení běhu programu. [36]

CancellationTokenSource token = new CancellationTokenSource();

(…)

//token.Cancel(); pro požadavek zrušení

if (cancellationTokenSource.Token.IsCancellationRequested)

{

return false; //zastaví program

}

## Výpočet prolomení hesla

Pro výpočet prolomení hesla používám vlastní třídu PasswordStrenghtCalculator, ve které jsou 3 jednoduché metody. Celý výpočet funguje na jednom vzorečku.

public static BigInteger Calculator(ulong passwordLenght,

ulong numberOfChars, BigInteger donePerSec,

out BigInteger speed, out bool overflowed)

{

overflowed = !TryPower(numberOfChars, passwordLenght,

out BigInteger number);

if (overflowed) (…)

speed = number / donePerSec;

return number;

}

Hlavní metoda je Calculator, která používá BigInteger. BigInteger je speciální dynamický datový typ, který pochází z třídy System.Numerics. Výhoda BigIntegeru oproti jednoduchým datovým typům jako Long je, že nemá žádnou horní či dolní mezi. Toho se docílí použitím více místa v paměti systému, takže velikost BigIntegeru je omezena velikostí RAM. [37]

Pro počítání mocniny jsem si udělal svoji vlastní jednoduchou TryPower metodu, která právě používá BigInteger.

private static bool TryPower(BigInteger basedValue, int exponent,

out BigInteger result)

{

result = 1;

try

{

for (int i = 1; i <= exponent; i++)

{

result \*= basedValue;

}

return true;

}

catch (Exception)

{

return false;

}

}

Funguje stejně jako TryParse u jednoduchých datových typů, akorát dělá mocninu o mocniteli větší jak 1, jelikož u tohoto vzorce nikdy nemůže být mocnitel menší jak 1.

Vzorec na celkový počet kombinací pro heslo. Počet znaků závisí na použití malých písmen (26 znaků), velkých písmen (26 znaků), číslic (10 znaků) a speciálních znaků jako třeba „!/@:? atd. (33 znaků).

Vzorec pro vypočítání celkové doby trvání neboli vyzkoušení všech možných kombinací. Kdybychom dělali útok hrubou silou, tak doba trvání je nejdelší čas, reálná doba trvání může být od instantní po dobu trvání. Většinou se může počítat s poloviční dobou doby trvání.

public static string Output(BigInteger numberSeconds)

{

BigInteger numberMilionYears = numberSeconds / 3155692600000000000;

BigInteger numberYears = (numberSeconds % 3155692600000000000)

/ 31556926;

BigInteger numberMonths = (numberSeconds % 31556926) / 2629749;

BigInteger numberDays = (numberSeconds % 2629749) / 86400;

BigInteger numberHours = (numberSeconds % 86400) / 3600;

BigInteger numberMinutes = (numberSeconds % 3600) / 60;

BigInteger numberSecondsLeft = numberSeconds % 60;

string s = "";

if (numberMilionYears > 0) s += numberMilionYears.ToString("N0") +

" " + Languages.Translate(575) + " " +

Languages.Translate(581) + ", ";

(…)

return s;

}

Metoda Output akorát vypíše dobu trvání v představitelné časové době.

## Slovníkový útok

Pro slovníkový útok používám vlastní třídu DictionaryAttack. Třída pracuje s více vlákny a používá metody, které byly v podkapitole MultiThreading vysvětleny. Také používá vlastnosti Get a Set pro zapouzdřenost.

public int Progress

{

get

{

if (LinesInTXT == 0) return 0;

int temp = (int)( CurrentLine / (double)LinesInTXT \* 100);

if (temp > 100) return 100;

else if (temp < 0) return 0;

else return temp;

}

}

Příklad použití vlastnosti Get pro získání progresu pro UI komponentu progressBar, která má rozmezí od 0 do 100 (int).

public void PasswordFinder(string fullPathToTXT, string[] passwords)

{

ResetValue();

using (StreamReader reader = new StreamReader(fullPathToTXT))

{

stopwatch.Start();

LinesInTXT = CountNumberOfLinesInFile(fullPathToTXT);

//Set Up

foundMatch = new bool[passwords.Count()];

lineFoundMatch = new long[passwords.Count()];

for (int i = 0; i < passwords.Count(); i++)

{

FoundMatch[i] = false;

lineFoundMatch[i] = -1;

}

while (!reader.EndOfStream && !UserAbandoned)

{

CurrentLine++;

string line = reader.ReadLine();

for (int i = 0; i < passwords.Count(); i++)

{

if (!FoundMatch[i]) //optimalizace

{

if (passwords[i] == line)

{

lineFoundMatch[i] = CurrentLine;

foundMatch[i] = true;

(…)

if (Array.TrueForAll(FoundMatch, value => value))

{

stopwatch.Stop();

return;

}

}

}

}

}

stopwatch.Stop();

}

}

PasswordFinder je hlavní metoda celého slovníkového útoku. Pro slovníkový útok potřebujeme textový soubor s hesly (fullPathToTXT) a hesla, které chceme najít (passwords). Metoda dovoluje hledání více hesel najednou. Spousta prvků je pro UI, jako třeba CurrentLine nebo použití Stopwatch pro měření času. Základ je ovšem přečíst si řádek a zkontrolovat se všemi hledanými hesly, jestli nejsou stejné. Pokud jsou, zapíše se řádek do proměnné lineFoundMatch a foundMatch se dá na true. Jestli jsou všechny prvky ve foundMatch nastavené na true,

if (Array.TrueForAll(FoundMatch, value => value))

našli jsme všechny hesla a algoritmus se může předčasně ukončit, jinak běží do konce souboru. Celá metoda je zanořena v Try a Catch.

## Útok duhovou tabulkou

Útok pomocí duhové tabulky je založen na stejném principu jako slovníkový útok. Jediný rozdíl je, že duhová tabulka už má před hashované data. Tabulka teda obsahuje nezahashovaný a zahashovaný vstup, kdyžto slovník obsahuje pouze nezahashovaný vstup. Příklad je (iloveyou=e4ad93ca07(…)) pro duhovou tabulku a (iloveyou) pro slovník.

Výhoda oproti slovníku je mnohem větší rychlost, jelikož se vstupní data nemusí hashovat. Hlavní nevýhodou oproti slovníku je velikost souboru. Slovník (pro tenhle příklad první 1. Mil hesel z rockyou.txt) má velikost 9 360KB, kdy duhová tabulka o stejném počtu hesel s hashovací funkcí SHA256 má 72 365KB. To je o 7krát více oproti slovníku. Obrana proti oběma útokům je použití soli a pepře.

Pro slovníkový útok mám 2 třídy. Jedna pro vygenerování duhové tabulky ze slovníku jménem RainbowTableGenerator a druhou pro samotný útok zvanou RainbowTableAttack. Obě třídy mají stejný základ jako třída pro slovníkový útok. Používají zapouzdřenost, cancellationToken pro ukončení procesu a mají metody pro použití více vláken.

### Generování duhové tabulky

Vytvoření duhové tabulky je velice jednoduché. Vezmeme slovník ve formátu .txt a po řádku generujeme duhovou tabulku.

public bool GenerateRainbowTable(string fileInputPath,

string fileOutputPath, Hasher.HashingAlgorithm hashingAlgorithm)

Potřebujeme cestu k slovníku, výstupní cestu a hashovací funkci.

using (StreamReader reader = new StreamReader(fileInputPath))

{

(…)

using (StreamWriter writer = new StreamWriter(fileOutputPath))

{

writer.WriteLine("algorithm==" + hashingAlgorithm.ToString());

while (!reader.EndOfStream)

{

if (cancellationTokenSource.Token.IsCancellationRequested)

{

RemoveFilesQuestion(fileOutputPath);

return false; // Stop if canceled

}

LinesProcessed++;

string line = reader.ReadLine();

string hash = hasher.Hash(line, hashingAlgorithm);

writer.WriteLine(line + "=" + hash);

}

}

}

Pomocí tříd StreamWriter a StreamReader zpracováváme po řádcích slovník a zapisujeme. Jako první řádek duhové tabulky je vždy jakou hashovací funkci používáme. Funkce je rozdělena pomocí dvou rovná se (==) a data pomocí jednoho (=).

algorithm==CRC32

123456=0972d361

12345=cbf53a1c

123456789=cbf43926

password=35c246d5

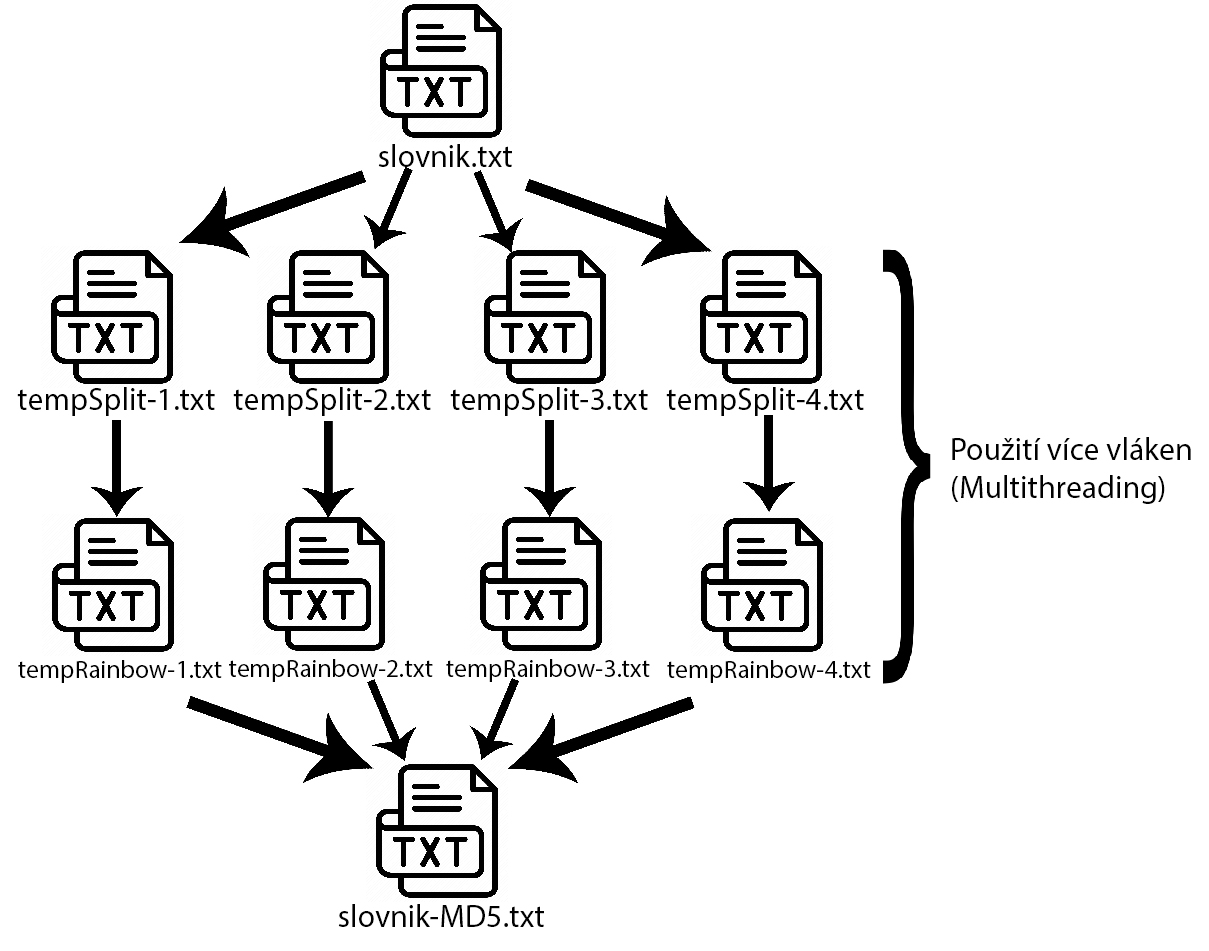
iloveyou=fc724cbd

princess=11ed43d0

Příklad začátku souboru duhové tabulky. Hashovací funkce je CRC32.

### Generování duhové tabulky pomocí více vláken

Generování duhové tabulky ze slovníku s multithreadingem vystavuje jeden zásadní problém. Více vláken nemůže přistupovat do jednoho souboru. Řešením je rozdělit soubor na více částí, díky čemuž si každé jedno vlákno vytvoří duhovou tabulku a pak se soubory sloučí do jednoho. Problém je příprava, protože rozdělování souborů a následné složení může dělat jenom jedno vlákno.



Na obrázku můžeme vidět přesně jak metoda GenerateRainbowTableMultiThread funguje při použití 4 vláken.

#### Metody pro mutlithreading

private void GenerateRainbowTableMultiThreadForSingleThread(string

fileInputPath, string fileOutputPath, Hasher.HashingAlgorithm

hashingAlgorithm)

{

try

{

using (StreamReader reader = new StreamReader(fileInputPath))

{

using (StreamWriter writer = new StreamWriter(fileOutputPath))

{

while (!reader.EndOfStream)

{

if (cancellationTokenSource.Token.

IsCancellationRequested) return;

string line = reader.ReadLine();

string hash = hasher.Hash(line, hashingAlgorithm);

writer.WriteLine(line + "=" + hash);

LinesProcessed++;

}

}

}

File.Delete(fileInputPath);

}

catch (Exception ex)

{

cancellationTokenSource.Cancel();

(…)

MessageBox.Show(Languages.Translate(11000) +

Environment.NewLine + ex.Message);

return;

}

}

Metoda GenerateRainbowTableMultiThreadForSingleThread je zavolaná pro každé jedno vlákno. Funguje stejně jako při použití jednoho vlákna.

(…)

List<Task> tasks = new List<Task>();

for (int i = 0; i < tempFilesInput.Length; i++)

{

string inputFile = tempFilesInput[i];

string outputFile = tempFilesOutput[i];

tasks.Add(Task.Run(() =>

{

if (cancellationTokenSource.Token.

IsCancellationRequested) return;

GenerateRainbowTableMultiThreadForSingleThread(inputFile,

outputFile, hashingAlgorithm);

}, cancellationTokenSource.Token));

}

(…)

Tohle je část metody,

public async Task<bool> GenerateRainbowTableMultiThread(

int numberOfThreadsUsed, string fileInputPath, string fileOutputPath,

Hasher.HashingAlgorithm hashingAlgorithm)

//Task<bool> because async method cant return anything unless like this

která právě zapíná pro každé vlákno metodu GenerateRainbowTableMultiThreadForSingleThread. Kvůli použití await musí být metoda async, jenže async metoda nemůže vracet základní datový typ, pouze Task<T>. To je důvod k použití Task<bool> v hlavičce metody.

#### Mazání dočasných souborů

Dočasné soubory jsou po dokončení generace vymazány, pokud nedošlo při generování k výjimce či ukončení procesu uživatelem.

private void RemoveFilesQuestion(string fileOutputPath)

{

if (MessageBox.Show(Languages.Translate(11003),

Languages.Translate(10030), MessageBoxButtons.YesNo,

MessageBoxIcon.Warning) == DialogResult.OK)

{

try

{

if (File.Exists(fileOutputPath))

{

File.Delete(fileOutputPath);

}

}

catch (Exception)

{

MessageBox.Show(Languages.Translate(11004),

Languages.Translate(10031), MessageBoxButtons.OK,

MessageBoxIcon.Information);

}

}

else //rename files to not cause problems

{

try

{

string rename = Directory.GetDirectories(fileOutputPath).FirstOrDefault();

string time = DateTime.UtcNow.ToString("yyyy,MM,dd-HH,mm,ss");

rename = Path.Combine(rename, "failedRainbowTable-" +

time + ".txt");

File.Move(fileOutputPath, rename);

}

catch (Exception ex)

{

MessageBox.Show(Languages.Translate(11007) +

Environment.NewLine + ex.Message, Languages.Translate(10020),

MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

Poté se pomocí metody RemoveFilesQuestion či RemoveFilesQuestionMultiThread program uživatele zeptá, jestli chce vymazat dočasné soubory, nebo je smazat. Jestli uživatel chce zachovat soubory, budou přejmenovány.

## Útok hrubou silou

Udělat útok hrubou silou není až tak těžké. Třída BruteForceAttack používá stejné principy jako předchozí třídy. Hlavní funkce je zkusit všechny možné kombinace znaků v určité délce a pokračovat, dokud se nenajde stejný hash jako u hesla. Uživatel musí zadat originální heslo nebo už zaheshované heslo, jaké všechny znaky použít (malé a velké písmena, číslice a speciální znaky), délka hledaného hesla pro zrychlení hledání, popřípadě začít od úplného začátku. Dodatečně časový limit a limit na pokusy.

public void SelectAllUsableChars(bool useLowerCase, bool useUpperCase,

bool useDigits, bool useSpecialChars)

{

int pocetZnaku = 0;

if (useLowerCase) pocetZnaku += 26;

if (useUpperCase) pocetZnaku += 26;

if (useDigits) pocetZnaku += 10;

if (useSpecialChars) pocetZnaku += 33;

int index = 0;

usableChars = new char[pocetZnaku];

if (useLowerCase)

{

for (char c = 'a'; c <= 'z'; c++)

{

usableChars[index++] = c;

}

}

(…) //stejný postup pro velké písmena a číslice

if (useSpecialChars)

{

// Add special characters

string specialChars = "!\"#$%&'()\*+,-

./:;<=>?@[\\] ^\_`{|}~";

foreach (char c in specialChars)

{

usableChars[index++] = c;

}

}

}

Metoda SelectAllUsableChars nastaví do pole znaků usableChars uživatelské vstupy, které má třída BruteForceAttack použít.

public BigInteger CalculateAllPossibleCombinations(

bool variablePasswordLength, int userPasswordLenght)

{

if (usableChars == null) return 0;

if (variablePasswordLength) // Variable password length

{

long temp = 0;

for (int length = 1; length <= maximumLenghtForBruteForce;

length++)

{

temp += Pow(usableChars.Length, length);

}

return temp;

}

else // Known password length

{

return Pow(usableChars.Length, userPasswordLenght);

}

}

Dále metoda CalculateAllPossibleCombinations spočítá všechny možné kombinace za použití BigInteger, aby výsledek nepřetekl. Metoda počítá variantou, že neznáme velikost hesla a počítá tak s maximem, které je nastaveno na začátku třídy (50 znaků, takže maximální počet kombinací je ).

public bool PasswordBruteForce((…))

if (useMultiThreading)

{

allPossibleCombinationsForOneThread = NumberOfAllPossibleCombinations

/ numberOfThreadsUsed;

BigInteger assignedStartIndex = allPossibleCombinationsForOneThread

\* threadID;

// Find the correct starting length

BigInteger tempCombinations = 0;

currentLength = 1;

while (assignedStartIndex >= tempCombinations +

CalculateAllPossibleCombinations(false, currentLength))

{

tempCombinations += CalculateAllPossibleCombinations(

false, currentLength);

currentLength++;

}

// Adjust index relative to the new length

index = assignedStartIndex - tempCombinations;

}

else

{

index = 0; // Single-threaded case starts from 0

}

Managementování více vláken je mnohem složitější než u jednoho vlákna. Jedno vlákno má začátek a konec. Dvě vlákna se musí rozdělit na dvě cca stejně velké půlky a každé vlákno musí začít na jiném indexu. To se musí provést i když neznáme délku.

//Move to next lenght

if (variablePasswordLenght && index >=

allPossibleCombinationsForCurrentLength)

{

index = 0;

currentLength++;

if (currentLength >= MaximumLenghtForBruteForce)

{

checkedAllPossibleCombinations = true;

break;

}

allPossibleCombinationsForCurrentLength =

CalculateAllPossibleCombinations(false, currentLength);

}

Když neznáme velikost hledaného hesla, musíme vyzkoušet všechny kombinace pro jednu délku a jakmile projedeme všechny kombinace pro jednu délku, přidáme k délce další znak a znova projedeme všechny kombinace. Jestli je celkový počet kombinací větší než maximální nastavená délka, metoda se ukončí.

private string GenerateText(BigInteger index, char[] allPossibleChars,

int length)

{

BigInteger baseSize = allPossibleChars.Length;

List<char> result = new List<char>();

while (index > 0)

{

result.Insert(0, allPossibleChars[(int)(index % baseSize)]);

index /= baseSize;

}

while (result.Count < length)

{

result.Insert(0, allPossibleChars[0]);

}

return new string(result.ToArray());

}

Metoda GenerateText si v hlavičce vezme index, všechny hledané znaky a délku a pomocí těchto dat vygeneruje text, který se poté zahashuje a vyzkouší. Příklad pro index 1, všechny možné znaky a délku 5 se vrátí aaaaa. Pro index 2 se aaaab, pro index 96 bude aaaba atd.

string tryText = GenerateText(index, usableChars, currentLength);

string hashedText = hasher.Hash(tryText, algorithm);

if (hashedText == userHashInput)

{

FoundPasswordBool = true;

FoundPassword = tryText; // Found the password

return true;

}

Vygenerovaný text podle indexu se zaheshuje a výstup se porovná s hashem hledaného hesla. Když jsou hashe stejné, našly se původní vstupní data. Pokud ne, pokračuje se dále.

Interlocked.Increment(ref attempts);

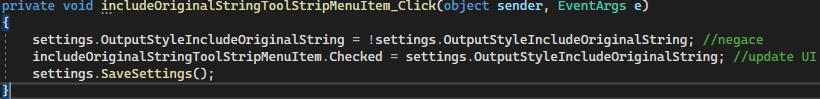
Interlocked.Increment se používá pro zvětšení celkových pokusů pro více vláken najednou.

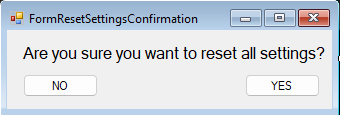
# Funkce Programu a UI

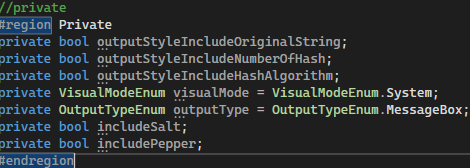
Pod tuto kapitolu patří všechno okolo programu a UI, které nejsou dostatečně technické či nějak spojené s hashováním. Patří sem nastavení výstupu, grafické rozhraní GUI, výběr jazyků, fungování tmavého či světlého režimu a práce se soubory.

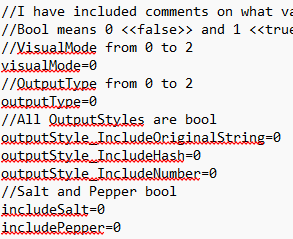
## Hlavní Formulář

## Nastavení

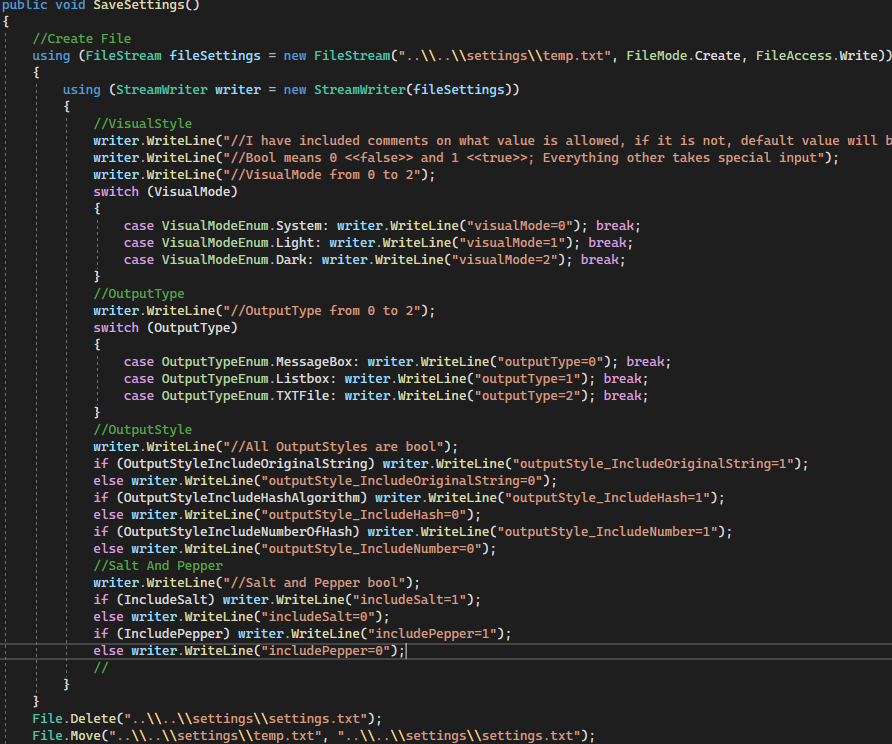
V programu jsou dva typy dat, které jsou potřeba uložit do souboru. První typ je přímo v UI v komponentě menu stripu, které fungují jako checkbox (buď ano nebo ne) či radiobutton (pouze jedna z několika možností). Příklad případu checkboxu je sůl a v output style přidání číselného řazení, jméno hashe a vstupní řetězce. Příklad radiobuttonu je „Output Type“, kde se dá zvolit „MessageBox“, „Listbox“ nebo do .txt souboru. Další typ dat jsou přímo v nastavení (ve stripmenu Options🡪Settings). Při kliknutí „Settings“ tlačítka se otevře „FormSettings“, což je modulární formulář (modulární znamená, že může být pouze jeden formulář a nedá se překlikávat mezi formuláři). Jediný rozdíl mezi tyto dvěma typy nastavení je kdy se uloží do souboru. Nastavení ve StripMenu se ukládá při každé změně.

Na obrázku jde vidět jeden z nastavení ve StripMenu (funguje jako checkBox).  
Ve formuláři je několik možností: Uložit (uloží vybrané nastavení) / Reset (všechny parametry se dají do základního přednastaveného nastavení) / Cancel (nic se neuloží). Při pokusu o reset vyskočí dialogové okno (obrázek dole), jestli chce uživatel doopravdy restartovat nastavení. Stejné s Cancel pokud byly provedeny jakékoliv změny (obrázek dole).

Nastavení používá vlastní skript, který má vždy načtené všechny proměnné (jde vidět na obrázku dole). Každá proměnná má vlastní Get a Set vlastnost.

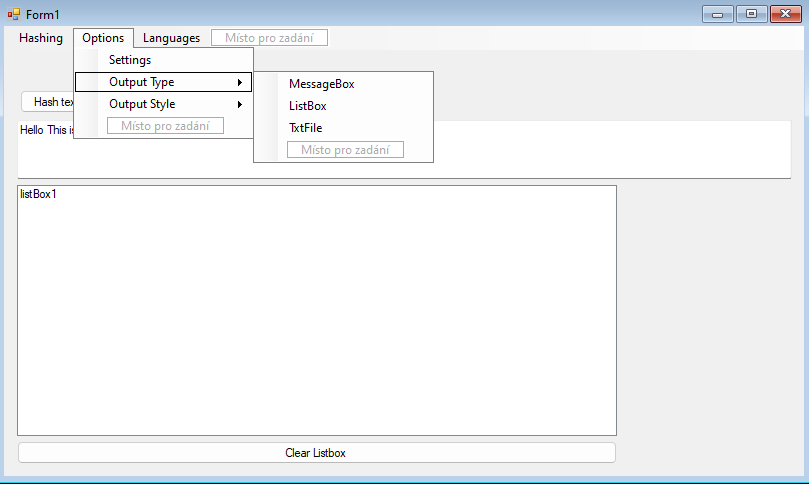
Díky tomuto systému je práce s nastavením velice jednoduchá a jednoduše rozšiřitelná. Soubor s nastavením má vlastní složku v projektu, kde je uložen settings.txt soubor (na obrázku jde vidět settings.txt soubor). Také podporuje komentáře pomocí // (jde vidět na obrázku).

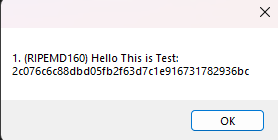
Jestli je soubor smazán či nějak poškozen, settings.txt je znovu vygenerován a uživatel je obeznámen.

Co se týče kodu, je velice jednoduchý. Přečte se řádek (pokud nezačíná //), pomocí String.Split se rozdělí na dva a přes Switch se zadávají data do proměnné a přes Int.Parse se zadávají hodnoty do proměnných. Proměnné bool jsou 0 či 1, string je text a enum funguje na indexech (visualMode a outputType).

Na obrázku jde vidět metoda SaveSettings(), která zapisuje data do settings.txt tak, že vytváří nový soubor, smaže starý a pomocí File.Move() přepíše jméno z temp.txt na settings.txt. Tímhle způsobem to je jednoduché a běžný uživatel si ničeho nevšimne.

## UI

User Interface (neboli uživatelské prostředí) je důležitá část tvorby jakéhokoliv softwaru. Pokud je UI špatné, tak to ztěžuje a zpomaluje práci s aplikací. Proto jsem se snažil udělat UI co nejvíce intuitivní a jednoduché. Toho se snažím docílit pomocí StripMenu komponentou v .NetFrameworku

Na obrázku jde vidět použití StripMenu v akci. V Hashing jsou další formuláře, které vedou do dalších formulářů a prací s hashema (Postupné hashování, Checksum souboru, simulace detekce kolize a hrubý hashovací útok). V Options je hlavní nastavení programu. „Settings“ otevře nový modulární formulář, kde je další nastavení (více v 3.6 Save/Load Systém). „Output Type“ je výstup, kam se zapíše výsledný hash, možnosti jsou

## Formátování výstupu

## Práce se soubory

## Jazyk

## Motivy

# Statistiky

Závěr

Vytvořená šablona maturitních prací obsahuje formální požadavky maturitních prací na SPŠT Třebíč. Jedná se zejména o upravené styly v dokumentu, podrobný popis jednotlivých částí maturitní práce a jejího obsahu, snadno editovatelné záhlaví a zápatí s automatickým číslováním stránek a propojení stylů se seznamy a obsahem.

Seznam použitých zdrojů

Seznam použitých symbolů a zkratek

|  |
| --- |
|  |

Seznam obrázků

[Obr. 2.1 Obsah **Chyba! Záložka není definována.**](#_Toc147493921)

[Obr. 2.2 Příklad umístění legendy obrázku **Chyba! Záložka není definována.**](#_Toc147493922)

Seznam tabulek

[Tab. 2.1 Legenda k tabulce **Chyba! Záložka není definována.**](#_Toc147493615)

Seznam příloh

Prázdná šablona maturitní práce

1. BCrypt je další hashovací funkce, která je ovšem značně pomalá. V roce vydání (1976) dokázala funkce zaheshovat skoro 4 hesla za sekundu. Vytvořena Nielsem Provosem a Davidem Mazièresem. [19] [↑](#footnote-ref-1)
2. Windows New Technology LAN Manager (NTLM) je bezpečnostní protokol, který slouží k ověřování identity uživatelů a integritě dat. NTLM umožňuje jednotné přihlašování **(SSO).** NTLM Používá MD4 a DES šifrovací funkci. [20] [↑](#footnote-ref-2)
3. Nejdůležitější funkce pro mě jako studenta, protože pracuji jak z domu, tak ze školy [↑](#footnote-ref-3)
4. CLI je zkratka pro Command Line Interface (rozhraní příkazového řádku) [↑](#endnote-ref-1)
5. GUI je zkratka pro Graphical User Interface (grafické uživatelské rozhraní) [↑](#endnote-ref-2)
6. SaaS je zkratka pro Software as a Servise (software jako služba**)** [↑](#endnote-ref-3)
7. Kanban (看板) z japonštiny znamená cedule či tabule [↑](#footnote-ref-4)
8. Just In Time (právě v čas) „umožňuje podniku vyrábět výrobky v určeném množství a určeném čase dle požadavků zákazníka.“ [28] [↑](#footnote-ref-5)
9. To-Do list znamená list věcí, které se ještě musí udělat. [↑](#endnote-ref-4)
10. Uniform Resource Locator (URL) znamená jednoznačné určení zdroje a používají to webové stránky jakožto doménovou adresu (příklad je seznam.cz) [↑](#endnote-ref-5)
11. celkově 12 tříd a 11 formulářů [↑](#footnote-ref-6)
12. Multithreading v překladu více vláknové řízení či použití více vláken [↑](#footnote-ref-7)