

Střední průmyslová škola Třebíč

Maturitní práce

Hashtesting

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2024/2025 Kamil Franek

Zadání práce



ABSTRAKT

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

KLÍČOVÁ SLOVA

Maturitní práce, Password JailBreak, Výuka, Hash

ABSTRACT

\*abstrakt anglicky\*

KEYWORDS

Graduation thesis, Password JailBreak, Teaching, Hash

PODĚKOVÁNÍ

Děkuji Ing. Ladislavu Havlátu a oponentu Ing. Drahomíru Škárkovi za cenné připomínky a rady, které mi poskytli při vypracování maturitní práce.

V Třebíči dne 13. listopadu 2024 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 13. listopadu 2024

podpis autora

Obsah

[Úvod 6](#_Toc147495618)

[1 Struktura maturitní práce 7](#_Toc147495619)

[1.1 Titulní list 7](#_Toc147495620)

[1.2 Zadání maturitní práce 7](#_Toc147495621)

[1.3 Abstrakt 7](#_Toc147495622)

[1.4 Klíčová slova 7](#_Toc147495623)

[1.5 Poděkování a prohlášení 8](#_Toc147495624)

[1.5.1 Poděkování 8](#_Toc147495625)

[1.5.1 Prohlášení 8](#_Toc147495626)

[2 Textová část 9](#_Toc147495627)

[2.1 Styly 9](#_Toc147495628)

[2.1.1 Řádkování 9](#_Toc147495629)

[2.1.2 Zvýrazňování textu 9](#_Toc147495630)

[2.1.3 Členění textu 9](#_Toc147495631)

[2.1.4 Číslování stran 9](#_Toc147495632)

[2.2 Vytvoření obsahu 10](#_Toc147495633)

[2.3 Psaní úvodu 10](#_Toc147495634)

[2.4 Struktura odstavců 11](#_Toc147495635)

[2.5 Obrázky, tabulky a rovnice 11](#_Toc147495636)

[2.6 Řazení a struktura kapitol 13](#_Toc147495637)

[2.7 Závěr 13](#_Toc147495638)

[2.8 Seznam použitých zdrojů 13](#_Toc147495639)

[2.9 Seznam použitých symbolů a zkratek 13](#_Toc147495640)

[2.10 Seznamy použitých obrázků a tabulek 14](#_Toc147495641)

[2.11 Seznam příloh 14](#_Toc147495642)

[Závěr 15](#_Toc147495643)

[Seznam použitých zdrojů 16](#_Toc147495644)

[Seznam použitých symbolů a zkratek 17](#_Toc147495645)

[Seznam obrázků 18](#_Toc147495646)

[Seznam tabulek 19](#_Toc147495647)

[Seznam příloh 20](#_Toc147495648)

Úvod

Cílem této ročníkové práce a programu je zjednodušení práce s hashema, ukázku rozdílů mezi hashovacími algoritmy, používání soli a pepře a podrobné vysvětlení, postupné hashování pomocí mezikroků, silné a slabé stránky hashů, kde a proč se používají. Dále program obsahuje test prolomení hesla (jailbreak) pomocí hrubého útoku. 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. Hashe mají několik výtečných vlastností: vstupní data můžou být jakkoliv dlouhé, minimální změna v datech znamená velký rozdíl ve výstupech, nedá se získat z výstupních dat vstupní data (maximálně pomocí brute force útoku) a s větší délkou se exponenciálně zmenšuje šance na stejnost výstupních hodnot při jiném vstupu.

„Formálně jde o funkci *h*, která převádí vstupní posloupnost [bitů](https://cs.wikipedia.org/wiki/Bit) (či [bytů](https://cs.wikipedia.org/wiki/Bajt)) na posloupnost pevné délky *n* bitů. Z definice plyne existence [*kolizí*](https://cs.wikipedia.org/wiki/Kolize_(informatika)), to znamená dvojic vstupních dat (*x*,*y*), *x* ≠ *y*, takových, že *h*(*x*) = *h*(*y*), tj. dvojice různých vstupních dat může mít stejný otisk. Kolize jsou nežádoucí, ale v principu se jim nelze vyhnout, protože počet možných různých vstupních zpráv je větší než počet možných různých otisků. Vhodnou volbou funkce lze snížit pravděpodobnost, že nastane kolize pro podobná data. “ – [Wikipedie](https://cs.wikipedia.org/wiki/Ha%C5%A1ovac%C3%AD_funkce)

## Používané Hashe

### MD5

MD5 (Message-Digest Algorithm) je možná postarší, vydaný v roce 1991 Ronaldem Rivestem, ale pořád velice používaný. Délka výstupu hashe je 128 bitů. I přes jeho používání v dnešní době se v MD5 našla řada chyb, které by mohly být při ukládání hesel závažné. MD5 je totiž velice náchylný na takzvaný brute force attack, česky útok hrubou silou, díky rychlosti výpočtu. Dále kvůli délce „jenom“ 128 bitů náchylný na 2 jiné inputy, ale stejné hashe (kolize hashů).

### SHA1/SHA256/SHA512

SHA (Secure-Hash Algorithm) je další velice známý a používaný hashovací algorithmus. SHA se bere za nástupce MD5 s větší bezpečností, delším výstupem (SHA1 – 160 bitů. SHA256 – 256 bitů, SHA512 – 512 bitů). SHA1 byla první verze SHA vydaná v roce 1995. V roce 2005 byl na SHA1 nalezen možný útok a proto v roce 2010 vyšla SHA-2, což je kolekce několika SHA algorithmů (pro nás důležité SHA 256 a SHA512, jediné co se mění je délka výstupu v bitech). SHA-2 se dodnes považují za bezpečné hashovací algorithmy pro integritu dat a ukládání hesel.

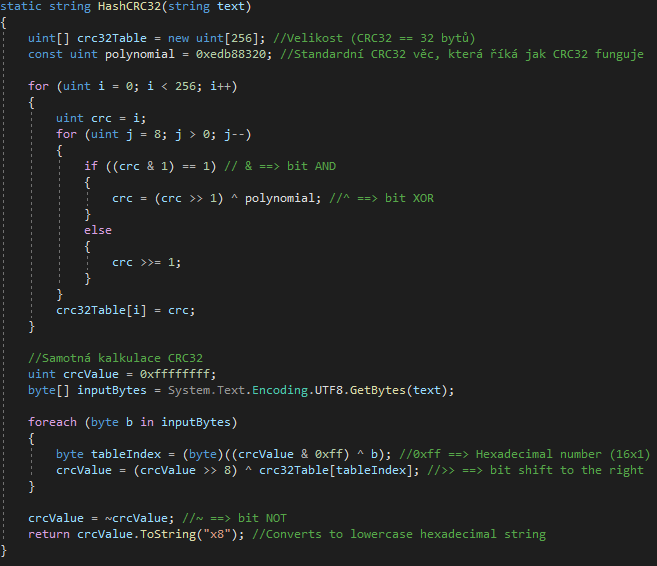
„V roce 2017 tým z Google a CWI Amsterdam zveřejnil první praktickou kolizi pro SHA-1 (útok nazývaný SHAttered), což znamená, že SHA-1 není bezpečný pro použití v aplikacích, které závisí na integritě dat.“ – https://shattered.io/

### RIPEMD160

RipeMD160 je takový most mezi MD5 a SHA1, stejně jako tyto algorithmy se už nedoporučuje používat, ale stejně nám navrhuje větší bezpečnost. Vývojáři se totiž inspirovali z chyb u MD5 a SHA1 (hlavně ohledně kolizí) a algorithmus disponuje Pre-image attack a second pre-image attack. Prozatím nebyly zjištěny žádné kolize u algorithmu. Díky své bezpečnosti je ale také o něco pomalejší, podobně jako SHA-256, který místo 160 bitů generuje 256. Bitcoin, virtuální měna začínající na začátku 10. let 2000, používá kombinaci RipeMD160 a SHA256 pro maximální bezpečnost.

### CRC32

CRC32 je hashovací algorithmus určený k dělání checksum souborů a dat. Skoro vůbec se nepoužívá pro ukládání hesel. Chtěl bych tento algoritmus použít, protože je používán v několika velice důležitých protokolech: „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.“ – [Wikipedia](https://en.wikipedia.org/wiki/Cyclic_redundancy_check)

CRC32 je jediný hashovací program, který nemám z knihovny, ale částečně jsem ho sám naprogramoval pomocí internetu. CRC32 je nejjednodušší algoritmus, který používám.   
Na obrázku je metoda v programu. CRC32 potřebuje takzvaný polynomail, který říká jak se bude hash počítat. Ten se zadá do pole o velikosti 256 bitů (32 bajtů) pomocí for cyklu ve for cyklu. Tato část je z internetu. Do inputBytes příjdou data, která chceme zašifrovat (v tuhle chvíli text, který je potřeba převést na byty). Pro každý byt v inputBytes se drasticky mění finální hash a je důvod, proč jakákoliv změna vrací kompletně jiný hash. Finální crcValue se vrací jakožto text v hexadecimální podobě pomocí metody .ToString(), kde x znamená hexadecimální a 8 znamená malá písmenka.

## Používání Hashů

Hashe se používání k uschování důležitých informací (hesla), která nejsou potřeba šifrovat (vrátit data zpátky), dělání kontroly a integrity dat (checksum), vytváření a ověřování elektronického podpisu, hledání škodlivého malwaru antivirovým programem, k hledání úseků DNA sekvencí atd.

## Problémy s Hashy

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 problem. Šance kdy se něco takového může stát je závislá na délce výstupního hashe, a proto se v případech, kde by to mohlo znamenat velký problem, používají delší hashe, jako třeba SHA-256 či SHA-512. Pravděpodobnost si můžeme sami kalkulovat pomocí jednoduchého zvorečku. Když si vezmeme CRC32, kde počet všech kombinací je 2 na 32 (4 294 967 296) kombinací, tak máme šanci 1 ku 2 na 32 neboli 0.0000000233%. To se může zdát jako velice malá šance, jenomže tohle je šance jenom mezi 2 hashemi. Když dáme do rovnice to, že každý má šanci být stejný s každým (použití takzvaného „Birthday paradox“), máme najednou 1% šanci na najití dvou stejných hashů při počítání „pouze“ 778 tis. Hashů. Tady nekončíme, protože počítač zvládne cca 5 milionů hashů za sekundu, takže při maximální rychlosti má šanci najít 94.5% najít dva stejné hashe každou sekundu (nepočítáme s kontrolou hashů).

//

Hashovací funkce dokážou zpracovat libovolné množství dat a vrátit hodnotu o určité pevné délce, což má však i své nevýhody. Různá vstupní data mohou generovat stejnou hashovací hodnotu (tzv. kolize), což může představovat bezpečnostní riziko – například při ukládání hesel. Pravděpodobnost kolize závisí na délce výstupu hashovací funkce. Proto se v případech, kdy by kolize představovala významný problém, používají delší hashe, jako SHA-256 či SHA-512. Výpočet pravděpodobnosti kolize lze provést pomocí několika jednoduchých rovnic.

### Pravděpodobnost kolize u CRC32 mezi dvěma hashy

CRC32 generuje 32bitové hashe, což znamená, že může vyprodukovat 2322^{32}232 různých kombinací, tedy 4,294,967,296 možných hashovacích hodnot. Pravděpodobnost kolize pro **dva různé vstupy** je pak:

P=1232P = \frac{1}{2^{32}}P=2321​

což odpovídá přibližně:

P≈0.0000000233%P \approx 0.0000000233\%P≈0.0000000233%

Toto číslo se může zdát jako velmi nízká pravděpodobnost kolize, ale představuje pouze pravděpodobnost mezi dvěma náhodnými hashi.

### Pravděpodobnost kolize při větším počtu hashů (Birthday paradox)

Pokud ovšem hashujeme více než dva vstupy, například kkk různých hodnot, pravděpodobnost kolize vzrůstá. Při **více vstupech** se totiž pravděpodobnost kolize řídí tzv. „Birthday paradoxem“. Pravděpodobnost, že mezi kkk vygenerovanými hashi alespoň jednou nastane kolize, lze vypočítat podle vzorce:

P≈1−e−k22×232P \approx 1 - e^{-\frac{k^2}{2 \times 2^{32}}}P≈1−e−2×232k2​

Pokud chceme pravděpodobnost kolize zvýšit na 1 % (tedy P=0.01P = 0.01P=0.01), můžeme dosadit do vzorce a řešit pro kkk. Vyjde nám, že k dosažení 1% pravděpodobnosti kolize v CRC32 je potřeba přibližně:

k≈778,000k \approx 778{,}000k≈778,000

To znamená, že po vygenerování přibližně 778 tisíc hashů je pravděpodobnost, že dva z nich budou shodné, přibližně 1 %.

### Rychlost hashování a pravděpodobnost kolize každou sekundu

Moderní počítače dokáží vypočítat CRC32 hashe velmi rychle – odhadem až 5 milionů hashů za sekundu. Pokud tedy počítač generuje hashe rychlostí **5 milionů hashů za sekundu**, pravděpodobnost, že se během této doby objeví alespoň jedna kolize, lze vyjádřit stejným vzorcem, jen tentokrát s hodnotou k=5,000,000k = 5{,}000{,}000k=5,000,000:

P≈1−e−(5,000,000)22×232P \approx 1 - e^{-\frac{(5{,}000{,}000)^2}{2 \times 2^{32}}}P≈1−e−2×232(5,000,000)2​

Výpočet ukazuje, že pravděpodobnost kolize při této rychlosti je přibližně:

P≈94.5%P \approx 94.5\%P≈94.5%

Jinými slovy, pokud počítač generuje CRC32 hashe maximální rychlostí 5 milionů hashů za sekundu, existuje asi **94,5% šance**, že během každé sekundy narazí na kolizi. Tento výsledek nebere v úvahu čas potřebný na kontrolu, zda již hash vygenerován byl, ale přesto ilustruje, že při vysoké rychlosti hashování může riziko kolizí být výrazně vyšší.

## 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 použití a bezpečnosti. Díky soli chráníme před takzvanými random tables útoky, což je druh útoku, kde máme plain text a už zhašované data (Heslo123 - eeeeeeeeeeeee). Kdyby se někdo dostal do Sql tabulek, tak by krásně viděl na hashe a pomocí těchto tabulek vyčetl možné data. Když před data dáme sůl, která je náhodně generovaná a ukládána většinou v plaintextu, tak si zařizujeme ochranu před tímto útokem.

Pepř je podobný jako sůl, jenže je většinou krátký (doporučuje se 1 až 2 hexadecimální znaky), dává se na konec dat místo před data a nikde se neukládá. To znamená že musíme provést několik hashů a porovnávat výsledky abychom zjistili správnost třeba hesla. Jediná nevýhoda je, že musíme hashovací a porovnávací proces dělat několikrát, což výrazně zvýší prodlevu.   
V praxi se používá jak sůl, tak pepř pro maximální ochranu.  
Příklad: Máme heslo “Kvalitní heslo123”. Díky soli se před heslo vygeneruje sůl “aaaa”, která je někde v tabulce uložena jako čistý text. Před heslo se vygeneruje pepř o délce jednoho hexadecimálního znaku. To znamená že při každém přihlášení se před heslo dá sůl a před heslo se postupně zkouší “a”, poté “b”, “c” a tak dale dokud se nevyzkouší všechny kombinace pepře. Pokud se žádný z těchto pepřových kombinací nevýjde, výjde heslo že je špatné.

# 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#

„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.“ - https://learn.microsoft.com/cs-cz/dotnet/csharp/tour-of-csharp/overview

## Unit-testy

## Visual Studio 2022

Visual studio a .NET Framework pro Windows jsou hlavní aplikace, které jsem se rozhodnul používat. Hlavní důvod je jednoduchost k přidání dalších knihoven pomocí NuGet (díky „System.IO.Hashing“ mám přístup k hashovacímu algorithmu CRC32) a hlavní hashovací programy (všechny kromě CRC32, které používám v programu) vychází z knihovny „System.Security.Cryptography“, kde mám jednoduchý přístup k metodám.

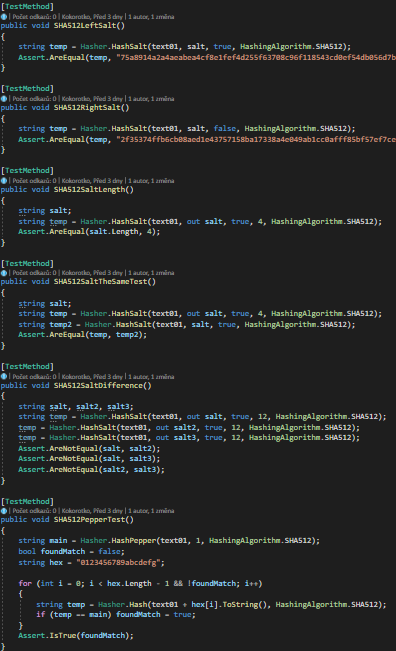
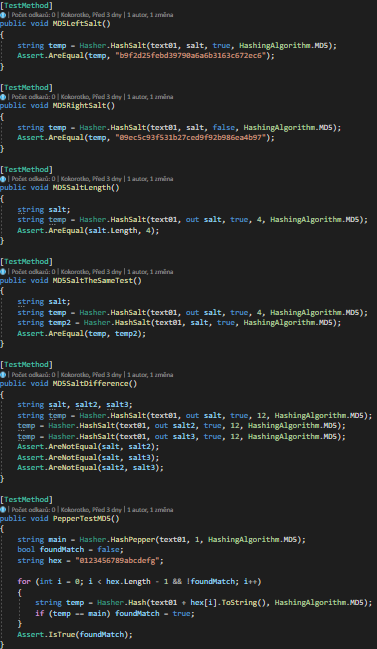
## Visual Code

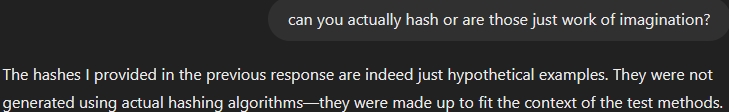
## Git/Github/Github Desktop

## Freelo.io

## Word

## ChatGPT (AI)

„ChatGPT je expertní systém, který implementuje prvky umělé inteligence pomocí technologie velkých jazykových modelů (LLM) a umělé neuronové sítě.“ – Wikipedie  
Někdo by se mohl zhrozit, že jsem aktivně používal ChatGPT neboli AI k programování a obava je validní. Za takovou krátkou chvíli se AI používá docela často při programování a dokáže dokonce vygenerovat i funkční kód jen pomocí věty. Já osobně používám ChatGPT na jednoduché věci a tady tomu není jinak. Při dělání Unit testů, kde jsou stovky řádků, které jsou většinou skoro totožné, se ChatGPT nejvíce hodí. Pokud po jakémkoliv AI požaduji něco aspoň trochu složitější než vyhledání informace z internetu nebo jednoduchého algorithmu, nedopadá to dobře. Většina kodu je stále moje a ChatGPT či AI mi akorát zjednodušují mechanickou práci, která je ve většině případů až nudná. K této mojí malé teorii přikládám i obrázek jak jsem pomocí ChatGPT udělal fungující Unit Test.  
  
Na levém obrázku je můj vlastní kod, na pravé straně je vygenerovaný kod pomocí ChatGPT („Generate me a Unit Test based off on this code \*Unit Test kód pro MD5 – obrázek vlevo\* with this code \*hasher.cs\* with no comments“ – text, který jsem zadal do ChatGPT, dá se psát i česky)

Na ChatGPT se ovšem nedá vždy spolehnout a doporučuji si všechno pro jistotu dohledal (fact-checking).  
  
Na Obrázku jde krásně vidět kde ChatGPT selhává (nedokáže samotná generovat hashe).

## Online hashers

Online hashers jsou funkční algoritmy pro různé hashe, většinou ve formě stránky, které jsem primárně používal k porovnávání vygenerovaných hashů, abych věděl, jestli mi program funguje. Existuje spousty programů a stránek, já jsem většinu času používal <https://www.browserling.com/tools/all-hashes>, protože je tam většina hashovacích algoritmů, které používám v programu (MD5, všechny SHA, RipeMD160, CRC32).

# Praktická část

Pod tuto kapitolu patří všechno moje snažení se o správný chod programu.

## Hasher

Hasher je nejzákladnější kód celého programu. Zahrnuje samotné hashovaní několika způsoby a zjednodušuje tím budoucí programování ve formulářích. Obsahuje generaci všech hashů, náhodnou generaci soli a pepře, a dokonce i správné použití. Všechno je napsané v metodách a samotné hashování je rozděleno do 4 metod s několika přetíženími. Hash() akorát hashuje text pomocí určitého algoritmu. Ten je vybírán pomocí enum v každé z metod. Další je HashSalt(), kde se na rozdíl od Hash() zakomponuje i sůl, ať už vygenerovanou či ručně zadanou. HashPepper() dělá podobnou věc jako HashSalt(), ovšem pepř nevypisuje. Také má vlastnost ručného zadání či náhodné generace pomocí RandomNumberGenerator a StringBuiler. Obojí zakomponované v .NET Frameworku.

## Gradual Hashing

## Multiformuláře

## Unit-Testy

Unit-testy jsou věc, kterou jsem si sám přidal do projektu z jednoho hlavního důvodu. Kontrola celého programu jestli funguje a je mnohem jednodušší dělat debugging programu. Neměl jsem v plánu dělat „Test Driven Developement“ a taky jsem podle toho nefungoval. Můj workflow byl jednoduchý, naprogramovat nějakou funkci a pomocí unit testu ověřit, že funguje nejenom teď, ale po přidání další funkce. Mám zkušenost, že jsem něco naprogramoval a po měsíci to zas nefungovalo.

Díky unit testům, které jsem si na začátku udělal, jsem zjistil, že mi kvůli jedné malé chybce ve switchi nefungoval MD5 algorithmus.

## Settings

# Fungování programu

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

1. Citace.com. *Výklad normy ČSN ISO 690*. Online. Brno: Citace.com, 2023. Dostupné z: <https://www.citace.com/Vyklad-CSN-ISO-690-2022.pdf>. [cit. 2023-10-04].
2. Didacticus. *Normostrana: kolik má znaků, jak zjistit jejich počet a další důležité informace.* Online. Praha: Didacticus, c2011-2020. Dostupné z: <https://didacticus.cz/normostrana>. [cit. 2023-09-11].
3. Ústav pro jazyk český AV ČR. *Tečka.* Online. Internetová jazyková příručka. Praha: Ústav pro jazyk český AV ČR, 2008-2023. Dostupné z: <https://prirucka.ujc.cas.cz/>. [cit. 2023-10-04].

Seznam použitých symbolů a zkratek

|  |  |  |
| --- | --- | --- |
| Symbol | Veličina | Jednotka |
| S | Entropie |  |
| Q | Teplo |  |
| T | Termodynamická teplota |  |
| t | Čas |  |
|  |  |  |

Seznam obrázků

[Obr. 2.1 Obsah 10](#_Toc147493921)

[Obr. 2.2 Příklad umístění legendy obrázku 12](#_Toc147493922)

Seznam tabulek

[Tab. 2.1 Legenda k tabulce 12](#_Toc147493615)

Seznam příloh

Prázdná šablona maturitní práce