

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

*Metódy útoku hrubou silou na TrueCrypt*  
Diplomová Práca

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

*Metódy útoku hrubou silou na TrueCrypt*  
Diplomová Práca

Študijný program: Informatika  
Študijný odbor: 2508 Informatika  
Školiace pracovisko: Katedra Informatiky  
Vedúci práce: RNDr. Richard Ostertág PhD.

## Podakovanie

# Abstrakt

Táto práca sa zaoberá ~~problémom~~ optimalizácie útokov hrubou silou. Rozoberá možnosti útoku na program TrueCrypt. ~~Taktiež sa~~ sústreďí na to ako využiť znalosti o často používaných heslách na vylepšenie efektívnosti útoku ~~hrubou silou~~. Hlavným cieľom práce je napísať program, ktorý sa snaží pomocou ~~algoritmu na~~ útok hrubou silou nájsť heslo pre dešifrovanie disku zašifrovaného pomocou programu TrueCrypt.

**KLÚČOVÉ SLOVÁ:** TrueCrypt, útoky hrubou silou, bezkontextové gramatiky, Markovov zdroj, používateľské heslá

# Abstract

Focus of this work is on optimizing brute-force attacks. It studies different methods to attack TrueCrypt. The main purpose of this thesis is to implement application that is able effectively recover password for encrypted disk with TrueCrypt

**KEYWORDS:** TrueCrypt, brute-force attacks, context-free grammars

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 TrueCrypt</b>	<b>2</b>
<b>2 Útoky hrubou silou</b>	<b>4</b>
2.1 Inkrementálny útok . . . . .	4
2.2 Slovníkový útok . . . . .	5
2.2.1 Prekrúcanie slov . . . . .	5
2.3 Hybridný útok . . . . .	5
2.4 SAT Solver . . . . .	6
<b>3 Používateľské heslá</b>	<b>7</b>
<b>4 Učenie</b>	<b>9</b>
4.1 Pravdepodobnostné bezkontextové gramatiky . . . . .	9
4.2 Markovovské zdroje . . . . .	10
<b>5 Implementácia</b>	<b>11</b>
5.1 Bezkontextová gramatika . . . . .	11
5.1.1 Tvorba gramatiky . . . . .	11
5.1.2 Počítanie pravdepodobností . . . . .	12
5.1.3 Generovanie hesiel . . . . .	14
5.2 Markovovský zdroj . . . . .	17
<b>6 Testy</b>	<b>20</b>
6.1 Časová náročnosť . . . . .	20
6.2 Výstupné heslá . . . . .	21
6.2.1 Heslá zo vstupného slovníka . . . . .	21
6.2.2 Miery presnosti . . . . .	25
6.2.3 Heslá z iného slovníka . . . . .	28
<b>7 Diskusia</b>	<b>29</b>
7.1 Možnosti zlepšenia nášho riešenia . . . . .	29

Obsah	Obsah
7.1.1	Izolovanie kódu na skúšanie kandidátov . . . . . 29
7.1.2	Velkosť potrebnej pamäte . . . . . 29
7.1.3	Kompletne generujúci Markovovský zdroj . . . . . 29
<b>8</b>	<b>Záver 30</b>
<b>Literatúra</b>	<b>31</b>

# Zoznam listingov

5.1	Úprava pravidiel na základe vstupného slova . . . . .	13
5.2	Pripočítanie výskytov k podmnožinám zložených neterminálov . . . . .	14
5.3	Generovanie všetkých susedných vektorov . . . . .	17



# Zoznam obrázkov

1.1	Schéma XTS algoritmu . . . . .	3
6.1	Počet vygenerovaných hesiel zo slovníka - Markovovský zdroj . . . . .	22
6.2	Počet vygenerovaných hesiel zo slovníka - Markovovský zdroj - Unikátne .	23
6.3	Počet vygenerovaných hesiel zo slovníka - Gramatika . . . . .	23
6.4	Počet vygenerovaných hesiel zo slovníka - dĺžka 6 . . . . .	24
6.5	Počet vygenerovaných hesiel zo slovníka - dĺžka 7 . . . . .	25
6.6	Počet vygenerovaných hesiel zo slovníka - dĺžka 8 . . . . .	26

# Úvod

V dnešnom svete fungujúcom na elektronických dátach, ktoré pre nás majú obrovskú cenu, sa ľudia snažia ~~udržať ich čo najviac v tajnosti~~. Tieto dáta sa dajú jednoducho ochrániť, ak k nim bude mať prístup len majiteľ. Toto avšak nie je najpoužiteľnejšie riešenie, keďže takmer každý počítač je pripojený do nejakej siete. Iná možnosť je mať ~~uložené tieto~~ dáta vo forme, ktorá bude dávať zmysel len ~~povereným~~ osobám, aj keď prístup k nim môžu mať aj iní ľudia. Na ~~toto~~ primárne slúži šifrovanie ~~pomocou kľúča~~.

Bezpečnosť ~~tohto~~ šifrovania závisí ~~vysoko~~ na utajení ~~tohto~~ kľúča. Preto je dôležité aby nebol ľahko ~~odhadnuteľný~~. Keďže počítače priniesli so sebou obrovskú výpočtovú silu, sú schopné robiť až niekoľko desiatok tisíc pokusov za sekundu snažiac sa uhádnuť tento kľúč [12]. Keďže rýchlosť tohto hľadania kľúča závisí hlavne od veľkosti prehľadávaného priestoru kľúčov, v praxi sa bežne používajú aspoň 256 bitov dlhé kľúče. Toto je ekvivalent 32 znakového používateľského hesla zloženého zo ~~z~~ ľubovoľných znakov ASCII tabuľky.

Takéto hľadanie kľúča sa nazýva útok hrubou silou. Jeho podstatou je postupné generovanie možných kľúčov a následne overenie ich správnosti. V tejto práci sa budeme venovať skúmaniu a implementácií algoritmov na generovanie týchto hesiel. Program na následne overenie správnosti týchto hesiel **nebudeme** implementovať, keďže jeho implementácia by mala obsahovať množstvo optimalizácií, ktoré **si** mimo **rozsah** tejto práce. Výstup našej práce bude program, ktorý bude generovať zoznam hesiel, ktoré sa dajú následne použiť v niektorom z voľne dostupných programov na **skúšanie** takýchto hesiel ako **napríklad** *hashCat*.



## TrueCrypt

TrueCrypt je šifrovací program poskytujúci používateľovi možnosť zašifrovať disk alebo jeho ľubovoľnú časť v počítači pomocou používateľom zvoleného hesla. Vývoj tohto programu bol ukončený v roku 2014 a podľa autorov nie je bezpečný, nakoľko jeho implementácia môže obsahovať bezpečnostné chyby. Následný bezpečnostný audit tohto programu neukázal žiadne závažné bezpečnostné chyby.

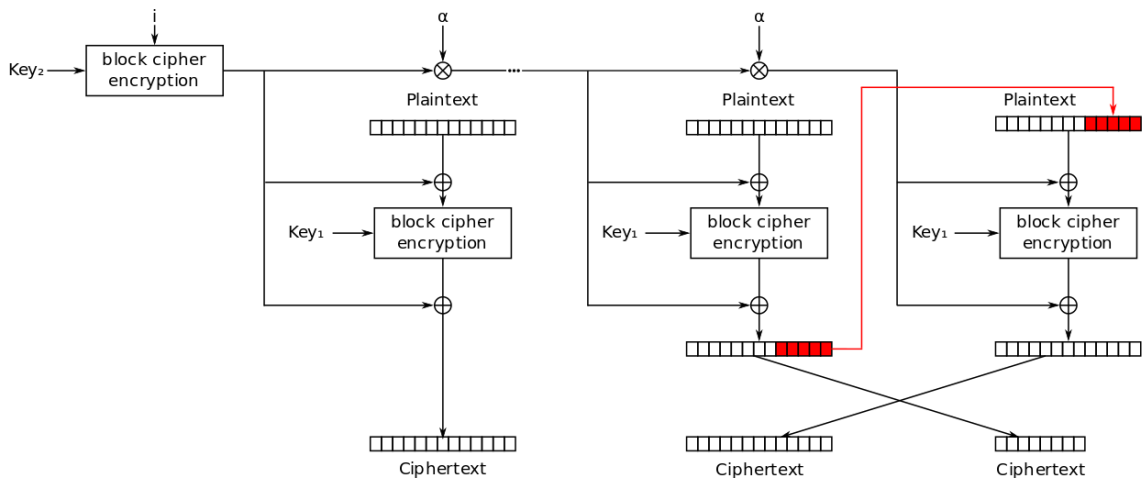
V septembri 2015 prišiel James Forshaw s informáciou o 2 chybách v programe TrueCrypt. Jedna z chýb umožňuje útočníkovi plný prístup k zašifrovaným partíciám iných používateľov, ktoré sú na tom istom počítači [5]. Druhá, závažnejšia chyba, umožňuje útočníkovi prístup k zvýšením právam zneužitím tvorby symbolického odkazu na písme diskov [4]. V tejto práci nebudeme využívať ani jednu z vyššie spomenutých chýb.

Ako sme spomínali v úvode program TrueCrypt slúži na zašifrovanie dát na používateľskom disku pomocou zvoleného hesla. K tomuto TrueCrypt používa niektorý zo šifrovacích algoritmov medzi ktoré patrí AES, Serpent alebo Twofish. Keďže sa jedná o blokové šifry, TrueCrypt používa algoritmus XTS na šifrovanie objemu dát väčšieho ako je 1 blok šifry. TrueCrypt taktiež poskytuje možnosť vybrať si jednu z podporovaných hešovacích funkcií ako RIPEMD-160, SHA-512 a Whirlpool.

Samotné šifrovanie prebieha vo viacerých fázach. Ako prvé sa vygeneruje náhodný kľúč vhodný na šifrovanie pomocou zvoleného algoritmu. Pomocou tohto kľúča sa zašifruje celá požadovaná partícia. Následne sa vytvorí hlavička pre túto partíciu obsahujúca verziu TrueCryptu, veľkosť šifrovanej partície a tento náhodne vygenerovaný kľúč. Hlavička taktiež obsahuje reťazec *TRUE* a CRC-32 kontrolné sumy na overenie jej správnosti. Táto celá hlavička sa opäť zašifruje pomocou zvoleného algoritmu. Keďže používateľské heslo nemusí byť dostatočne dlhé aby bolo použité ako šifrovací kľúč, pri-

## 1. TrueCrypt

pojí sa k nemu 512 bitový náhodný **reťazec** a celé sa to pomocou algoritmu PBKDF2 zmení na kľúč vhodný pre zvolené šifrovanie.



Obr. 1.1: **Schéma XTS algoritmu**

Vďaka zdrojovým kódom voľne prístupným na internete, sme mali možnosť si ich prehliadnuť a hľadať v nich možnosť rýchleho overovania kandidátov na hľadané používateľské heslo. Naším cieľom pri tomto hľadaní bolo nájsť časť kódu, ktorá je zodpovedná za prijatie používateľského vstupu a jeho následne spracovanie. Predpokladáme, že toto spracovanie zahŕňa transformáciu tohto vstupu na kľúč, ktorým sa program následne posnaží rozšifrovať hlavičku partície. Následne sa program posnaží rozšifrovať hlavičku pomocou tohto kľúča a overí, či dáta v nej dávajú zmysel. Predpokladáme, že toto zhŕňa nájsť reťazca TRUE medzi dátami a že overovacie sumy CRC-32 súhlasia s obsahom hlavičky. V tomto momente program vyhlási heslo za správne a následne sa pomocou kľúča vyextrahovaného z tejto hlavičky pokúsi rozšifrovať zvyšok partície. Bohužiaľ vrámci tejto práce **sme nemali čas** izolovať kód zodpovedný za vyššie popísané overenia vstupného hesla. Zvyšok tejto práce sa venuje generovaniu slovníka z ktorého budeme brať kandidátov na hľadané heslo.

## Útoky hrubou silou

Základným princípom útokov hrubou silou hľadanie správneho riešenia pomocou skúšania veľkého množstva kandidátov. Spôsob skúšania kandidátov sa môže líšiť od situácie, avšak veľmi často máme prístupnú zašifrovanú respektíve zahešovanú verziu hľadaného reťazca. Keďže hešovacie algoritmy sú dizajnované tak aby nebolo možné z hešu vyrobiť pôvodný reťazec, musíme pre vyskúšanie kandidáta zahešovať tohto kandidáta a následne porovnať výsledný heš s tým od správneho hesla. Metód akými sa dajú títo kandidáti generovať je mnoho a nižšie si predstavíme pár z nich.

Všetky v praxi používané algoritmy používajú kľúče dĺžky aspoň 256 bitov, čo je ekvivalent reťazca dĺžky 32 zloženého zo ľubovoľných znakov ASCII tabuľky. V praxi je veľmi nepravdepodobné, že používateľ bude mať takéto dlhé heslo založené na takej abecede. Práve preto sa v tejto práci sústredíme na používateľské heslá, pretože majú omnoho menší počet možných reťazcov. Možností pre 256 bitový kľúč je  $2^{256}$  zatiaľ čo možností pre 16 miestne heslo zložené z veľkých, malých písmen, čísl a niektorých často používaných znakov je približne  $2^{101}$ , čo už je dosť signifikantné zmenšenie počtu možností.

### 2.1 Inkrementálny útok

Inkrementálna metóda patrí medzi najzákladnejšie útoky hrubou silou a často krát je práve to, čo sa myslí pod útokom hrubou silou. Podstatou tohto útoku je vyskúšanie všetkých kandidátov. Ak prejdeme cez celý priestor reťazcov, museli sme určite prejsť aj cez konkrétny reťazec, ktorý hľadáme. Táto metóda má tým pádom 100 percentnú úspešnosť. Jej problém avšak spočíva v množstve reťazcov, ktoré musíme vyskúšať. Vo väčšine prípadoch vieme obmedziť hľadanie maximálnou dĺžkou hľadaného výrazu a abecedou znakov z ktorej sa daný výraz skladá. Používateľské heslá majú maximálnu dĺžku okolo 16 znakov a sú zložené prevažne z asi 80 rôznych znakov. Pre takéto reťazce,

ktorých je  $80^{16}$ , by nám vygenerovanie všetkých trvalo približne  $9^{16}$  rokov pri skúšaní 1 milióna reťazcov za sekundu.

## 2.2 Slovníkový útok

Častokrát existuje ešte menšia množina reťazcov, ktoré majú omnoho väčšiu šancu, že medzi nimi bude hľadaný výraz. Toto je pravda špeciálne pri hľadaní používateľských hesiel, nakoľko používatelia volia heslá aby boli zapamätateľné. Vďaka tomu existuje relatívne malá množina reťazcov, ktoré keď vyskúšame máme vysokú šancu úspechu. V takomto prípade je najlepšie zostaviť slovník takýchto reťazcov, ktoré potom postupne skúšame. Táto metóda väčšinou rýchlejšie nájde heslo ako vyššie spomínaná inkrementálna metóda. Avšak jej úspešnosť závisí hlavne od tohto vstupného slovníka. V dnešnom svete keď takmer každá služba vyžaduje heslo od používateľa, existuje veľa verejne prístupných zoznam najčastejšie používaných hesiel, ktoré slúžia ako veľmi dobrý základ pre tento útok.

### 2.2.1 Prekrúcanie slov

Samotný slovníkový útok väčšinou pokrýva takmer zanedbateľné percento všetkých možných výrazov spadajúcich do priestoru hesiel danej abecedy a dĺžky. Preto sa spolu s touto metódou často používa prekrúcanie slov. Podstatou je rozšírenie vstupného slovníka o alternatívne verzie vstupných hesiel za účelom rozšírenia prehľadaného priestoru reťazcov. Bežne sa to dosahuje definovaním zoznam pravidiel popisujúcich transformáciu slova. Tieto pravidlá budú následne aplikované na jednotlivé vstupné slová a tým vzniknú potenciálne nové reťazce, ktoré sa nenachádzajú vo vstupnom slovníku. Tieto pravidlá môžu transformovať slovo rôznymi spôsobmi od pridania prefixu či sufixu cez zmenu veľkostí písmen alebo vynechanie spoluhlások. Mnohé programy zaoberajúce sa útokmi hrubou silou podporujú vlastný jednoduchý jazyk na popisanie týchto pravidiel.

## 2.3 Hybridný útok

V tejto práci sa venujeme implementácii útoku, ktorý je spojením vyššie uvedených. Naším hlavným cieľom je nájsť správne heslo k particii zašifrovanej programom TrueCrypt. Keďže chceme toto heslo nájsť v ľubovoľnom veľkom konečnom čase, budeme náš algoritmus implementovať, tak aby vygeneroval všetky možné reťazce zo vstupnej abecedy kratšie ako nami stanovená dĺžka. V tomto sa bude veľmi podobáť na inkrementálny útok. Avšak v našom prípade predpokladáme, že na vstupe dostaneme ešte slovník obsahujúci zoznam hesiel. Predpokladáme, že tento zoznam je usporiadaný podľa pravdepodobnosti správnosti hesiel v ňom. Naš algoritmus si na základe týchto

hesiel upraví pravdepodobnosť vygenerovania jednotlivých reťazcov aby následne mohol na výstup dávať heslá od najpravdepodobnejšieho. V práci sme implementovali 2 spôsoby ako spracovať vstupné dáta.

## 2.4 SAT Solver

Táto metóda útoku hrubou silou je založená na probléme splniteľnosti boolovského výrazu. Ako vstup tohto algoritmu je boolovský výraz, väčšinou v konjunktívnom normálnom tvare, pre ktorý sa daný algoritmus snaží nájsť ohodnotenia boolovských premenných ~~také~~, aby všetky formuly tohto výrazu boli pravdivé. Algoritmus postupne rekurzívne prehľadáva všetky možnosti nastavenia jednotlivých premenných. Po nastavení niektorej premennej skontroluje, či žiaden výskyt tejto premennej nespôsobil konflikt, čiže ohodnotil formulu tak že sa stala nesplniteľnou. V tomto prípade sa algoritmus vráti do momentu kedy bol výraz nekonfliktný a odtiaľ sa snaží postupovať inou cestou.

Náš problém sa dá pretransformovať na vstup pre takýto SAT solver. V našom prípade by sme vedeli šifrovací algoritmus prepísať do konjunktívnej normálnej formy. Ako výstup tohto algoritmu je reťazec takmer náhodných bitov, ktoré dopredu poznáme, pretože tieto sú fyzicky uložené na disku. Neznámymi v tomto ~~celom~~ boolovskom výrazu sú vstupné dáta a kľúč pomocou ktorého boli tieto dáta zašifrované. Naším predpokladom je, že veľkú časť týchto dát by sme vedeli určiť keďže ide o dopredu známe informácie ako reťazec TRUE, použitú verziu programu TrueCrypt a podobne. Vďaka týmto informáciám by sa vedel SAT solver skôr rozhodnúť, že niektoré ohodnotenie premenných nie je možné, kvôli konfliktu, ktorý by vznikol.

V našej práci sme ~~tejto metóde nevenovali veľkú pozornosť~~, keďže ~~používanie SAT solverov v kryptografii je už vo veľkom preskúmané~~. Existuje viacero prác, ktoré do podrobnej rozoberajú správanie týchto solverov v prípade, že majú dopredu určené niektoré bity vstupu alebo výstupu. Taktiež existujú práce, ktoré sa venujú optimalizácii behu SAT solveru čo zahŕňa optimalizáciu poradia ohodnotenia premenných alebo miesta na ktoré sa algoritmus vráti v prípade konfliktu.

## Používateľské heslá

Aj napriek tomu ako sú používateľské heslá vo všeobecnosti ľahko prelomiteľné ~~mes~~ sú najčastejšie používaný spôsob autentifikácie používateľa. Tento trend sa pravdepodobne ani v najbližšej budúcnosti nebude meniť. Hlavným dôvodom slabých používateľských hesiel sú obmedzenia ľudskej pamäte. Ak by si používatelia nemuseli pamätať heslo používali ~~by~~ heslá s najväčšou entropiou. Tie by boli najdlhšie možné povolené systémom, zložené z náhodne vybraných znakov povolených týmto systémom ~~v poradi, ktoré by bolo kompletne náhodné~~ a neexistovala by žiadna iná možnosť ako túto sekvenciu dostať na základe inej informácie.

Tento spôsob tvorby hesiel je presný opak toho k čomu je prispôbena ľudská myseľ. Ľudia sú schopní zapamätať si sekvenciu znakov dlhú približne sedem znakov plus mínus dva znaky vo svojej krátkodobej pamäti. Taktiež aby si človek zapamätal takúto sekvenciu, táto sekvencia nemôže byť kompletne náhodná ale musí skladať zo známych kusoch informácie ako sú slová. Nakoniec ľudská myseľ funguje veľmi dobre vďaka redundancii informácie, čiže človeku sa ľahšie pamätajú veci, ktoré si vie odvodiť z viacerých iných kusoch informácií.

Mnohé systémy používajúce heslá ako spôsob autentifikácie používateľa dávajú používateľovi rady ako si zvoliť bezpečné heslo. Na základe informácií spomenutých na začiatku tejto kapitoly by heslo malo byť dostatočne dlhé, skladajúce sa z rozumne veľkej abecedy znakov a malo by byť ľahko zapamätateľné. Väčšina týchto systémov sa sústreďí hlavne na prvé dve podmienky tvorby hesla a to že by malo používateľove heslo spĺňať minimálnu dĺžku a obsahovať aspoň jeden z každej kategórie veľké, malé písmena, číslce a špeciálne znaky. Týmto dávajú dôraz na ochranu proti útokom hrubou silou oproti zapamätateľnosti hesla.

Mnoho používateľov, ktorí boli prezentovaný minimálnymi nárokmi na heslo, si vyvinulo spôsob na generovanie takýchto hesiel. Tento spôsob zahŕňal výber slova, ktorému



zväčšili prvé písmeno a pridali sufix skladajúci sa z číslíc a špeciálnych znakov. Takéto spôsoby zakladajúce na jednoduchej transformácii slova sa veľmi rýchlo ukázali neefektívne keď sa počas posledného desaťročia náramne zvýšil výkon počítačov. Tie boli schopné skúsiť veľké množstvo transformácií pre každé slovo slovníka.

Začali sa objavovať mnohé mnemotechnické pomôcky umožňujúce generovať používateľské heslá. Jedna z často sa vyskytujúcich doporučovala vytvorenie si extrémne dlhého slovného spojenia. Účelom tejto metódy bola ochrana proti útokom hrubou silou zväčšením priestoru potenciálnych hesiel pomocou zvýšenia dĺžky samotného hesla. Ďalšia veľmi často používaná metóda bola založená na tvorbe hesla zobratím prvých znakov slov z frázy, ktorú používateľ vymyslel. Heslá založené na mnemotechnických pomôckach sa málokedy vyskytujú v slovníkoch používaných pri útokoch hrubou silou. To avšak neznamená, že sú bezpečnejšie ako bežné heslá [8].

V tejto práci sa snažíme vyvinúť algoritmus, ktorý dostane na vstupe slovník **s** heslami. Tento slovník ma slúžiť na naučenie algoritmu metódy tvorby a používania hesiel pre konkrétneho používateľa. Mal by zahŕňať ukážky hesiel, ktoré sú vytvorené podobnými metódami ako používateľ vytvára heslá pre potreby svojej autentifikácie.

## Učenie

Ako sme spomínali vyššie v texte v tejto práci sa budeme zaoberať útokom, ktorý dostane na vstupe slovník a na základe tohto slovníka bude generovať heslá zoradené podľa pravdepodobností. Kvalita výsledného zoznamu bude závisieť od schopnosti algoritmu správne sa naučiť ohodnotiť pravdepodobností jednotlivých reťazcov. V tejto práci kladieme dôraz na skúmanie možností generovania reťazcov použitím bezkontextových gramatík, avšak implementovali sme taktiež algoritmus používajúci Markovovské zdroje, ktorý použijeme na porovnanie s bezkontextovými gramatikami.

## 4.1 Pravdepodobnostné bezkontextové gramatiky

Bezkontextové gramatiky sú definované štyrmi parametrami. Množinou neterminálov, ktoré slúžia ako premenné pri odvodzovaní vetnej formy. Množinou terminálov, ktoré tvoria reálny obsah výslednej vetnej formy. Túto množinu tvorí vstupná abeceda symbolov a je disjunktná s neterminálmi. Vetná forma obsahujúca len terminálne symboly sa nazýva terminálna vetná forma. Ďalej je potrebné zadať počiatočný neterminál z ktorého sa bude každá vetná forma odvídať. Nakoniec potrebujeme poznať množinu prepisovacích pravidiel, ktoré definujú spôsob akým sa menia neterminály na ďalšie vetné formy. Pri bezkontextových gramatikách majú prepisovacie pravidlá tvar

$$N \rightarrow (N \cup \Sigma)^*$$

kde  $N$  vyjadruje množinu neterminálov a  $\Sigma$  je množina terminálov. Tieto pravidlá vyjadrujú schopnosť neterminálu zmeniť sa na ľubovoľnú vetnú formu, bez ohľadu na kontext v ktorom sa nachádza. V našej práci sa budeme venovať špeciálnym bezkontextovým gramatikám, ktorých každé prepisovacie pravidlo má priradenú pravdepodobnosť. Suma pravdepodobností jedného neterminálu bude vždy rovná 1. Vďaka týmto pravdepodobnostiam dokážeme ohodnotiť nami generované heslá a zoradiť ich podľa ich pravdepodobností. Pravdepodobnosť ľubovoľnej vetnej formy získame súčinom pravdepodobností prepisovacích pravidiel použitých na jej odvodenie.

Odvedenia vetných foriem, čiže sekvencie použitých prepisovacích pravidiel, tvoria strom odvedenia danej gramatiky. Je možné aby v takomto strome existovali 2 rôzne cesty odvedenia, ktoré na koniec vygenerujú rovnakú vetnú formu. Tejto vlastnosti bezkontextových gramatík sa budeme snažiť vyhnúť vytvorením pravidiel tak aby ľubovoľná terminálna vetná forma mala práve 1 spôsob odvedenia v danej gramatike. Zámerom tohto obmedzenia je zamedzenie generovania duplikátov, keďže predpokladáme, že pri skúšaní jedného hesla viac krát sa výsledok tohto pokusu nezmení.

## 4.2 Markovovské zdroje

Bezkontextové gramatiky, ktoré sme implementovali si odvádzajú vetné formy výberom prepisovacieho pravidla s najvyššou pravdepodobnosťou. Taktiež si pamätajú, ktoré pravidlá už použili aby sa vyhli odvodeniu jednej terminálnej vetnej formy viac krát. Keďže gramatika si počas generovania reťazca pamätala celú postupnosť použitých prepisovacích pravidiel vyžadovala veľmi veľa pamäte. Preto sme implementovali náhodný proces prechádzajúci cez priestor stavov. Tento náhodný proces spĺňa Markovovskú vlastnosť, ktorá je popísaná ako takzvaná bezpamätovosť. Hovorí o tom, že distribúcia pravdepodobností nasledujúceho stavu závisí len od terajšieho stavu a nezáleží na sekvencií udalostí, ktoré mu predchádzali. Vďaka tejto vlastnosti je potrebná pamäť konštantná. Jediné čo si tento algoritmus pamätá, je tabuľka pravdepodobností pomocou ktorej sa rozhoduje aký najbližší symbol vygeneruje. Pre konštantne veľký prefix si Markovovský zdroj poráta pravdepodobností nasledujúcich znakov. Pri tomto spôsobe učenia sa dá pre ľubovoľný stav vypočítať pravdepodobnosť s akou sa Markovovský zdroj dostane do tohto stavu.

Pomocou základnej implementácie Markovovho zdroja dokáže algoritmus využívajúci tento zdroj generovať len heslá zložené z kombinácií znakov, ktoré videl na vstupe. Toto pre nás spôsobuje problém, keďže by sme chceli aby náš algoritmus v konečnom čase vygeneroval všetky možné reťazce. Na vyriešenie tohto problému sme sa rozhodli definovať pravdepodobnosti pre kombinácie znakov, ktoré sa na vstupe nevyskytli.

# Implementácia

## 5.1 Bezkontextová gramatika



### 5.1.1 Tvorba gramatiky

Pri tvorbe gramatiky sme **potrebovali** zaistiť aby gramatika spĺňala určité podmienky. Prvou z nich je schopnosť gramatiky vygenerovať všetky reťazce zo vstupnej abecedy kratšie ako používateľom zadaná maximálna dĺžka. Druhou podmienkou je aby každé terminálne slovo, ktoré gramatika generuje malo práve **jeden strom odvodenia**. Nakoniec by sme chceli aby algoritmus, ktorý bude pomocou tejto gramatiky generovať heslá bol **deterministický**.



**Jednoduché neterminály** Prvý typ neterminálov, ktoré budeme nazývať jednoduché, obsahuje pravidlá na **zterminálne** generovaného slova. Keďže naša vstupná abeceda obsahuje **okolo 70 znakov**, medzi **ne** patria veľké a malé písmena, cifry a niektoré často používané symboly, rozhodli sme sa ich rozdeliť do jednotlivých skupín. Pre každú z týchto skupín sme vytvorili neterminál, ktorý bude reprezentovať sekvenciu **pevnej dĺžky** zloženú zo znakov danej skupiny. V gramatike tieto neterminály vyjadrujeme pomocou prvého písmena anglického názvu danej skupiny.

- U - veľké písmena
- L - malé písmena
- D - cifry
- S - symboly

Každý jednoduchý neterminál sa teda skladá z písmena vyjadrujúceho skupinu znakov, ktoré generuje, a čísla popisujúceho dĺžku sekvencie na pravej strane pravidiel tohto neterminálu. Ako napríklad jednoduchý neterminál  $D_1$  vyjadruje pravidla  $D_1 \rightarrow 1|2...9|0$ .

Keďže všetkých **variácií veľkostí**  $k$  pri  $n$  prvkoch je  $n^k$  **rozhodli sme sa** zdefinovať maximálnu veľkosť jednoduchého neterminálu, vyjadrujúcu maximálne povolené  $k$ .

~~môže byť obrovské množstvo rozhodli sme sa zdefinovať maximálnu dĺžku sekvencie generovanej jednoduchým neterminálom.~~

**Zložené neterminály** Jednoduché neterminály nám pomáhajú vyjadrovať sekvenciu znakov práve jedného z vyššie vymenovaných typov. Aby sme boli schopní popísať ľubovoľný reťazec tvorený znakmi vstupnej abecedy, budeme tieto jednoduché neterminály skladať do skupín, zložených neterminálov. Tieto neterminály vyjadrujú vždy jeden možný predpis pre terminálne slovo. Napríklad neterminál  **$U_1L_3D_4$**  vyjadruje všetky terminálne slová začínajúce na veľké písmeno nasledované tromi malými písmenami, ukončené štvoricou čífer.

Ďalej taktiež **nedovoľujeme aby** sa vyskytovali 2 jednoduché neterminály rovnakého typu za sebou. V prípade, že potrebujeme popísať sekvenciu terminálnych znakov jedného typu dlhšiu ako povolené maximum (popísané vyššie), rozdelíme túto sekvenciu do viacerých jednoduchých neterminálov pažravým algoritmom, čiže každý z týchto neterminálov zoberie maximálny možný počet znakov sekvencie. Ak zoberieme heslo pozostávajúce z 9 čífer ako napríklad jedno z najpoužívanějších *123456789* a máme najvyššiu povolenú dĺžku jednoduchého neterminálu nastavenú na 4, toto heslo bude v našej gramatike zapísané ako  $D_4D_4D_1$ . Tento spôsob nám zaručí, že nevzniknú dva rôzne zložené neterminály vyjadrujúce ten istý predpis terminálneho slova.

Počiatočný neterminál gramatiky  $Z$  bude obsahovať pravidlá prepisujúce tento neterminál na **niektorý** zo zložených alebo jednoduchých neterminálov. Tento spôsob generovania gramatiky **spĺňa obe pravidlá**, ktoré sme popisovali v úvode tejto kapitoly.

### 5.1.2 Počítanie pravdepodobností

Ako sme spomínali v úvode **textu**, nami generované pokusy o nájdenie hesla chceme prispôbiť potrebám jednotlivých používateľom, ktorí sa snažia získať svoje stratené heslo. Aby sme vedeli čo najlepšie vyhovieť týmto používateľom, potrebujeme **upraviť** našu gramatiku. Tu **prichádzajú do pozornosti** pravdepodobnosti jednotlivých pravidiel našej gramatiky. Naším cieľom je nastaviť našu gramatiku tak, aby generovala heslá podľa pravdepodobnosti použitia daným používateľom. Úspešnosť tohto učenia gramatiky bude **drastický** **záležať** od kvality vstupných dát.

Vzhľadom na to, že v dnešnom svete používatelia používajú rôzne služby, ktoré každá odporúča mať jedinečné heslo, používatelia používajú niekoľko hesiel naraz. Tieto heslá by si radi všetky pamätali a preto **si** často vytvoria pre seba charakteristický spôsob

tvorby a zapamätania si týchto hesiel. V ideálnom prípade by sme chceli aby naše vstupné dáta pozostávali z čo najväčšieho počtu hesiel vytvorených pomocou tohto charakteristického spôsobu, keďže každé upresnenie informácií o hľadanom hesle nám **zvýši** rýchlosť nájdenia tohto hesla.

Keďže cieľom našej práce je nájsť heslo so 100% pravdepodobnosťou, čo v najhoršom prípade znamená vygenerovať všetky možné reťazce kratšie ako zadaná maximálna dĺžka hesla, tak základnú gramatiku s pravidlami vieme vygenerovať **dopredu** a pravidla tejto gramatiky sa budú meniť len pri zmene maximálnej dĺžky hesla. Pri používaní nášho algoritmu s rôznymi slovníkmi nevyžaduje vyrábanie novej gramatiky až do momentu kedy sa rozhodneme generovať heslá s inou maximálnou dĺžkou. Pravdepodobnosť prepisovacích pravidiel generujúcich terminálne sekvencie budeme rátať ako percento výskytov danej terminálnej sekvencie spomedzi všetkých sekvencií spadajúcich pod tento neterminál. Práve kvôli tomuto spôsobu sme pridali v implementácii možnosť napísať do vstupného slovníku počty výskytov jednotlivých hesiel, aby mal používateľ možnosť zdôrazniť dôležitosť hesla. Vstupné slovníky, ktoré neskôr používame v našich testoch majú formát, kde na každom riadku je heslo s počtom jeho výskytov oddelené medzerou.

```
1 for i in range(1, len(word)):
2     if (word[i] in lower) and (currentNet != '\L\'):
3         rule += currentNet + str(i-startI)
4         rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
5         ruleCount[currentNet + str(i-startI)] += occ
6         startI = i
7         currentNet = '\L\'
8         currentSubstring=''
9     elif (word[i] in upper) and (currentNet != '\U\'):
10        rule += currentNet + str(i-startI)
11        rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
12        ruleCount[currentNet + str(i-startI)] += occ
13        startI = i
14        currentNet = '\U\'
15        currentSubstring=''
```

Zdrojový kód 5.1: Úprava pravidiel na základe vstupného slova

Pri počítaní pravdepodobností zložených neterminálov máme viacero možností ako postupovať.

**Priamo zo vstupného slovníka** Prvý spôsob ako postupovať **bol** identický s tým pre jednoduché neterminály. Pre každé pravidlo gramatiky prepisujeme počiatkový neterminál na zvolený zložený neterminál vypočítame jeho pravdepodobnosť ako pomer počtu výskytov tohto neterminálu a výskytov všetkých neterminálov dohromady. Tento spôsob môže mať ešte **2 varianty**.

- Do výskytov počítame len výskyty hesiel ktoré sú presne reprezentované daným neterminálom
- Do výskytov započítame aj výskyty kedy je zvolený neterminál podreťazcom iného neterminálu

V oboch týchto variantoch počítame počty výskytov jednoduchých neterminálov. Rozdiel medzi týmito variantami ukážeme na príklade. Majme na vstupe heslo, ktoré je reprezentované zloženým neterminálom  $U_2L_3D_2$ , použitím prvého variantu tento zložený neterminál vygeneruje jedno zvýšenie počtu výskytov a to pre tento konkrétny neterminál. Ukážka kódu implementujúceho prvý variant 5.1. Druhý variant by na tomto neterminály vyvolal 2 navýšenia počtu výskytov a to osobitne pre zložené neterminály  $U_2L_3$  a  $U_2L_3D_2$ . Aby sme upravili náš program na druhý variant pridali sme riadky kódu znázornené v 5.2 pre **každý krok kedy** sa mení typ pozorovaného jednoduchého neterminálu.

```

1 if not rule in rulez['Z']:
2     rulez['Z'][rule] = 1
3 rulez['Z'][rule] += occurrences
4 ruleCount['Z'] += occurrences

```

Zdrojový kód 5.2: Pripočítanie výskytov k podmnožinám zložených neterminálov

**Rekurzívne** Ďalší spôsob spočíva v tom, že zo vstupného slovníka vypočítame pravdepodobnosti len pre jednoduché neterminály. Následne pre zložené neterminály počítame pravdepodobnosti ako súčin pravdepodobností jednoduchých neterminálov, ktoré daný neterminál obsahuje.



### 5.1.3 Generovanie hesiel

Dôležitým aspektom používania bezkontextových gramatík je práve spôsob generovania hesiel. Naším hlavným cieľom bolo generovanie hesiel pomocou gramatiky od najpravdepodobnejšieho z nich. Tieto heslá generujeme tak, že počiatkový neterminál



rozpíšeme na najpravdepodobnejší zložený neterminál. Následne jednoduché neterminály, z ktorých sa tento zložený neterminál skladá, prepíšeme postupne ich najpravdepodobnejšími terminálnymi vetnými formami. K tomu sme potrebovali utriediť všetky pravidlá pre jednotlivé neterminály zostupne podľa ich pravdepodobnosti. Toto utriedenie nám umožnilo pamätať si len indexy posledne použitých pravidiel jednotlivých neterminálov, ktoré práve rozpisujeme. Týmto spôsobom dokážeme popísať **stromy odvodenia jednotlivých hesiel ako  $k$ -tice čísel** vyjadrujúce poradie použitých pravidiel vrámci ich neterminálov. Kde jedno číslo slúži na určenie vybratého zloženého neterminálu a zvyšné vyjadrujú poradie použitých pravidiel  $k - 1$  jednoduchých neterminálov, z ktorých sa tento zložený neterminál skladá.

Na začiatku je heslo s najvyššou pravdepodobnosťou popísané vektorom samých núl. Z tohto bodu rozbehneme algoritmus prehľadávania do šírky s použitím prioritnej fronty. Ako prvé si do fronty pridáme všetky možné vektory indexov vzdialené od aktuálneho práve o 1, čiže také kde sa niektorý z indexov zvýši o jedna zatiaľ čo ostatné ostanú nezmenené. Do fronty pridávame dvojice vektor a pravdepodobnosť tohto vektoru. Pravdepodobnosť jednotlivých vektorov rátame ako súčin pravdepodobností pravidiel na ktoré ukazujú. Keďže každý čo pridávame sa líši práve v jednom indexe,  $p[t + 1] = p[t] / p_i[t] * p_i[t + 1]$ . Keď už sme pridali všetky takéto susedné vektory, vyberieme z fronty ten s najvyššou pravdepodobnosťou a na ňom celý tento proces opäť zopakujeme.

Týmto spôsobom by sme ale generovali veľké množstvo rovnakých vektorov, ktoré by sme dostali zmenou indexov v inom podarí. Napríklad ak by sme zvýšili najprv index na pozícii 1 a potom 3, dostali by sme to isté ako keby sme zvýšili na pozícii 3 a potom 1. Preto zavedieme ešte špeciálne číslo, ktoré nazveme **radom** vektoru. Rád vektoru bude číslo určujúce pozíciu najvyššieho zmeneného indexu. Zároveň pri generovaní susedných vektorov dovolíme meniť indexy len na pozíciách vyšších alebo rovných ako je aktuálny rád vektora. Týmto budeme generovať terminálne slová **zlava doprava** a vďaka tomu nebudeme generovať duplikáty, ktoré by sa od seba líšili len v poradí v akom sme rozpisali neterminály na terminály.



Vo vyššie uvedených tabuľkách 5.1 sme demonštrujeme 2 kroky nášho algoritmu na generovanie hesiel. V tomto príklade sa sústreďíme na generovanie rôznych terminálnych slov zo zloženého neterminálu  $U_1L_3D_2$ . V ľavej tabuľke môžeme vidieť zadané prepisovacie pravidlá pre tento neterminál aj s pravdepodobnosťami, ktoré majú priradené. V pravej tabuľke simulujeme obsah našej prioritnej fronty, kde dvojitou vodorovnou čiarou sú oddelené stavy tejto fronty v rôznych krokoch. V počiatočnom stave máme vo fronte prvý prvok ukazujúci na najpravdepodobnejšie heslo generované z definovaných pravidiel. Algoritmus tento prvok vyberie z fronty a následne tam vloží



Tabuľka 5.1: Ukážka krokov algoritmu pre neterminál  $U_1L_3D_2$ 

Ľavá strana	Pravá strana	p	i	p	řád	vektor	slovo
$U_1$	A	0.7	0	0.336	0	[0, 0, 0]	Aminf47
$U_1$	B	0.2	1	0.168	1	[0, 1, 0]	Afmfi47
$U_1$	C	0.1	2	0.096	0	[1, 0, 0]	Bminf47
$L_4$	minf	0.6	0	0.084	2	[0, 0, 1]	Aminf42
$L_4$	fmfi	0.3	1	0.096	0	[1, 0, 0]	Bminf47
$L_4$	dipl	0.1	2	0.084	2	[0, 0, 1]	Aminf42
$D_2$	47	0.8	0	0.056	1	[0, 2, 0]	Adipl47
$D_2$	42	0.2	1	0.042	2	[0, 1, 1]	Afmfi42



prvky označujúce heslá vzdialené práve na 1 zmenu použitého pravidla. Tieto novo pridané prvky sú automaticky zoradené podľa pravdepodobností vďaka tomu, že na pozadí je naša fronta reprezentovaná haldou.

V druhom kroku algoritmu vyberie prvok z najvyššou pravdepodobnosťou. Opäť do fronty pridáme prvky vyjadrujúce heslá vzdialené na 1 zmenu použitého pravidla. Tu si treba všimnúť, že nepridali sme prvok hovoriaci o vektore [1, 1, 0], keďže rád práve vytiahnutého vektora je 1, čiže môžeme meniť len indexy 1 a 2, ktoré sú väčšie rovné ako rád vektora. Týmto spôsobom algoritmus pokračuje až dokým nevygeneruje požadovaný počet hesiel alebo nevyprázdni frontu. Fronta sa môže vyprázdniť len ak prejdeme cez všetky možné heslá, keďže jediný moment kedy nepribudne žiaden prvok do fronty je ak rád vektora bude rovný jeho dĺžke a v poslednom jednoduchom neterminály sme použili už všetky jeho pravidlá.

Tento priamočiary prístup ku generovaniu spĺňa všetky naše požiadavky na generované heslá. Veľkosť fronty sa môže veľmi radikálne zmeniť na základe rozloženia pravdepodobností vrámci neterminálov. Preto by toto miesto bolo vhodné na použitie nejakej heuristiky. Bohužiaľ vrámci tejto práce sa nám nepodarilo nájsť vhodné heuristiky, ktoré by zmenšili pamäťovú náročnosť zatiaľ čo by čo najlepšie uchovali poradie hesiel.

V 5.3 môžeme vidieť kus kódu zodpovedný za napĺňanie prioritnej fronty ďalšími kandidátmi na najbližšie vygenerované heslo. Premenná *task* je usporiadaná dvojica, kde prvý člen tejto dvojice vyjadruje rád vektora, ktorý je uložený ako druhá časť tejto dvojice. Algoritmus prejde od člena určeného radom vektora až po koniec vektora a pre každý prvok posunie index ukazujúci na aktuálne použitý prvok. Taktiež vypočíta pravdepodobnosť hesla reprezentovaného novým stavom vektora. Túto pravdepodobnosť spolu s usporiadanou dvojicou obsahujúcou zmenený rád vektora a samotný vektor vloží do prioritnej fronty.

```

1 for x in range(task[0],len(task[1])):
2     tmp = copy.deepcopy(task[1])
3     newpriority = priority / rulez[net[(x-1)*2:x*2]][tmp[x]][1]
4     tmp[x] += 1
5     if tmp[x] >= len(rulez[net[(x-1)*2:x*2]]):
6         continue
7     newpriority = newpriority * rulez[net[(x-1)*2:x*2]][tmp[x]][1]
8     newtask = (x, tmp)
9     add_task(newtask, newpriority)

```

Zdrojový kód 5.3: Generovanie všetkých susedných vektorov

Veľkým nedostatkom použitia bezkontextových gramatík je veľkosť pamäte, ktorú potrebuje. Samotný zápis gramatiky na disku v tvare JSON mal pri gramatike generujúcej 12 znakové heslá okolo 1 gigabajtu. Avšak táto veľkosť nemá v dnešnej dobe až taká problematická, keďže existujú kvalitné kompresné algoritmy.


Väčší problém nastáva s pamäťou použitou pri samotnom generovaní hesiel z gramatiky. Prioritná fronta, ktorú používame častokrát dosahuje obrovské veľkosti presahujúce desiatky gigabajtov. Tento problém by sa dal riešiť zmenou algoritmu použitého pri generovaní gramatiky, ideálne takého, čo si potrebuje pamätať len gramatiku samotnú a konštantne veľa informácie k tomu. Toto sa nám bohužiaľ nepodarilo v tejto práci dosiahnuť.

Poslednou vecou čo sme riešili v implementácii bezkontextových gramatík bola možnosť prerušovaného generovania, kedy používateľ má možnosť generovať požadované heslá po ľubovoľných veľkých častiach.




## 5.2 Markovovský zdroj

Po implementácii vyššie uvedeného algoritmu na generovanie hesiel pomocou pravdepodobnostných bezkontextových gramatík a odhalení nedostatkov čo sa týka pamätovej náročnosti sme sa rozhodli implementovať ešte jednu metódu. Tou je Markovovský zdroj. Ako sme písali v predošlej kapitole, jedná sa o náhodný proces, ktorý spĺňa podmienku bezpamätivosti. Markovovské zdroje sa veľmi často používajú práve pri generovaní prirodzeného jazyka. Práve preto boli vhodným kandidátom pre generovanie hesiel na základe znalostí získaných zo vstupného slovníka.

Bohužiaľ táto metóda nespĺňa ani jednu z podmienok, ktoré sme si na začiatku definovali. ~~Ako bolo písané jedná sa o náhodný proces, čiže dve od seba rôzne spustenia môžu viesť k rôznym výsledkom.~~ Druhá podmienka o generovaní duplikátov taktiež nie je splnená, keďže tomuto zdroju nič nebráni k tomu vygenerovať viac krát počas behu to isté slovo ~~a nič by mu v tom nemalo ani brániť, vzhľadom na bezpamätovosť.~~ A na koniec Markovovské zdroje nemusia generovať všetky možné reťazce kratšie ako zadaná maximálna dĺžka. 



Aj keď prvé dve podmienky nevedia Markovovské zdroje splniť už priamo z definície, s tretou sme sa pokúsili niečo vymyslieť. Ako prvé sme sa pokúšali inicializovať všetky počty výskytu na 1 namiesto 0. Toto avšak spôsobilo, že sa zdroj relatívne ľahko dostal medzi prefixy, ktoré neboli definované, kde všetky znaky majú rovnakú pravdepodobnosť. Dôsledkom tohto bolo cyklenie sa v týchto neznámych stavoch, čoho výsledkom boli dlhé nezmyselné reťazce znakov. Preto sme sa snažili nájsť spôsob ako nastaviť pravdepodobnosti nevidených stavov na nenulové, avšak dostatočne malé aby sa v nich samotný algoritmus necyklil.


Pri takto definovanom Markovovskom zdroji dokážeme všetky stavy tohto zdroja rozdeliť do dvoch množín prefixov. Videné prefixy sú také, ktoré aspoň raz nastali pri učení podľa vstupného slovníka. Takéto stavy majú pre aspoň 1 znak slovníka nenulovú pravdepodobnosť. Druhou väčšou skupinou prefixov sú nevidené prefixy, ktoré sa nevyskytli nikde vo vstupnom slovníku a preto pre všetky znaky našej abecedy je pravdepodobnosť prechodu nulová. Keďže chcem upraviť náš Markovovský zdroj tak aby mal možnosť generovať všetky možné heslá, potrebujeme tieto nulové pravdepodobnosti zmeniť na nenulové. 

Videné stavy nemusia mať určené pravdepodobnosti pre všetky znaky nášho vstupného slovníka. Pre tieto znaky nastavíme pravdepodobnosti, ktoré v pôvodnom algoritme majú nulovú pravdepodobnosť, na ~~nenulovú~~ hodnotu  $\varepsilon$ . Táto hodnota by mala vyjadrovať pravdepodobnosť prechodu zo stavu v ktorom je prefix známy z dát vo vstupnom súbore do stavu kedy prefix je neznámy a pravdepodobnosti všetkých znakov sú nulové. Domnievame sa, že pre správne fungovanie algoritmu by hodnota  $\varepsilon$  mala byť niekoľko násobne menšia ako najnižšia známa pravdepodobnosť pre daný prefix. Veľkosť tejto konštanty necháme na určenie používateľovi nášho programu.

Po prechode nášho algoritmu do stavu, ktorý nebol videný počas inicializácie programu potrebujeme nastaviť pravdepodobnosti všetkých znakov našej abecedy. Pravdepodobnosť jednotlivých znakov nastavíme na hodnotu  $\varepsilon$  ak sa jedná o znak pomocou ktorého v ďalšom kroku algoritmu vznikne videný prefix. V prípade, že znak dostane

naš algoritmus do stavu s iným neznámym prefixom, nastavíme tomuto znaku pravdepodobnosť  $\delta$ , ktorá je niekoľko krát menšia ako  $\varepsilon$ . Opäť výber tejto konštanty necháme na používateľa. Použitím hodnôt  $\varepsilon$  a  $\delta$  by sme mali dostať algoritmus používajúci Markovovský zdroj stavu, kedy v konečnom čase dokáže vygenerovať ľubovoľný počet hesiel.

V rámci implementácie sme algoritmus používajúci Markovovský zdroj upravili tak aby nepoužíval žiadnu pamäť navyše. Jediná informácia, ktorú si tento algoritmus pamätá je tabuľka pravdepodobností nasledujúceho znaku po danom prefixe. Túto informáciu si pamätá len pre znaky a prefixy, ktoré sa vyskytujú vo vstupnom slovníku, ktorý dostal na vstupe. V prípade, že sa algoritmus dostane do stavu, kedy sa aktuálny prefix nenachádzal vo vstupnom slovníku, prejde cez všetky znaky vstupnej abecedy a každému priradí pravdepodobnosť  $\varepsilon$  ak sa vygenerovaním tohto znaku dostane do známeho prefixu alebo pravdepodobnosť  $\delta$  ak prídanie tohto znaku vedie do ďalšieho stavu s neznámym prefixom.  

Rozdiely vo veľkosti  $\varepsilon$  oproti najmenej nenulovej pravdepodobnosti pre ten prefix a  $\delta$  od  $\varepsilon$  by mali zaručiť, že algoritmus sa snaží preferovať heslá, ktoré sa skladajú z kombinácií znakov videných na vstupe. Výsledky tohto algoritmu pre rôzne nastavené hodnoty koeficientov  $\varepsilon$  a  $\delta$  sú znázornené v kapitole Testy. 

V tomto prípade sme zvažovali aj použitie možnosti zmeny týchto pravdepodobností počas behu programu podobne ako je pri simulovanom žíhaní. Zo začiatku by sme tieto pravdepodobnosti nastavili na relatívne nízke hodnoty a s počtom hesiel vygenerovaných našim algoritmom by sa tieto hodnoty zväčšovali aby mal algoritmus vyššiu tendenciu dostať sa aj k menej pravdepodobným heslám. Bohužiaľ v časovom rámci tejto práce sme nenašli priestor na implementáciu a preskúmanie takto upraveného algoritmu.

# KAPITOLA 6

## Testy

V predošlej kapitole sme bližšie popísali implementáciu našich algoritmov. Táto kapitola sa zameriava na evaluáciu výsledkov z týchto algoritmov. Keďže celá táto práca sa zaoberá implementáciou algoritmov na generovanie hesiel pomocou vstupného slovníka, bolo potrebné si nájsť vhodný vstupný slovník. Podarilo sa nám nájsť online zdroj slovníkov [2], ktorý obsahuje slovníky s usporiadanými heslami podľa pravdepodobnosti výskytu. Slovník je formátovaný v dvoch stĺpcoch, kde prvý obsahuje informáciu o počte výskytov daného heslá a druhý je samotné heslo.

### 6.1 Časová náročnosť

V tomto základnom teste sme spustili naše algoritmy a počítali čas behu algoritmov pre jednotlivé parametre spustenia. Pre všetky testy používame slovník *phpbb* stiahnutý z [2]. Slovník sme upravovali aby obsahoval len heslá zodpovedajúce maximálnej dĺžke hesiel, ktoré generuje bezkontextová gramatika. Týmto sme znížili pravdepodobnosť Markovovského zdroja generovať heslá dlhšie ako stanovené maximum pre gramatiku. Stĺpec *d* označuje maximálnu dĺžku generovaných hesiel zatiaľ čo stĺpec *p* vyjadruje maximálnu veľkosť jednoduchého neterminálu v prípade bezkontextovej gramatiky zatiaľ čo pri Markovovskom zdroji vyjadruje dĺžku prefixu podľa ktorého sa rozhoduje. Tieto časové testy boli spúšťané na procesore Intel® Core™ i5-4690K s rýchlosťou 3.50GHz na operačnom systéme Windows 10. Namerané hodnoty zobrazené v tabuľke sú uvedené v sekundách.

Tabuľka 6.1: Časy pre slovník phpbb

	d	p	GEN	10000	50000	100000	500000	1000000	5000000
CFG	6	2	0.103	0.754	3.504	7.097	35.109	71.129	369.057
CFG	6	3	0.32	2.696	5.441	9.028	37.121	72.181	354.763
CFG	6	4	6.053	2.621	5.53	8.96	37.515	73.744	367.655
CFG	6	5	156.141	47.595	50.767	54.054	81.957	116.699	397.522
Markov	6	2	—	1.284	3.936	7.469	34.533	68.029	337.949
Markov	6	3	—	2.218	4.839	8.28	35.406	69.88	347.756
Markov	6	4	—	4.833	7.926	10.874	38.463	81.074	373.771
	d	p	GEN	10000	50000	100000	500000	1000000	5000000
CFG	7	2	0.21	0.808	3.646	7.011	35.11	70.911	361.186
CFG	7	3	0.419	0.84	3.607	7.156	36.138	71.242	361.462
CFG	7	4	6.182	3.022	5.459	9.041	37.589	73.397	364.075
CFG	7	5	158.024	53.649	52.981	56.456	84.178	119.034	402.945
Markov	7	2	—	1.648	4.361	7.753	34.48	68.228	349.683
Markov	7	3	—	2.864	5.575	9.602	37.016	71.108	346.98
Markov	7	4	—	8.19	10.772	13.921	43.673	79.005	379.443
	d	p	GEN	10000	50000	100000	500000	1000000	5000000
CFG	8	2	0.588	0.905	3.662	7.094	36.179	71.394	369.456
CFG	8	3	0.818	0.939	3.71	7.338	36.871	74.814	362.259
CFG	8	4	6.758	3.015	5.182	8.512	34.945	68.242	335.426
CFG	8	5	159.983	51.749	49.595	53.055	78.314	110.507	374.249
Markov	8	2	—	2.462	5.029	8.441	34.9	68.778	337.725
Markov	8	3	—	4.679	7.333	10.563	38.329	71.626	345.412
Markov	8	4	—	12.857	16.485	24.572	54.914	87.455	428.908

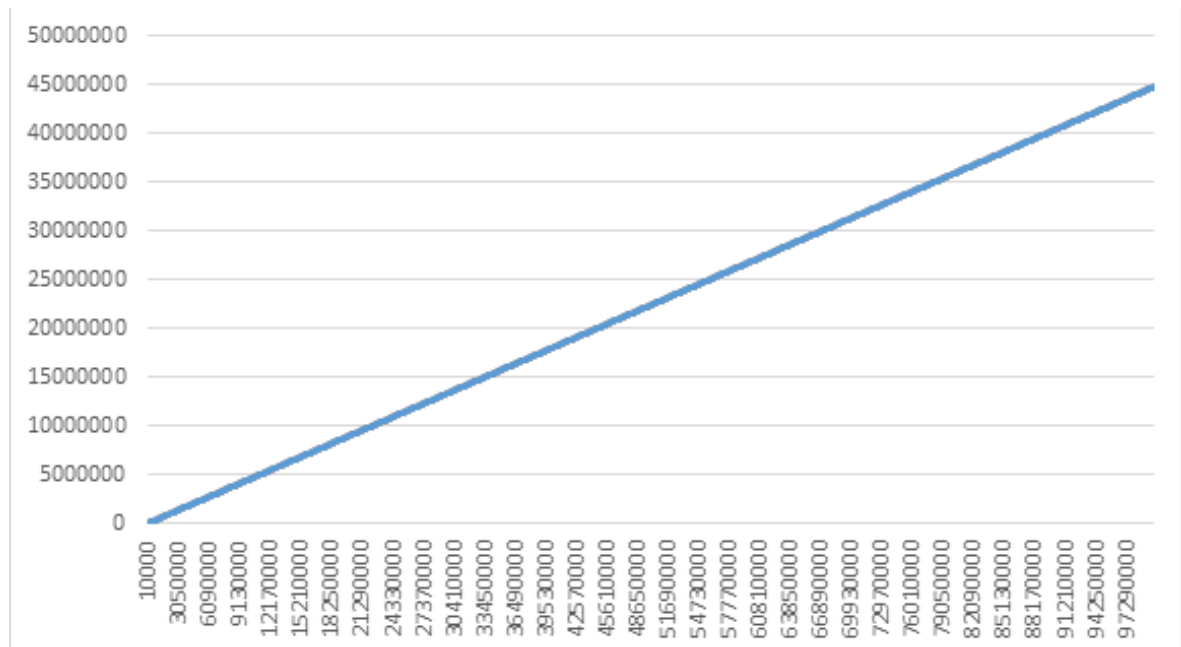
## 6.2 Výstupné heslá

Po overení časovej zložitosti generovania hesiel pomocou jednotlivých algoritmov sme na výsledné heslá aplikovali viaceré metriky. Cieľom týchto testov bolo ukázať výhody a slabiny jednotlivých algoritmov a spraviť ich vzájomne porovnanie v zmysle šancí na nájdenie hľadaného heslá. Vzhľadom na to, že v praxi sa trendy medzi používanými heslami môžu meniť a časom by mohla drvivá väčšina ľudí používať bezpečné heslá, tieto metriky nie sú záväzne a nevytvádzajú o tom ako sa budú jednotlivé algoritmy správať ak by boli použité v praxi.

### 6.2.1 Heslá zo vstupného slovníka

Ako prvú metriku sme skúmali koľko hesiel zo slovníka gramatika vygenerovala po vygenerovaní určitého počtu hesiel. Na grafoch, ktoré boli výstupom tohto testu sme na vodorovnej osi znázornili počet hesiel vygenerovaných gramatikou zatiaľ čo na vertikálnej osi je percento hesiel slovníka, ktoré sa medzi nimi nachádzajú. Pri tomto teste sme nechali oba algoritmy generovať 100 miliónov hesiel k čomu bol použitý slovník obsahujúci 13 331 008 rôznych hesiel dĺžky 12 a menej znakov.

Nižšie vidíme grafy znázorňujúce vývoj počtu vygenerovaných hesiel, ktoré sa nachádzajú v slovníku. Zatiaľ čo prvý zobrazuje koľko unikátnych hesiel bolo vygenerovaných pomocou algoritmu využívajúceho Markovovské zdroje, tie následovne ukazujú vyššie popísanú metriku ohľadom počtu vygenerovaných hesiel patriacich do vstupného slovníka.

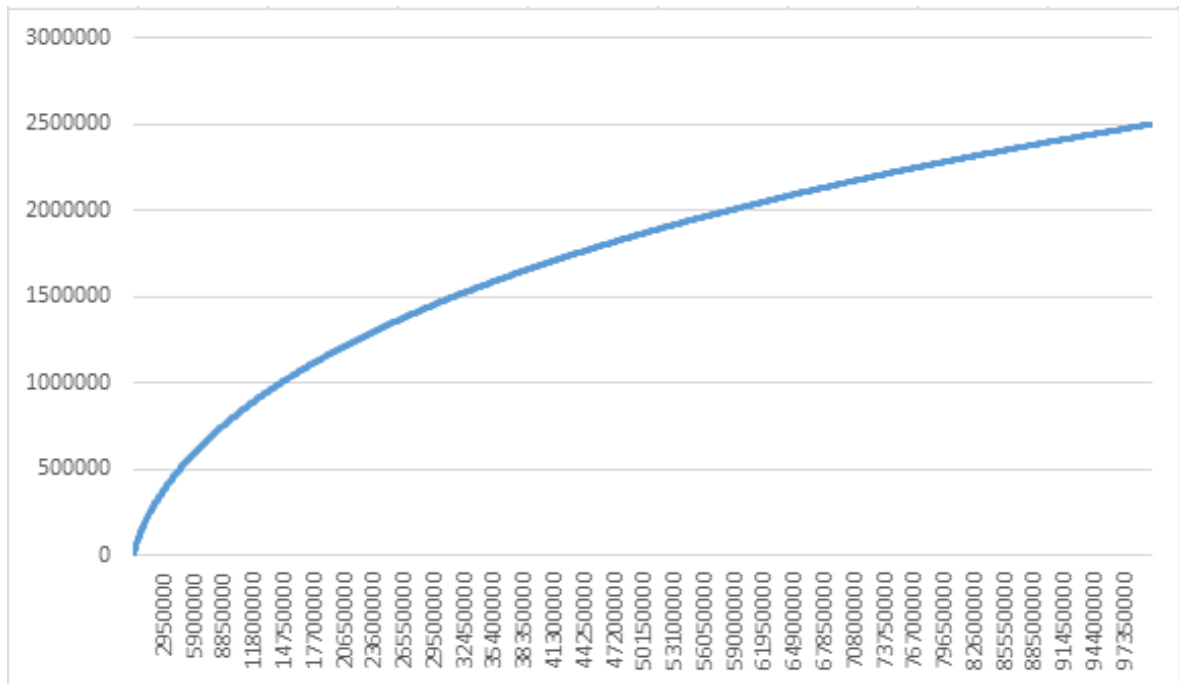


Obr. 6.1: Počet vygenerovaných hesiel zo slovníka - Markovovský zdroj

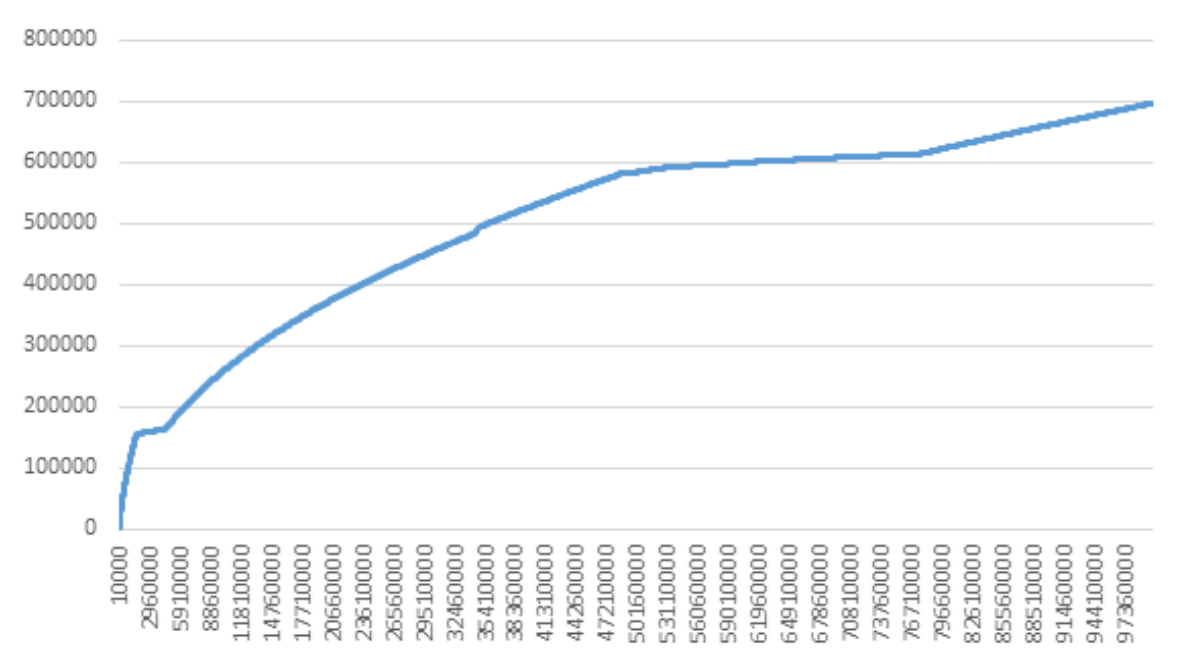
Tento algoritmus vygeneroval len niečo málo cez 42 miliónov unikátnych hesiel. Z týchto unikátnych hesiel sa približne 2,5 milióna nachádza vo vstupnom slovníku. Z týchto grafov môžeme vidieť, že Markovov zdroj má problém so zbytočným generovaním veľkého množstva hesiel duplicitne. Naopak je vidieť, že učiaci algoritmus založený na prefixoch o veľkosti  $k$  je efektívny čo sa týka napodobenia pravdepodobností, ktoré dostal na vstupe.

Na 6.3 vidíme priebeh hodnôt pre nami definované a implementované riešenie pomocou bezkontextových gramatík. Toto riešenie má na rozdiel od Markovovho zdroja omnoho pomalší rast počtu hesiel patriacich do slovníka. Pri 100 miliónoch generovaných hesiel to je niečo málo pod 700 tisíc. Dôvod pre takéto relatívne malé percento vygenerovaných hesiel zo slovníka môže byť práve vlastnosť gramatiky učiť sa vzory hesiel. Keďže vo vstupnom slovníku existovalo málo hesiel, ktoré mali obrovský počet výskytov, gramatika sa zamerala na generovanie hesiel s veľmi podobným vzorom.

Ďalej sme taktiež skúmali ako sa správajú nami implementované algoritmy na menších dátach. Na obrázku 6.4 je znázornený graf priebehu generovania hesiel, ktoré sa nachádzajú vo vstupnom slovníku. Na vodorovnej osi je ukázaný počet vygenerovaných hesiel, v tomto prípade to bolo 5 miliónov hesiel. Výška čiar určuje množstvo



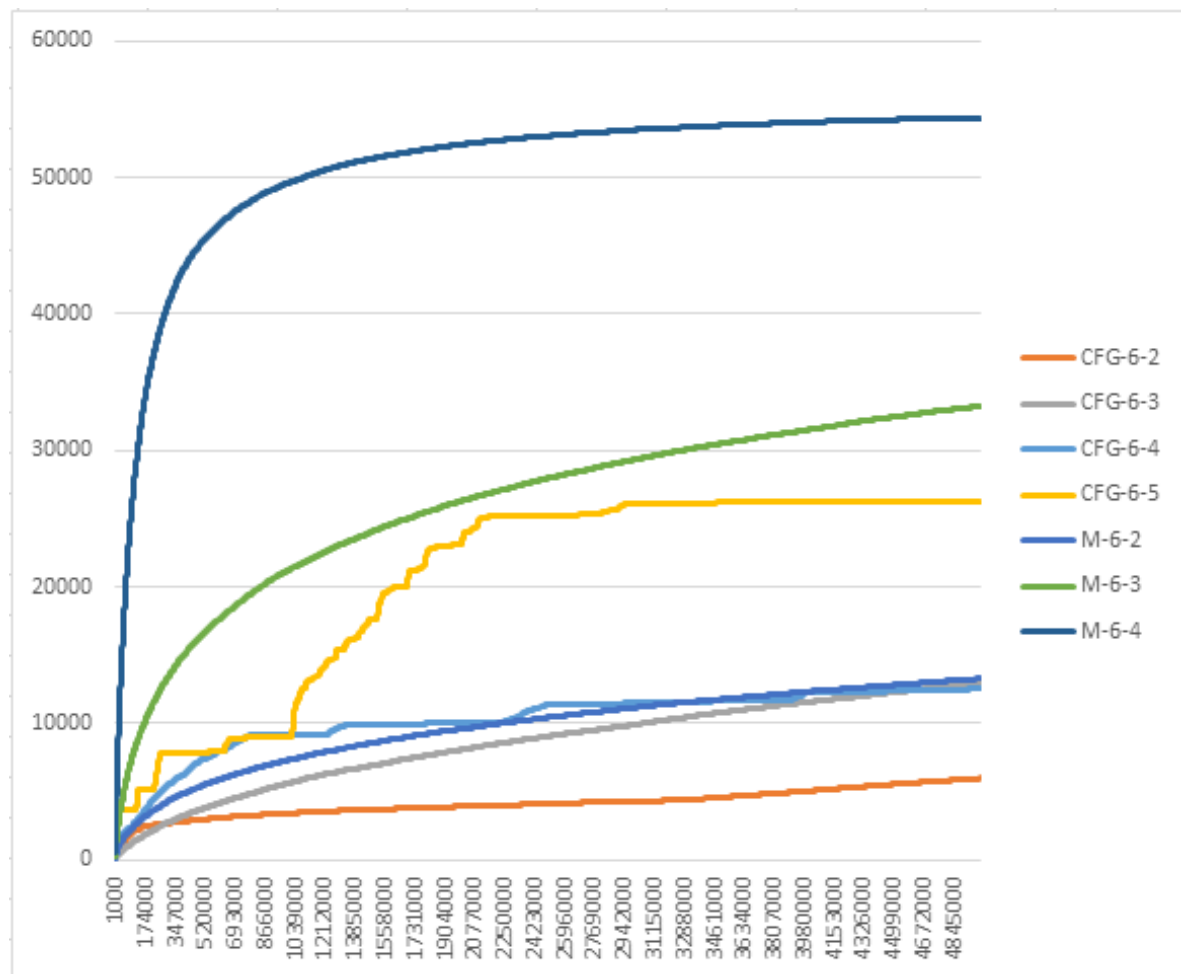
Obr. 6.2: Počet vygenerovaných hesiel zo slovníka - Markovovský zdroj - Unikátné



Obr. 6.3: Počet vygenerovaných hesiel zo slovníka - Gramatika

hesiel, ktoré boli nájdené vo vstupnom slovníku. Pre Markovovské zdroje sa toto číslo počíta z počtu unikátnych hesiel, ktoré boli vygenerované. Všimli sme si, že priebehy jednotlivých algoritmov sa náramne podobajú logaritmickému krivke. Taktiež si môžeme všimnúť, že algoritmus používajúci Markovovské zdroje je v tejto metrike opäť lepší ako algoritmus používajúci pravdepodobnostné bezkontextové gramatiky. Použili sme slovník phpbb stiahnutý z [2], ktorý sme upravili aby všetky heslá mali dĺžku najviac 6 znakov. Takto upravený slovník mal nakoniec 55 744 rôznych hesiel.



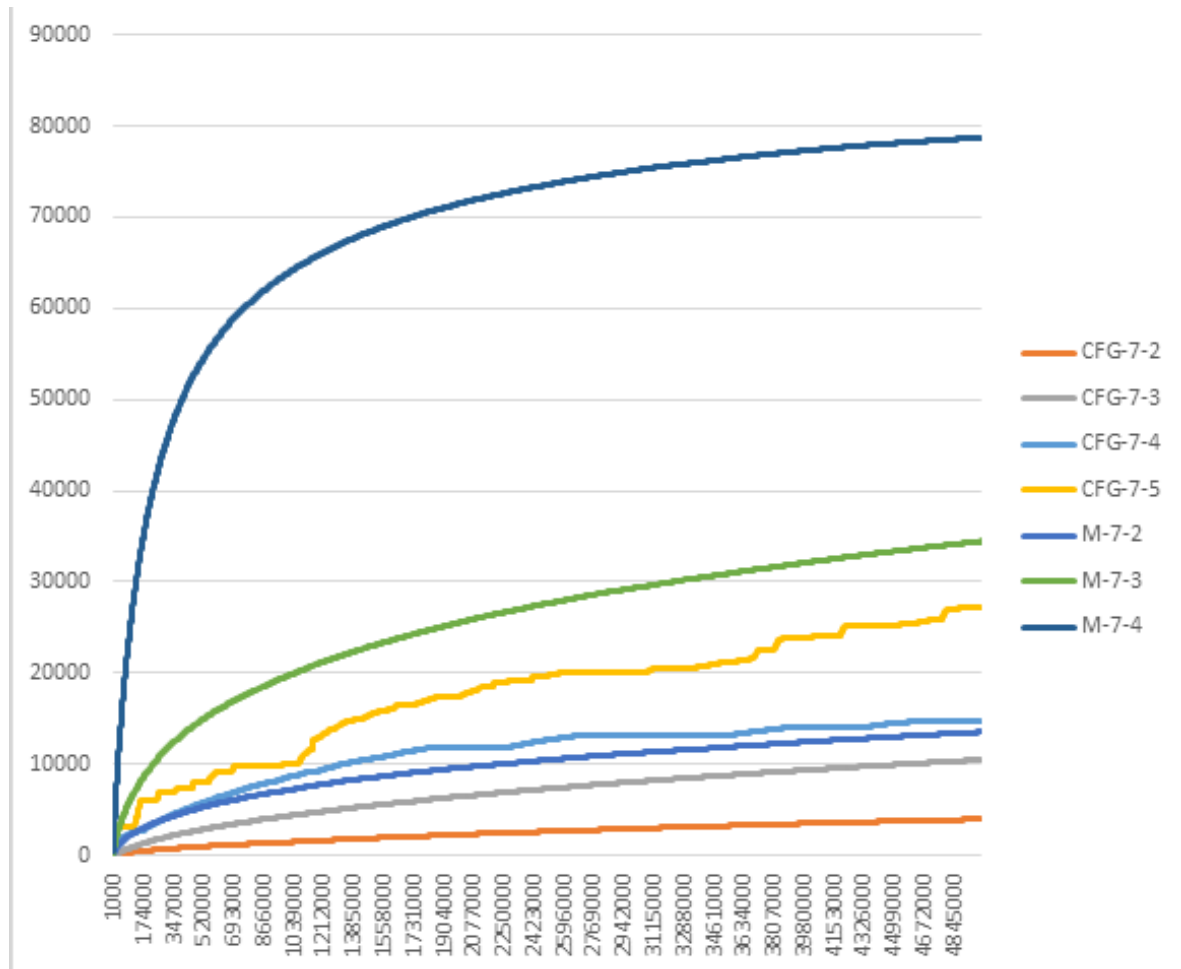


Obr. 6.4: Počet vygenerovaných hesiel zo slovníka - dĺžka 6

Po zhliadnutí rovnakého grafu pre algoritmy pustené pre generovanie hesiel dĺžky 7 a s upraveným slovníkom phpbb obsahujúcim heslá najviac dĺžky 7 sme si všimli, že množstvo hesiel zo vstupného slovníka generovaných našimi algoritmami sa nezmenilo až na Markovovské zdroje s prefixom 4. V tomto teste bola veľkosť vstupného slovníka 88 416 hesiel.

Nakoniec sme tento istý test opäť spustili na všetkých algoritmoch. Tentokrát vstupné parametre a slovník boli nastavené na generovanie hesiel maximálnej dĺžky 8. Takto upravený slovník phpbb obsahoval 143 675 hesiel z ktorých až 100 tisíc bolo vygenerovaných algoritmom používajúcim Markovovské zdroje s prefixom nastaveným na dĺžku 4. Avšak taktiež ako pri dĺžke 7, ostatné inštancie algoritmov nepresiahli ani 40 tisíc vygenerovaných hesiel.

Na základe 6.2 a 6.3 vidíme, že nami navrhnutá metóda pomocou bezkontextových gramatík síce negeneruje veľa hesiel zo vstupného slovníka počas prvých miliónov vygenerovaných hesiel. Tento problém by sa dal vyriešiť, tým že by sme vždy ako prvé na výstup poslali všetky heslá zo slovníka, keďže ten býva zanedbateľne malý oproti veľkosti priestoru hesiel, ktorý musíme prehľadať aby sme definitívne našli hľadané heslo.



Obr. 6.5: Počet vygenerovaných hesiel zo slovníka - dĺžka 7

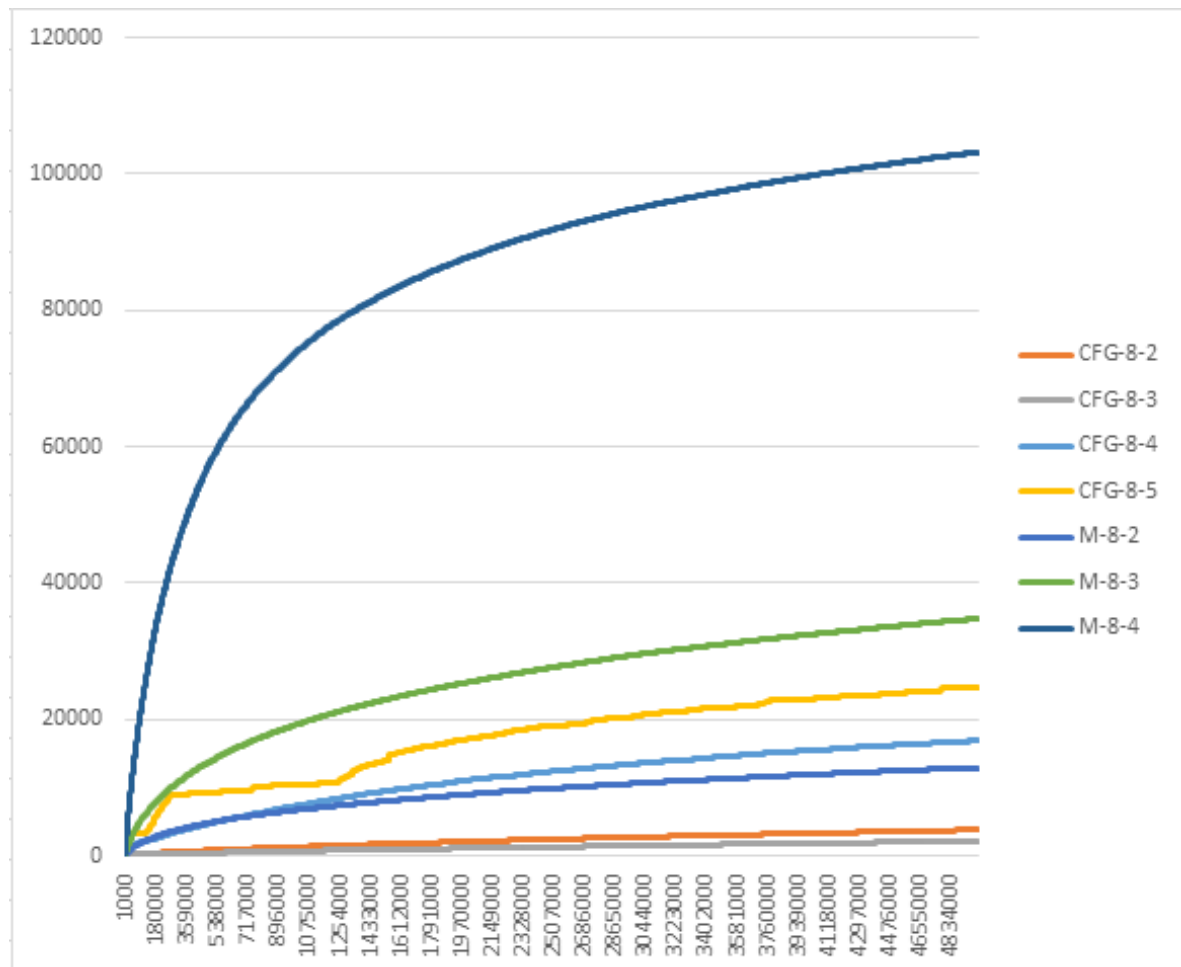
Pri analyzovaní týchto dát nemôžeme zabudnúť, že terajšia verzia algoritmu používajúceho Markovovský zdroj negeneruje celý priestor hesiel určenej dĺžky.

### 6.2.2 Miery presnosti

Pod týmto pojmom rozumieme metriky popisujúce nielen kvantitu nami generovaných hesiel patriacich do slovníka, ale snažia sa bližšie vyhodnotiť ako rýchlo sa gramatika dostane k heslám, ktoré boli podľa vstupného slovníka označené za najpravdepodobnejšieho.

Stĺpce tabuľky 6.2 vyjadrujú hodnoty jednotlivých metrík pre daný algoritmus.

- *PPS* - Priemerná Pozícia v Slovníku - Vyjadruje priemernú pozíciu vo vstupnom slovníku hesiel, ktoré boli vygenerované algoritmom na výstupe
- *PPS %* - Priemerná Pozícia v Slovníku - Vyjadruje percentuálnu pozíciu vrámci slovníku hesiel, ktoré boli vygenerované algoritmom na výstupe



Obr. 6.6: Počet vygenerovaných hesiel zo slovníka - dĺžka 8

- *RPSV* - Rozdiel Pozície v Slovníku a na Výstupe - Rozdiel v pozícií na vstupe a na výstupe algoritmu prenasobený percentuálnym počtom výskytov vo vstupnom slovníku
- *OPSV* - Odchýlka Pozície v Slovníku a na Výstupe - Absolútna hodnota rozdielu v pozícií na vstupe a na výstupe algoritmu prenasobená percentuálnym počtom výskytov vo vstupnom slovníku

$$\frac{\sum_{i=1}^k ((indG_i - indS_x) * \frac{v_{ind_x}}{\sum_{j=1}^n v_j})}{k}$$

Vzorec vyjadrujúci mieru RPSV, kde  $n$  vyjadruje počet hesiel vo vstupnom slovníku a  $k$  je počet výskytov hesiel zo vstupného slovníka medzi generovanými. Hodnota  $indG_i$  určuje poradie  $i$ -tého heslá nachádzajúceho sa vo vstupnom aj výstupnom slovníku rámci generovaného slovníka. Hodnota  $indS_i$  vyjadruje tú istú hodnotu pre vstupný slovník. Hodnoty  $v_i$  sú počty výskytov hesiel zadefinované vo vstupnom slovníku.

Tabuľka 6.2: Miery presnosti

	d	p	PPS	PPS %	RPSV
CFG	6	2	31328	56.199	36.567
CFG	6	3	26358	47.285	37.749
CFG	6	4	31033	55.670	19.673
CFG	6	5	28554	51.224	21.397
CFG	7	2	47454	53.672	26.489
CFG	7	3	40343	45.629	25.831
CFG	7	4	46739	52.863	16.648
CFG	7	5	46718	52.840	21.715
CFG	8	2	88521	61.612	16.139
CFG	8	3	61113	42.536	22.384
CFG	8	4	70500	49.069	15.028
CFG	8	5	75570	52.598	14.749
Markov	6	2	25219	45.241	28.307
Markov	6	3	25959	46.568	14.815
Markov	6	4	27705	49.701	3.649
Markov	7	2	39090	44.212	21.289
Markov	7	3	40283	45.561	13.321
Markov	7	4	43232	48.896	4.700
Markov	8	2	61688	42.936	15.745
Markov	8	3	66401	46.216	9.701
Markov	8	4	68291	47.532	4.596
CFG	12	4	3781788	28.3	5.879
Markov	12	4	4609712	34.5	2.191

Vzorec pre mieru OPSV je takmer identický s vyššie uvedeným vzorcom, jediný rozdiel je v absolútnej hodnote rozdiel medzi pozíciami na vstupe a výstupe.

$$\frac{\sum_{i=1}^k (|indG_i - indS_x| * \frac{v_{ind_x}}{\sum_{j=1}^n v_j})}{k}$$

Priemerná pozícia v slovníku ukazuje ako pravdepodobné heslá v priemernom prípade generuje náš algoritmus. Keďže toto číslo sa často krát zdá veľké, prikkladáme k nemu v druhom stĺpci jeho percentuálnu hodnotu. Hodnoty  $d$  a  $p$  vyjadrujú maximálnu dĺžku hesiel v slovníku a dĺžku použitých prefixov v Markovovskom zdroji. Pre hodnoty 6,7,8 parametru  $d$  sme použili slovník *phppbb* upravený na heslá relevantnej dĺžky. Pre hodnoty  $d$  rovné 12 sme použili omnoho robustnejší slovník *rockyou*, skladajúci sa z takmer 14 miliónov unikátnych hesiel.

Z tabuľky 6.2 môžeme vidieť že obom našim algoritmom prospieva navýšenie vstupnej informácie o heslách. Toto je vidieť na jak na percentuálnych hodnotách priemernej pozície v slovníku tak aj na rozdieloch pozícií medzi vstupným a vygenerovaným slovní-

kom. Hodnota rozdielov pozícií má vyjadrovať presnosť generovania hesiel v správnom poradí, alebo v poradí blízkom tomu zo vstupného slovníka.

### 6.2.3 Heslá z iného slovníka

Ako ďalšie sme s rozhodli simulovať postupné generovanie hesiel zo slovníka, ktorý je rôzny od toho aký sme dostali na vstupe. Pri tomto teste sme zobrali heslá vygenerované našimi algoritmami, ktoré na vstupe dostali slovník *rockyou*. Následne sme spravili testy počtu vygenerovaných hesiel, tentokrát avšak z iného ako vstupného slovníka. V tomto prípade sme použili slovník *phpbb*. Týmto by sme chceli ukázať schopnosť nášho algoritmu vygenerovať slovník použiteľný v praxi.

Výstupný graf opäť na vodorovnej osi zobrazuje počet hesiel vygenerovaných našim algoritmom. Na zvislej osi je zobrazené percento hesiel z cieľového slovníka, ktoré boli vygenerované našim algoritmom. Dá sa pozorovať, že algoritmus využívajúci bežne definované Markovovské zdroje stále našiel najvyššie percento hesiel z cieľového slovníka. Avšak v prípade upravenia Markovovského zdroja, tak aby generoval všetky možné heslá vidíme, že ... TODO(dopocitava sa)

## Diskusia

### 7.1 Možnosti zlepšenia nášho riešenia

#### 7.1.1 Izolovanie kódu na skúšanie kandidátov

Bohužiaľ sa nám kvôli nedostatku času nepodarilo izolovať kód programu TrueCrypt, ktorý je zodpovedný za overenie správnosti hesla zadaného používateľom. Nájdenie tohto kódu by mohlo pomôcť pri implementácii rýchleho algoritmu na overovanie nami generovaných kandidátov. Tento nedostatok sa dá avšak nahradiť použitím niektorého z voľne dostupných programov určených na útoky hrubou silou, ktorému ako vstupný slovník dodáme slovník vygenerovaný nami implementovaným programom.

#### 7.1.2 Veľkosť potrebnej pamäte

Najväčším nedostatkom samotného algoritmu využívajúceho bezkontextové gramatiky je množstvo pamäte potrebné na jeho beh. Tento nedostatok spôsobuje možnosť použitia tohto algoritmu len v prostredí s obrovským množstvom RAM. Odstránenie tohto nedostatku vyžaduje vývoj algoritmu, ktorý dokáže generovať terminálne vetné formy gramatiky s použitím malého množstva pamäte.

#### 7.1.3 Kompletne generujúci Markovovský zdroj

Podarilo sa nám implementovať Markovovský zdroj, ktorý v konečnom čase vygeneruje všetky možné heslá zo vstupnej abecedy. Pri riešení tohto problému sme zadefinovali konštanty  $\delta$  a  $\varepsilon$ , ktoré slúžia na zadefinovanie prechodov medzi známymi a neznámymi stavmi. Možný vylepšením by bolo upravovanie týchto konštánt podľa počtu hesiel, ktoré sme už vygenerovali, aby sme zvýšili šancu vygenerovania zvyšku existujúcich hesiel. Toto avšak zahŕňa výskum, ktorý by sa dal rozobrať v samostatnej práci.

## Záver

V tejto práci sme sa zaoberali útokmi hrubou silou na program TrueCrypt. Podrobne sme si naštudovali fungovanie tohto programu a usúdili, že najlepším miestom útoku, na disku zašifrovaný týmto programom, bude používateľské heslo pomocou ktorého sa generujú šifrovacie kľúče. Na základe tohto poznatku sme si naštudovali možnosti útokov hrubou silou, ktoré by sme mohli pri riešení nášho problému použiť.

Pri implementácii nášho riešenia používajúceho bezkontextové gramatiky sme používali znalosti o používateľských heslách k nastaveniu nášho algoritmu tak aby optimalizoval poradie generovaných hesiel podľa pravdepodobností ich správnosti. Tento algoritmus sme následne porovnávali so zaužívanou metódou Markovovských zdrojov. Keďže náš algoritmus spĺňal konkrétnejšie podmienky pri generovaní hesiel ako sú determinizmus, generovanie celého priestoru hesiel a vyhnutie sa duplikátom navrhli sme zmeny v modeli Markovovských zdrojov. Tieto zmeny sme taktiež implementovali a výsledný algoritmus sme taktiež porovnali s našim riešením.

Nami navrhnutá metóda využívajúca bezkontextové gramatiky dopadla v testoch veľmi podobne ako Markovovské zdroje, čo hodnotíme ako pozitívny výsledok tejto práce. Nedostatky tejto metódy a jej aplikácie na program TrueCrypt sme rozobrali v kapitole Diskusia.

# Literatúra

- [1] Francesco Bergadano, Bruno Crispo, and Giancarlo Ruffo. High dictionary compression for proactive password checking. *ACM Trans. Inf. Syst. Secur.*, 1(1):3–25, November 1998.
- [2] Ron Bowes. Slovníky s heslami - <https://wiki.skullsecurity.org/Passwords>, 2015.
- [3] François Delebecque and Jean-Pierre Quadrat. Optimal control of markov chains admitting strong and weak interactions. *Automatica*, 17(2):281–296, 1981.
- [4] James Forshaw. Truecrypt 7 derived code/windows: Drive letter symbolic link creation eop - <https://bugs.chromium.org/p/project-zero/issues/detail?id=538>, 2015.
- [5] James Forshaw. Truecrypt 7 derived code/windows: Incorrect impersonation token handling eop - <https://bugs.chromium.org/p/project-zero/issues/detail?id=537>, 2015.
- [6] E. F. Gehring. Choosing passwords: security and human factors. In *Technology and Society, 2002. (ISTAS'02). 2002 International Symposium on*, pages 369–373, 2002.
- [7] Aaron L. F. Han, Derek F. Wong, and Lidia S. Chao. Password cracking and countermeasures in computer security: A survey. *CoRR*, abs/1411.7803, 2014.
- [8] Cynthia Kuo, Sasha Romanosky, and Lorrie Faith Cranor. Human selection of mnemonic phrase-based passwords. In *Proceedings of the Second Symposium on Usable Privacy and Security*, SOUPS '06, pages 67–78, New York, NY, USA, 2006. ACM.
- [9] David Malone and Kevin Maher. Investigating the distribution of password choices. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 301–310, New York, NY, USA, 2012. ACM.
- [10] Simon Marechal. Advances in password cracking. *Journal in computer virology*, 4(1):73–81, 2008.



- [11] Eugene H Spafford. Observing reusable password choices. 1992.
- [12] ZDNet. 25 gpus devour password hashes at up to 348 billion per second - <http://www.zdnet.com/article/25-gpus-devour-password-hashes-at-up-to-348-billion-per-second>, 2012.
- [13] Moshe Zviran and William J. Haga. A comparison of password techniques for multilevel authentication mechanisms. *The Computer Journal*, 36(3):227–237, 1993.