

Střední průmyslová škola Třebíč

Maturitní práce

Hashtesting

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2024/2025 Kamil Franek

Zadání práce

ABSTRAKT

Maturitní práce na téma hashování, hashovací funkce, jejich využití a rozdíly v hashovacích funkcích. Zabývá se problematikou spojenou s hashováním a vysvětlení použití hashování v IT s příklady pomocí programu. Tento dokument popisuje použitou technologii, praktiky a vytváření 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 a kybernetické útoky na hashe a zobrazuje rozdíl mezi použitím a nepoužitím soli a pepře.

KLÍČOVÁ SLOVA

maturitní práce, výuka, hash, kybernetický útok, multithreading

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 basci and advanced hash and file handling features to increase efficiency. It also has simulations of login and cyber attacks on hashes and shows differences between using and not using salt and 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 27. 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 27. března 2025

podpis autora

Obsah

[Úvod 8](#_Toc193956701)

[1 Teorie hashovaní 9](#_Toc193956702)

[1.1 Použití Hashů 9](#_Toc193956703)

[1.2 Použité Hashe 9](#_Toc193956704)

[1.2.1 MD5 9](#_Toc193956705)

[1.2.2 SHA1/SHA256/SHA512 10](#_Toc193956706)

[1.2.3 RIPEMD160 11](#_Toc193956707)

[1.2.4 CRC32 11](#_Toc193956708)

[1.3 Sůl, Pepř a jejich používání 12](#_Toc193956709)

[1.4 Kybernetické útoky na hashe 13](#_Toc193956710)

[1.4.1 Pravděpodobnost kolize (Narozeninový paradox) 14](#_Toc193956711)

[1.5 Útoky 16](#_Toc193956712)

[1.5.1 Duhové tabulky a slovníkové útoky 17](#_Toc193956713)

[1.5.2 Útok hrubou silou 18](#_Toc193956714)

[1.5.3 Pass the Hash 18](#_Toc193956715)

[2 Použité programy a technologie 20](#_Toc193956716)

[2.1 C# a .NET 20](#_Toc193956717)

[2.2 Multithreading 20](#_Toc193956718)

[2.2.1 Problémy s Multithreadingem 20](#_Toc193956719)

[2.2.2 Async a Await 21](#_Toc193956720)

[2.2.3 Interlocked.Increment 21](#_Toc193956721)

[2.2.4 Invoke 22](#_Toc193956722)

[2.2.5 CancellationTokenSource 22](#_Toc193956723)

[2.3 Visual Studio 2022 22](#_Toc193956724)

[2.4 Visual Code 22](#_Toc193956725)

[2.5 Git/Github a Github Desktop 23](#_Toc193956726)

[2.6 Freelo.io 24](#_Toc193956727)

[2.7 Wayback Machine 25](#_Toc193956728)

[2.8 Online hashers 25](#_Toc193956729)

[3 Základy hashování a hlavní formulář 26](#_Toc193956730)

[3.1 Hashovací funkce a metody 26](#_Toc193956731)

[3.2 Hlavní Formulář 29](#_Toc193956732)

[3.2.1 Strip menu 29](#_Toc193956733)

[3.3 Záznam 35](#_Toc193956734)

[3.4 Třída Settings 35](#_Toc193956735)

[3.5 Třída FormManagement 40](#_Toc193956736)

[3.6 Jazyk a lokalizace 41](#_Toc193956737)

[3.7 Frekvence UI formulář 45](#_Toc193956738)

[3.8 Nastavení CPU a vláken formulář 46](#_Toc193956739)

[3.9 Multi-Hashování 47](#_Toc193956740)

[4 Dodatečné funkce programu 49](#_Toc193956741)

[4.1 Postupné hashování 49](#_Toc193956742)

[4.1.1 Kod pro postupné hashování 49](#_Toc193956743)

[4.1.2 Formulář pro postupné hashování 49](#_Toc193956744)

[4.2 Formulář pro vícenásobné hashování 50](#_Toc193956745)

[4.3 Práce se solí a pepřem 51](#_Toc193956746)

[4.3.1 Kód pro práci se solí a pepřem 51](#_Toc193956747)

[4.3.2 Formulář pro práci se solí/pepřem 54](#_Toc193956748)

[4.4 Kontrolní součet 58](#_Toc193956749)

[4.4.1 Kód pro kontrolní součet 59](#_Toc193956750)

[4.4.2 Formulář pro kontrolu souborů 60](#_Toc193956751)

[4.5 Hledání kolizí 61](#_Toc193956752)

[4.5.1 Kód 62](#_Toc193956753)

[4.5.2 Formulář pro hledání kolizí 65](#_Toc193956754)

[5 Útoky na hashe 67](#_Toc193956755)

[5.1 Slovníkový útok 67](#_Toc193956756)

[5.1.1 Kód pro slovníkový útok 67](#_Toc193956757)

[5.1.2 Formulář pro slovníkový útok 70](#_Toc193956758)

[5.2 Výpočet prolomení hesla 71](#_Toc193956759)

[5.2.1 Kód 71](#_Toc193956760)

[5.2.2 Formulář Čas k prolomení kalkulátor 73](#_Toc193956761)

[5.3 Útok duhovou tabulkou 74](#_Toc193956762)

[5.3.1 Generování duhové tabulky 74](#_Toc193956763)

[5.3.2 Generování duhové tabulky pomocí více vláken 75](#_Toc193956764)

[5.3.3 Formulář pro duhovou tabulku 79](#_Toc193956765)

[5.4 Útok hrubou silou 80](#_Toc193956766)

[5.4.1 Kód 80](#_Toc193956767)

[5.4.2 Formulář pro útok hrubou silou 85](#_Toc193956768)

[6 Statistika 86](#_Toc193956769)

[6.1 Vyhledávání kolizí 86](#_Toc193956770)

[6.2 Generování duhové tabulky 87](#_Toc193956771)

[6.3 Útok hrubou silou 90](#_Toc193956772)

[Závěr 91](#_Toc193956773)

[Seznam použitých zdrojů 92](#_Toc193956774)

[Seznam použitých symbolů a zkratek 93](#_Toc193956775)

[Seznam obrázků 94](#_Toc193956776)

[Seznam tabulek 95](#_Toc193956777)

[Seznam příloh 96](#_Toc193956778)

Úvod

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

# Teorie hashovaní

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

## Použití hashů

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

## Použité hashe

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

### MD5

MD5 (Message-Digest Algorithm) pochází z rodiny „Message-Digest“ neboli algoritmus na strávení zprávy. Předchůdci MD5 jsou hashovací funkce MD2 a MD4, všechny tři vytvořeny a vydány Ronaldem Rivestem. MD2 byl vydán v roce 1989, MD4 jakožto pokračovatel v 1990 a MD5 jakožto vylepšená verze MD5 v roce 1991. MD5, na rozdíl od svých předchůdců, je docela složitý algoritmus na rozlousknutí. Používá 4 kola, místo 3 kol jako MD4, a pomocí matematických operací s 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. [1][2]



Obrázek Příklad 1. kola hashovací funkce MD5

### SHA1/SHA256/SHA512

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

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

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

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

### RIPEMD160

RipeMD160 je hashovací funkce, která měla za účel nahradit MD4 a MD5, stejně jako SHA1. Hlavní rozdíl je, že RipeMD160 byla vyvinuta v EU jakožto součást projektu RIPE[[1]](#footnote-1). RipeMD160 byla vytvořena Hansem Dobbertinem, Antoonem Bosselaersem a Bartem Preneelem. Spolu s RipeMD128, RipeMD256 a RipeMD320 byly vydány v roce 1996. Všechny tyto verze vstávají z originální RipeMD hashovací funkce, která byla vydána roku 1992. Dnes všechny tyto verze RipeMD nejsou doporučovány používat, i když v lepších verzích RipeMD (vydány v 1996) nebyly nalezeny žádné kolize. [8]

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

### 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.“ [10], a CRC16, které se používá v Bluetooth, SD, X.25, v různých mobilních sítích, USB a mnoho dalších. [11]

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

#### Výpočet CRC

Základ celého výpočtu CRC jsou polynomy. Každý jeden bit v určité délce může být zapsán do polynomu. Příklad (1101) může být zapsán jako . poslední bit reprezentuje 1, druhý bit x, třetí atd. CRC32 má až a CRC16 až .

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]](#footnote-2). [10] [11]

## Sůl, Pepř a jejich používání

Hashovací sůl a pepř jsou další vrstvou pro bezpečnost hashování. Sůl je náhodně vygenerována před generováním a dává se před samotnými daty. Může mít jakoukoliv délku, záleží na správci, který bude hodnotu ukládat. Díky soli se chráníme před takzvanými 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. [13]

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.

Všechny hashe jsou šifrované v hashovací funkci MD5.  


Výpis Příklad rozdílu použití soli a pepře na výstupní hash

## 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 . To znamená že CRC32 má šanci 1 ku (skoro 4.3 miliardy kombinací) neboli 0.0000000233 %. To se může zdát jako velice malá šance, jenomže tohle je šance jenom mezi 2 hashy. Když započítáme šanci každého s každým použitím narozeninového paradoxu, kdy hledáme kolize každého s každým, počet všech kombinací se nám sníží na .

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

Tabulka Šance na najití kolize

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

### Pravděpodobnost kolize (Narozeninový paradox)

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

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

Tato upravená verze není až tak přesná, ale dá se mnohem snadněji vypočítat. [15, strana 11 a 12]

k = počet vygenerovaných hashů

e = Eulerovo číslo (přibližně 2.718)

N = počet možných hashů ()

P ≥ zaokrouhlená šance na kolizi (mezi 0 a 1 / 0 % a 100 %)

Graf 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 najití jakékoliv jedné kolize. Pokud chceme najít pravděpodobnost kolize v určité šanci, třeba na 23 % (tedy p=0.23), můžeme si poupravit naši rovnici a získat vzorec (ln je přirozený logaritmus neboli logaritmus o základu e)..

Po dosazení vznikne , n ≐ 47 382. Po projití 47 382 hashů máme 23% šanci na naleznutí kolize.

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

Tabulka Šance na nalezení kolize pro SHA256 a RipeMD-160

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

## Útoky

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

### Duhové tabulky a slovníkové útoky

Slovníkový útok a útok pomocí duhové tabulky můžou na první pohled vypadat jako jeden a ten samý útok, jenže mají jeden hlavní rozdíl. Oba útoky jsou nějaká databáze potenciálních hesel, jediný rozdíl je, že duhová tabulka vyhledává už zaheshovaný 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 zaheshovanými 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 zaheshovanému heslu. Díky délce hashe můžeme docela dobře zjistit použitou funkci. Každá funkce má svoji vlastní tabulku. Místo abychom jako u slovníkového útoku hledali počátek, hledáme radši shodu ve finálním hashy. Jakmile ho najdeme, tak s ním máme i originální vstup. Největší obtíže tomuto stylu útoku dělá používání soli a pepře při hashování vstupu. Jelikož skladování jak vstupu, tak výstupu může být u některých hashovacích funkcí náročné na místo.

Tabulka Rozdíly mezi útokem pomocí slovníku a duhové tabulky

|  |  |  |
| --- | --- | --- |
| **Vlastnosti** | **Slovníkový útok** | **Duhová tabulka** |
| Rychlost hashování | Záleží na použité funkci | Před hashováno |
| Náročnost na uložiště | 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 |

### 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 . [17] V našem příkladu máme milionů možných kombinací. To může znít jako hodně, ale když jedna grafická karta RTX 4090 dokáže zvládnout 200 tisíc hashů za sekundu BCryptu[[3]](#footnote-3) a 300 milionů hashů NTLM [[4]](#footnote-4) [17], tak 100 milionu najednou nezní tak krásně.

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

### Pass the Hash

Hlavní důvod, proč se hashování používá pro hesla je zabránit tomu, aby si někdo při komunikaci mohl jen tak vzít data a přečíst 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 Linuxu[[5]](#footnote-5). [21] [22]

# Použité programy a technologie

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

## C# a .NET

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

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

## Multithreading[[6]](#endnote-1)

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. [33]

### Problémy s Multithreadingem

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



Výpis Příklad použití volatile

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

### Async a await



Výpis 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. [34]

#### Lambda výraz

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



Výpis Příklad bez lambda výrazu



Výpis Příklad s lambda výrazem

### 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. [35]



Výpis Příklad použití thread safe pro zvýšení proměnné

### Invoke

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



Výpis Příklad použití metody Invoke

### CancellationTokenSource

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



Výpis Příklad použití třídy CancellationTokenSource

## 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 frameworkem. Tuto knihovnu používám pro všechny hashe použité v programu, až na CRC32. Dále poskytuje spousty UI elementů a práci s formulářem, jednoduché přidání unit-testů, chybové hlášení, krokování programu, statistika využití CPU a RAM a mnoho dalšího.

## Visual Code

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

## Git/Github a Github Desktop

„Git je verzovací systém, pomocí kterého ukládáte své projekty a veškeré jejich verze. Je to distribuovaný systém správy verzí. To znamená, že k celému kódu i jeho historii se vývojář dostane z jakéhokoliv počítače.[[7]](#footnote-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.“ [24]

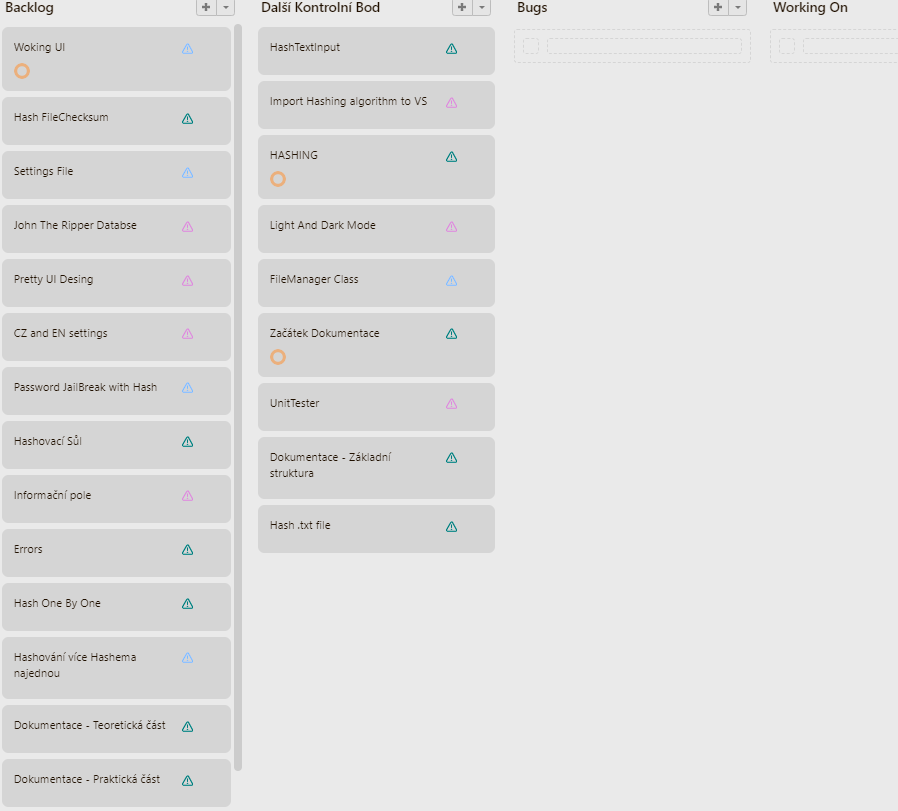
Github je webová stránka, kde se dají procházet a ukládat různé Git projekty. Github nabízí spousty služeb, článků, a dokonce nabízí i předplatné pro ještě více funkcí. Jedna z jejich nejnovějších zaměření je Copilot, což je umělá inteligence (AI), která má dělat programování o něco jednodušší. Nejdůležitější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

„Díky tomuto systému je spolupráce na projektu bezpečnější a jednodušší. Možná i to je důvod, proč Git používá více než 87 % vývojářů.“ [24]

Github desktop je počítačová aplikace, která funguje stejně jako příkazový řádek v Git, akorát místo CLI[[8]](#endnote-2) to je GUI[[9]](#endnote-3) a dělá to Git uživatelsky přívětivé.

## Freelo.io

„Freelo je [online](https://cs.wikipedia.org/wiki/Online) aplikace s cílem usnadnit [řízení projektů](https://cs.wikipedia.org/wiki/%C5%98%C3%ADzen%C3%AD_projekt%C5%AF) a zvýšit efektivitu podnikatelů a firem. Řadí se mezi [SaaS](https://cs.wikipedia.org/wiki/SaaS)[[10]](#endnote-4) aplikace a v červnu 2021 překročilo hranici 50 000 uživatelů.“ „Mezi hlavní funkce Freela patří Projekty, To-Do listy, Diskuse, Kalendář a Finance. Jejich cílem je usnadnění finanční správy projektu.“ [27]

Freelo je založeno na agilním projektovém řízení kanban[[11]](#footnote-7). Kanban je založen na systému výroby Just In Time[[12]](#footnote-8) (JIT), která byla vymyšlena v japonské automobilové firmě Toyota. Zakladatel je Taiičiho Óno. Kanban zjednodušuje práci několika lidí a její management. Díky kanbanu jde krásně vidět co kdo dělá, co bude dělat a co se musí ještě udělat. V mém případě Freelo používám jako to-do list[[13]](#endnote-5). [29]

Obrázek Příklad rozvrhnutí práce pomocí Freela

## Wayback Machine

Wayback Machine je digitální archiv, který je součástí Internet Archive. Internet Archive je nezisková organizace, původem z USA, která se snaží zachovat co nejvíce z internetu. Od webových stránek, po videa, audia, obrázky, programy a další. V dnešní době mají uloženo přes 960 miliard stránek v rozmezí 1996 až konce 2024. [31]

Nejlepší je, že pro uživatele je všechno k podívání zadarmo bez potřeby účtu. U webových stránek stačí mít pouze URL[[14]](#endnote-6) adresu. Záznamů může být několik a člověk si může vybrat. Odkaz na Internet Archive <https://archive.org> a Wayback Machine <https://web.archive.org>

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

## Online 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 spousty programů a stránek, já jsem používal <https://www.browserling.com/tools/all-hashes>, kvůli použití všech hashovacích funkcí, které používám v programu.

# 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.

## 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 *Hashing* *Algorithm*, díky kterému se vybírá, kterou hashovací funkci použít. V základu je na výběr MD5, SHA1, SHA256, SHA512, RipeMD-160 a CRC32.

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



Výpis Metoda Hash ve třídě Hasher

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



Výpis 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.



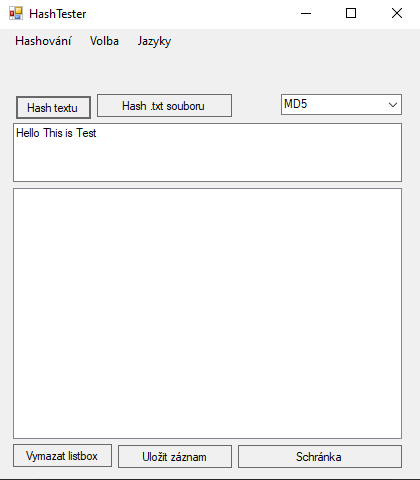
Výpis Metoda HashCRC32 ve třídě Hasher



Výpis 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.

## Hlavní Formulář



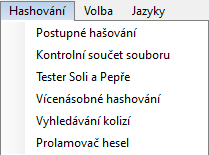
Obrázek Vzhled hlavního formuláře

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*.

### 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.

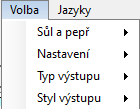
#### Hashování



Obrázek 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.

#### Volba



Obrázek Podsekce Volba ve strip menu

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



Obrázek Podsekce Sůl a pepř v podsekci Volba



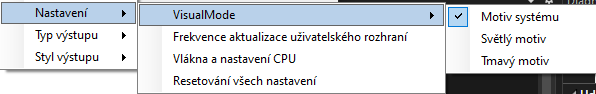
Obrázek Podsekce Sůl a pepř se zapnutými možnostmi

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



Výpis Obsluha menu strip tlačítka použít sůl

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



Obrázek Podsekce VisualMode v nastavení

Pod sekcí 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*.

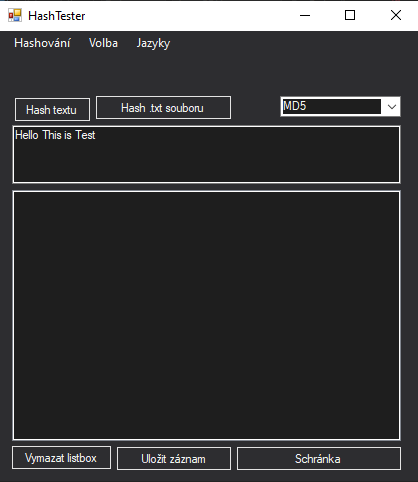
#### Motivy

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.



Výpis 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 Hlavní formulář ve tmavém motivu

#### 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.



Obrázek Všechny přednastavené barvy ve třídě Colors [(https://learn.microsoft.com/en-us/dotnet/api/system.windows.media.colors?view=windowsdesktop-9.0]((https:/learn.microsoft.com/en-us/dotnet/api/system.windows.media.colors?view=windowsdesktop-9.0))



Výpis 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[[15]](#endnote-7). 90 % času je hodnota světlejší nebo tmavší šedá.

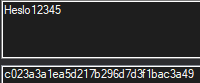
#### Uživatelské preference

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

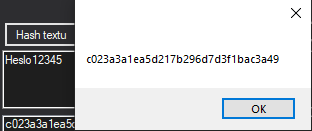


Obrázek Ukázka strip menu typu výstupu

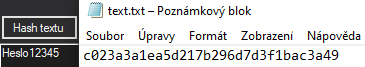
Typ výstupu funguje jako komponenta radiobutton. Program podporuje výstup do MessageBox, Listboxu (záznamu) a do nového .txt souboru.



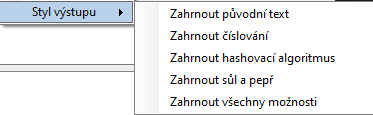
Příklad do záznamu.



Příklad do MessageBox

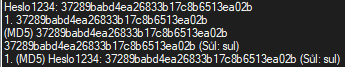


Příklad do souboru.



Obrázek Podsekce Styl výstupu v nastavení

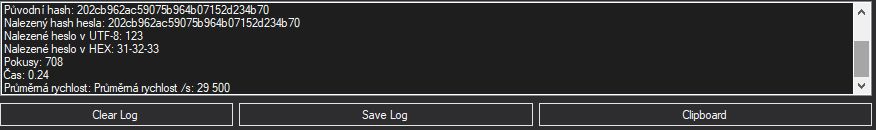
Styl výstupu formátuje výsledný text podle několika možností. Všechny možnosti fungují jako checkbox. Když jsou všechny 4 možnosti zahrnuty, zaškrtne se „Zahrnout všechny možnosti“. Pomocí toho se dají rychle nastavit či odebrat všechny možnosti. Pro každodenní práci s hashy to není moc užitečné, ale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



Postupný příklad všech možností samostatně a poté všechny najednou.

## 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.



Příklad použití záznamu. Vymazat záznam odstraní všechny položky ze záznamu. Tlačítko uložit záznam otevře SaveFileDialog a následně všechny položky uloží ve formátu .txt. 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.



Výpis Metoda při stiknutí 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.

## Třída Settings

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



Výpis Příklad použití enum a vlastnosti ve třídě Settings



Výpis 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.



Výpis 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.



Výpis Metoda ResetSettings ve třídě Settings



Výpis 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 zvlášť 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.



Výpis Textový soubor settings.txt ve složce Settings

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



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

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

## Třída FormManagement

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



Výpis 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.

## Jazyk a lokalizace

V tuto dobu jsou podporované dva jazyky, angličtina a čeština. Bez problému by stačila čeština, ale jelikož je program dělán za pomocí GitHubu, na který to mám v plánu i vydat, tak jsem se rozhodl přeložit celý program do angličtiny za pomocí DeepL online překladače, jenž funguje na AI[[16]](#endnote-8).



Výpis Příklad, jak vytvořit datový typ Dictionary

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



Výpis 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*.



Výpis 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.



Výpis Metoda LoadDictionary ve třídě Languages

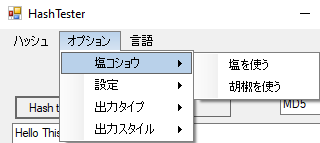
Přiřazení dat do slovníku je akorát čtení dat ze souboru, který je už předpřipraven. Jediná nevýhoda je, že všechny data jsou uložené v paměti, ale když budeme brát v úvahu, že celý soubor pro češtinu zabírá akorát 14KB místa, tak si nemyslím, že to bude dělat problém i na starších zařízeních. Druhá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



Výpis 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 Příklad stip menu v japonském jazyce



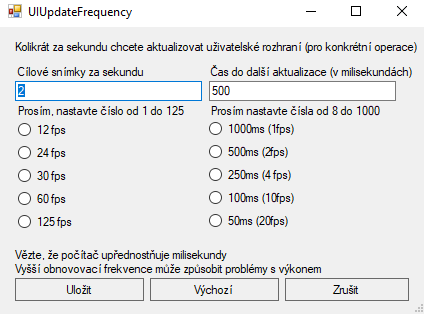
Výpis Japonský překlad formuláře



Výpis Český překlad formuláře

## 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 Formulář pro nastavení frekvence UI

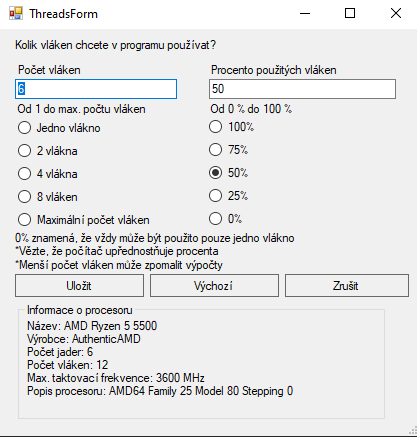
Formulář obsahuje 2 části, obě nastavují jednu věc. Nalevo je cílová snímková frekvence (FPS – snímky za sekundu) a vpravo čas do další aktualizace. Když se změní hodnota vlevo, nastaví se i hodnota vpravo, protože obě znamenají stejnou věc. FPS pracuje na jednotkách Hz, zatímco čas na periodě. Proto se dají tak jednoduše zaměnit. Pod textboxy se nachází několik radiobuttonů pro rychlé nastavení hodnoty. Ani jedna hodnota není stejná. Do *Settings* je ukládaná 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.



Výpis Příklad použití časovače a periody

## Nastavení CPU a vláken formulář

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



Obrázek 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.



Výpis Příklad použití proměnné v programu

## 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.



Výpis 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.



Výpis Část kódu pro hashování více hashovacích funkcí ve třídě Hasher

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

# Dodatečné funkce programu

## 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.

### 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.

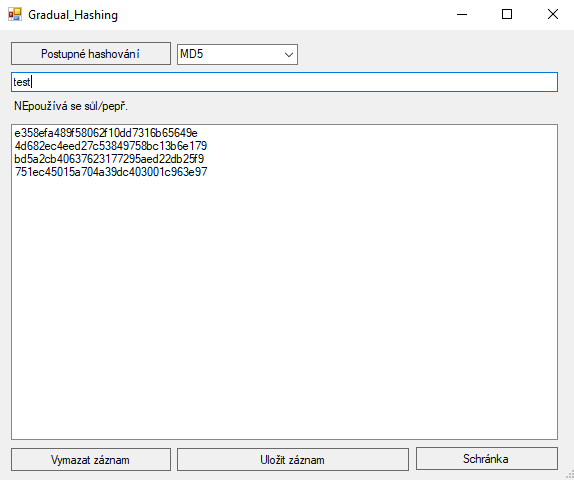


Výpis Metoda GradualHashing ve třídě Hasher

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.

### Formulář pro postupné hashování

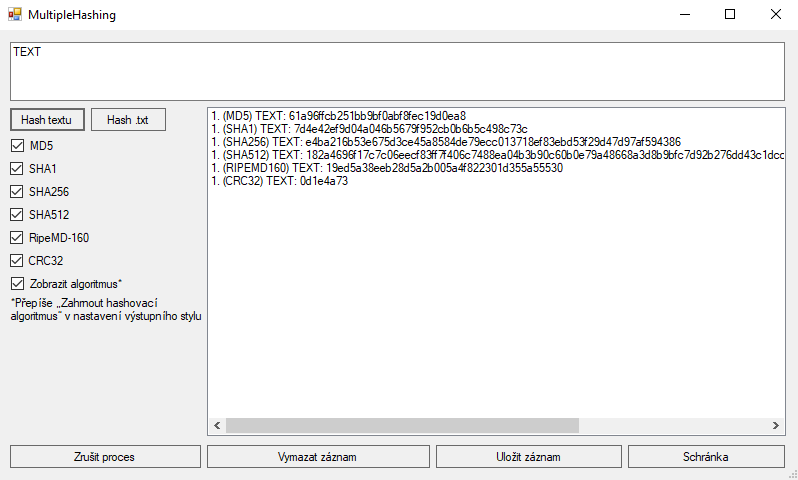
Formulář je velice jednoduchý, většina obrazovky je komponenta listbox pro záznam. Kvůli výukovým použitím se nepoužívá sůl/pepř i přes nastavení uživatele.



Obrázek Formulář postupného hashování

## 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.



Obrázek Formulář pro vícenásobné hashování

## 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.

### 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.



Výpis 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 vrací zpátky, kde jsou metodou *IsUsingSaltAndPepper* zpracovány a uloženy.



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



Výpis Metoda CheckPepper ve třídě Hasher

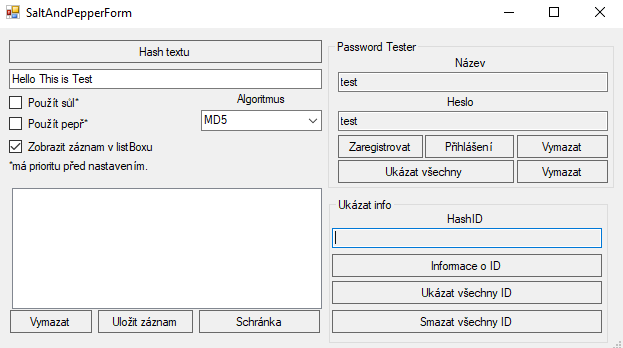
Hledání pepře je poměrně těžké, ale je to výhodné pro maximální bezpečnost. Musíme projít všechny možné kombinace a postupně porovnávat vygenerovaný hash s cílovým hashem. Pokud nalezneme shodu, znamená to, že jsme našli odpovídající vstupní text. Pokud žádná kombinace neodpovídá, originální text a hash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

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

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

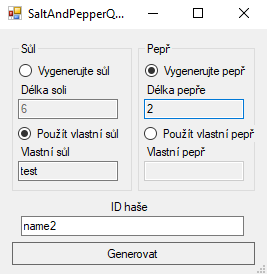
### Formulář pro práci se solí/pepřem

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



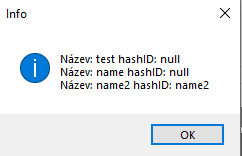
Obrázek Formulář pro práci se solí a pepřem

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.



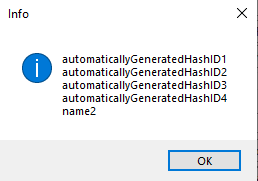
Obrázek Formulář pro zadání soli a pepře

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 zadané HashID, data se neuloží.



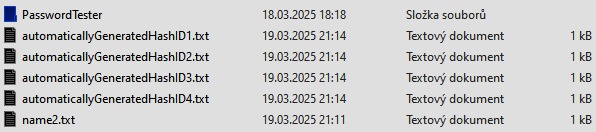
Obrázek Výstup tlačítka Ukázat všechny uživatele

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



Obrázek Výstup tlačítka Ukázat všechny ID

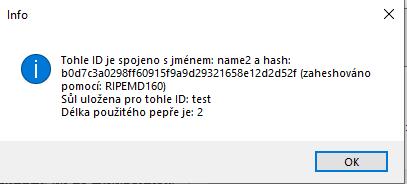
Tlačítko „ukázat všechny ID“ ukáže všechny ID ve složce HashData.



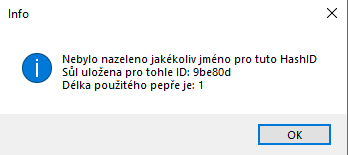
Obrázek Složka HashData



Výpis Data v souboru name2.txt



Obrázek Výstup tlačítka Informace o ID



Obrázek Výstup tlačítka Informace o ID bez uživatele



Výpis Metoda ShowIDInfo

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

## Kontrolní součet

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

### Kód pro kontrolní součet

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



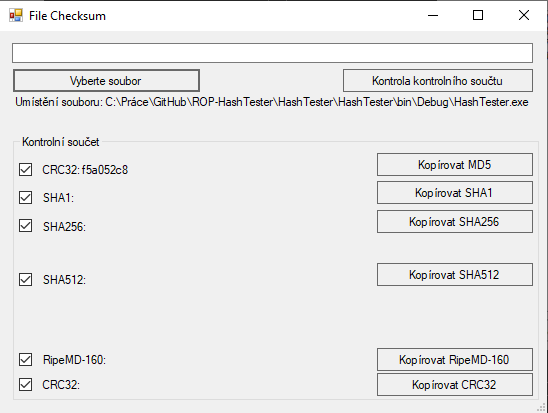
Výpis Metoda FileChecksum ve třídě Hasher

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.

### 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 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.



Výpis Velice jednoduchý rozhodovací proces hashovací funkce podle délky hashe

## 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.

### Kód



Výpis 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.



Výpis 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.



Výpis 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.



Výpis Příklad vygenerovaného textového souboru

Další část kolizí je metoda *CheckCollision*, která vezme uživatelovi vstupy a zkontroluje, jestli se po zaheshování 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.

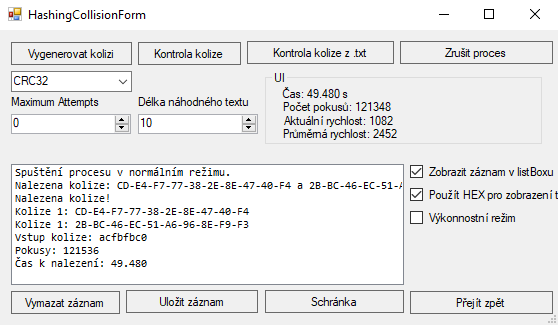


Výpis Část metody pro zpracování textového souboru

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.

### Formulář pro hledání kolizí

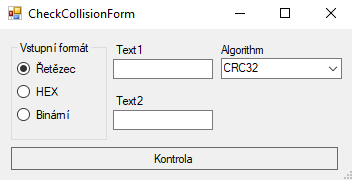
Formulář vyhledává kolizi dvou různých vstupů se stejným výstupem.



Obrázek Formulář pro generování kolizí

Formulář má 3 hlavní tlačítka, *Vygenerovat kolizi*, *Kontrola kolize* a *Kontrola kolize z .txt. \*\*\** 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 Formulář pro kontrolu kolize

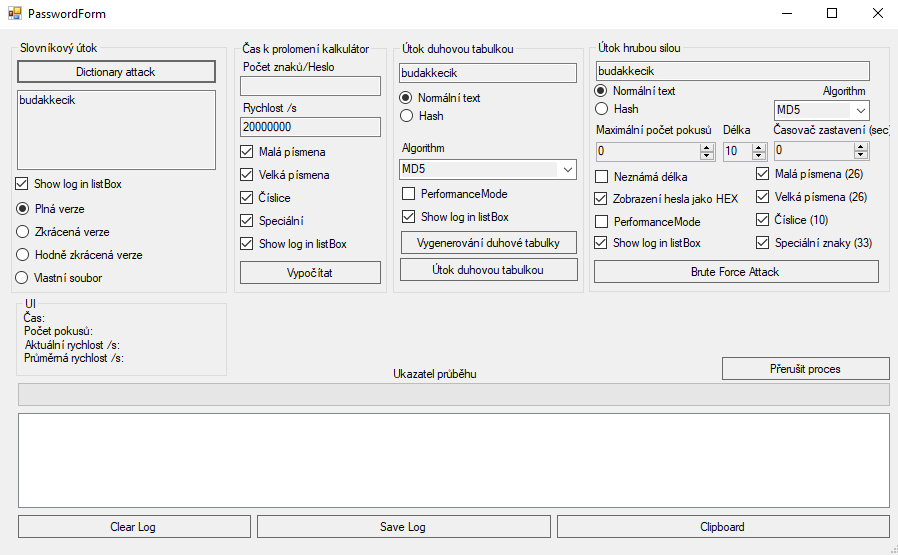
Kontrola kolize spustí další formulář, kam uživatel zadá vstup ve formátu UTF-8, hexadecimální číslo nebo binární číslo. Ve složce *Collisions* je pár souborů s kolizemi, které se dají použít.



Výpis Textový soubor CRC32-Prefab.txt ve složce Collisions

# Útoky na hashe

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



Obrázek 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.

## Slovníkový útok

### 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.



Výpis Vlastnost Progress ve třídě DictionaryAttack

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



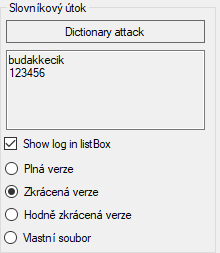
Výpis Metoda PasswordFinder ve třídě DictionaryAttack

*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 přečíst si řádek a zkontrolovat se všemi hledanými hesly, jestli nejsou stejné. Pokud jsou, zapíše se řádek do proměnné *lineFoundMatch* a *foundMatch* se dá na true. Jestli jsou všechny prvky ve *foundMatch* nastavené na true, našli jsme všechny hesla a algoritmus se může předčasně ukončit, jinak běží do konce souboru. Celá metoda je zanořena v try a catch.



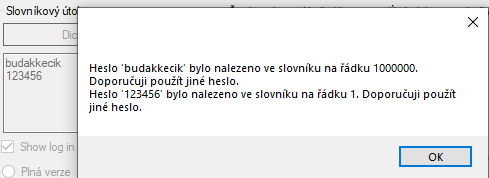
Výpis Zkouška, zda v poli byly najité všechny vstupy

### Formulář pro slovníkový útok



Obrázek Groupbox Slovníkový útok

První z leva 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 stejné 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 Ukázka slovníkového útoku s více vstupy



Obrázek Ukázka výstupu slovníkového útoku s více vstupy v záznamu

Slovníkový útok podporuje více vstupů zároveň.

## Výpočet prolomení hesla

### Kód

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



Výpis 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. [37]

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



Výpis 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.

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

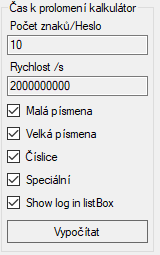
Vzorec pro vypočítání celkové doby trvání neboli vyzkoušení všech možných kombinací. Kdybychom dělali útok hrubou silou, tak doba trvání je nejdelší čas, reálná doba trvání může být od instantní po dobu trvání. Většinou se může počítat s poloviční dobou doby trvání.



Výpis Metoda Output ve třídě PasswordStrenghtCalculator

Metoda Output akorát vypíše dobu trvání v představitelné časové době.

### Formulář Čas k prolomení kalkulátor



Obrázek Groupbox Čas k prolomení kálkulátor

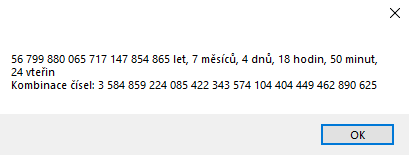
Kalkulátor či kalkulačka simuluje čas potřeba pro prolomení hesla při útoku hrubou silou. Kalkulačka funguje velice jednoduše na už zmíněném vzorci. První textbox podporuje vstup celočíselného nezáporného čísla reprezentující délku hesla, nebo přímo samotné heslo.



Výpis 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 Příklad výstupu při délce hesla 20 a všech 95 znaků

## 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.

### Generování duhové tabulky

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



Výpis Hlavička metody GenerateRainbowTable ve třídě RainbowTableGenerator

Potřebujeme cestu k slovníku, výstupní cestu a hashovací funkci.



Výpis Tělo metody GenerateRainbowTable ve třídě RainbowTableGenerator

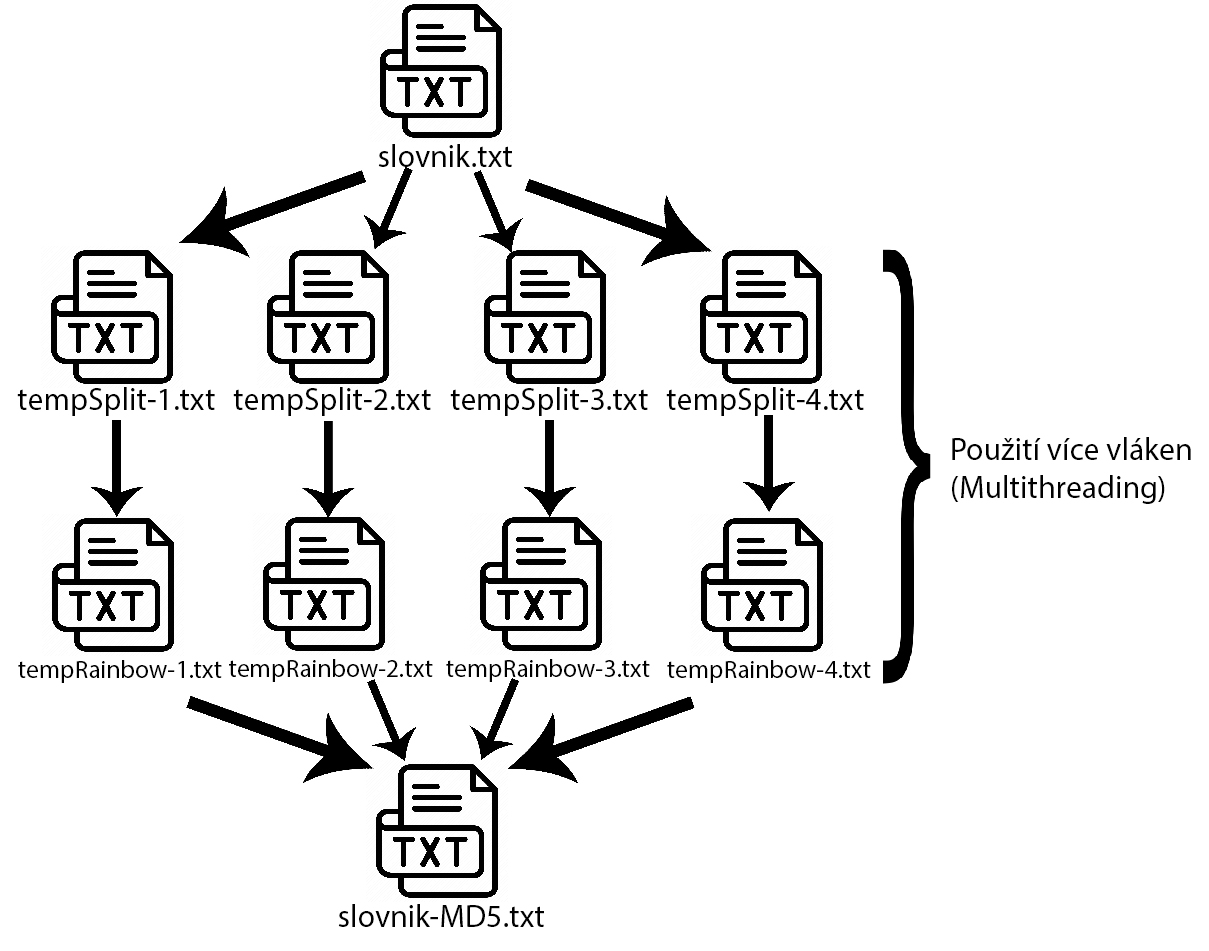
Pomocí tříd *StreamWriter* a *StreamReader* zpracováváme po řádcích slovník a zapisujeme. Jako první řádek duhové tabulky je vždy jakou hashovací funkci používáme. Funkce je rozdělena pomocí dvou rovná se (==) a data pomocí jednoho (=).



Výpis Příklad začátku duhové tabulky v textovém souboru

### 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 Fungování metody GenerateRainbowTableMultiThread pro 4 vlákna

#### Metody pro mutlithreading



Výpis Metoda GenerateRainbowTableMultiThreadForSingleThread

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



Výpis 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.

#### Mazání dočasných souborů

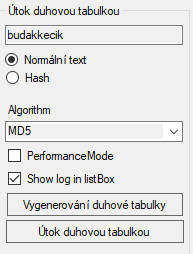
Dočasné soubory jsou po dokončení generace vymazány, pokud nedošlo při generování k výjimce či ukončení procesu uživatelem.



Výpis 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.

### Formulář pro duhovou tabulku



Obrázek Groupbox Útok duhovou tabulkou

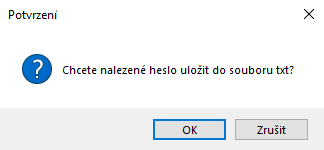
Do textboxu se dává hledaný vstup ve formě textu nebo hashe, záleží na nastavení radiobuttonu. Pokud je nastaven radiobutton *Text*, vstup v textboxu bude zaheshován podle vybraného algoritmu. Checkbox *Performance Mode* nastavuje použití multithreadingu. Checkbox *Show log in listBox* nastavuje ukládání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

#### Generování duhové tabulky

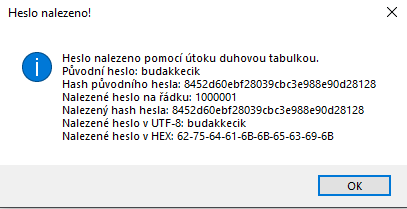
Tlačítko „Vygenerovat duhovou tabulku“ otevře *OpenFileDialog* pro vybrání slovníku a *SaveFileDialog* pro uložení duhové tabulky. Oba dva dialogy musí být úspěšné. Poté se zapíná metoda *GenerateRainbowTable* či *GenerateRainbowTableMultiThread* ze třídy *RainbowTableGenerator*.

#### Útok duhovou tabulkou

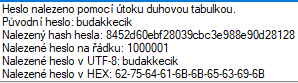
Tlačítko „Útok duhovou tabulkou“ otevře *OpenFileDialog* pro vybrání duhové tabulky a poté spustí metodu *PerformRainbowAttack* ze třídy *RainbowTableAttack*.



Obrázek Dotaz pro výstup nalezené shody při útoku duhovou tabulkou



Obrázek Výstup nalezené shody při útoku duhovou tabulkou



Obrázek Výstup nalezené shody při útoku duhovou tabulkou v záznamu

## Útok hrubou silou

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



Výpis Metoda SelectAllUsableChars ve třídě BruteForceAttack

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



Výpis 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 ).



Výpis 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.



Výpis 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.



Výpis Metoda GenerateText ve třídě BruteForceAttack

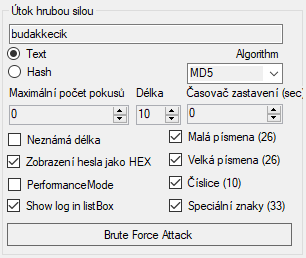
Metoda *GenerateText* si v hlavičce vezme index, všechny hledané znaky a délku a pomocí těchto dat vygeneruje text, který se poté zahashuje a vyzkouší. Příklad pro index 1, všechny možné znaky a délku 5 se vrátí *aaaaa*. Pro index 2 je *aaaab*, pro index 96 bude *aaaba* atd.



Výpis Příklad použití metody GeneratText

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

### Formulář pro útok hrubou silou



Obrázek 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.



Výpis 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*.

# 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).

## Vyhledávání kolizí

Průměrná rychlost za 3 minuty.

Čas na najití kolize v CRC32. Průměrný čas je spočítán průměrem 10 pokusů. UI bylo nastaveno na 30fps.

## Generování duhové tabulky

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

Další vlákna jsou o maličko rychlejší kvůli vytváření temp souborů, což dělá jedno jádro.

Průměrná rychlost je prakticky stejná jako aktuální rychlost,

## Útok hrubou silou

Závěr

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

Seznam použitých zdrojů

Seznam použitých symbolů a zkratek

|  |
| --- |
|  |

Seznam obrázků

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

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

Seznam tabulek

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

Seznam příloh

Prázdná šablona maturitní práce

1. RIPE je zkratka RACE Integrity Primitives Evaluation, 1988-1992 [↑](#footnote-ref-1)
2. Realizace kódu je dostupná později v praktické části [↑](#footnote-ref-2)
3. BCrypt je další hashovací funkce, která je ovšem značně pomalá. V roce vydání (1976) dokázala funkce zaheshovat skoro 4 hesla za sekundu. Vytvořena Nielsem Provosem a Davidem Mazièresem. [19] [↑](#footnote-ref-3)
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. [20] [↑](#footnote-ref-4)
5. (Windows: <https://github.com/byt3bl33d3r/pth-toolkit>) (Kali: [https://www.kali.org/tools/passing-the-hash/).](https://www.kali.org/tools/passing-the-hash/).%20) [↑](#footnote-ref-5)
6. Multithreading v překladu více vláknové řízení či použití více vláken [↑](#endnote-ref-1)
7. Nejdůležitější funkce pro mě jako studenta, protože pracuji jak z domu, tak ze školy [↑](#footnote-ref-6)
8. CLI je zkratka pro Command Line Interface (rozhraní příkazového řádku) [↑](#endnote-ref-2)
9. GUI je zkratka pro Graphical User Interface (grafické uživatelské rozhraní) [↑](#endnote-ref-3)
10. SaaS je zkratka pro Software as a Servise (software jako služba**)** [↑](#endnote-ref-4)
11. Kanban (看板) z japonštiny znamená cedule či tabule [↑](#footnote-ref-7)
12. Just In Time (právě v čas) „umožňuje podniku vyrábět výrobky v určeném množství a určeném čase dle požadavků zákazníka.“ [28] [↑](#footnote-ref-8)
13. To-Do list znamená list věcí, které se ještě musí udělat. [↑](#endnote-ref-5)
14. Uniform Resource Locator (URL) znamená jednoznačné určení zdroje a používají to webové stránky jakožto doménovou adresu (příklad je seznam.cz) [↑](#endnote-ref-6)
15. RGB znamená Red Green a Blue, což je formát pro nastavení barvy. Každá barva má hodnotu od 0 do 255. 0 ==> nesvítí a 255 ==> svítí nejvíce. Černá barva jsou všechny hodnoty na 0 a bílá všechny hodnoty na 255. [↑](#endnote-ref-7)
16. AI, artificial inteligence, je počítačově simulovaný algoritmus založený na neuronech a strojovém učení. [↑](#endnote-ref-8)