



SPŠT

Střední průmyslová škola Třebíč

Maturitní práce

HASHTESTER

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2024/2025 Kamil Franek

Zadání práce



SPŠT

Střední průmyslová škola Třebíč
Manželů Curieových 734, 674 01 Třebíč

Zadání ročníkové práce

Obor studia: **18-20-M/01 Informační technologie**

Celé jméno studenta:	Kamil Franek	Školní rok:	2024/2025
Třída:	ITA4		
Číslo tématu:	55		
Název tématu:	Hashovací algoritmy a jejich využití		
Rozsah práce:	15 - 25 stránek textu		

Specifické úkoly, které tato práce řeší:

Teoreticky zpracujte problematiku hashování. S pomocí nejméně dvou hashovacích algoritmů vytvořte aplikaci na demonstraci procesu hashování. Vstupem bude textový ručně zadaný textový řetězec a soubor. Naprogramujte samotné hashovací funkce s možností volby stupně zabezpečení a délky výstupního hodnoty. Navrhněte způsob testování dané hashovací funkce a způsob demonstrace změny výstupní hodnoty na změně vstupní hodnoty a tyto funkčnosti naprogramujte. Aplikaci realizujte v prostředí VS, jazyk C#, sdílení a ukládání přes GitHub.

Termín odevzdání: **28. března 2025, 23.00**

Vedoucí projektu: **Ing. Ladislav Havlát**

Oponent: **Ing. Drahomír Škárka**

Schválil: **Ing. Petra Hrbáčková, ředitelka školy**

ABSTRAKT

Maturitní práce na téma hashování, hashovací funkce, jejich využití a rozdíly v hashovacích funkcích. Zabývá se prací a problematikou spojenou s hashováním a vysvětlení použití hashování v IT s příklady pomocí programu. Tento dokument popisuje použitou technologii, praktiky a fungování samotného programu a všeho okolo. Výsledný program disponuje základními i rozšířenými funkcemi práce s hashy a soubory pro zvýšení efektivity práce. Dále disponuje simulací přihlašování, kybernetickými útoky a zobrazuje rozdíl mezi použitím a nepoužitím soli a pepře.

KLÍČOVÁ SLOVA

maturitní práce, výuka, hash, kybernetický útok, více vláknové řízení

ABSTRACT

Graduation thesis on hashing, hashing functions, their uses and differences in hashing functions. It discusses issues related to hashing and explains the use of hashing in IT with examples using a program. This document describes the technology used, the practices and the creation of the program itself and everything around it. The resulting program has basic and advanced hash and file handling features to increase efficiency. It also has simulations of login, cyber attacks and shows differences between using and not using salt and or pepper.

KEYWORDS

graduation thesis, education, hash, cyber-attack, multithreading

PODĚKOVÁNÍ

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

V Třebíči dne 28. 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 28. března 2025

podpis autora

Obsah

Úvod.....	8
1 Teorie hashování	9
1.1 Využití hashů	9
1.2 Použité hashe	9
1.2.1 MD5	9
1.2.2 SHA1/SHA256/SHA512	10
1.2.3 RIPEMD160.....	11
1.2.4 CRC32.....	11
1.3 Sůl, Pepř a jejich používání.....	12
1.4 Kybernetické útoky na hashe.....	13
1.4.1 Pravděpodobnost kolize (Narozeninový paradox)	14
1.5 Útoky	17
1.5.1 Duhové tabulky a slovníkové útoky.....	17
1.5.2 Útok hrubou silou.....	18
1.5.3 Pass the Hash	19
2 Použité programy a technologie.....	20
2.1 C# a .NET.....	20
2.2 Multithreading	20
2.2.1 Problémy s Multithreadingem.....	20
2.2.2 Async a await	21
2.2.3 Interlocked.Increment	22
2.2.4 Invoke.....	22
2.2.5 CancellationTokenSource	22
2.3 Visual Studio 2022.....	22
2.4 Visual Code	23
2.5 Git/Github a Github Desktop.....	23
2.6 Freelo.io.....	24
2.7 Wayback Machine.....	25
2.8 Online hashery.....	25
3 Základy hashování a hlavní formulář	26
3.1 Hashovací funkce a metody	26
3.2 Hlavní Formulář	28

3.2.1	Strip menu	29
3.3	Záznam.....	34
3.4	Třída Settings	35
3.5	Třída FormManagement	39
3.6	Jazyk a lokalizace.....	40
3.7	Frekvence UI formulář	45
3.8	Nastavení CPU a vláken formulář.....	46
3.9	Multi-Hashování.....	47
4	Pokročilé funkce programu.....	48
4.1	Postupné hashování.....	48
4.1.1	Kód pro postupné hashování.....	48
4.1.2	Formulář pro postupné hashování.....	49
4.2	Formulář pro vícenásobné hashování	50
4.3	Práce se solí a pepřem.....	50
4.3.1	Kód pro práci se solí a pepřem.....	51
4.3.2	Formulář pro práci se solí/pepřem	54
4.4	Kontrolní součet	58
4.4.1	Kód pro kontrolní součet.....	58
4.4.2	Formulář pro kontrolu souborů	59
4.5	Hledání kolizí.....	61
4.5.1	Kód.....	61
4.5.2	Formulář pro hledání kolizí.....	64
5	Útoky na hashe	66
5.1	Slovníkový útok	67
5.1.1	Kód pro slovníkový útok.....	67
5.1.2	Formulář pro slovníkový útok.....	69
5.2	Výpočet prolomení hesla	70
5.2.1	Kód.....	70
5.2.2	Formulář Čas k prolomení kalkulátor	72
5.3	Útok duhovou tabulkou	73
5.3.1	Generování duhové tabulky	73
5.3.2	Generování duhové tabulky pomocí více vláken	75
5.3.3	Formulář pro duhovou tabulku	78
5.4	Útok hrubou silou.....	79

5.4.1	Kód.....	79
5.4.2	Formulář pro útok hrubou silou	84
6	Statistika	85
6.1	Vyhledávání kolizí.....	85
6.2	Generování duhové tabulky	87
6.3	Útok hrubou silou.....	88
	Závěr.....	90
	Seznam použitých zdrojů	91
	Seznam výpisů	95
	Seznam grafů	98
	Seznam zkratk	99
	Seznam obrázků	100
	Seznam tabulek	102

Úvod

Cílem této ročníkové práce a programu je zjednodušení práce s hashy a pro výukové účely na školách. Proto je program pod licencí open source a se záměrem použití jako výukový nástroj. Další hlavní myšlenka byla jednoduchost používání i pro někoho, kdo nemá velké zkušenosti s počítačem.

V první kapitole popisuji teorii hashování, druhy hashovacích funkcí, rozdíly mezi nimi, kde se používají a další. Součástí je pár grafů, tabulek a rovnic pro vysvětlení několika jevů. Velkou částí práce se zabývám kybernetickými útoky na hashe, jejich vysvětlení fungování, efektivitu a jak se proti nim bránit.

V druhé kapitole se nachází použité programy, technologie a stránky díky kterým jsem byl schopen vytvořit program. Samotný program je napsán v jazyce C#.

Třetí kapitola je o popisu základního fungování programu a rozdělení do tříd a formulářů. Kapitola obsahuje obrázky UI programu a části kódu, které jsou poté popsány po třídách či metodách.

Čtvrtá kapitola se zaměřuje na pokročilou práci s hashy, používání soli a pepře v programu a hledání kolizí pomocí náhodné generace.

Pátá kapitola je celá o simulaci pár nejznámějších útoků, kde se snažíme dostat vstupní data z už zahashovaného výstupu. Útoky jsou slovníkový útok, útok pomocí duhové tabulky a útok hrubou silou.

V poslední kapitole se nachází spousta grafů a statistik o technologii multithreadingu, které program využívá pro najetí kolize, generování duhové tabulky a při útoku hrubou silou. Všechny grafy jsou popsány a navrhuji v nich své domněnky.

V závěru je celkové hodnocení práce z mého pohledu, spokojeností s finální verzí programu a moje vlastní dojmy.

1 Teorie hashování

Hashování je matematický algoritmus pro převod dat do předem určitého dlouhého výstupu podle algoritmu tzv. hashovací funkce. Hashe mají několik výtečných vlastností, vstupní data mohou 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, znamená, že proces je jednosměrný. Díky tomu se hash bere jako unikátní otisk vstupních dat. [1]

1.1 Využití hashů

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

1.2 Použité hashe

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

1.2.1 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 maticemi vypočítá výstup. Délka výstupu hashe je 128 bitů.

I přes jeho používání v tehdejší a dnešní době se v MD5 našla řada chyb, které by mohly být při ukládání hesel závažné. MD5 je totiž poměrně náchylný na takzvaný útok hrubou silou. [2][3]

```

/* Round 1. */
/* Let [abcd k s i] denote the operation
   a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

```

Obrázek 1 Příklad 1. kola hashovací funkce MD5

1.2.2 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]). [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 funkce pro integritu dat a ukládání hesel. [4][5]

„Google oznámil, že se mu podařilo prolomit bezpečnost hashovací funkce SHA-1. Od první publikované slabiny 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.“ [6]

Přikládám URL adresu projektu SHattered: <https://shattered.io/>

1.2.3 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 RIPE¹. 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. [7]

1.2.4 CRC32

CRC32 je hashovací funkce z rodiny cyclic redundancy check neboli cyklický redundantní součet 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.*“ [8] a CRC16, které se používá v Bluetooth, SD, X.25, v různých mobilních sítích, USB a mnoho dalších. [9] Wesley Petersonovi byla za návrh CRC udělena cena „Japan Prize“ v roce 1999. [10] Co se týče hashování důležitých dat, klíčů a bezpečnostních prvků nemá CRC mnoho využití, ale jakožto funkce pro kontrolní součet jich má nespočet.

1.2.4.1 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 $x^3 + x^2 + 1$. poslední bit reprezentuje 1, druhý bit x , třetí x^2 atd. CRC32 má až x^{31} a CRC16 až x^{15} .

¹ RIPE je zkratka RACE Integrity Primitives Evaluation, 1988-1992

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 doprava, 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)². [8][9]

1.3 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 útoky s duhovými tabulkami 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. [11]

Jako 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 dále, dokud se nevyzkouší všechny kombinace pepře. Pokud žádný z těchto pepřových kombinací nevejde, vstup je rozdílný od uloženého hashe.

² Realizace kódu je dostupná později v praktické části

Všechny hashe jsou šifrované v hashovací funkci MD5.

(bez soli) `TestingPassword123` == `1d898af5dbe7c9e07fc473e248f623a1`
(se soli) `a0_XTestingPassword123` == `5a6c1e14762baf73406b7267a8afae88`
(sůl i pepř) `a0_XTestingPassword123C` == `d7d0d822ce9faea482a0c5ae372d0ed0`

Výpis 1 Příklad rozdílu použití soli a pepře na výstupní hash

1.4 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 SHA-256 či SHA-512.

Pravděpodobnost si můžeme sami vypočítat pomocí jednoduchého vzorce. Počet všech kombinací u hashovací funkce je rovna $2^{\text{počet bitů na výstupu}}$. To znamená že CRC32 má šanci 1 ku 2^{32} (skoro 4.3 miliardy kombinací) neboli 0.0000000233 %. To se může zdát jako velice malá šance, jenomže tohle je šance jenom mezi 2 hashy. Když započítáme šanci každého s každým použitím narozeninového paradoxu, kdy hledáme kolizi každého s každým, počet všech kombinací se nám sníží na $2^{\frac{\text{počet počet bitů na výstupu}}{2}}$.

„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.“ [12]

Tabulka 1 Šance na najít kolizi

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

1.4.1 Pravděpodobnost kolize (Narozeninový paradox)

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.

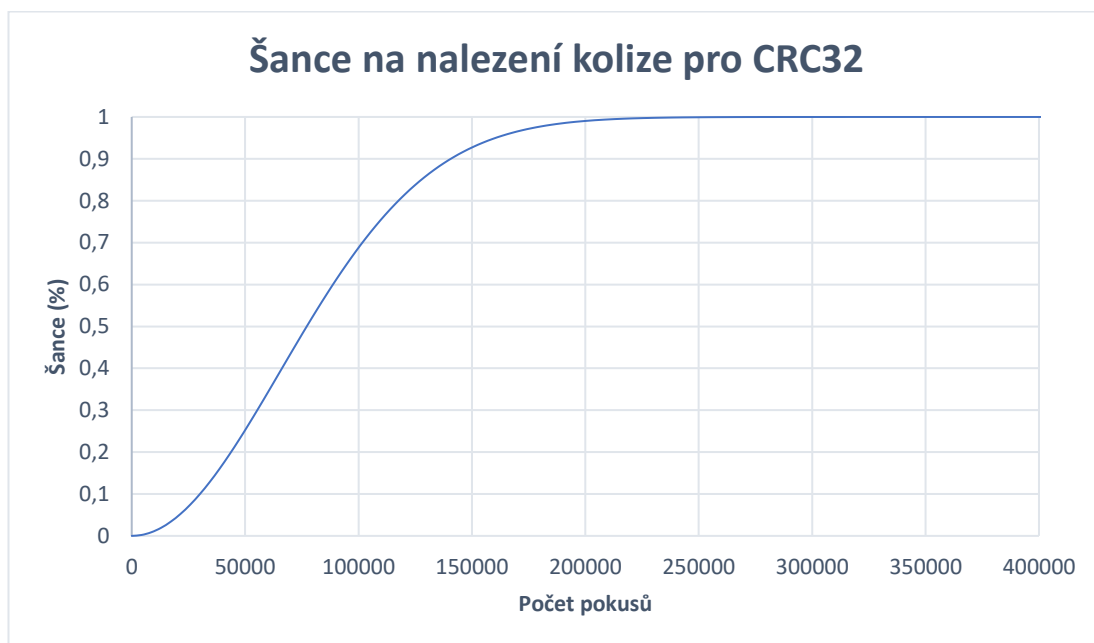
$$p = 1 - \frac{N!}{(N - k)! N^k}$$

Tento vzorec je velice přesný, ovšem pro počítání skoro až nereálný, jelikož při $N = 2^{160}$ jako to je u RipeMD-160 naprosto vyloučeno. Naštěstí si můžeme vzorec upravit do mnohem lepší formy.

$$p \geq 1 - e^{\frac{-k*(k-1)}{2N}}$$

Tato upravená verze není až tak přesná, ale dá se mnohem snadněji vypočítat. [13][12]

Graf 1 Šance na nalezení kolize pro CRC32



Jak můžeme vidět, hledání kolizí je exponenciální. Díky narozeninovému paradoxu se nám drasticky zmenšuje počet projití, které je potřeba na najetí jakékoliv jedné kolize. Pokud chceme najít pravděpodobnost kolize v určité šanci, třeba na 23 % (tedy $p=0.23$), můžeme si upravit naši rovnici a získat vzorec (\ln je přirozený logaritmus neboli logaritmus o základu e).

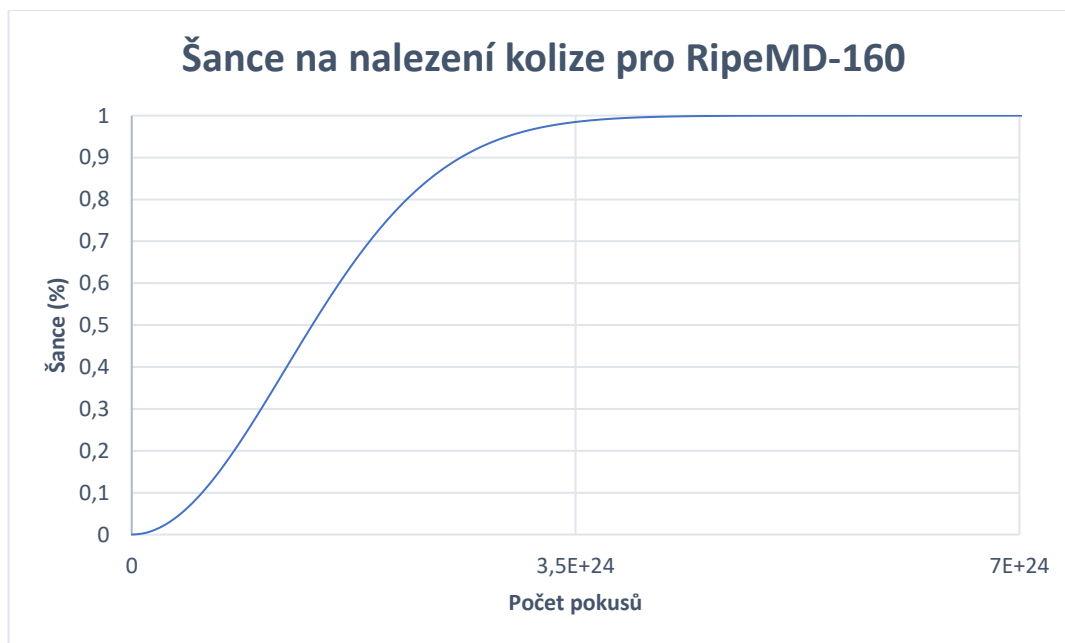
$$n = \sqrt{2N * \ln \frac{1}{1-p}}$$

Po dosazení vznikne $n = \sqrt{2(2^{32}) * \ln \frac{1}{1-0.23}}$, $n \doteq 47\,382$. Po projití 47 382 hashů máme 23% šanci na nalezení kolize.

Tento způsob výpočtu má dva problémy. Problém jedna je, že $P \neq 1.00$ (100%), jelikož bychom se dostali k dělení 0 v jmenovateli $\ln \frac{1}{1-p} \implies \ln \frac{1}{0}$, proto musíme dosadit co nejbližší číslo menší než 1. Další problém je zaokrouhlování, protože pomocí jedné metody ($2^{\frac{\text{počet počet bitů na výstupu}}{2}}$) nám vychází, že 50 % na najetí kolize je zaokrouhleně 65 tisíc, jenže pomocí našeho upraveného vzorečku pro p nám vychází zaokrouhleně 74 tisíc (tento útvar jde vidět v grafu).



Graf 2 Šance na nalezení kolize pro SHA256



Graf 3 Šance na nalezení kolize pro RipeMD-160

Tady dokazují, 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.

1.5 Útoky

Jak už bylo zmíněno, na hashe existuje pár specifických útoků. Většinou jsou útoky zamýšlené na získání hesla či nějakého klíče či certifikátu, jako to používají soubory cookies na webových prohlížečích. [14] Mezi nejrozšířenější jsou duhové tabulky a slovníkové útoky, útok hrubou silou a útok pass the hash.

1.5.1 Duhové tabulky a slovníkové útoky

Slovníkový útok a útok pomocí duhové tabulky můžou na první pohled vypadat jako jeden a ten samý útok, jenže mají jeden hlavní rozdíl. Oba útoky jsou nějaká databáze potenciálních hesel, jediný rozdíl je, že duhová tabulka vyhledává už zahashovaný 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 zahashovanými hesly. Zkoušíme všechna možná hesla ze seznamu, dokud nenajdeme shodu. 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 14 milionů často používaných hesel.

Útok s duhovými tabulkami použijeme, když jsme se nějakým způsobem dostali k už zahashovanému 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 některých hashovacích funkcí náročné na místo.

Tabulka 2 Rozdíly mezi útokem pomocí slovníku a duhové tabulky

Vlastnosti	Slovníkový útok	Duhová tabulka
Rychlost hashování	Záleží na použité funkci	Před hashováno
Náročnost na uložení	Malá (vstup)	Velká (vstup i výstup)
Použitelnost	Platí na všechny funkce	Pro specifickou funkci
Rychlost	Pomalá (hashování)	Rychlá (hledání shody)
Efektivita	Bezbranný proti soli/pepři	Bezbranný proti soli/pepři

1.5.2 Útok hrubou silou

Útok hrubou silou je nejzákladnější možný útok. 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 $Počet\ možných\ znaků^{Délka\ hesla}$. [15] V našem příkladu máme $10^8 = 100$ milionů možných kombinací. To může znít jako hodně, ale když jedna grafická karta RTX 4090 dokáže zvládnout 200 tisíc hashů za sekundu BCrypt³ a 300 milionů hashů NTLM⁴, tak 100 milionu najednou nezní tak krásně.

³ BCrypt je další hashovací funkce, která je ovšem značně pomalá. V roce vydání (1976) dokázala funkce zahashovat skoro 4 hesla za sekundu. Vytvořena Nielsem Provosem a Davidem Mazièrem. [18]

⁴ 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. [16]

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

1.5.3 Pass the Hash

Hlavní důvod, proč se hashování používá pro hesla je zabránit tomu, aby si někdo při komunikaci mohl jen tak vzít data a přechít si naše heslo. Když by někdo odposlouchával za hashované heslo, tak musí pomocí dalšího útoku 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 za hashované 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 sada nástrojů pro zkoušení tohoto útoku přímo ve Windows či Kali Linuxu⁵. [18][19]

⁵ (Windows: <https://github.com/byt3bl33d3r/pth-toolkit>) (Kali: <https://www.kali.org/tools/passing-the-hash/>).

2 Použité programy a technologie

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

2.1 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.“ [20]***Chyba! Nenalezen zdroj odkazů.**

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#.

2.2 Multithreadingⁱ

Multithreading je používání více logických 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. [21]

2.2.1 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. [21]

```
volatile bool foundCollision = false;
```

Výpis 2 Příklad použití volatile

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

2.2.2 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);
    (...)
}
```

Výpis 3 Příklad použití async a await v obsluze pro tlačítko

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. [22]

2.2.2.1 Lambda výraz

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

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

Výpis 4 Příklad bez lambda výrazu

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

Výpis 5 Příklad s lambda výrazem

2.2.3 Interlocked.Increment

Další důležitá třída je `Interlocked`, specificky metoda `Interlocked.Increment(ref)`, která zaručuje správné zpracování problému `x++` mezi vlákny. Problém je, když přijde více požadavků najednou, protože vlákno musí počkat, než se akce dokončí. [23]

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

Výpis 6 Příklad použití thread safe pro zvýšení proměnné

2.2.4 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`. [24]

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

Výpis 7 Příklad použití metody Invoke

2.2.5 CancellationTokenSource

Poslední důležitá třída je `CancellationTokenSource` pro ukončení běhu programu. [25]

```
CancellationTokenSource token = new CancellationTokenSource();  
(...)  
//token.Cancel(); pro požadavek zrušení  
if (cancellationTokenSource.Token.IsCancellationRequested)  
{  
    return false; //zastaví program  
}
```

Výpis 8 Příklad použití třídy CancellationTokenSource

2.3 Visual Studio 2022

Hlavní důvod vybrání si Visual Studia a .NET Framework je jednoduchost k přidání knihoven a manipulace s nimi. Hashování je provedeno přes `System.Security.Cryptography` knihovnu, která je v základu s .NET Framework. 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.

2.4 Visual Code

Visual Code je univerzální editor kódu. Je nejlepší s použitím jazyka Python či HTML 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 tříd, které jsem poté naimportoval do Visual Studia.

2.5 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.⁶ 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.“
[26]

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 menším uložištěm.

„Díky tomuto systému je spolupráce na projektu bezpečnější a jednodušší. Možná i to je důvod, proč Git používá více než 87 % vývojářů.“ [26]

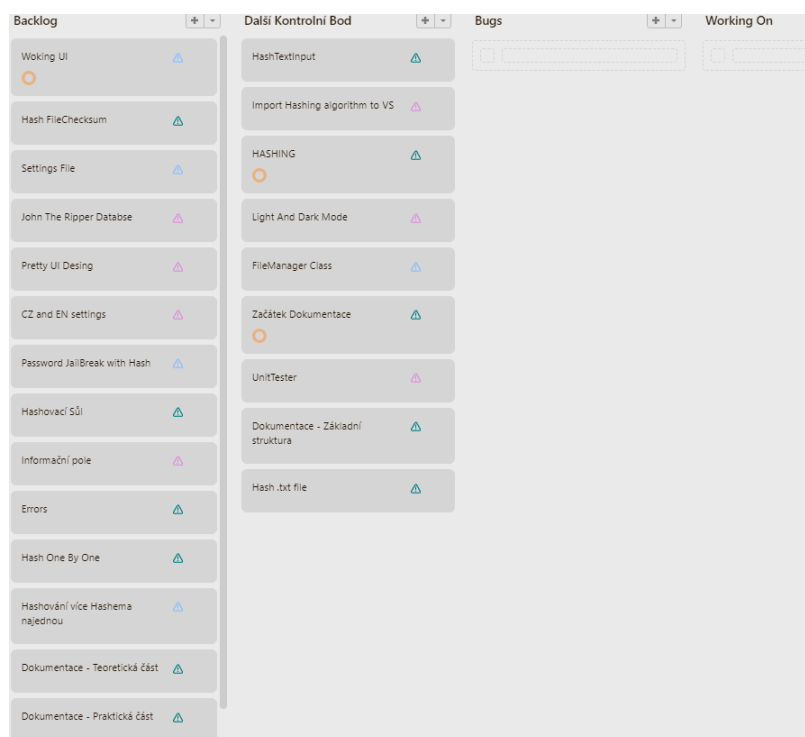
Github desktop je počítačová aplikace, která funguje stejně jako příkazový řádek v Git, akorát místo CLIⁱⁱ to je GUIⁱⁱⁱ a dělá to Git uživatelsky přívětivé.

⁶ Nejdůležitější funkce pro mě jako studenta, protože pracuji jak z domu, tak ze školy

2.6 Freelo.io

„Freelo je *online* aplikace s cílem usnadnit *řízení projektu* a zvýšit efektivitu podnikatelů a firem. Řadí se mezi *SaaS*^{iv} 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 kanban⁷. Kanban je založen na systému výroby Just In Time⁸ (JIT), která byla vymyšlena v japonské automobilové firmě Toyota. Zakladatel je Taiichi Ōno. 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^v. [28]



Obrázek 2 Příklad rozvrhnutí práce pomocí Freela

⁷ Kanban (看板) z japonštiny znamená cedule či tabule

⁸ Just In Time (právě v čas) „umožňuje podniku vyrábět výrobky v určeném množství a určeném čase dle požadavků zákazníka.“ [29]

2.7 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. [30]

Nejlepší je, že pro uživatele je všechno k podívání zadarmo bez potřeby účtu. U webových stránek stačí mít pouze URL^{vi} 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 nejpresnější datum, ale je to lepší než nic.

2.8 Online hashery

Online hashery jsou funkční algoritmy pro různé hashe, většinou ve formě stránky, které jsem primárně používal k porovnávání vygenerovaných hashů pro unit testy. Existuje spousta programů a stránek, já jsem používal <https://www.browserling.com/tools/all-hashes>, kvůli použití všech hashovacích funkcí, které používám v programu.

3 Základy hashování a hlavní formulář

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

3.1 Hashovací funkce a metody

Hlavní funkce programu jsou hashovací funkce a práce s nimi, proto jsem si vytvořil svoji vlastní třídu jménem *Hasher*. *Hasher* používá public enumerátor *HashingAlgorithm*, 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;
    }
}
```

Výpis 9 Metoda Hash ve třídě Hasher

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
    }
}
```

Výpis 10 Metoda HashMD5 ve třídě Hasher

Pro příklad HashMD5 je privátní metoda, která se stará o samotné hashování. Třída MD5 a další jsou součástí systémové třídy názvu *System.Security.Cryptography*, který je součástí .NET Framework. 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.

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.

```
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
}
```

Výpis 11 Metoda HashCRC32 ve třídě Hasher

```

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;
    }
}

```

Výpis 12 Metoda CRC32Table ve třídě Hasher, která je volána metodou HashCRC32

Pro generování tabulky se používá předem určená konstanta polynom. Ostatní CRC funkce jako například CRC16 budou mít polynom jiný.

3.2 Hlavní Formulář

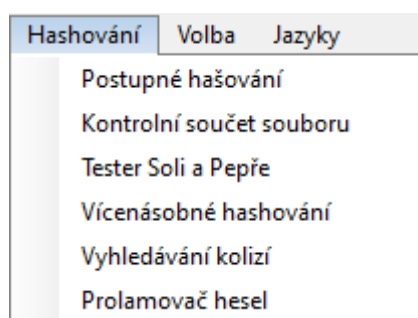
Obrázek 3 Vzhled hlavního formuláře

Hlavní formulář je první věc, kterou uživatel po spuštění programu uvidí. Disponuje jednoduchým víceřádkovým textboxem, komponentou strip menu, výběrem hashovacích funkcí a tlačítka. Hlavní formulář používá metody ze třídy *Hasher*.

3.2.1 Strip menu

Nahoře v hlavním formuláři se nachází komponenta strip menu, která slouží pro nastavení, uživatelské preference a jako přístup k dalším formulářům. Menu má tři hlavní sekce.

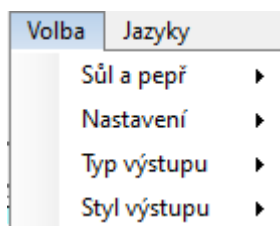
3.2.1.1 Hashování



Obrázek 4 Podsekce Hashování ve strip menu

Sekce hashování je pro otevření dalších formulářů, které všechny mají jinou funkci. Všechny fungují jako tlačítko. Každý jeden formulář je upřesněn později.

3.2.1.2 Volba

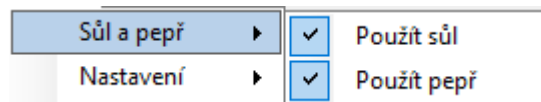


Obrázek 5 Podsekce Volba ve strip menu

Volba je sekce pro nastavení programu a uživatelské preference. Všechny informace jsou uloženy ve třídě *Settings*.



Obrázek 6 Podsekce Sůl a pepř v podsekci Volba



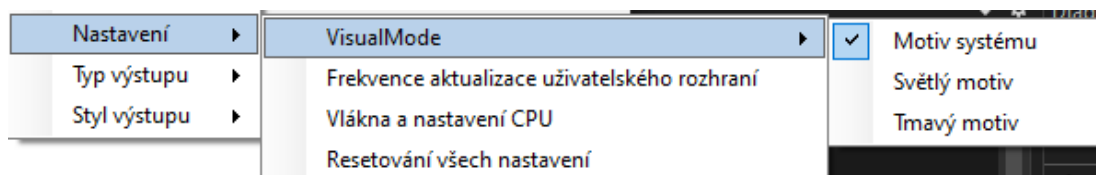
Obrázek 7 Podsekce Sůl a pepř se zapnutými možnostmi

Pod volbou sůl a pepř se nachází položky použít sůl a použít pepř, které fungují jako komponenty checkbox. V základu jsou možnosti vypnuté.

```
private void includeSaltToolStripMenuItem_Click_1(...)
{
    includeSaltToolStripMenuItem.Checked =
        !includeSaltToolStripMenuItem.Checked; //Negace tlačítka
    Settings.UseSalt = includeSaltToolStripMenuItem.Checked;
}
```

Výpis 13 Obsluha menu strip tlačítka použít sůl

Kód pro obsluhu je velice jednoduchý, tlačítko se nastaví na opačnou hodnotu a uloží do třídy *Settings*.



Obrázek 8 Podsekce VisualMode v nastavení

Pod sekci nastavení je několik možností. Motiv slouží pro nastavení vzhledu programu. Frekvence aktualizace uživatelského rozhraní (GUI) a vlákna a nastavení CPU jsou dodatečné formuláře, které mají vlastní sekci v dokumentaci. Resetování všech nastavení nastaví základní parametry ve třídě *Settings*.

3.2.1.3 Motivy

Všechny formuláře podporují světlý a tmavý režim. Základní hodnota je motiv systému, která vezme hodnotu z registru počítače. Světlý motiv nastaví světlý a tmavý zase tmavý. Možnosti fungují jako radiobutton, takže může být nastavena pouze jedna z možností. Ve třídě *Settings* je hodnota jako datový typ enum.

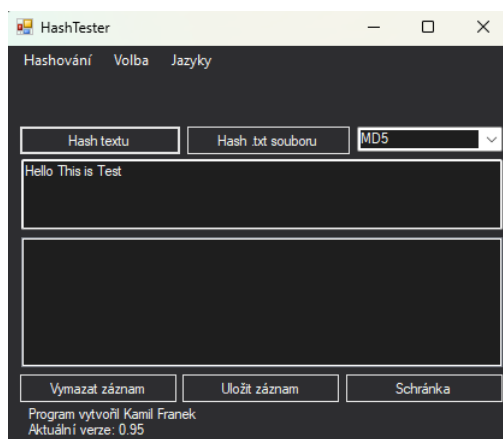
```

private static bool RegistryUseLightMode()
{
    string registryKey =
        @"Software\Microsoft\Windows\CurrentVersion\Themes\Personalize";
    string valueName = "AppsUseLightTheme";
    using (RegistryKey key = Registry.CurrentUser.OpenSubKey(registryKey))
    {
        if (key != null)
        {
            object value = key.GetValue(valueName);
            if (value is int intValue)
            {
                if (intValue == 1) return true;
                else return false;
            }
        }
    }
    Console.WriteLine("Couldnt find Registry for AppsUseLightTheme.");
    Console.WriteLine("Settings theme as light");
    return true;
}

```

Výpis 14 Metoda RegistryUseLightMode ve třídě Settings

Práce s registry je poměrně jednoduchá. Existuje třída *RegistryKey* pro práci s registry v operačním systému Windows. Stačí si akorát vzít registr, přečíst ho a zpracovat. Nejtěžší bylo najít kde se nachází specifický klíč. Naštěstí je to veřejná informace a použití registrů je velice častá záležitost pro aplikace a weby. Windows má v sobě aplikaci RegEdit pro případné upravování registrů.



Obrázek 9 Hlavní formulář ve tmavém motivu

3.2.1.4 Vlastní barevná paleta

.Net Framework nepodporuje tak docela tmavé barvy a formuláře. Jeden z důkazů je fakt, že MessageBox nejde nastavit na tmavé pozadí. Dále existuje spousta přednastavených barev ve třídě Colors (jak jde vidět na obrázku), ale většinou jsou buď moc světlé a barevné, nebo čistě černé. Proto jsem se rozhodl udělat si vlastní barevnou paletu.

AliceBlue	#FFF0F8FF	DarkTurquoise	#FF00CED1	LightSeaGreen	#FF20B2AA	PapayaWhip	#FFFEEFD5
AntiqueWhite	#FFFAEBD7	DarkViolet	#FF9400D3	LightSkyBlue	#FF87CEFA	PeachPuff	#FFFFDAB9
Aqua	#FF00FFFF	DeepPink	#FFFF1493	LightSlateGray	#FF778899	Peru	#FFCD853F
Aquamarine	#FF7FFFD4	DeepSkyBlue	#FF008FFF	LightSteelBlue	#FFB0C4DE	Pink	#FFFFC0CB
Azure	#FFF0FFFF	DimGray	#FF696969	LightYellow	#FFFFFFE0	Plum	#FFDDA0DD
Beige	#FFF5F5DC	DodgerBlue	#FF1E90FF	Lime	#FF00FF00	PowderBlue	#FFB0E0E6
Bisque	#FFFFE4C4	Firebrick	#FFB22222	LimeGreen	#FF32CD32	Purple	#FF800080
Black	#FF000000	FloralWhite	#FFFFFFAF0	Linen	#FFFAF0E6	Red	#FFFF0000
BlanchedAlmond	#FFFFEBCD	ForestGreen	#FF228B22	Magenta	#FFFF00FF	RosyBrown	#FFB8C8F8
Blue	#FF0000FF	Fuchsia	#FFFF00FF	Maroon	#FF800000	RoyalBlue	#FF4169E1
BlueViolet	#FF8A2BE2	Gainsboro	#FFDCDCDC	MediumAquamarine	#FF66CDAA	SaddleBrown	#FF8B4513
Brown	#FFA52A2A	GhostWhite	#FFF8F8FF	MediumBlue	#FF0000CD	Salmon	#FFFA8072
BurlyWood	#FFDEB887	Gold	#FFD700	MediumOrchid	#FFBA55D3	SandyBrown	#FFD4A460
CadetBlue	#FF66CCEE	Goldenrod	#FFDAA520	MediumPurple	#FF9370DB	SeaGreen	#FF2E8B57
Chartreuse	#FF7FFF00	Gray	#FF808080	MediumSeaGreen	#FF3CB371	SeaShell	#FFF5F5EE
Chocolate	#FFD2691E	Green	#FF008000	MediumSlateBlue	#FF7B68EE	Sienna	#FFA0522D
Coral	#FF7F7F50	GreenYellow	#FFADFF2F	MediumSpringGreen	#FF00FA9A	Silver	#FFC0C0C0
CornflowerBlue	#FF6495ED	Honeydew	#FFF0FF0	MediumTurquoise	#FF48D1CC	SlateBlue	#FF6A5ACD
Cornsilk	#FFFFFF8D	HotPink	#FFFF69B4	MediumVioletRed	#FFC71585	SlateGray	#FF708090
Crimson	#FFDC143C	IndianRed	#FFCD5C5C	MidnightBlue	#FF191970	Snow	#FFFFFFFA
Cyan	#FF00FFFF	Indigo	#FF4B0082	MintCream	#FFF5FFFA	SpringGreen	#FF00FF7F
DarkBlue	#FF00008B	Ivory	#FFFFFFF0	MistyRose	#FFFEE4E1	SteelBlue	#FF4682B4
DarkCyan	#FF008B8B	Khaki	#FFF0E68C	Moccasin	#FFFEE4B5	Tan	#FFD2B48C
DarkGoldenrod	#FFB8860B	Lavender	#FFE6E6FA	NavajoWhite	#FFFDEAD	Teal	#FF008080
DarkGray	#FFA9A9A9	LavenderBlush	#FFFFFFF05	Navy	#FF000080	Thistle	#FFD8BFD8
DarkGreen	#FF006400	LawnGreen	#FF7FCF00	OldLace	#FFFD5E6	Tomato	#FFF6347
DarkKhaki	#FFBDB76B	LemonChiffon	#FFFFFACD	Olive	#FF808000	Transparent	#00FFFFFF
DarkMagenta	#FF8B008B	LightBlue	#FFADD8E6	OliveDrab	#FF6B8E23	Turquoise	#FF40E0D0
DarkOliveGreen	#FF556B2F	LightCoral	#FFF08080	Orange	#FFFA500	Violet	#FFEE82EE
DarkOrange	#FFFF8C00	LightCyan	#FEE0FFFF	OrangeRed	#FFFA4500	Wheat	#FFF5DEB3
DarkOrchid	#FF9932CC	LightGoldenrodYellow	#FFFAFAD2	Orchid	#FFDA70D6	White	#FFFFFFF0
DarkRed	#FF8B0000	LightGray	#FFD3D3D3	PaleGoldenrod	#FFEE8AA	WhiteSmoke	#FFF5F5F5
DarkSalmon	#FFE9967A	LightGreen	#FF90EE90	PaleGreen	#FF98FB98	Yellow	#FFFF0000
DarkSeaGreen	#FF8FBC8F	LightPink	#FFF0B6C1	PaleTurquoise	#FFAFEEEE	YellowGreen	#FF9ACD32
DarkSlateBlue	#FF483D8B	LightSalmon	#FFFA07A	PaleVioletRed	#FFDB7093		
DarkSlateGray	#FF2F4F4F						

Obrázek 10 Všechny přednastavené barvy ve třídě Colors

(<https://learn.microsoft.com/enus/dotnet/api/system.windows.media.colors?view=windowsdesktop-9.0>)


```

public class CustomColorTable : ProfessionalColorTable
{
    private readonly bool isDarkTheme;

    public CustomColorTable(bool isDarkTheme)
    {
        this.isDarkTheme = isDarkTheme;
    }

    public override Color MenuStripGradientBegin
    {
        get
        {
            if (isDarkTheme) return Color.FromArgb(45, 45, 48);
            // Dark mode background
            else return SystemColors.Control;
            // Default light mode background
        }
    }
    (...)
}

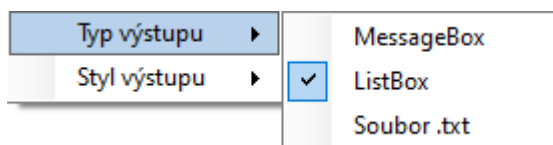
```

Výpis 15 Příklad třídy CustomColorTable a vlastnosti MenuStripGradientBegin

Rychlý příklad třídy *CustomColorTable*, která dědí od *ProfessionalColorTable*. V konstruktoru stačí zadat světlý či tmavý režim. Některé vlastnosti vrací přímo třídu *Colors* či *SystemColors*, u některých je nastavena hodnota v RGB^{vii}. 90 % času je hodnota světlejší nebo tmavší šedá.

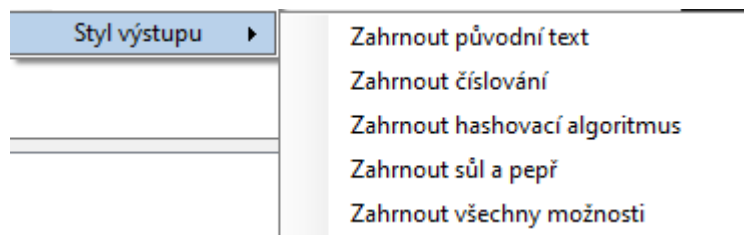
3.2.1.5 Uživatelské preference

Zpátky do sekce nastavení, Typ výstupu a Styl výstupu jsou nastavení pro uživatelskou preferenci.



Obrázek 11 Ukázka strip menu typu výstupu

Typ výstupu funguje jako komponenta radiobutton. Program podporuje výstup do MessageBox, Listboxu (záznamu) a do nového .txt souboru.



Obrázek 12 Podsekce Styl výstupu v nastavení

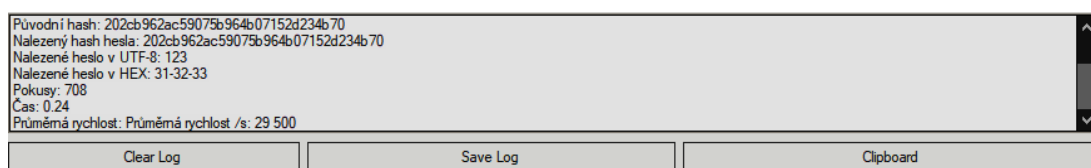
Styl výstupu formátuje výsledný text podle několika možností. Všechny možnosti fungují jako checkbox. Když jsou všechny 4 možnosti zahrnuty, zaškrtně se *Zahrnout všechny možnosti*. Pomocí toho se dají rychle nastavit či odebrat všechny možnosti. Pro každodenní práci s hashy to není moc užitečné, ale pro amatérské ukládání či výuku to má své využití.

```
Heslo 1234: 37289babd4ea26833b17c8b6513ea02b
1. 37289babd4ea26833b17c8b6513ea02b
(MD5) 37289babd4ea26833b17c8b6513ea02b
37289babd4ea26833b17c8b6513ea02b (Sůl: sul)
1. (MD5) Heslo 1234: 37289babd4ea26833b17c8b6513ea02b (Sůl: sul)
```

Obrázek 13 Příklad všech možností samostatně a poté všechny najednou.

3.3 Záznam

Většina formulářů obsahuje komponentu listbox a 3 tlačítka, která dohromady slouží pro záznam neboli log. Do záznamu se dají ukládat výsledné hashe a slouží celkově jako informace o tom, co program dělá.



Obrázek 14 Fungování záznamu

Příklad použití záznamu. Vymazat záznam odstraní všechny položky ze záznamu. Tlačítko uložit záznam otevře *SaveFileDialog* a následně všechny položky uloží jako textový soubor. Schránka funguje jako zkratka CTRL + C. Uživatel si označí jeden prvek v záznamu a po kliknutí se text zkopíruje do počítačové schránky, kde se pak pomocí zkratky CTRL + V může použít dále.

```
private void buttonClipboard_Click(object sender, EventArgs e)
{
    try
    {
        if (listBoxLog.SelectedItem != null)
            Clipboard.SetText(listBoxLog.SelectedItem.ToString());
        else (...)
    }
    catch (Exception)
    {
        (...)
    }
}
```

Výpis 16 Metoda při stisknutí tlačítka kopírovat ve formuláři

Samotná operace je na jeden řádek s použitím statické třídy Clipboard a metody SetText. Listbox pro záznam je nastaven, aby povoloval pouze jeden označený text najednou.

3.4 Třída Settings

Třída Settings je statická třída, která nastavuje všechny možnosti programu a ukládá je do souboru. Třída používá zapouzdřenost, vlastnosti a enum.

```
public static OutputTypeEnum OutputType
{
    get { return outputType; }
    set { outputType = value; }
}

public enum OutputTypeEnum
{
    MessageBox,
    Listbox,
    TXTFile
}
```

Výpis 17 Příklad použití enum a vlastnosti ve třídě Settings

```
public static string DirectoryPathToSettings
{
    get
    {
        string path = Path.Combine(DirectoryExeBase, "Settings/");
        if (!Directory.Exists(path)) Directory.CreateDirectory(path);
        return path;
    }
}
```

Výpis 18 Vlastnost DirectoryPathToSettings

Některé vlastnosti jsou pro práci se soubory, jako třeba *DirectoryPathToSettings*. Všechny tyto vlastnosti jsou pouze get a používají cestu hlavního .exe souboru.

Třída disponuje čtyřmi metodami, *ResetSettings*, *SaveSettings*, *LoadSettings* a *InitialFolderChecker*. Při zapnutí programu se jako první spouští *InitialFolderChecker*, který kontroluje existenci všech složek a souborů potřebných ke správném běhu programu. Funguje to jako taková malá instalace. V základu program obsahuje anglický jazyk ve složce *Languages*, několik kolizí ve složce *Collisions* a prvních 5 tisíc řádku *rockyou.txt*. Systém složek jsem zvolil, protože to dovoluje flexibilnější práci s programem. Pro příklad, uživatel si může přidat jakýkoliv seznam do složky *Wordlists*. Jestli se stane chyba v tomto kroku, nebude program fungovat a sám se vypne. Program musí mít zapisovací a čtecí práva na soubory, jinak se ani nenačte jazyk.

```
//Collisions
if (!Directory.Exists(Settings.DirectoryPathToCollisions))
{
    Directory.CreateDirectory(Settings.DirectoryPathToCollisions);
}
string s = Path.Combine(
    Settings.DirectoryPathToCollisions, "_collisionInfo.txt");
if (!File.Exists(Path.Combine(s)))
{
    using (StreamWriter writer = new StreamWriter(s))
    {
        //Zapsání souboru
        (...)
    }
}
```

Výpis 19 Část metody InitialFolderChecker ve třídě Settings

Další metoda je *LoadSettings*, která se zavolá po *InitialFolderChecker* při zapnutí programu. Ve složce *Settings* je soubor *settings.txt*, ve kterém je uloženo všechno nastavení programu. Jestli soubor neexistuje, spustí se metoda *ResetSettings*, která nastaví základní hodnoty. *SaveSettings* vezme tyto nastavené hodnoty a uloží je do souboru.

```

public static void ResetSettings()
{
    VisualMode = VisualModeEnum.System;
    OutputType = OutputTypeEnum.Listbox;
    OutputStyleIncludeHashAlgorithm = false;
    OutputStyleIncludeNumberOfHash = false;
    OutputStyleIncludeOriginalString = false;
    OutputStyleIncludeSaltPepper = false;
    UseSalt = false;
    UsePepper = false;
    SaveSettings();
}

```

Výpis 20 Metoda ResetSettings ve třídě Settings

```

public static void SaveSettings()
{
    try
    {
        //Create File
        (...)
        //Create Directory if it doesnt exist
        if (!Directory.Exists(Settings.DirectoryPathToSettings))
        {
            Directory.CreateDirectory(Settings.DirectoryPathToSettings);
        }
        using (FileStream fileSettings = new FileStream(...))
        {
            using (StreamWriter writer = new StreamWriter(fileSettings))
            {
                (...)
                switch (VisualMode)
                {
                    case VisualModeEnum.System:
                        writer.WriteLine("visualMode=0"); break;
                    (...)
                }
                (...)
            }
        }
        File.Delete(settingsPathToFileSettings);
        File.Move(settingsPathToFileTemp, settingsPathToFileSettings);
    }
    catch (UnauthorizedAccessException)
    {
        (...)
        Application.Exit();
    }
    (...)
}

```

Výpis 21 Metoda SaveSettings ve třídě Settings

Metoda *SaveSettings* vytvoří dočasný soubor, do kterého jsou zapsané všechny potřebné informace (v kódu jsem nechal příklad enum motivu), poté je originální *settings.txt* soubor smazat a dočasný soubor je přejmenován. Stejně jako *InitialFolderSetup* je potřeba práva na zápis a čtení, pokud tyto práva nejsou, aplikace se ukončí.

Tento druh ukládání mám z postarší videohry *Need For Speed: Most Wanted 2005*, kterou jsem hrál jako dítě. Všechna volba nastavení je v textovém souboru. Když se přešlo z 4:3 monitorů na 16:9 a zvláště na Full HD, tak to vnitřní nastavení ve hře nepodporovalo, i když herní engine ano. Díky tomu, že nastavení byl jeden textový soubor se dala velikost obrazovky ručně nastavit velice jednoduše, a dokonce tam bylo i pár nastavení, která ani ve hře nebyla dostupná. Je to tak jednoduché a uživatelsky přívětivé řešení, že jsem na této filozofii postavil i svůj program.

```
//Varování! Pokud za znakem = není nic, nastaví se výchozí hodnota.  
//Bool znamená 0 <<false>> a 1 <<true>>; vše ostatní vyžaduje specifický vstup  
//Připojil jsem komentáře o tom, jaké hodnoty jsou povoleny. Jinak se nastaví  
výchozí hodnota  
//Vizuální režim od 0 do 2  
visualMode=0  
//UpdateUI v milisekundách  
//<<8 - 1000>> celé číslo  
UIupdateInMS=500  
//Počet maximálně použitých vláken v procentech (%)  
//<<1 - 100>> celé číslo  
threadsUsagePercentage=50  
//Preferovaný jazyk  
language=Čeština  
//Typ výstupu od 0 do 2  
outputType=1  
//Všechny OutputStyles jsou bool  
outputStyle_IncludeOriginalString=0  
outputStyle_IncludeHash=0  
outputStyle_IncludeNumber=0  
outputStyle_IncludeSaltPepper=0  
//Sůl a pepř bool  
useSalt=0  
usePepper=0
```

Výpis 22 Textový soubor settings.txt ve složce Settings

Stačí si otevřít textový soubor, upravit a hotovo. Dokonce systém podporuje poznámky, takže uživatel ví, co může a nemůže zadat. Jako bonus jsou poznámky přeloženy podle jazyka.

```
//Metoda LoadSettings
case "outputType":
{
    try
    {
        if (data[1] == "0") OutputType = OutputTypeEnum.MessageBox;
        else if (data[1] == "1") OutputType = OutputTypeEnum.Listbox;
        else if (data[1] == "2") OutputType = OutputTypeEnum.TXTFile;
        else OutputType = OutputTypeEnum.Listbox;
    }
    catch (Exception)
    {
        OutputType = OutputTypeEnum.Listbox;
    }
    break;
}
}
```

Výpis 23 Příklad čtení nastavení ze souboru v metodě LoadSettings ve třídě Settings

Pokud zadá nějaký vstup špatně, tak bude nastavena základní hodnota v metodě *LoadSettings*.

3.5 Třída FormManagement

Statická třída *FormManagement* je třída, která usnadňuje práci s formuláři, jelikož se spousta věcí v nich opakuje.

```
public static void SaveLog(ListBox listbox, Form form);
public static int NumberOfThreadsToUse();
public static bool UseMultiThread();
public static bool UseLightMode();
private static bool RegistryUseLightMode();
public static void SetUpFormTheme(Form form);
private static void ApplyThemeToMenu(ToolStrip menuStrip,
                                     Color backColor, Color textColor);
private static void ApplyThemeToMenuItem(ToolStripMenuItem menuItem,
                                         Color backColor, Color textColor);
```

Výpis 24 Metody ve třídě FormManagement

SetUpFormTheme je metoda, která inicializuje třídu *CustomColorTable* a nastavuje motiv. Metoda má pod sebou všechny soukromé metody. *SaveLog* ukládá záznam do souboru. Metody *NumberOfThreadsToUse* a *UseMultiThread* by dávali větší smysl ve třídě *Settings*, upřímně nevím, proč jsem je dal sem.

3.6 Jazyk a lokalizace

V tuto dobu jsou podporované dva jazyky, angličtina a čeština. Bez problému by stačila čeština, ale jelikož je program dělán za pomoci GitHubu, na který to mám v plánu i vydat, tak jsem se rozhodl přeložit celý program do angličtiny za pomoci DeepL online překladače, jenž funguje na AI^{viii}.

```
private static Dictionary<int, string> dictionary = null;
```

Výpis 25 Příklad, jak vytvořit datový typ Dictionary

Statická třída *Languages* pracuje na datovém typu *Dictionary* a ID systému, kterým jsem se inspiroval z Android Studio.

```
public static string[] AllLanguages()
{
    List<string> list = new List<string>();
    string[] temp = Directory.GetFiles(Settings.DirectoryToLanguages);
    foreach (string s in temp)
    {
        if (!Path.GetFileName(s).StartsWith("_"))
        {
            list.Add(Path.GetFileNameWithoutExtension(s));
            //Console.WriteLine(s);
        }
    }
    return list.ToArray();
}
```

Výpis 26 Metoda AllLanguages ve třídě Languages

Na začátku programu se spustí metoda *AllLanguages*, která ze složky *Languages* vytáhne všechny textové soubory. Ty se poté bez přípony dají do strip menu v hlavním formuláři pomocí metody *AddLanguagesToMenu*.


```

private void AddLanguagesToMenu()
{
    string[] array = Languages.AllLanguages();
    if (array != null && array.Length != 0)
    {
        bool firstItem = true;
        string dictionaryNameLoad = "";
        (...)
        foreach (string item in array)
        {
            ToolStripMenuItem newItem = new ToolStripMenuItem(item);
            newItem.Name = item;
            if (possibleNames.Contains(newItem.Name))
            {
                dictionaryNameLoad = newItem.Name;
                firstItem = false;
            }
            else if (firstItem)
            {
                dictionaryNameLoad = newItem.Name;
                firstItem = false;
            }
            //Set item to do something
            newItem.Click += (sender, e) =>
            {
                Languages.LoadDictionary(newItem.Name);
                foreach (ToolStripItem menuItem in
                    languagesToolStripMenuItem.DropDownItems)
                {
                    if (menuItem is ToolStripMenuItem toolStripMenuItem)
                    {
                        toolStripMenuItem.Checked = false;
                    }
                }
                newItem.Checked = true;
                Settings.SelectedLanguage = newItem.Name;
                FormUISetUpLanguages(); //set new language
                Settings.SaveSettings();
            };
            languagesToolStripMenuItem.DropDownItems.Add(newItem);
        }
        Languages.LoadDictionary(dictionaryNameLoad);
        foreach (ToolStripItem menuItem in
            languagesToolStripMenuItem.DropDownItems)
        {
            if (menuItem is ToolStripMenuItem toolStripMenuItem &&
                toolStripMenuItem.Name == dictionaryNameLoad)
            {
                toolStripMenuItem.Checked = true;
                break;
            }
        }
    }
}

```

Výpis 27 Metoda AddLanguagesToMenu ve třídě Languages

Ke každému jazyku se přidá metoda, která se spustí po kliknutí. Ta akorát nastaví, aby byla jediná zaškrtnuta, uloží nastavení a přepne jazyk, aby používal jiný jazyk.

```
public static bool LoadDictionary(string nameOfLanguage)
{
    try
    {
        string pathToFile = GetPath(nameOfLanguage);
        Console.WriteLine("Dictionary path to files: " + pathToFile);
        if (string.IsNullOrEmpty(pathToFile) || !File.Exists(pathToFile))
        {
            (...)
            return false;
        }

        dictionary = new Dictionary<int, string>();
        currentlyUsedLanguage = nameOfLanguage;

        using (FileStream fileStream = new FileStream(pathToFile,
                                                    FileMode.Open, FileAccess.Read))
        using (StreamReader reader = new StreamReader(fileStream))
        {
            while (!reader.EndOfStream)
            {
                string line = reader.ReadLine();
                (...)
                {
                    string[] split = line.Split(new string[] { "==" });
                    if (split.Length == 2 && int.TryParse(split[0],
                                                            out int id))
                    {
                        dictionary[id] = split[1];
                    }
                }
            }
        }
        return true;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Missing Translation for: " + nameOfLanguage);
        (...)
        return false;
    }
}
```

Výpis 28 Metoda LoadDictionary ve třídě Languages

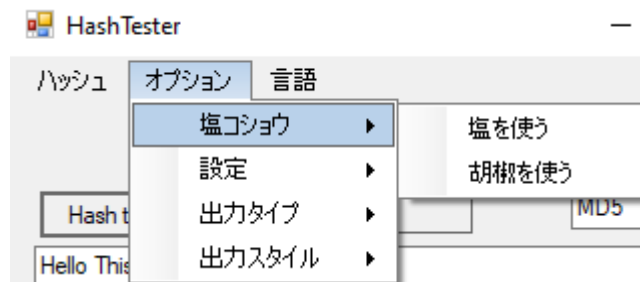
Přiřazení dat do slovníku je akorát čtení dat ze souboru, který je už předpřipraven. Jediná nevýhoda je, že všechny data jsou uloženy v paměti, ale když budeme brát v úvahu, že celý soubor pro češtinu zabírá akorát 14KB místa, tak si nemyslím, že to bude dělat problém i na starších zařízeních. Druhá možnost by byla zatěžovat disk a CPU čtením přímo z disku, a to bych řekl, že je ještě horší.

```
public static string Translate(int id)
{
    if (dictionary == null && !LoadDictionary("English")) return "error";
    if (dictionary != null && dictionary.ContainsKey(id))
    {
        return dictionary[id];
    }
    else
    {
        Console.WriteLine("Missing Translation: " + id);
        return "Translation Missing";
    }
}
```

Výpis 29 Metoda Translate ve třídě Languages

Metoda, která má přes 600 odkazů je metoda Translate, která vezme ID a pomocí slovníku vrátí přeložený text. Jestli pro nějaké ID neexistuje překlad, nastaví se *Translation Missing for ID*.

Díky takovému systému není žádný problém udělat překlad do více jazyků, stačí akorát přidat do složky *Languages* přidat soubor a je hotovo. Pro vyzkoušení jsem si pomocí DeepL za minutu udělat japonskou lokalizaci.



Obrázek 15 Příklad stip menu v japonském jazyce

//ツールストリップメニュー <<0-30>>

- 13==UI更新頻度
- 14==スレッドとCPUの設定
- 15==全設定リセット
- 16==システムテーマ
- 17==明るいテーマ
- 18==暗いテーマ
- 23==言語
- 24==ファイル.txt
- 25==原文を含める
- 26==ナンバリングを含める
- 27==ハッシュ・アルゴリズムを含める
- 28==ソルト&ペッパーを含める
- 29==すべてのオプションを含める

Výpis 30 Japonský překlad formuláře

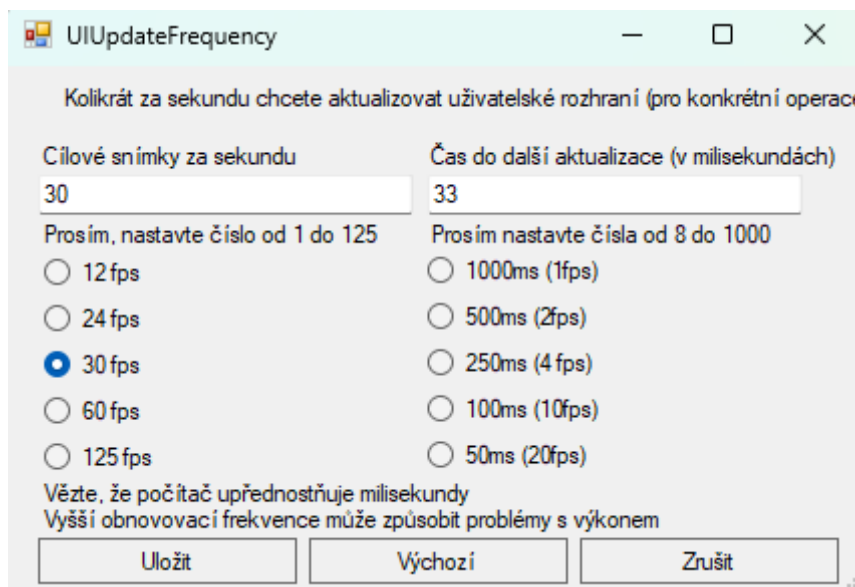
//ToolStrip Menu <<0-30>>

- 13==Frekvence aktualizace uživatelského rozhraní
- 14==Vlákna a nastavení CPU
- 15==Resetování všech nastavení
- 16==Motiv systému
- 17==Světlý motiv
- 18==Tmavý motiv
- 23==Jazyky
- 24==Soubor .txt
- 25==Zahrnout původní text
- 26==Zahrnout číslování
- 27==Zahrnout hashovací algoritmus
- 28==Zahrnout sůl a pepř
- 29==Zahrnout všechny možnosti

Výpis 31 Český překlad formuláře

3.7 Frekvence UI formulář

Ve volbách, nastavení ve strip menu je *Frekvence aktualizace UI*, která ukáže modulární formulář *UIUpdateFrequency* pro nastavení použité frekvence v programu.



Obrázek 16 Formulář pro nastavení frekvence UI

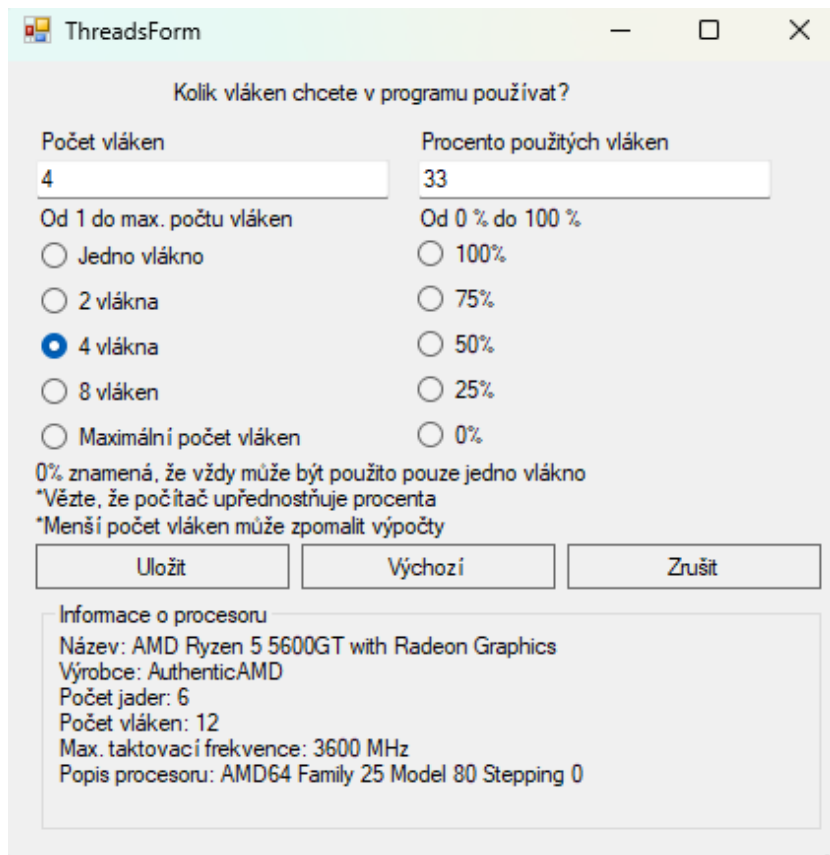
Formulář obsahuje 2 části, obě nastavují jednu věc. Nalevo je cílová snímková frekvence (FPS – snímky za sekundu) a vpravo čas do další aktualizace. Když se změní hodnota vlevo, nastaví se i hodnota vpravo, protože obě znamenají stejnou věc. FPS pracuje na jednotkách Hz, zatímco čas na periodě. Proto se dají tak jednoduše zaměnit. Pod textboxy se nachází několik radiobuttonů pro rychlé nastavení hodnoty. Ani jedna hodnota není stejná. Do *Settings* je ukládána hodnota v periodě, protože to je jednodušší k nastavení časovače. Doporučená hodnota je 24fps, standardní hodnota je 30fps pro dojem plynulosti, ale hodně záleží na CPU.

```
//Timer
timerRainbowTableGen.Interval = Settings.UpdateUIms;
timerRainbowTableGen.Tick += UpdateUIRainbowTable;
timerRainbowTableGen.Enabled = true;
```

Výpis 32 Příklad použití časovače a periody

3.8 Nastavení CPU a vláken formulář

Hned pod formulářem s frekvencí je formulář pro nastavení počtu vláken. Tento formulář funguje na úplně stejném principu.



Obrázek 17 Formulář pro nastavení počtu logických jader

Pokud je procento použitých vláken 0 %, nikdy se nebude používat systém více vláken. Doporučené procento je 50 % či 75 % pro většinu akcí a 100% pro hledání kolizí.

```
//MultiThread
if (checkBoxPerformanceModeRainbowTable.Checked
    && FormManagement.UseMultiThread())
{
    int numberOfThreadsToUse = FormManagement.NumberOfThreadsToUse();
    (...)
}
```

Výpis 33 Příklad použití proměnné v programu

3.9 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)
```

Výpis 34 Přetížení metody ProcessingHash

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);
}
```

Výpis 35 Část kódu pro hashování více hashovacích funkcí ve třídě Hasher

4 Pokročilé funkce programu

4.1 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.

4.1.1 Kód pro postupné hashování

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;
}
```

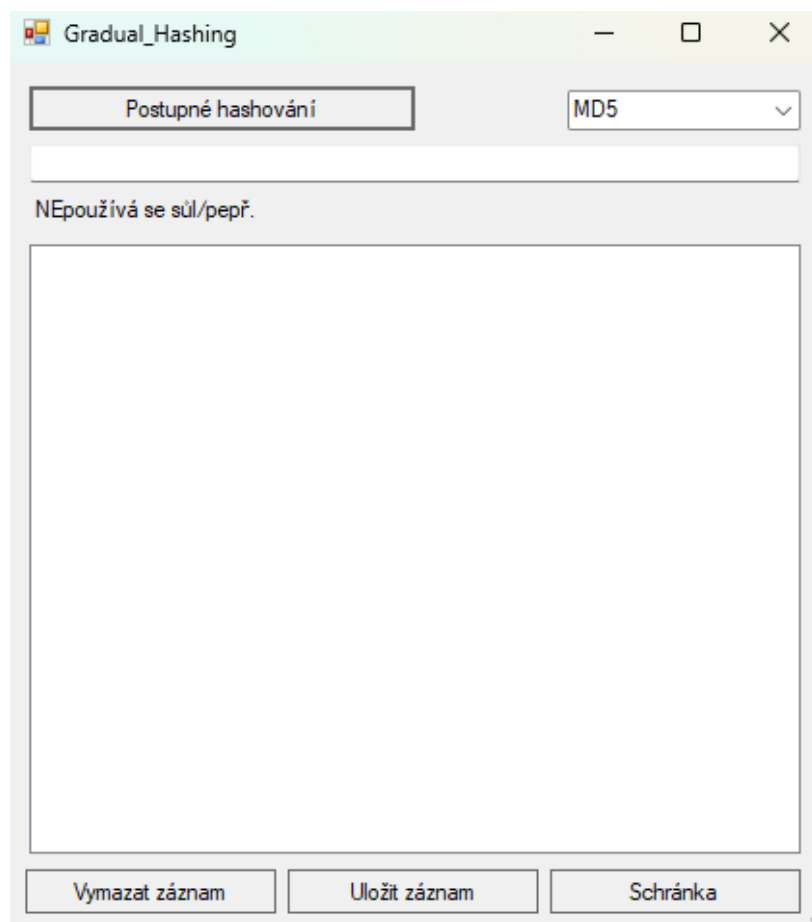
Výpis 36 Metoda GradualHashing ve třídě Hasher

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.

4.1.2 Formulář pro postupné hashování

Formulář je velice jednoduchý, většina obrazovky je komponenta listbox pro záznam.

Kvůli výukovým použitím se nepoužívá sůl/pepř i přes nastavení uživatele.



The screenshot shows a Windows application window titled "Gradual_Hashing". At the top, there is a button labeled "Postupné hashování" and a dropdown menu currently set to "MD5". Below these is a single-line text input field. A message "NEpoužívá se sůl/pepř." is displayed above a large, empty listbox that occupies the central portion of the window. At the bottom, there are three buttons: "Vymazat záznam", "Uložit záznam", and "Schránka".

Obrázek 18 Formulář postupného hashování

4.2 Formulář pro vícenásobné hashování

Formulář dělá to stejné jako je základ hlavního formuláře, ovšem si uživatel může pomocí checkboxů vybrat jaké hashovací funkce použít. Podporuje se více vstupů najednou, vstup z textového souboru a *Zobrazit algoritmus* možnost, která zahrne ve výstupu hashovací funkci, stejně jako možnost u stylu výstupu. *Zobrazit algoritmus* má přednost před normálním nastavením.

The screenshot shows a window titled "MultipleHashing". It has a text input area at the top containing "test" and "test2". Below the input are two buttons: "Hash textu" and "Hash .txt". To the left of the output area is a list of checkboxes for selecting hash algorithms: MD5 (checked), SHA1 (checked), SHA256 (unchecked), SHA512 (unchecked), RipeMD-160 (unchecked), CRC32 (checked), and "Zobrazit algoritmus*" (checked). Below these checkboxes is a note: "*Přepíše „Zahmout hashovací algoritmus“ v nastavení výstupního stylu". The output area on the right displays the results for each input and selected algorithm. At the bottom of the window are four buttons: "Přejít zpět", "Vymazat záznam", "Uložit záznam", and "Schránka".

Input	MD5	SHA1	CRC32
test	098f6bcd4621d373cade4e832627b4f6	a94a8fe5ccb19ba61c4c0873d391e987982fbbd3	d8777e0c
test2	ad0234829205b9033196ba8197a872b	109f4b3c50d7b0df729d299bcbf8e9ef9066971f	13bb8d58

Obrázek 19 Formulář pro vícenásobné hashování

4.3 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.

4.3.1 Kód pro práci se solí a pepřem

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
}
```

Výpis 37 Metoda IsUsingSaltAndPepper ve třídě Hasher

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 vrátí 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);
    }
}
```

Výpis 38 Metoda SaveSalt ve třídě Hasher

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 ukládána 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;
}

```

Výpis 39 Metoda CheckPepper ve třídě Hasher

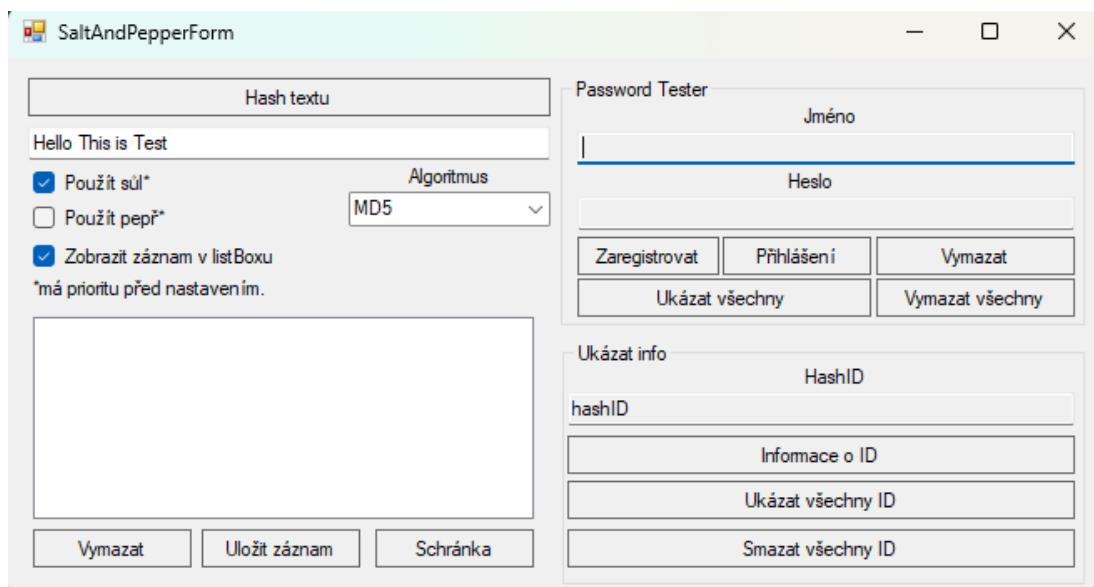
Hledání pepře je poměrně těžké, ale je to výhodné pro maximální bezpečnost. Musíme projít všechny možné kombinace a postupně porovnávat vygenerovaný hash s cílovým hashem. Pokud nalezneme shodu, znamená to, že jsme našli odpovídající vstupní text. Pokud žádná kombinace neodpovídá, originální text a hash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

Já používám pro pepř znaky UTF-8 neboli prvních 256 znaků Ascii, dokonce i prvních 32, které jsou netisknutelné, jelikož všude v programu je možnost vypisování 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).

4.3.2 Formulář pro práci se solí/pepřem

Formulář pracuje ze solí a pepřem a pomocí textového souboru simuluje přihlášení třeba na webovou stránku.

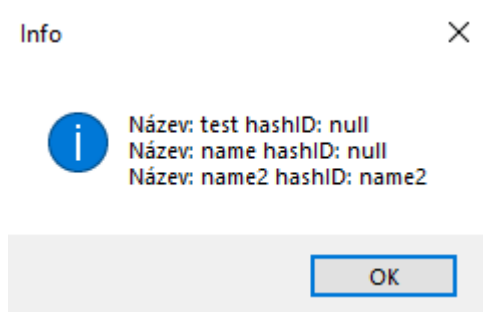


Obrázek 20 Formulář pro práci se solí a pepřem

Na levé straně je část pro normální hashování, jakožto je u hlavního formuláře. Na pravé straně nahoře je přihlášení a dole informace o hashech. Formulář používá složku *HashData* a podsložku *PasswordTester* pro funkci. V *PasswordTester* se nachází textový soubor *nameTable.txt*, který ukládá přihlašovací data. *HashID* slouží pro identifikaci a získání informací o soli a pepři.

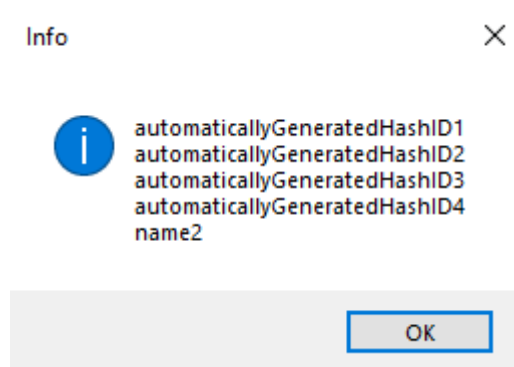
Obrázek 21 Formulář pro zadání soli a pepře

Při jakémkoliv použití soli či pepře se zobrazí další formulář, kde si může uživatel vybrat mezi náhodnou generací či vlastní. *HashID* se v případě registrace použije samo, jelikož je stejné se jménem, jinak se samo vygeneruje. Jestli není zadán HashID, data se neuloží.



Obrázek 22 Výstup tlačítka Ukázat všechny uživatele

Tlačítko *Ukázat všechny* ukáže všechny zaregistrované uživatele a jejich *HashID*. *Null* znamená, že nepoužívají sůl/pepř.



Obrázek 23 Výstup tlačítka Ukázat všechny ID

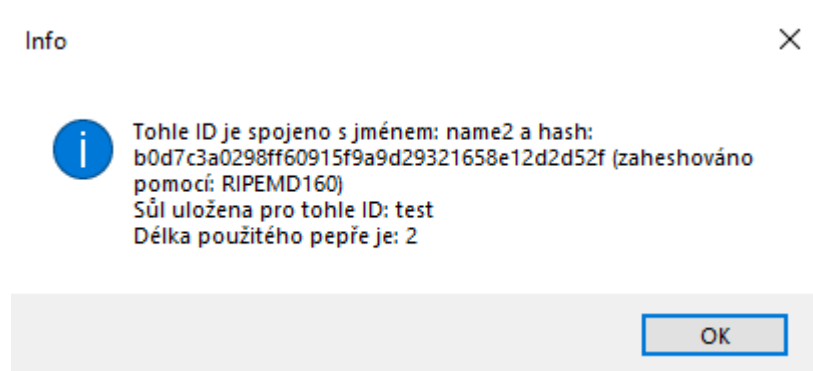
Tlačítko *Ukázat všechny ID* ukáže všechny ID ve složce HashData.

	PasswordTester	18.03.2025 18:18	Složka souborů	
	automaticallyGeneratedHashID1.txt	19.03.2025 21:14	Textový dokument	1 kB
	automaticallyGeneratedHashID2.txt	19.03.2025 21:14	Textový dokument	1 kB
	automaticallyGeneratedHashID3.txt	19.03.2025 21:14	Textový dokument	1 kB
	automaticallyGeneratedHashID4.txt	19.03.2025 21:14	Textový dokument	1 kB
	name2.txt	19.03.2025 21:11	Textový dokument	1 kB

Obrázek 24 Složka HashData

```
salt==test
pepperLength==2
```

Výpis 40 Data v souboru name2.txt



Obrázek 25 Výstup tlačítka Informace o ID


```

private List<string> ShowIDInfo()
{
    string pathNameTable =
        Path.Combine(Settings.DirectoryToPasswordTester, "nameTable.txt");
    string pathHashID = Path.Combine(Settings.DirectoryToHashData,
                                      textBoxHashID.Text + ".txt");

    bool foundIDinNameTable = false;
    List<string> info = new List<string>();
    //checkNameTable
    if (File.Exists(pathNameTable))
    {
        using (StreamReader reader = new StreamReader(pathNameTable))
        {
            while (!reader.EndOfStream && !foundIDinNameTable)
            {
                (...) //Najde v souboru
            }
        }
        if (!foundIDinNameTable) info.Add(Languages.Translate(635));
    }
    else
    {
        saltAndPepper.GenerateNameTableFile();
        info.Add(Languages.Translate(636));
    }
    //check hashID
    if (File.Exists(pathHashID))
    {
        hasher.LoadSalt(textBoxHashID.Text, out string salt,
                                                                out int pepperLenght);
        if (!String.IsNullOrEmpty(salt))
            info.Add(Languages.Translate(637) + ": " + salt);
        if (pepperLenght > 0) info.Add(
            Languages.Translate(638) + ": " + pepperLenght);
    }
    else
    {
        info.Add(Languages.Translate(639));
    }
    return info;
}

```

Výpis 41 Metoda ShowIDInfo

Metoda *ShowAllInfo* prohledá všechny soubory ve složce *HashData* po *hashID* a poté prohledá *nameTable.txt* v *HashData/PasswordTester* po *hashID*.

4.4 Kontrolní součet

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

4.4.1 Kód pro kontrolní součet

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)
    {
        (...)
    }
}
```

Výpis 42 Metoda FileChecksum ve třídě Hasher

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 proměnné 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.

4.4.2 Formulář pro kontrolu souborů

Formulář slouží hlavně pro kontrolu integrity souborů, ale může se použít jako otisk pro důležité dokumenty. Při stahování z internetu se někdy dává i hash jakožto kontrola integrity, díky formuláři si může uživatel udělat kontrolní součet souboru a zjistit, zda se soubor stáhnul správně.

Obrázek 26 Formulář pro generování kontrolního součtu

Uživatel si může buď vybrat soubor, nebo zadat hash do textboxu a poté soubor. Jestliže kontrola probíhá přes tlačítko *Vyberte soubor*, uživatel musí pomocí komponenty checkbox zaškrtnout jaké algoritmy by chtěl. Pokud ovšem probíhá kontrola přes kontrolu součtu, musí být splněna délka jednoho z hashů (8 pro CRC32, 64 pro SHA256 atd.).

Pokud je délka 40, program se pomocí checkboxu zeptá, jestli se jedná o RipeMD-160 nebo SHA1 (oba používají délku 40–160 bitů). Vpravo si uživatel může zkopírovat do schránky hashe. To funguje stejně jako kopírování u záznamu.

```
switch (checksum.Length)
{
    case 32: fileAlgorithm = Hasher.HashingAlgorithm.MD5; break;
    case 40:
    {
        //SHA1 and RipeMD160 are the same lenght
        if (MessageBox(...)) == DialogResult.Yes)
            fileAlgorithm = Hasher.HashingAlgorithm.SHA1;
        else fileAlgorithm = Hasher.HashingAlgorithm.RIPEMD160;
        break;
    }
    case 64: fileAlgorithm = Hasher.HashingAlgorithm.SHA256; break;
    case 128: fileAlgorithm = Hasher.HashingAlgorithm.SHA512; break;
    case 8: fileAlgorithm = Hasher.HashingAlgorithm.CRC32; break;
}
```

Výpis 43 Velice jednoduchý rozhodovací proces hashovací funkce podle délky hashe

4.5 Hledání kolizí

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

4.5.1 Kód

```
private bool GenerateCollision((...))
{
    (...)
    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);
        }
        (...)
    }
}
```

Výpis 44 Metoda `GenerateCollision` ve formuláři `HashingCollisionForm`

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));
```

Výpis 45 Počítání náhodného vstupu pomocí třídy `Random`

ThreadNumber se používá při generaci náhodného vstupu 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);
}
```

Výpis 46 Metoda GenerateRandomString pro generování náhodného vstupu

Metoda *GenerateRandomString* nám zajišťuje náhodný text 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.

```
Algorithm=CRC32
<STRING>
ĚÀÔ÷, +%ÜPs
+ _ □ · ^ h L Í é
<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
```

Výpis 47 Příklad vygenerovaného textového souboru

Další část kolizí je metoda *CheckCollision*, která vezme uživateli vstupy a zkontroluje, jestli se po zahashování rovnají či nikoliv. Vstup může být i přes soubor. Program obsahuje několik předpřipravených souborů. Ve složce *Collisions* je také textový soubor *_collisionInfo.txt*, který popisuje vytváření vlastních souborů.

```

using (OpenFileDialog soubor = new OpenFileDialog())
{
    (...)
    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);
            (...)
        }
    }
}

```

Výpis 48 Část metody pro zpracování textového souboru

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.

4.5.2 Formulář pro hledání kolizí

Formulář vyhledává kolizi dvou různých vstupů se stejným výstupem.

Obrázek 27 Formulář pro generování kolizí

Formulář má 3 hlavní tlačítka, *Vygenerovat kolizi*, *Kontrola kolize* a *Kontrola kolize z .txt*, což jsou stejné funkce, akorát se mění formát vstupních dat. Vygenerovat kolizi začne náhodně generovat vstupní data, hashovat je a zapisovat do listu, dokud se nenajde shoda. Výkonnostní režim je použití více vláken najednou, kde každé vlákno má vlastní list. Je to mnohem jednodušší udělat než udělat jeden společný velký list kvůli problémům mezi vlákny. Maximum pokusů ukončí program, pokud se přesáhne počet pokusů. *Hex pro zobrazení* při nalezení kolize zobrazí vstup v hexadecimálním zápisu společně s UTF-8.

Co se týče generování jsou podporované pouze CRC32, RipeMD-160, MD5 a SHA1. Důvod je, že pro tyto hashovací funkce je aspoň nějaká šance na nalezení kolize. CRC32 je doporučená, protože většinou do minuty se najde nějaká kolize. MD5 má potvrzeno několik kolizí, ale vyhledávání je minimálně proces pro spousty moderních GPU. SHA1 byla do 2017 bezkolizní, ale vyhledávání přes krátký text je takřka nemožné. Stejně jako u postupného hashování doporučuji použití pouze pro ukázání, jak kolize fungují u hashů.

Obrázek 28 Formulář pro kontrolu kolize

Kontrola kolize spustí další formulář, kam uživatel zadá vstup ve formátu UTF-8, hexadecimální číslo nebo binární číslo. Ve složce *Collisions* je pár souborů s kolizemi, které se dají použít.

Algorithm=CRC32

<STRING>

i!%ÓÁ!çø¼

#±JÁöâ«ú

<HEX>

8D-B7-EC-BC-D3-C1-21-A2-6F-BE

23-B1-4A-81-C1-F5-E2-90-AB-FA

Výpis 49 Textový soubor CRC32-Prefab.txt ve složce Collisions

5 Útoky na hashe

Existuje spousta útoků na hashe, já jsem naprogramoval 3 z nich. Všechny útoky jsou pod jedním velkým formulářem, který je členěn na 5 částí.

The screenshot shows the PasswordForm application window with the following sections:

- Slovníkový útok (Dictionary attack):** Includes a text input field with 'budakkecik', a 'Dictionary attack' button, and checkboxes for 'Show log in listBox', 'Plná verze', 'Zkrácená verze', 'Velice zkrácená verze', and 'Vlastní soubor'.
- Čas k prolomení kalkulačtor (Brute Force Attack):** Includes a 'Počet znaků/Heslo' field, a 'Rychlost /s' field with '2000000000', checkboxes for 'Malá písmena (26)', 'Velká písmena (26)', 'Číslice (10)', 'Speciální (33)', and 'Show log in listBox', and a 'Vypočítat' button.
- Útok duhovou tabulkou (Rainbow Table Attack):** Includes a 'budakkecik' text input, radio buttons for 'Text' and 'Hash', an 'Algorithm' dropdown set to 'MD5', checkboxes for 'PerformanceMode', 'Show log in listBox', and buttons for 'Vygenerovat duhovou tabulku' and 'Útok duhovou tabulkou'.
- Útok hrubou silou (Hash Attack):** Includes a 'budakkecik' text input, radio buttons for 'Text' and 'Hash', an 'Algorithm' dropdown set to 'MD5', fields for 'Maximální počet pokusů' (0), 'Délka' (10), and 'Časovač zastavení (sec)' (0), checkboxes for 'Neznámá délka', 'Zobrazení hesla jako HEX', 'PerformanceMode', 'Show log in listBox', and checkboxes for 'Malá písmena (26)', 'Velká písmena (26)', 'Číslice (10)', and 'Speciální znaky (33)', and a 'Brute Force Attack' button.

At the bottom, there is a 'UI' section with statistics: 'Čas: 1.715', 'Zpracovává se řádek: 1 000 000', 'Aktuální rychlost /s: 628 303', and 'Průměrná rychlost /s: 583 090'. A progress bar labeled 'Ukazatel průběhu' is shown, along with a 'Přerušit proces' button. A log area displays: 'Nalezeno 'budakkecik' na řádku: 1000000' and 'Heslo 'budakkecik' bylo nalezeno ve slovníku na řádku 1000000. Doporučuji použít jiné heslo.' At the very bottom are buttons for 'Clear Log', 'Save Log', and 'Clipboard'.

Obrázek 29 Formulář Prolamovač Hesel

První 4 části se nacházejí ve vrchní části formuláře a jsou rozděleny pomocí komponenty groupbox. Spodní část je celá jedna část a zahrnuje listBox se záznamem a nastavení, tlačítko *Přerušit proces* pro ukončení jakékoliv činnosti, komponentu progress bar a groupbox jménem UI, které ukazuje čas, počet pokusů, aktuální rychlost mezi updaty UI a průměrnou rychlost.

5.1 Slovníkový útok

5.1.1 Kód pro 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 třídy.

```
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;
    }
}
```

Výpis 50 Vlastnost Progress ve třídě DictionaryAttack

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;
                        }
                    }
                }
            }
        }
    }
}

```

Výpis 51 Metoda PasswordFinder ve třídě DictionaryAttack

PasswordFinder je hlavní metoda celého slovníkového útoku. Pro slovníkový útok potřebujeme textový soubor s hesly *fullPathToTXT* a hesla, které chceme najít. Metoda dovoluje hledání více hesel najednou. Spousta prvků je pro UI, jako třeba *CurrentLine* nebo použití *Stopwatch* pro měření času. Základ je ovšem přechíst 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, našli jsme všechny hesla a algoritmus se může předčasně ukončit, jinak běží do konce souboru. Celá metoda je zanořena v try a catch.

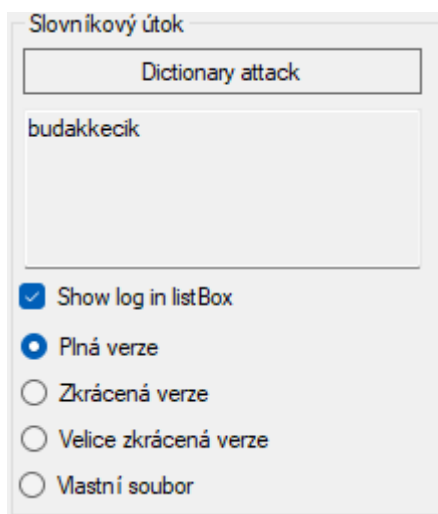
```

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

```

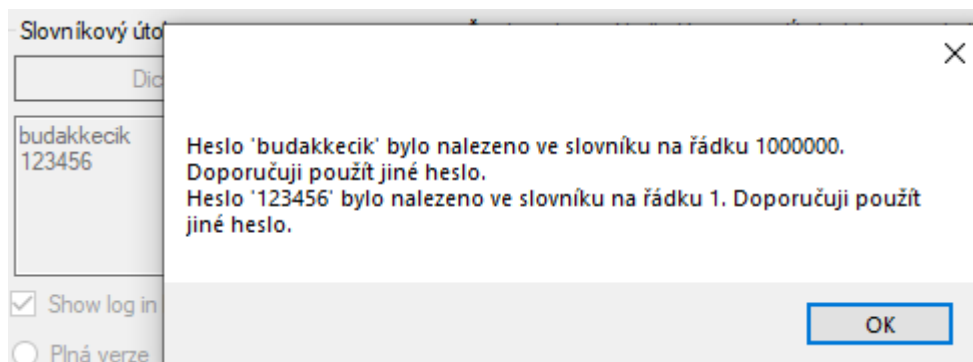
Výpis 52 Zkouška, zda v poli byly najité všechny vstupy

5.1.2 Formulář pro slovníkový útok



Obrázek 30 Groupbox Slovníkový útok

První zleva je groupbox jménem *Slovníkový útok*, který obsahuje hlavní tlačítko, textové pole s více řádky, checkbox pro ukládání do záznamu a 4 radiobuttony. Radiobutton říká, který slovník použijeme pro slovníkový útok. Slovníkový útok nepoužívá hashe, na rozdíl od útoku duhovou tabulkou. Jediné, co dělá je, že projíždí soubor a hledá stejný text. Každý radiobutton je enabled podle existence specifického souboru ve složce *WordLists*. Plná verze je spojena se souborem *rockyou.txt*, který obsahuje přes 14 mil. hesel. Zkrácená verze je spojena s *rockyouShort.txt*, která obsahuje 1 mil. hesel a stejně pro Velice zkrácenou verzi. Radiobutton *Vlastní soubor* je vždy vybratelný a ukáže *OpenFileDialog* pro uživatele, aby si vybral vlastní soubor, který bude použit jako slovník.



Obrázek 31 Ukázka slovníkového útoku s více vstupy

```
Nalezeno '123456' na řádku: 1
Nalezeno 'budakkecik' na řádku: 1000000
Heslo 'budakkecik' bylo nalezeno ve slovníku na řádku 1000000. Doporučuji použít jiné heslo.
Heslo '123456' bylo nalezeno ve slovníku na řádku 1. Doporučuji použít jiné heslo.
```

Obrázek 32 Ukázka výstupu slovníkového útoku s více vstupy v záznamu

Slovníkový útok podporuje více vstupů zároveň.

5.2 Výpočet prolomení hesla

5.2.1 Kód

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;
}
```

Výpis 53 Metoda Calculator ve třídě PasswordStrenghtCalculator

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. [31]

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;
    }
    (...)
}
```

Výpis 54 Metoda TryPower pro mocnění BigIntegeru

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.

$$\text{Počet kombinací} = \text{Počet znaků}^{\text{délka hesla}}$$

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).

$$\text{Doba trvání (s)} = \frac{\text{Počet kombinací}}{\text{Rychlost za sekundu}}$$

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 numberSecondsLeft = numberSeconds % 60;

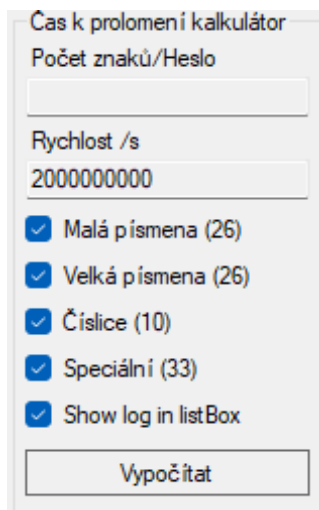
    string s = "";
    if (numberMilionYears > 0) s += numberMilionYears.ToString("N0") +
        " " + Languages.Translate(575) + " " +
        Languages.Translate(581) + ", ";
    (...)
    return s;
}

```

Výpis 55 Metoda Output ve třídě PasswordStrenghtCalculator

Metoda Output akorát vypíše dobu trvání v představitelné časové době.

5.2.2 Formulář Čas k prolomení kalkulátor



Obrázek 33 Groupbox Čas k prolomení kalkulátor

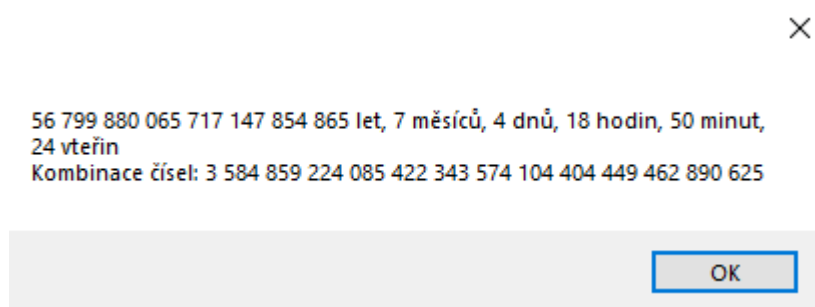
Kalkulátor či kalkulačka simuluje čas potřeba pro prolomení hesla při útoku hrubou silou. Kalkulačka funguje velice jednoduše na už zmíněném vzorci. První textbox podporuje vstup celočíselného nezáporného čísla reprezentující délku hesla, nebo přímo samotné heslo.


```
if (!int.TryParse(textBoxCrackLenght.Text, out passwordLenght))
    passwordLenght = textBoxCrackLenght.Text.Length;
```

Výpis 56 Rozhodující proces použití délky či samotné heslo

Rozhodující proces funguje na metodě *TryParse*, jestli se povede z textboxu udělat číslo, použije se číslo, jestli číslo přeteče či obsahuje něco jiného než číslovky, použije se délka textu v textboxu. V *Rychlost* /s textboxu musí být číslo, jinak program nepustí uživatele dále. Limit délky čísla není díky použití dynamické proměnné *BigInteger*.

Uživatel si může přidat/odebrat počet použitých znaků pomocí checkboxů, kde je také v závorkách napsáno kolik znaků přiřadí.



Obrázek 34 Příklad výstupu při délce hesla 20 a všech 95 znaků

5.3 Útok duhovou tabulkou

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.

5.3.1 Generování duhové tabulky

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

```
public bool GenerateRainbowTable(string filePath,
                                string outputPath, Hasher.HashingAlgorithm hashingAlgorithm)
```

Výpis 57 Hlavička metody GenerateRainbowTable ve třídě RainbowTableGenerator

Potřebujeme cestu k slovníku, výstupní cestu a hashovací funkci.

```
using (StreamReader reader = new StreamReader(filePath))
{
    (...)
    using (StreamWriter writer = new StreamWriter(outputPath))
    {
        writer.WriteLine("algorithm==" + hashingAlgorithm.ToString());
        while (!reader.EndOfStream)
        {
            if (cancellationTokenSource.Token.IsCancellationRequested)
            {
                RemoveFilesQuestion(outputPath);
                return false; // Stop if canceled
            }
            LinesProcessed++;
            string line = reader.ReadLine();
            string hash = hasher.Hash(line, hashingAlgorithm);
            writer.WriteLine(line + "=" + hash);
        }
    }
}
```

Výpis 58 Tělo metody GenerateRainbowTable ve třídě RainbowTableGenerator

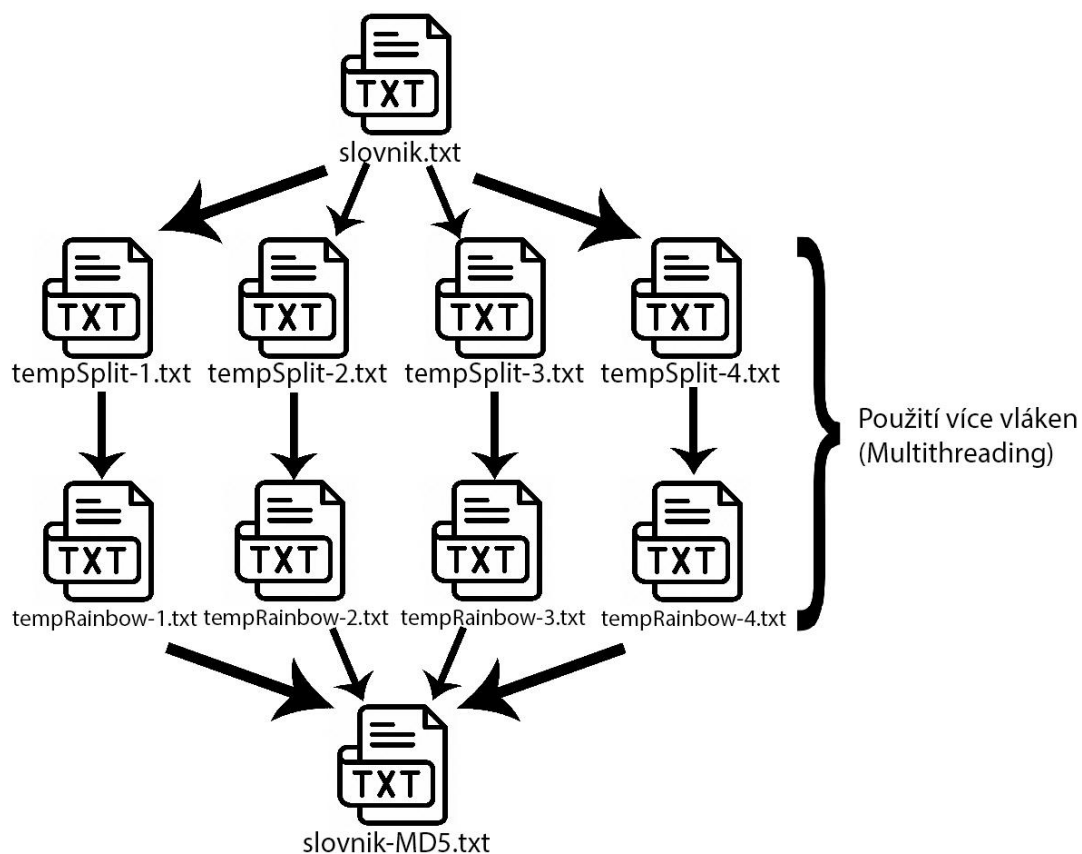
Pomocí tříd *StreamWriter* a *StreamReader* zpracováváme po řádcích slovník a zapisujeme. Jako první řádek duhové tabulky je vždy jakou hashovací funkci používáme. Funkce je rozdělena pomocí dvou rovná se (==) a data pomocí jednoho (=).

```
algorithm==CRC32
123456=0972d361
12345=cbf53a1c
123456789=cbf43926
password=35c246d5
iloveyou=fc724cbd
princess=11ed43d0
```

Výpis 59 Příklad začátku duhové tabulky v textovém souboru

5.3.2 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.



Obrázek 35 Fungování metody *GenerateRainbowTableMultiThread* pro 4 vlákna

5.3.2.1 Metody pro multithreading

```
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;
    }
}
```

Výpis 60 Metoda GenerateRainbowTableMultiThreadForSingleThread

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

```
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
```

Výpis 61 Hlavička metody GenerateRainbowTableMultiThread

Metoda spouští pro každé vlákno předchozí metodu. 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.

5.3.2.2 Mazání dočasných souborů

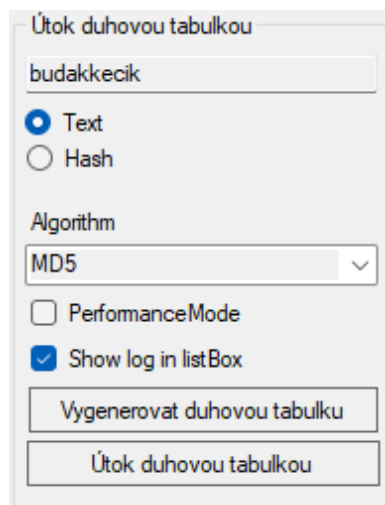
Dočasné soubory jsou po dokončení generace vymazány, pokud nedošlo při generování k výjimce či ukončení procesu uživatelem.

```
private void RemoveFilesQuestion(string filePath)
{
    if (MessageBox.Show(Languages.Translate(11003),
        Languages.Translate(10030), MessageBoxButtons.YesNo,
        MessageBoxIcon.Warning) == DialogResult.OK)
    {
        try
        {
            if (File.Exists(filePath))
            {
                File.Delete(filePath);
            }
        }
        (...)
    }
    else //rename files to not cause problems
    {
        try
        {
            string rename =
                Directory.GetDirectories(filePath).FirstOrDefault();
            string time = DateTime.UtcNow.ToString("yyyy,MM,dd-HH,mm,ss");
            rename = Path.Combine(rename, "failedRainbowTable-" +
                time + ".txt");
            File.Move(filePath, rename);
        }
        (...)
    }
}
```

Výpis 62 Metoda RemoveFilesQuestion

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.

5.3.3 Formulář pro duhovou tabulku



Obrázek 36 Groupbox *Útok duhovou tabulkou*

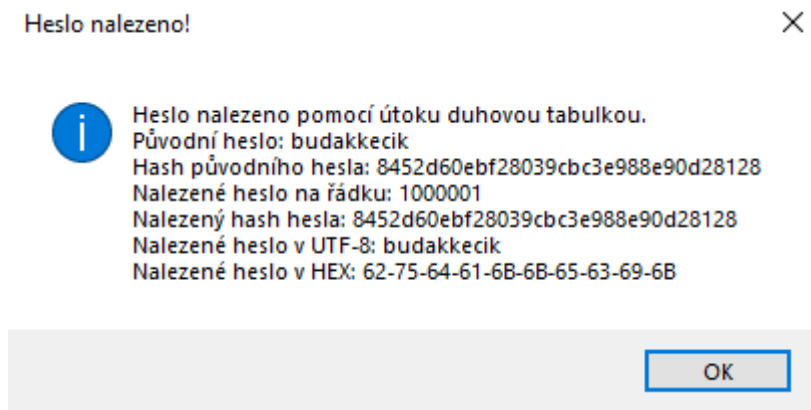
Do textboxu se dává hledaný vstup ve formě textu nebo hashe, záleží na nastavení radiobuttonu. Pokud je nastaven radiobutton *Text*, vstup v textboxu bude zaheshován podle vybraného algoritmu. Checkbox *Performance Mode* nastavuje použití multithreadingu. Checkbox *Show log in listBox* nastavuje ukládání do záznamu.

5.3.3.1 Generování duhové tabulky

Tlačítko *Vygenerovat duhovou tabulku* otevře *OpenFileDialog* pro vybrání slovníku a *SaveFileDialog* pro uložení duhové tabulky. Oba dva dialogy musí být úspěšné. Poté se zapíná metoda *GenerateRainbowTable* či *GenerateRainbowTableMultiThread* ze třídy *RainbowTableGenerator*.

5.3.3.2 Útok duhovou tabulkou

Tlačítko *Útok duhovou tabulkou* otevře komponentu *OpenFileDialog* pro vybrání duhové tabulky a poté spustí metodu *PerformRainbowAttack* ze třídy *RainbowTableAttack*.



Obrázek 37 Výstup nalezené shody při útoku duhovou tabulkou

```
Heslo nalezeno pomocí útoku duhovou tabulkou.  
Původní heslo: budakkecik  
Nalezený hash hesla: 8452d60ebf28039cbc3e988e90d28128  
Nalezené heslo na řádku: 1000001  
Nalezené heslo v UTF-8: budakkecik  
Nalezené heslo v HEX: 62-75-64-61-6B-6B-65-63-69-6B
```

Obrázek 38 Výstup nalezené shody při útoku duhovou tabulkou v záznamu

5.4 Útok hrubou silou

5.4.1 Kód

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 zahashované heslo, jaké všechny znaky použít (malé a velká písmena, číslice a speciální znaky), délka hledaného hesla pro zrychlení hledání, popřípadě začít od úplného začátku.

```

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;
        }
    }
}

```

Výpis 63 Metoda SelectAllUsableChars ve třídě BruteForceAttack

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


```

public BigInteger CalculateAllPossibleCombinations(
    bool variablePasswordLength, int userPasswordLength)
{
    if (usableChars == null) return 0;
    if (variablePasswordLength) // Variable password length
    {
        long temp = 0;
        for (int length = 1; length <= maximumLengthForBruteForce;
            length++)
        {
            temp += Pow(usableChars.Length, length);
        }
        return temp;
    }
    else // Known password length
    {
        return Pow(usableChars.Length, userPasswordLength);
    }
}

```

Výpis 64 Metoda CalculateAllPossibleCombinations ve třídě BruteForceAttack

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 95^{50}).

```

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
}

```

Výpis 65 Metoda PasswordBruteForce ve třídě BruteForceAttack

Spravování 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);
}

```

Výpis 66 Logika posunu hledané délky ve třídě BruteForceAttack

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());
}
```

Výpis 67 Metoda GenerateText ve třídě BruteForceAttack

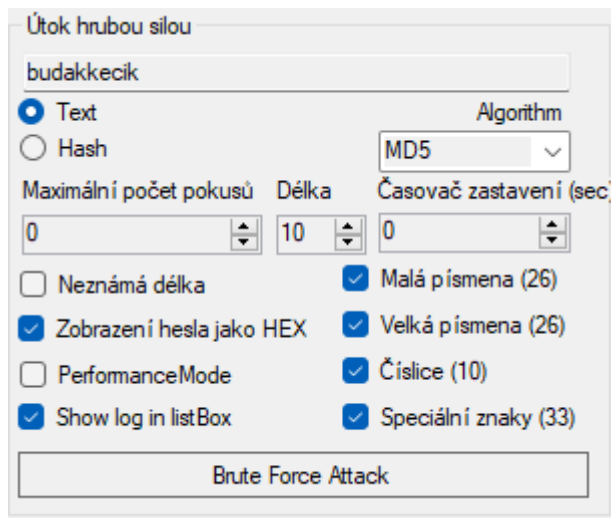
Metoda *GenerateText* si v hlavičce vezme index, všechny hledané znaky a délku a pomocí těchto dat vygeneruje text, který se poté zaheshuje a vyzkouší. Příklad pro index 1, všechny možné znaky a délku 5 se vrátí *aaaaa*. Pro index 2 je *aaaab*, pro index 96 bude *aaaba* atd.

```
string tryText = GenerateText(index, usableChars, currentLength);
string hashedText = hasher.Hash(tryText, algorithm);
if (hashedText == userHashInput)
{
    FoundPasswordBool = true;
    FoundPassword = tryText; // Found the password
    return true;
}
```

Výpis 68 Příklad použití metody GenerateText

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.

5.4.2 Formulář pro útok hrubou silou



Obrázek 39 Groupbox Útok hrubou silou

Formulář disponuje stejným systémem vstupu jako groupbox *Útok duhovou tabulkou* a také výběrem znaků jako groupbox *Čas k prolomení Kalkulátor*. Dále máme limit počtu pokusů a limit podle časovače. Jestli je vstup 0 na limitu, nebude se limit používat. Délka je nastavení hledané délky a automaticky se zadává podle délky v textu, jestli je ovšem radiobutton *Text* aktivní.

```
if (radioButtonRegularBruteForce.Checked)
{
    numericUpDownLenght.Value = textBoxBruteForce.Text.Length;
    checkBoxUnknownLenghtBruteForce.Checked = false;
}
else if (radioButtonBruteForceHashed.Checked)
{
    checkBoxUnknownLenghtBruteForce.Checked = true;
}
```

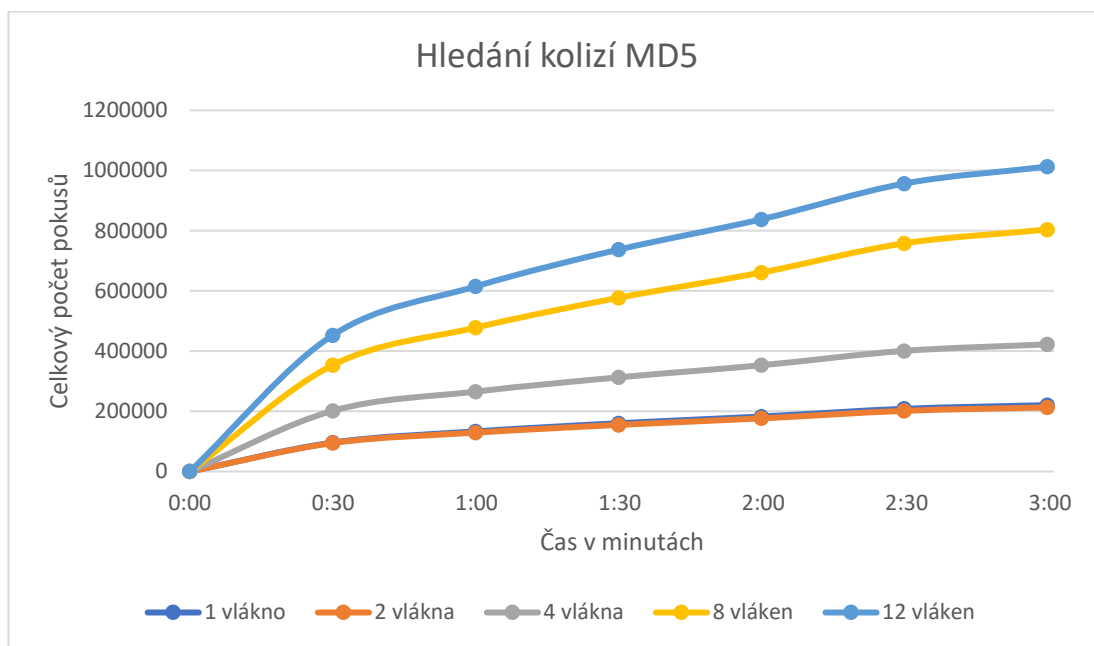
Výpis 69 Logika vstupu textboxu v groupboxu Útok hrubou silou

Pokud je délka nastavena na hodnotu 0 nebo checkbox *Neznámá délka* je zaškrtnuto, bude se začínat od délky 1 a postupně se bude zvětšovat. Checkboxy *PerformanceMode* a *Show log in listBox* fungují stejně jako u předchozích groupboxů. Checkbox *Zobrazení hesla jako HEX* přidá do výstupu výstupní hodnotu v Hex. Tlačítko *Brute Force Attack* započne útok hrubou silou pomocí třídy *BruteForceAttack*.

6 Statistika

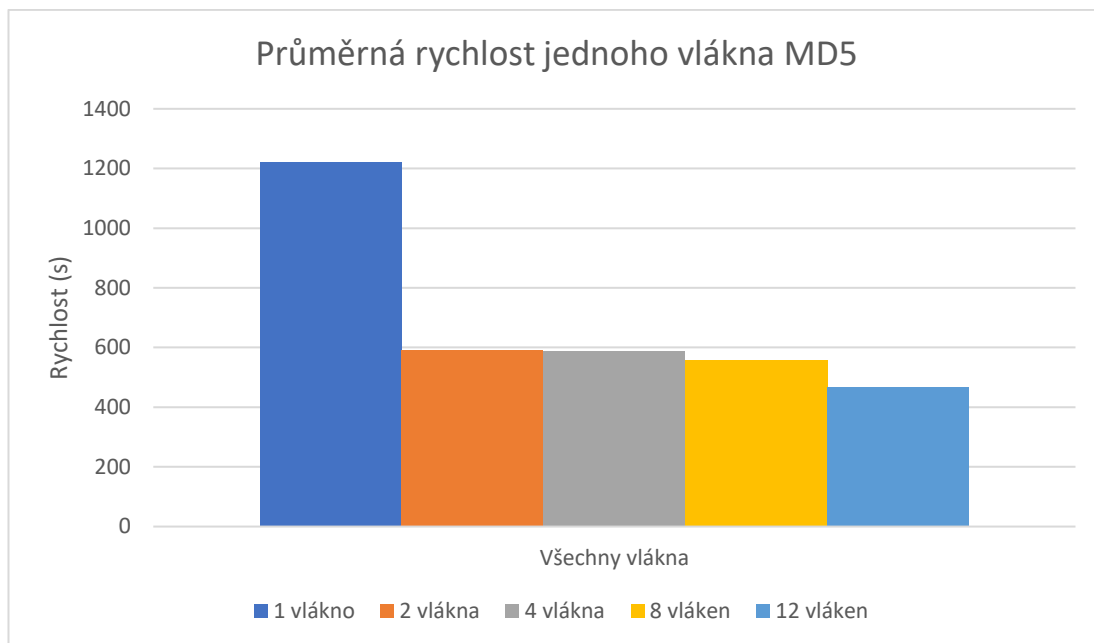
V této kapitole jsem graficky znázornil efektivitu použití několika vláken oproti použití pouze jednoho. Každý graf je okomentován. Parametry systému pro všechny testy je Operační systém Windows 10, procesor AMD Ryzen 5 5500, který disponuje 6 jádry (12 logických procesorů) a 16 GB paměti RAM. Všechno ukládání na disk probíhá do Samsung SSD 970 Evo Plus (SSD Mk.2).

6.1 Vyhledávání kolizí



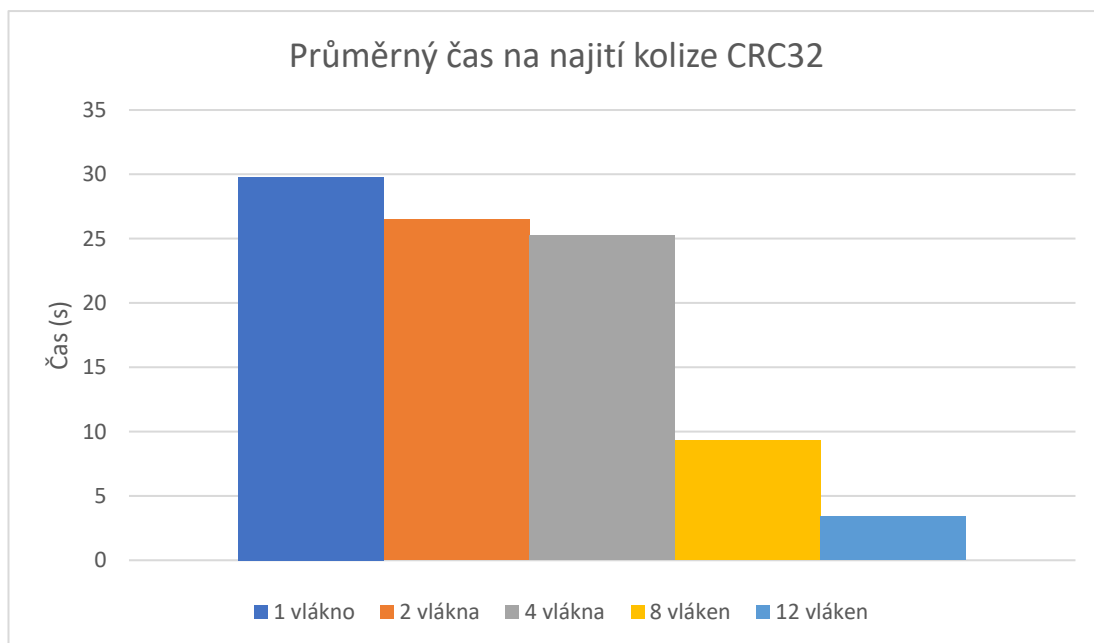
Graf 4 Rychlost hledání kolizí MD5 za určitou dobu

Graf znázorňuje 2 problémy programu. První je problém s použitím více vláken, kde rozdíl mezi jedním vláknem a dvěma vlákny je prakticky k nerozeznání. Tento jev jsem čekal, ale nečekal jsem až takto blízko u sebe. Metody jako *Increment* mezi vlákny potřebují přečíst a poté zapsat. Aby nedošlo ke ztrátě dat, tak musí jedno vlákno počkat, než druhé dodělá tuto operaci a pak až může pokračovat používat proměnnou. Druhý problém je s hledáním kolizí, kde čím déle hledáme a zvětšujeme list, tím více musíme procházet a porovnávat hodnotu k listu a tím pomalejší je hledání.



Graf 5 Průměrná rychlost při hledání kolize pro jedno vlákno

Tento graf nám ukazuje využití jednoho vlákna oproti celku neboli kolik dokáže jedno vlákno zpracovat za sekundu v průměru. Jakmile použijeme více vláken, rychlost pro jedno jádro dramaticky klesne o cca polovinu v tomto případě, ale zůstává poměrně konzistentní při více vláknech.

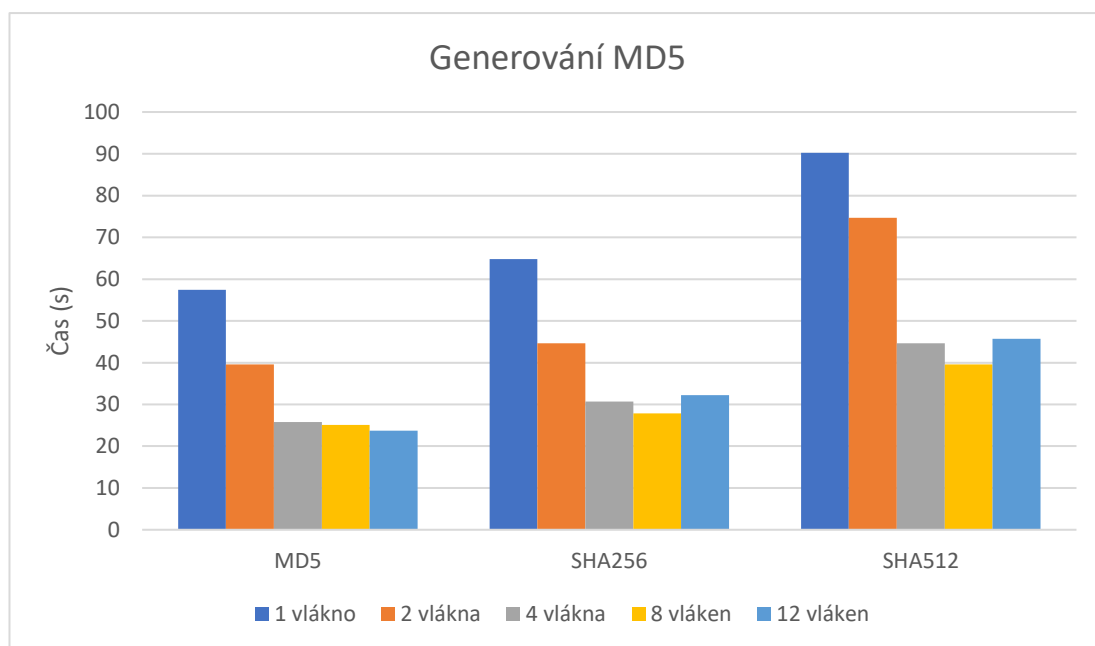


Graf 6 Průměrný čas na najetí kolize v CRC32

Průměrný čas je spočítán průměrem 10 pokusů. UI bylo nastaveno na 30fps. Můžeme vidět dramatické zrychlení v hledání mezi čtyřmi a osmi vlákny. Nemám tušení, co toto zrychlení způsobuje, ale nakonec je hledání kolizí jenom o štěstí a průměr z 10 pokusů je docela málo, bohužel nemůžu z časových důvodů zkoušet vícekrát.

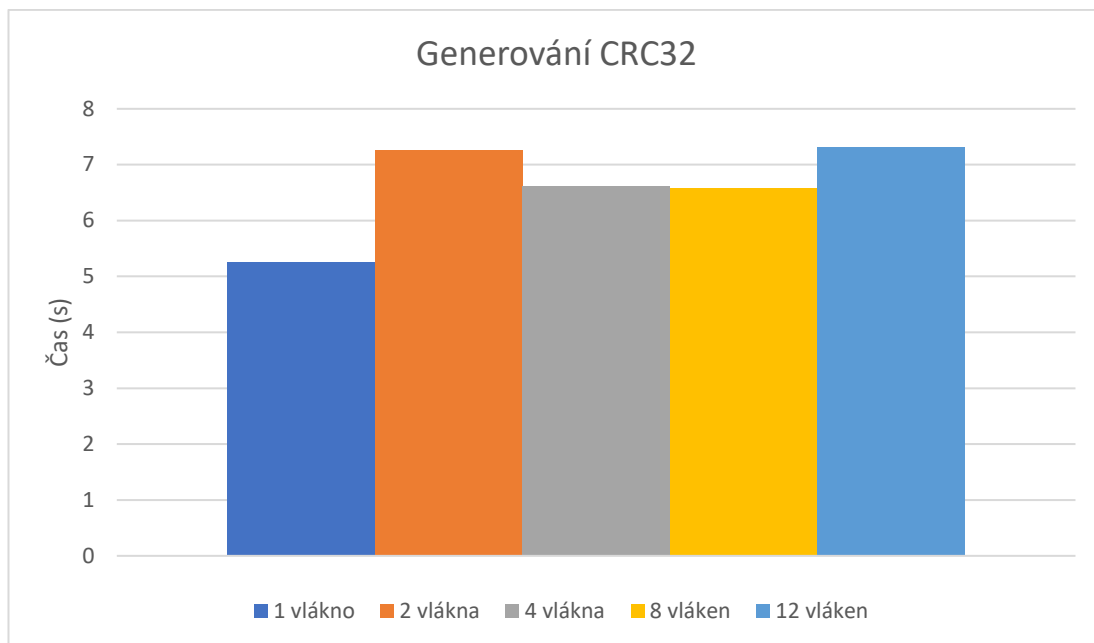
6.2 Generování duhové tabulky

Pro všechny je použit soubor rockyou.txt s 14 344 391 hesly.



Graf 7 Generování duhové tabulky

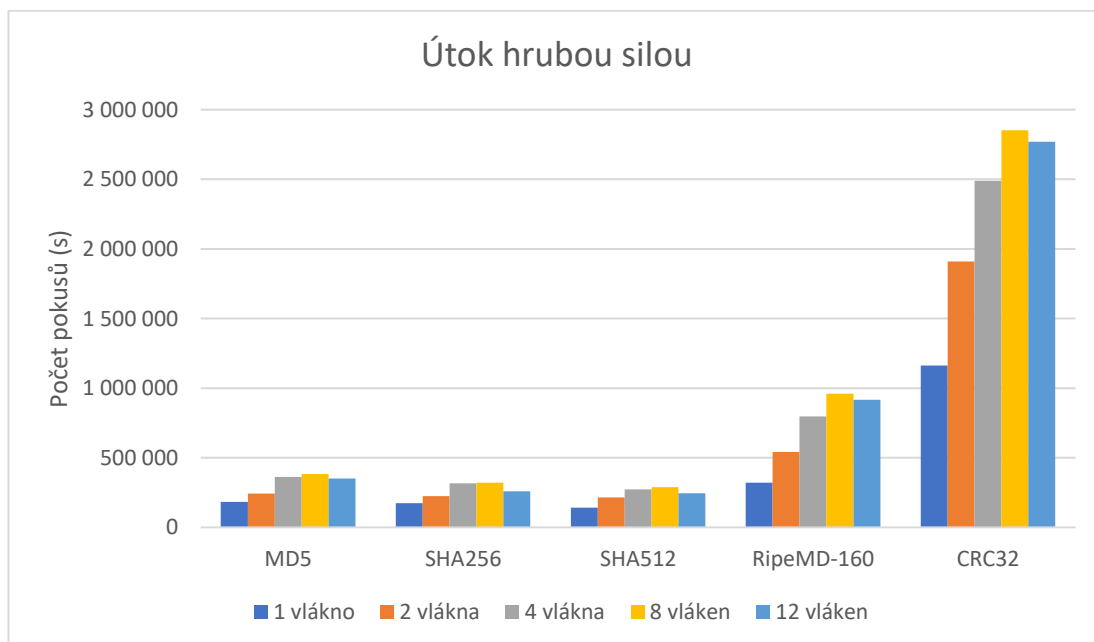
Na grafu můžeme vidět čas vytváření duhové tabulky ze souboru rockyou.txt. SHA512 je nejpomalejší, protože celkový soubor zabírá nejvíce místa a je nejpomalejší na kalkulaci. MD5 je pravý opak, a proto je mnohem rychlejší. Posun mezi jedním a dvěma vlákny je velký, a to i přes fakt, že se originální soubor musí rozdělit na menší a poté znova složit. Při 12 vláknech tento proces u SHA512 zbrzdil o necelých 10 sekund a je důvod, proč je výhodnější použít jen 8 vláken.



Graf 8 Generování duhové tabulky pro CRC32

Na rozdíl od ostatních hashovacích funkcí se u CRC32 vyplatí použít jedno vlákno. Rozdělení trvá až moc dlouhou dobu a díky rychlosti CRC32 je lepší to prostě neřešit.

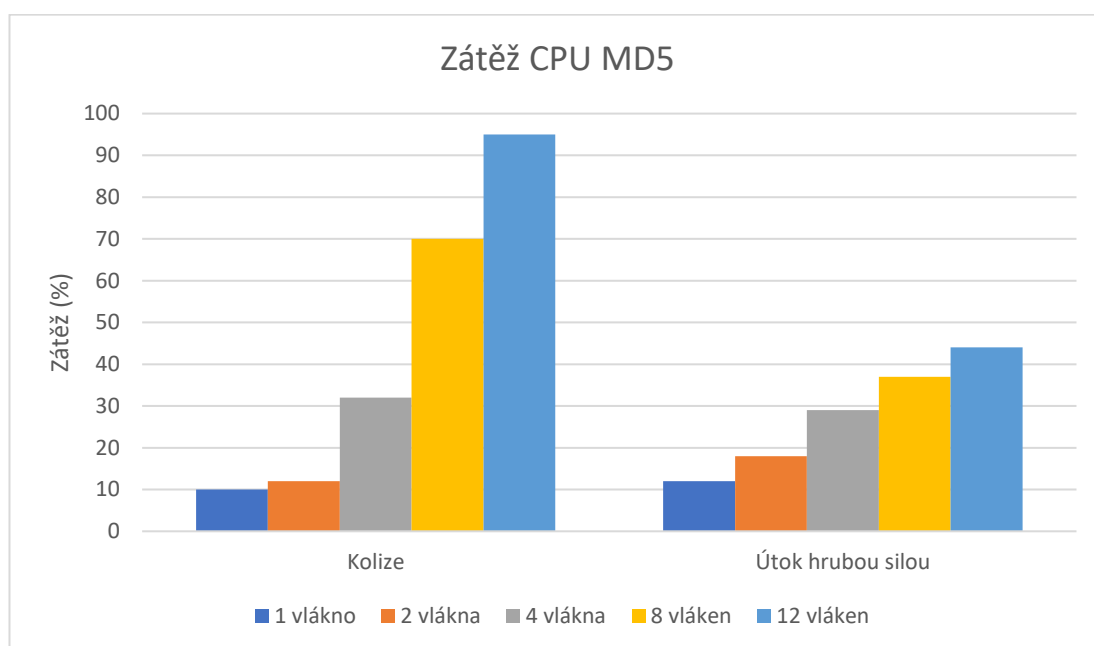
6.3 Útok hrubou silou



Graf 9 Útok hrubou silou

Graf nám ukazuje rychlost útoku hrubou silou a proč bychom neměli používat všech 12 vláken. Všude až na plný výkon můžeme vidět zvětšení rychlosti. Problém je, že se musí vypisovat i hodnoty do UI a celkově musí UI zůstat funkční pro zrušení hledání. Tyto akce se provádí na dalším vlákně, ale když používáme všechnu sílu CPU, tak si musíme někde trochu síly odebrat, což znamená menší rychlost.

Tentokrát jsem přidal RipeMD-160 do grafu a nečekal jsem, jak rychlý je. Na začátku jsem zjistil, že SHA1 a MD5 mají skoro stejnou kalkulaci a kvůli stejnému počtu bitu na výstupu jsem to stejné čekal i od RipeMD-160.



Graf 10 Zátěž CPU při multithreadingu s MD5

Jedna věc, které jsem si všimnul je, že díky čekání se nepoužívá plného potenciálu CPU. Na tomto případě to zrovna jde vidět rozdíl použití CPU při hledání kolizí a provádění útoku hrubou silou. Při obou bylo použita MD5, u CRC32 se takovýto jev skoro nevyskytuje a CPU při útoku hrubou silou pracuje na 90 %. Myslel jsem si, že to bude hashovací funkcí a že to zpomaluje výpočet, ale tím pádem by měl být stejný výsledek i u hledání kolize. Je to fascinující jev, ale nemám žádné logické vysvětlení. Jediný můj nápad je, že to je hashovací funkcí a při hledání kolize se počítá i projíždění listu, které zabírá více úsilí jak vypočítání hashe.

Závěr

Na závěr své písemné práce bych chtěl říct, že jsem velice spokojen se svým programem a jeho funkčností. Vadí mi ovšem nekonzistentnost kódu, která je způsobena učením se, hlavně u používání více vláken, a předěláváním kódu. Nebál bych se říci, že třetina času mi zabral debugging a použití Unit-testů, které jsem nakonec v půlce vypracovávání vynechal úplně. Už teď mě napadají další vylepšení a nápady a doufám, že program bude někomu prospěšný, ať už k výuce či k osobnímu použití. Psaní práce jsem si užil a pomocí teoretické části jsem se naučil něco málo víc o hashech.

Seznam použitých zdrojů

- [1] ŠTRÁFELDA, Jan. Co je hash či hashování. Online. [2008], aktualizováno 13.01.2025. Dostupné z: <https://www.strafelda.cz/hash>. [cit. 2025-01-13].
- [2] RIVEST, Ronald. The MD5 Message-Digest Algorithm. Technická specifikace. USA: MIT Laboratory for Computer Science and RSA Data Security, Inc., duben 1992. Dostupné také z: <https://datatracker.ietf.org/doc/html/rfc1321>. [cit. 2025-01-19].
- [3] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY [NIST]. Secure Hash Standard (SHS). Online. Srpen 2015. Dostupné z: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>. [cit. 2025-01-19].
- [4] EASTLAKE, D. a JONES, P. US Secure Hash Algorithm 1 (SHA1). Technická specifikace. USA: Cisco Systems, září 2001. Dostupné také z: <https://datatracker.ietf.org/doc/html/rfc1321>. [cit. 2025-01-19].
- [5] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY [NIST]. Secure Hashing. Online. INTERNET ENGINEERING TASK FORCE [IETF]. Computer Security Resource Center (CSRC). Říjen 2007, aktualizováno 5. května 2011. Dostupné z: https://web.archive.org/web/20110625054822/http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html. [cit. 2025-01-19].
- [6] KRČMÁŘ, Petr. SHA-1 není bezpečná, Google ukázal kolizi. Online. Root. 24. 7. 2017, Dostupné z: <https://www.root.cz/clanky/sha-1-neni-bezpecna-google-ukazal-kolizi/>. [cit. 2025-01-19].
- [7] DEPARTEMENT ELEKTROTECHNIEK (ESAT), The hash function RIPEMD-160. Online. [2005], aktualizováno 13. února 2012. Dostupné z: <https://homes.esat.kuleuven.be/~bosselae/ripemd160.html>. [cit. 2025-01-20].
- [8] BLACK, Richard, UNIVERSITY OF CAMBRIDGE. Fast CRC32 in Software. Online. Department of Computer Science and Technology. 18. 2. 1994. Dostupné z: <https://www.cl.cam.ac.uk/research/srg/projects/fairisle/bluebook/21/crc/crc.html>. [cit. 2025-01-20].

- [9] Cyclic redundancy check: Cyklický redundantní součet. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, listopad 2004. Dostupné z: https://en.wikipedia.org/wiki/Cyclic_redundancy_check. [cit. 2025-01-20].
- [10] THE JAPAN PRIZE FOUNDATION, Laureates of the Japan Prize. Online. [2012]. Dostupné z: https://www.japanprize.jp/en/prize_prof_1999_peterson.html. [cit. 2025-01-20].
- [11] DEFUSE SECURITY. Salted Password Hashing - Doing it Right. Online. Crack Station. [2012], aktualizováno 28. září 2021. Dostupné z: <https://crackstation.net/hashing-security.htm>. [cit. 2025-01-21].
- [12] OŠŤÁDAL, Radim. Teoretický základ a přehled kryptografických hashovacích funkcí. PDF, Technická specifikace. Brno: Masarykova univerzita, 2012. Dostupné z: https://is.muni.cz/www/ostadal/hash_overview.pdf. [cit. 2025-01-22]
- [13] KOZLÍK, Andrew. Hashovací funkce. PDF, Technická specifikace. Praha: Univerzita Karlova, Matematicko fyzikální fakulta, [2024]. Dostupné také z: https://www.karlin.mff.cuni.cz/~kozlik/udk_mat/hash.pdf.
- [14] NORD VPN s.a. Cookie hash. Online. Nord VPN. C2025 Dostupné z: <https://nordvpn.com/cybersecurity/glossary/cookie-hash/>. [cit. 2025-01-22].
- [15] KASPERSKY LAB. Brute Force Attack: Definition and Examples. Online. Kaspersky. [2019] C2025. Dostupné z: <https://www.kaspersky.com/resource-center/definitions/brute-force-attack>. [cit. 2025-01-23].
- [16] VAIDEESWARAN, Narendran. NTLM Explained. Online. CrowdStrike Holdings, Inc. 2011. Dostupné z: <https://www.crowdstrike.com/en-us/cybersecurity-101/identity-protection/windows-ntlm/>. [cit. 2025-01-23].
- [17] CHICK3NMAN. Hashcat v6.2.6 benchmark. Online. MICROSOFT. Github. 2022, aktualizováno února 2024. Dostupné z: <https://gist.github.com/Chick3nman/32e662a5bb63bc4f51b847bb422222fd>. [cit. 2025-01-23].

- [18] BAKER, Kurt. Pass-the-Hash Attack. Online. CrowdStrike Holdings, Inc. 2011.
Dostupné z: <https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/pass-the-hash-attack/>. [cit. 2025-01-23].
- [19] CORESECURITY. Pass-the-Hash Toolkit for Windows. Online. Fortra. [2020].
Dostupné z: <https://www.coresecurity.com/core-labs/publications/pass-hash-toolkit-windows>. [cit. 2025-01-24].
- [20] MICROSOFT. Prohlídka jazyka C#. Online. Microsoft Learn. 09.05.2024.
Dostupné z: <https://learn.microsoft.com/cs-cz/dotnet/csharp/tour-of-csharp/overview>. [cit. 2025-01-13].
- [21] KIRVAN, Paul. What is multithreading? Online. TechTarget. C1999-2015.
Dostupné z: <https://www.techtarget.com/whatis/definition/multithreading>. [cit. 2025-03-14].
- [22] MICROSOFT. Scénáře asynchronního programování. Online. Microsoft Learn. 22.03.2025. Dostupné z: <https://learn.microsoft.com/cs-cz/dotnet/csharp/asynchronous-programming/async-scenarios>. [cit. 2025-03-14].
- [23] MICROSOFT. Interlocked.Increment Metoda. Online. Microsoft Learn. C2025.
Dostupné z: <https://learn.microsoft.com/cs-cz/dotnet/api/system.threading.interlocked.increment?view=net-9.0>. [cit. 2025-03-14].
- [24] MICROSOFT. Control.Invoke Metoda. Online. Microsoft Learn. C2025.
Dostupné z: <https://learn.microsoft.com/cs-cz/dotnet/api/system.windows.forms.control.invoke?view=windowsdesktop-8.0>. [cit. 2025-03-14].
- [25] MICROSOFT. CancellationTokenSource Třída. Online. Microsoft Learn. C2025. Dostupné z: <https://learn.microsoft.com/cs-cz/dotnet/api/system.threading.cancellationtokensource?view=net-8.0>. [cit. 2025-03-14].
- [26] PRAHA CODING SCHOOL. Co je to Git, GitHub a proč byste je měli znát? Online. Praha Coding. [2025]. Dostupné z: <https://prahacoding.cz/co-je-to-git-github/>. [cit. 2025-01-24].

- [27] Freelo. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, 03.01.2018. Dostupné z: <https://cs.wikipedia.org/wiki/Freelo>. [cit. 2025-01-25].
- [28] Just-in-time výroba. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, 11.01.2007. Dostupné z: https://cs.wikipedia.org/wiki/Just-in-time_v%C3%BDroba. [cit. 2025-01-25].
- [29] DICTIONARY, Kanban. Online. [2016]. Dostupné z: <https://www.dictionary.com/browse/Kanban>. [cit. 2025-01-25].
- [30] INTERNET ARCHIVE. About IA. Online. Internet Archive .org. [2001]. Dostupné z: <https://archive.org/about/>. [cit. 2025-01-25].
- [31] KARIA, Ravi. BigInteger Data Type in C#. Online. TutorialTeacher. C2024. Dostupné z: <https://www.tutorialsteacher.com/articles/biginteger-type-in-csharp>. [cit. 2025-03-14].

Seznam výpisů

Výpis 1 Příklad rozdílu použití soli a pepře na výstupní hash	13
Výpis 2 Příklad použití volatile	21
Výpis 3 Příklad použití async a await v obsluze pro tlačítko	21
Výpis 4 Příklad bez lambda výrazu	21
Výpis 5 Příklad s lambda výrazem.....	21
Výpis 6 Příklad použití thread safe pro zvýšení proměnné.....	22
Výpis 7 Příklad použití metody Invoke	22
Výpis 8 Příklad použití třídy CancellationTokenSource	22
Výpis 9 Metoda Hash ve třídě Hasher	26
Výpis 10 Metoda HashMD5 ve třídě Hasher	26
Výpis 11 Metoda HashCRC32 ve třídě Hasher	27
Výpis 12 Metoda CRC32Table ve třídě Hasher, která je volána metodou HashCRC32	28
Výpis 13 Obsluha menu strip tlačítka použít sůl	30
Výpis 14 Metoda RegistryUseLightMode ve třídě Settings	31
Výpis 15 Příklad třídy CustomColorTable a vlastnosti MenuStripGradientBegin....	33
Výpis 16 Metoda při stisknutí tlačítka kopírovat ve formuláři	35
Výpis 17 Příklad použití enum a vlastnosti ve třídě Settings	35
Výpis 18 Vlastnost DirectoryPathToSettings	35
Výpis 19 Část metody InitialFolderChecker ve třídě Settings.....	36
Výpis 20 Metoda ResetSettings ve třídě Settings	37
Výpis 21 Metoda SaveSettings ve třídě Settings	37
Výpis 22 Textový soubor settings.txt ve složce Settings	38
Výpis 23 Příklad čtení nastavení ze souboru v metodě LoadSettings ve třídě Settings	39
Výpis 24 Metody ve třídě FormManagement	39
Výpis 25 Příklad, jak vytvořit datový typ Dictionary	40
Výpis 26 Metoda AllLanguages ve třídě Languages	40
Výpis 27 Metoda AddLanguagesToMenu ve třídě Languages.....	41
Výpis 28 Metoda LoadDictionary ve třídě Languages	42
Výpis 29 Metoda Translate ve třídě Languages.....	43
Výpis 30 Japonský překlad formuláře	44
Výpis 31 Český překlad formuláře	44

Výpis 32 Příklad použití časovače a periody	45
Výpis 33 Příklad použití proměnné v programu	46
Výpis 34 Přetížení metody ProcessingHash	47
Výpis 35 Část kódu pro hashování více hashovacích funkcí ve třídě Hasher	47
Výpis 36 Metoda GradualHashing ve třídě Hasher	48
Výpis 37 Metoda IsUsingSaltAndPepper ve třídě Hasher	51
Výpis 38 Metoda SaveSalt ve třídě Hasher.....	52
Výpis 39 Metoda CheckPepper ve třídě Hasher	53
Výpis 40 Data v souboru name2.txt.....	56
Výpis 41 Metoda ShowIDInfo	57
Výpis 42 Metoda FileChecksum ve třídě Hasher	58
Výpis 43 Velice jednoduchý rozhodovací proces hashovací funkce podle délky hashe	60
Výpis 44 Metoda GenerateCollision ve formuláři HashingCollisionForm	61
Výpis 45 Počítání náhodného vstupu pomocí třídy Random.....	61
Výpis 46 Metoda GenerateRandomString pro generování náhodného vstupu.....	62
Výpis 47 Příklad vygenerovaného textového souboru.....	62
Výpis 48 Část metody pro zpracování textového souboru.....	63
Výpis 49 Textový soubor CRC32-Prefab.txt ve složce Collisions	65
Výpis 50 Vlastnost Progress ve třídě DictionaryAttack	67
Výpis 51 Metoda PasswordFinder ve třídě DictionaryAttack	68
Výpis 52 Zkouška, zda v poli byly najité všechny vstupy	68
Výpis 53 Metoda Calculator ve třídě PasswordStrenghtCalculator.....	70
Výpis 54 Metoda TryPower pro mocnění BigInteger.....	71
Výpis 55 Metoda Output ve třídě PasswordStrenghtCalculator	72
Výpis 56 Rozhodující proces použití délky či samotné heslo.....	73
Výpis 57 Hlavička metody GenerateRainbowTable ve třídě RainbowTableGenerator	74
Výpis 58 Tělo metody GenerateRainbowTable ve třídě RainbowTableGenerator ...	74
Výpis 59 Příklad začátku duhové tabulky v textovém souboru	74
Výpis 60 Metoda GenerateRainbowTableMultiThreadForSingleThread.....	76
Výpis 61 Hlavička metody GenerateRainbowTableMultiThread.....	76
Výpis 62 Metoda RemoveFilesQuestion	77
Výpis 63 Metoda SelectAllUsableChars ve třídě BruteForceAttack	80

Výpis 64 Metoda CalculateAllPossibleCombinations ve třídě BruteForceAttack	81
Výpis 65 Metoda PasswordBruteForce ve třídě BruteForceAttack	82
Výpis 66 Logika posunu hledané délky ve třídě BruteForceAttack	82
Výpis 67 Metoda GenerateText ve třídě BruteForceAttack	83
Výpis 68 Příklad použití metody GenerateText.....	83
Výpis 69 Logika vstupu textboxu v groupboxu Útok hrubou silou.....	84

Seznam grafů

Graf 1 Šance na nalezení kolize pro CRC32.....	15
Graf 2 Šance na nalezení kolize pro SHA256.....	16
Graf 3 Šance na nalezení kolize pro RipeMD-160	16
Graf 4 Rychlost hledání kolizí MD5 za určitou dobu	85
Graf 5 Průměrná rychlost při hledání kolize pro jedno vlákno	86
Graf 6 Průměrný čas na najetí kolize v CRC32.....	86
Graf 7 Generování duhové tabulky	87
Graf 8 Generování duhové tabulky pro CRC32.....	88
Graf 9 Útok hrubou silou	88
Graf 10 Zátěž CPU při multithreadingu s MD5	89

Seznam zkratek

Zkratka	Význam Zkratky
k	Počet vygenerovaných hashů
e	Eulerovo číslo (přibližně 2.718)
N	Počet možných hashů
P	Zaokrouhlena šance na kolizi

Seznam obrázků

Obrázek 1 Příklad 1. kola hashovací funkce MD5	10
Obrázek 2 Příklad rozvrhnutí práce pomocí Freela	24
Obrázek 3 Vzhled hlavního formuláře	28
Obrázek 4 Podsekce Hashování ve strip menu	29
Obrázek 5 Podsekce Volba ve strip menu.....	29
Obrázek 6 Podsekce Sůl a pepř v podsekcí Volba	30
Obrázek 7 Podsekce Sůl a pepř se zapnutými možnostmi	30
Obrázek 8 Podsekce VisualMode v nastavení	30
Obrázek 9 Hlavní formulář ve tmavém motivu	31
Obrázek 10 Všechny přednastavené barvy ve třídě Colors (https://learn.microsoft.com/enus/dotnet/api/system.windows.media.colors?view=windowsdesktop-9.0)	32
Obrázek 11 Ukázka strip menu typu výstupu	33
Obrázek 12 Podsekce Styl výstupu v nastavení	34
Obrázek 13 Příklad všech možností samostatně a poté všechny najednou.....	34
Obrázek 14 Fungování záznamu	34
Obrázek 15 Příklad strip menu v japonském jazyce	44
Obrázek 16 Formulář pro nastavení frekvence UI.....	45
Obrázek 17 Formulář pro nastavení počtu logických jader	46
Obrázek 18 Formulář postupného hashování.....	49
Obrázek 19 Formulář pro vícenásobné hashování	50
Obrázek 20 Formulář pro práci se solí a pepřem	54
Obrázek 21 Formulář pro zadání soli a pepře	55
Obrázek 22 Výstup tlačítka Ukázat všechny uživatele	55
Obrázek 23 Výstup tlačítka Ukázat všechny ID	56
Obrázek 24 Složka HashData.....	56
Obrázek 25 Výstup tlačítka Informace o ID	56
Obrázek 27 Formulář pro generování kontrolního součtu	59
Obrázek 28 Formulář pro generování kolizí	64
Obrázek 29 Formulář pro kontrolu kolize.....	65
Obrázek 30 Formulář Prolamovač Hesel	66
Obrázek 31 Groupbox Slovníkový útok	69
Obrázek 32 Ukázka slovníkového útoku s více vstupy	70

Obrázek 33 Ukázka výstupu slovníkového útoku s více vstupy v záznamu	70
Obrázek 34 Groupbox Čas k prolomení kalkulátor	72
Obrázek 35 Příklad výstupu při délce hesla 20 a všech 95 znaků	73
Obrázek 36 Fungování metody GenerateRainbowTableMultiThread pro 4 vlákna ..	75
Obrázek 37 Groupbox Útok duhovou tabulkou	78
Obrázek 39 Výstup nalezené shody při útoku duhovou tabulkou	79
Obrázek 40 Výstup nalezené shody při útoku duhovou tabulkou v záznamu	79
Obrázek 41 Groupbox Útok hrubou silou	84

Seznam tabulek

Tabulka 1 Šance na najít kolize	14
Tabulka 2 Rozdíly mezi útokem pomocí slovníku a duhové tabulky	18

ⁱ Multithreading v překladu více vláknové řízení či použití více vláken

ⁱⁱ CLI je zkratka pro Command Line Interface (rozhraní příkazového řádku)

ⁱⁱⁱ GUI je zkratka pro Graphical User Interface (grafické uživatelské rozhraní)

^{iv} SaaS je zkratka pro Software as a Service (software jako služba)

^v To-Do list znamená list věcí, které se ještě musí udělat.

^{vi} Uniform Resource Locator (URL) znamená jednoznačné určení zdroje a používají to webové stránky jakožto doménovou adresu (příklad je seznam.cz)

^{vii} RGB znamená Red Green a Blue, což je formát pro nastavení barvy. Každá barva má hodnotu od 0 do 255. 0 ==> nesvíí a 255 ==> svítí nejvíce. Černá barva jsou všechny hodnoty na 0 a bílá všechny hodnoty na 255.

^{viii} AI, artificial intelligence, je počítačově simulovaný algoritmus založený na neuronech a strojovém učení.