Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Metódy útoku hrubou silou na TrueCrypt

Diplomová Práca

2016 **Martin Strapko**

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.

2016 **Martin Strapko**

Poďakovanie

Abstrakt

Táto práca sa zaoberá problémom optimalizácie útokov hrubou silou. Rozoberá možností útoku na program TrueCrypt. Taktiež sa sústredí na to ako využiť znalostí o často používaných heslách na vylepšenie efektívnosti útokou 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.

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

Abstract

Focus of this work is on optimizing brute-force attacks. It studies diffrent 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

Úvod **1**

1 TrueCrypt **2**

2 Útoky hrubou silou **4**
2.1 Inkrementálny útok 4
2.2 Slovníkový útok 5
2.2.1 Prekrúcanie slov 5
2.3 Hybridný útok 5

3 Používateľské heslá **7**

4 Učenie **8**
4.1 Pravdepodobnostné bezkontextové gramatiky 8
4.2 Markovské zdroje 9

5 Implementácia **10**
5.1 Bezkontextová gramatika 10
5.1.1 Tvorba gramatiky 10
5.1.2 Počítanie pravdepodobností 11
5.1.3 Generovanie hesiel 14
5.2 Markovský zdroj 16

6 Testy **18**
6.1 Časová náročnosť 18
6.2 Výstupné heslá 18
6.2.1 Heslá zo slovníka 19
6.2.2 Miery presnosti 23

Literatúra **26**

Zoznam listingov

5.1 Úprava pravidiel na základe vstupného slova 12
5.2 Pripočítanie výskytov k podmnožinám zložených neterminálov 13
5.3 Generovanie všetkých susedných vektorov 16

Zoznam obrázkov

1.1 Schéma XTS algoritmu 3

6.1 Počet vygenerovaných hesiel zo slovníka - Markovský zdroj 20
6.2 Počet vygenerovaných hesiel zo slovníka - Markovský zdroj - Unikátne . . 21
6.3 Počet vygenerovaných hesiel zo slovníka - Gramatika 21
6.4 Počet vygenerovaných hesiel zo slovníka - dĺžka 6 22
6.5 Počet vygenerovaných hesiel zo slovníka - dĺžka 7 23
6.6 Počet vygenerovaných hesiel zo slovníka - dĺžka 8 24

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 ním 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 ním môžu mať aj iný ľudia. Na toto primárne slúži šifrovanie pomocou kľúča.

Bezpečnosť tohto šifrovania záleží vysoko na utajený 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 záleží 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 ľubovolných 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*.

KAPITOLA 1

TrueCrypt

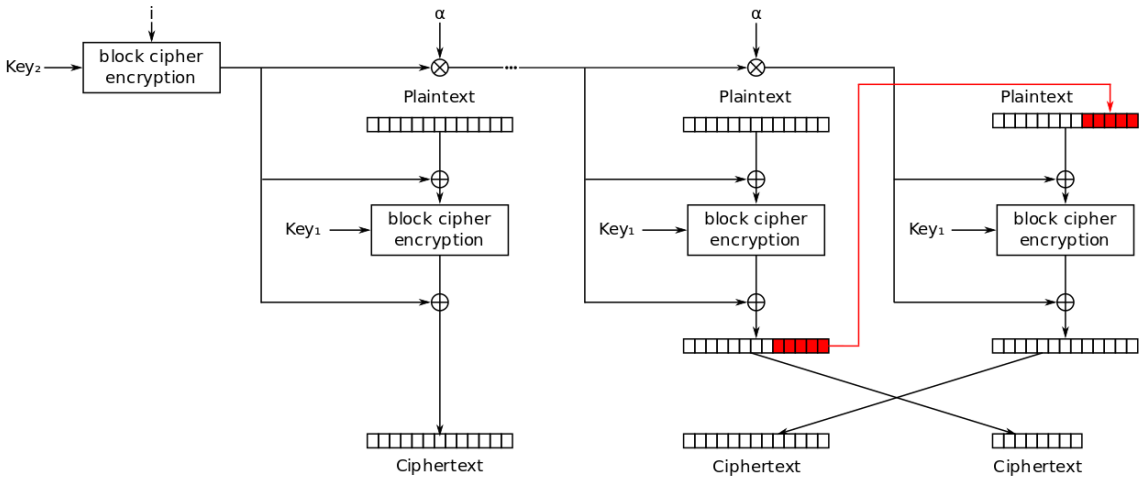
TrueCrypt je šifrovací program poskytujúci používateľovi možnosť zašifrovať disk alebo jeho ľubovoľnú časť v počítači pomocou používateľom zvoleného hesla. Vývoj tohto programu bol ukončený v roku 2014 a podľa autorov nie je bezpečný, nakoľko jeho implementácia môže obsahovať bezpečnostné chyby. Následný 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, závažnejšie chyba, umožňuje útočníkovi prístup k zvýšením právam zneužitím tvorby symbolického odkazu na písmena 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-

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

KAPITOLA 2

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 ľubovolných znakov ASCII tabuľky. V praxi je veľmi nepravdepodobné, že používateľ bude mať takéto dlhé heslo založené na tak veľkej 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, číslic 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 mávajú maximálnu dĺžku okolo 16 znakov a sú zložene 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 menšie 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 záleží 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 docieľuje 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 popísanie týchto pravidiel.

2.3 Hybridný útok

V tejto práci sa venujeme implementácií útoku, ktorý je spojením vyššie uvedených. Našim hlavným cieľom je nájdenie správneho hesla k particií zašifrovanej programom TrueCrypt. Keďže chceme toto heslo nájsť v ľubovoľné 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 podobať 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. Náš algoritmus si na základe týchto

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

KAPITOLA 3

Používateľské heslá

Aj napriek tomu ako sú používateľské heslá vo všeobecnosti napadnuteľné dnes sú najčastejšie používaný spôsob autentifikácie používateľa. Tento trend sa pravdepodobne ani v najbližšej budúcnosti nebude meniť. Každý rok pribúda viac a viac hesiel zatiaľ sa sa stávajú čím ďalej tým ľahšie napadnuteľné.

KAPITOLA 4

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 Markovské 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 zadefinovať počiatočný neterminál z ktorého sa bude každá vetná forma odvádzať. Nakoniec potrebujeme poznať množinu prepisovacích pravidiel, ktoré definujú spôsob akým sa menia neterminály na ďalšie vetné formy. Pri bezkontextových gramatikách majú prepisovacie pravidlá tvar

$$N -> (N \cup \Sigma)^*$$

kde N vyjadruje množinu neterminálov a Σje množina terminálov. Tieto pravidlá vyjadrujú schopnosť neterminálu zmeniť sa na ľubovolnú 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 ľubovolnej vetnej formy získame súčinom pravdepodobností prepisovacích pravidiel použitých na jej odvodenie.

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

4.2 Markovské 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 Markovskú vlastnosť, ktorá je popísaná ako takzvaná bezpamäťovosť. Hovorí o tom, že distribúcia pravdepodobností následujúceho 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 Markovský zdroj poráta pravdepodobností následujúcich znakov. Pri tomto spôsobe učenia sa dá pre ľubovolný stav vypočítať pravdepodobnosť s akou sa Markovský zdroj dostane do tohto stavu.

KAPITOLA 5

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 zterminalnenie 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 zadefinovať maximálnu veľkosť jednoduchého neterminálu, vyjadrujúcu maximálné povolené k.

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

Zložené neterminály Jednoduché neterminály nám pomáhajú vyjadrovať sekvenciu znakov práve jedného z vyššie vymenovaných typov. Aby sme boli schopný popísať ľubovolný 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 písmena, ukončené štvoricou cifier.

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 (popísane 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 cifier ako napríklad jedno z najpoužívanejších *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 prispôsobiť 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 pravdepodobností jednotlivých pravidiel našej gramatiky. Našim 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 používanie nášho algoritmu s rôznymi slovníkmi nevyžaduje vyrábanie novej gramatiky až do momentu kedy sa rozhodneme generovať heslá s inou maximálnou dĺžkou. Pravdepodobností 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 implementácií 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.

```
for i in range(1, len(word)):
  if (word[i] in lower) and (currentNet != \'L\'):
    rule += currentNet + str(i-startI)
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    ruleCount[currentNet + str(i-startI)] += occ
    startI = i
    currentNet = \'L\'
    currentSubstring=''
  elif (word[i] in upper) and (currentNet != \'U\'):
    rule += currentNet + str(i-startI)
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    ruleCount[currentNet + str(i-startI)] += occ
    startI = i
    currentNet = \'U\'
    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 prepisujúce počiatočný 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] += occurences
4 ruleCount['Z'] += occurences
```

Zdrojový kód 5.2: Pripočítanie výskytov k podmnožinám zložených neterminálov

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

Všetky vyššie spomenuté metódy na počítanie pravdepodobností pravidiel gramatiky sme implementovali. Ich vzájomne porovnanie ako aj porovnanie s inými bežne používanými programami je vidieť v kapitole Výsledky. Bohužiaľ sme zistili, že gramatika vytvorená naším algoritmom zaberala príliš veľa miesta na disku.

5.1.3 Generovanie hesiel

Dôležitým aspektom používania bezkontextových gramatík je práve spôsob generovania hesiel. Našim hlavným cieľom bolo generovanie hesiel pomocou gramatiky od najpravdepodobnejšieho z nich. Tieto heslá generujeme tak, že počiatočný 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 poradia 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, $p[t+1] = p[t]/pi[t] * pi[t+1]$. 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 pozíciach vyšších alebo rovných ako je aktuálny rád vektora. Týmto budeme generovať terminálne slová zľava doprava a vďaka tomu nebudeme generovať duplikáty, ktoré by sa od seba líšili len v poradí v akom sme rozpísali neterminály na terminály.

Vo vyššie uvedených tabuľkách 5.1 sme demonštrujeme 2 kroky nášho algoritmu na generovanie hesiel. V tomto príklade sa sústredíme na generovanie rôznych terminálnych slov zo zloženého neterminálu $U_1 L_3 D_2$. V ľavej tabuľke môžme vidieť zadefino-

Tabuľka 5.1: Ukážka krokov algoritmu pre neterminál $U_1L_3D_2$

Ľavá strana	Pravá strana	p	i
U_1	A	0.7	0
U_1	B	0.2	1
U_1	C	0.1	2
L_4	minf	0.6	0
L_4	fmfi	0.3	1
L_4	dipl	0.1	2
D_2	47	0.8	0
D_2	42	0.2	1

p	rád	vektor	slovo
0.336	0	[0, 0, 0]	Aminf47
0.168	1	[0, 1, 0]	Afmfi47
0.096	0	[1, 0, 0]	Bminf47
0.084	2	[0, 0, 1]	Aminf42
0.096	0	[1, 0, 0]	Bminf47
0.084	2	[0, 0, 1]	Aminf42
0.056	1	[0, 2, 0]	Adipl47
0.042	2	[0, 1, 1]	Afmfi42

vané prepisovacie pravidla 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 prvky označujúce heslá vzdialené práve na 1 zmenu použitého pravidla. Tieto novo pridané prvky sú automatický 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 pravidlá. Tu si treba všimnúť, že nepridali sme prvok hovoriaci o vektore [1, 1, 0], keďže rád práve vytiahnutého vektora je 1, čiže môžme 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 fronta. 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 rozpoloženia 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 môžme vidieť kus kódu zodpovedný za napĺňanie prioritnej fronty s 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.

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

Zdrojový kód 5.3: Generovanie všetkých susedných vektorov

5.2 Markovský zdroj

Po implementácií vyššie uvedeného algoritmu na generovanie hesiel pomocou pravdepodobnostných bezkontextových gramatík a odhalení nedostatkov čo sa týka pamäťovej náročnosti sme sa rozhodli implementovať ešte jednu metódu. Tou je Markovský zdroj. Ako sme písali v predošlej kapitole, jedná sa o náhodný proces, ktorý spĺňa podmienku bezpamäťovosti. Markovské zdroje sa veľmi často používajú práve pri generovaní prirodzeného jazyka. Práve preto boli vhodný kandidát 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 písane 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 zdroje nič nebráni k tomu vygenerovať viac krát počas behu to isté slovo a nič by mu v tom nemalo ani brániť, vzhľadom na bezpamäťovosť. A na koniec Markovské zdroje nemusia generovať všetky možné reťazce kratšie ako zadaná maximálna dĺžka.

Aj keď prvé dve podmienky nevedia Markovské zdroje splniť už priamo z definície, s treťou sme sa pokúsili niečo vymyslieť. Ako prvé sme sa pokúšali inicializovať všetky počty výskyt 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 pravdepodobností nevidených stavov na nenulové, avšak dostatočne malé aby sa v nich samotný algoritmus necyklil.

KAPITOLA 6

Testy

V predošlej kapitole sme bližšie popísali implementáciu našich algoritmov. Táto kapitola sa zameriava na evalváciu 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 Markovského 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 respektíve dĺžku prefixu podľa ktorého sa rozhoduje Markovský zdroj. 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.

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

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

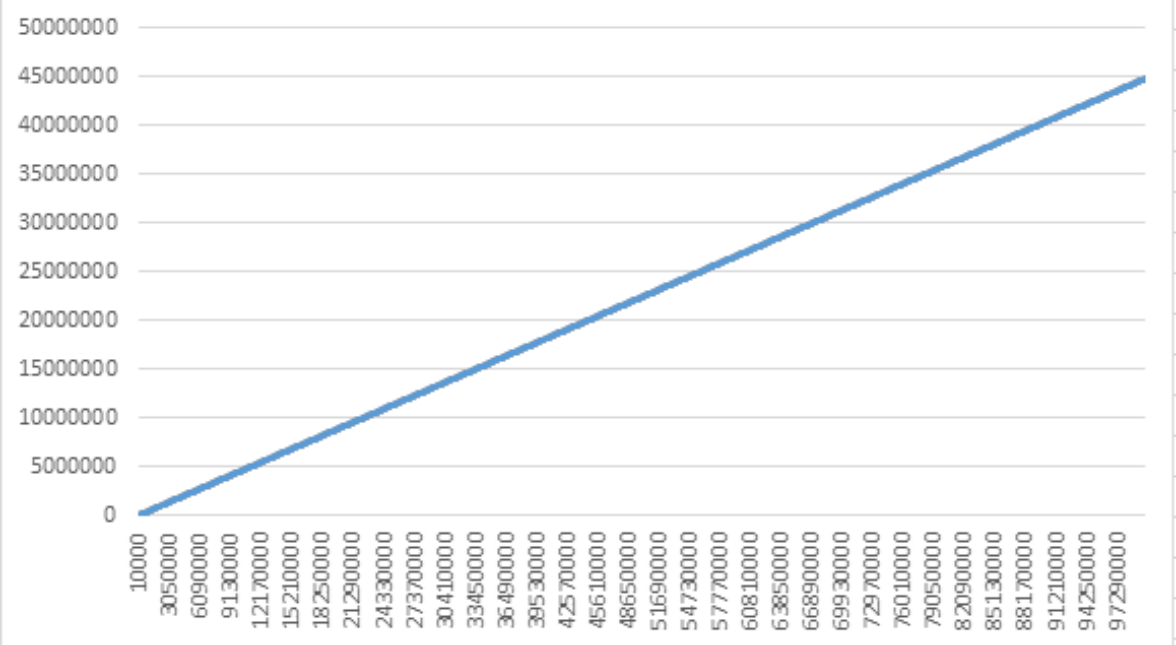
a slabiny jednotlivých algoritmov a spraviť ich vzájomne porovnanie v zmysle šancí na nájdenie hľadaného heslá. Vzhľadom na to, že v praxi sa trendy medzi používanými heslami môžu meniť a časom by mohla drvivá väčšina ľudí používať bezpečné heslá, tieto metriky nie sú záväzne a nevypovedajú o tom ako sa budú jednotlivé algoritmy správať ak by boli použité v praxi.

6.2.1 Heslá zo 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 počet 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. Prvý graf popisuje priebeh trendu v pohľade na všetky vygenerované heslá. Vidíme, že počet hesiel zo slovníka lineárne stúpa s počtom vygenerovaných he-

siel. Vidíme, že zo 100 miliónov vygenerovaných je takmer polovica zložená z hesiel, ktoré sa nachádzajú v slovníku. Toto číslo je trojnásobok všetkých hesiel slovníka, toto je spôsobené tým, že Markov zdroj nekontroluje, ktoré heslá už boli vygenerované a veľké množstvo ich opakuje.

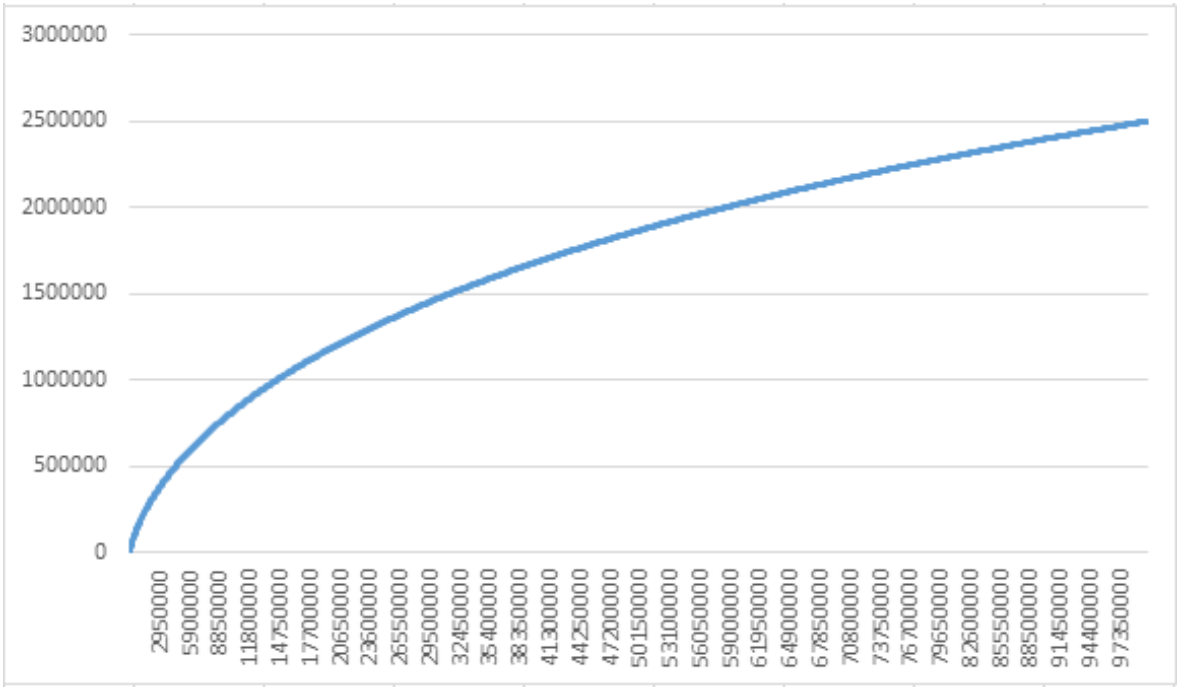


Obr. 6.1: Počet vygenerovaných hesiel zo slovníka - Markovský zdroj

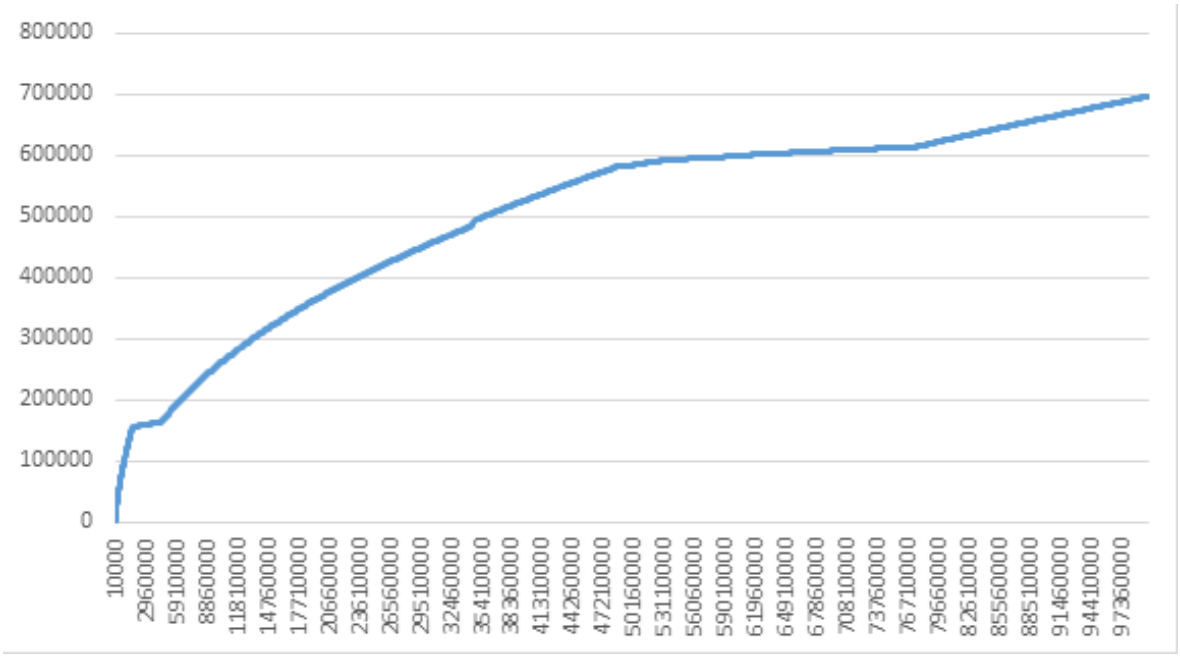
Tento algoritmus vygeneroval len niečo málo cez 42 miliónov unikátnych hesiel. Z týchto unikátnych hesiel sa približne 2,5 milióna nachádza vo vstupnom slovníku. Z týchto grafov môžme vidieť, že Markov zdroj má problém so zbytočným generovaním veľkého množstva hesiel duplicitne. Naopak je vidieť, že 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 Markovho 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 ma 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 hesiel, ktoré boli nájdene vo vstupnom slovníku. Pre Markovské zdroje sa toto číslo počíta z



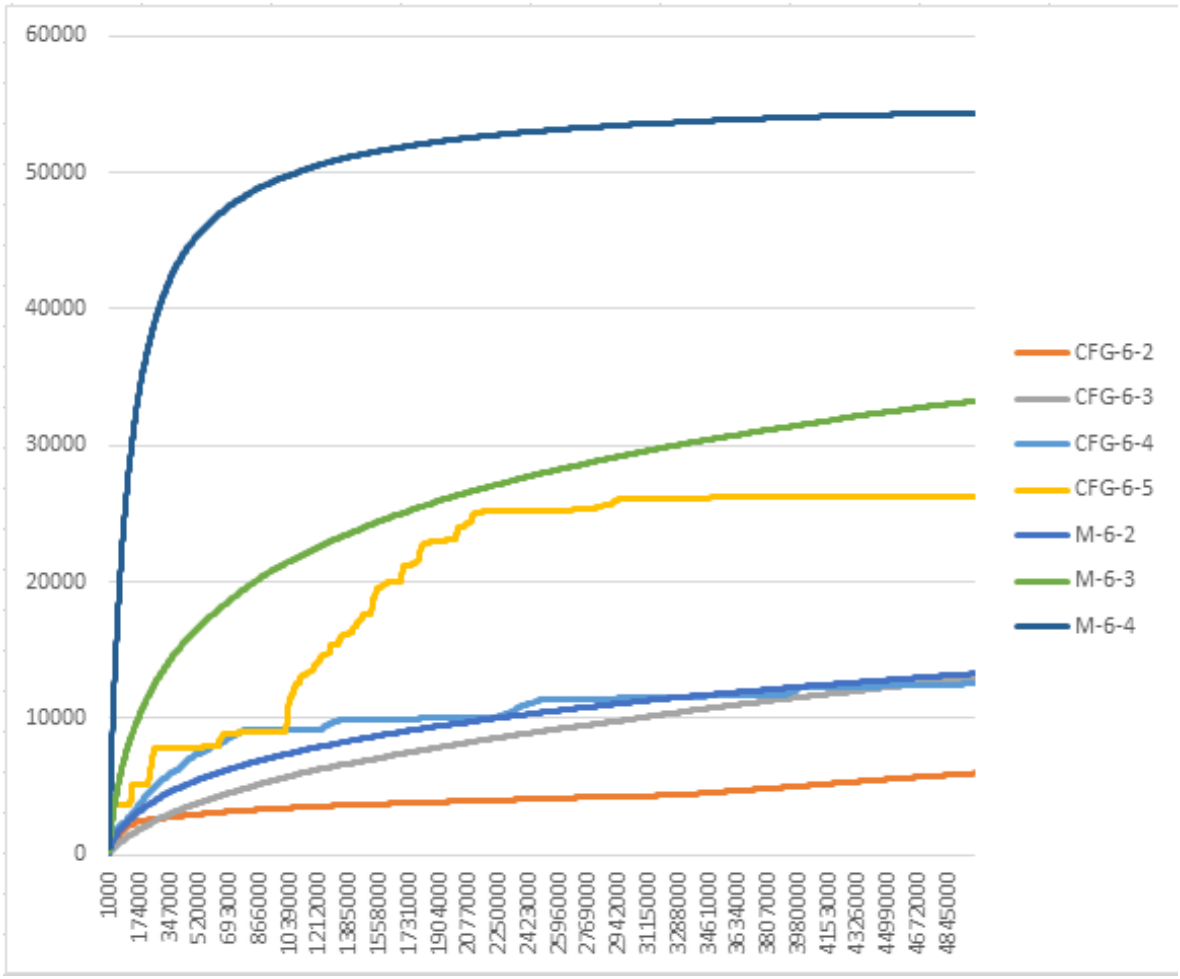
Obr. 6.2: Počet vygenerovaných hesiel zo slovníka - Markovský zdroj - Unikátne



Obr. 6.3: Počet vygenerovaných hesiel zo slovníka - Gramatika

počtu unikátnych hesiel, ktoré boli vygenerované. Všimli sme si, že priebehy jednotlivých algoritmov sa náramne podobajú logaritmickej krivke. Taktiež si môžme všimnúť, že algoritmus používajúci Markovské zdroje je v tejto metriky 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.

Po zhliadnutí rovnakého grafu pre algoritmy pustené pre generovanie hesiel dĺžky

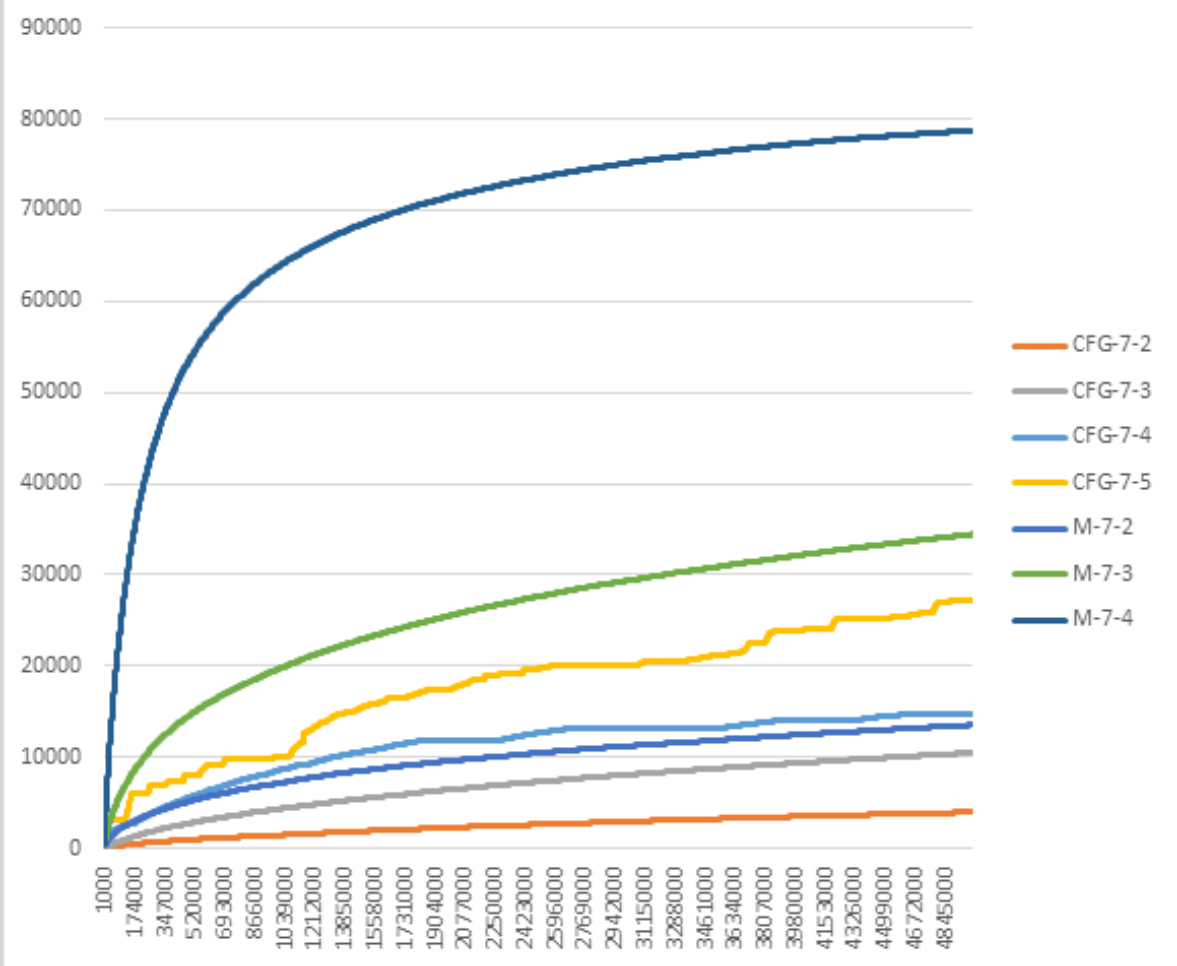


Obr. 6.4: Počet vygenerovaných hesiel zo slovníka - dĺžka 6

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 Markovské 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 slonvík phpbb obsahoval 143 675 hesiel z ktorých až 100 tisíc bolo vygenerovaných algoritmom používajúcim Markovské 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. Pri analyzovaní týchto dát nemôžme zabudnúť, že terajšia verzia algoritmu používajú-



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

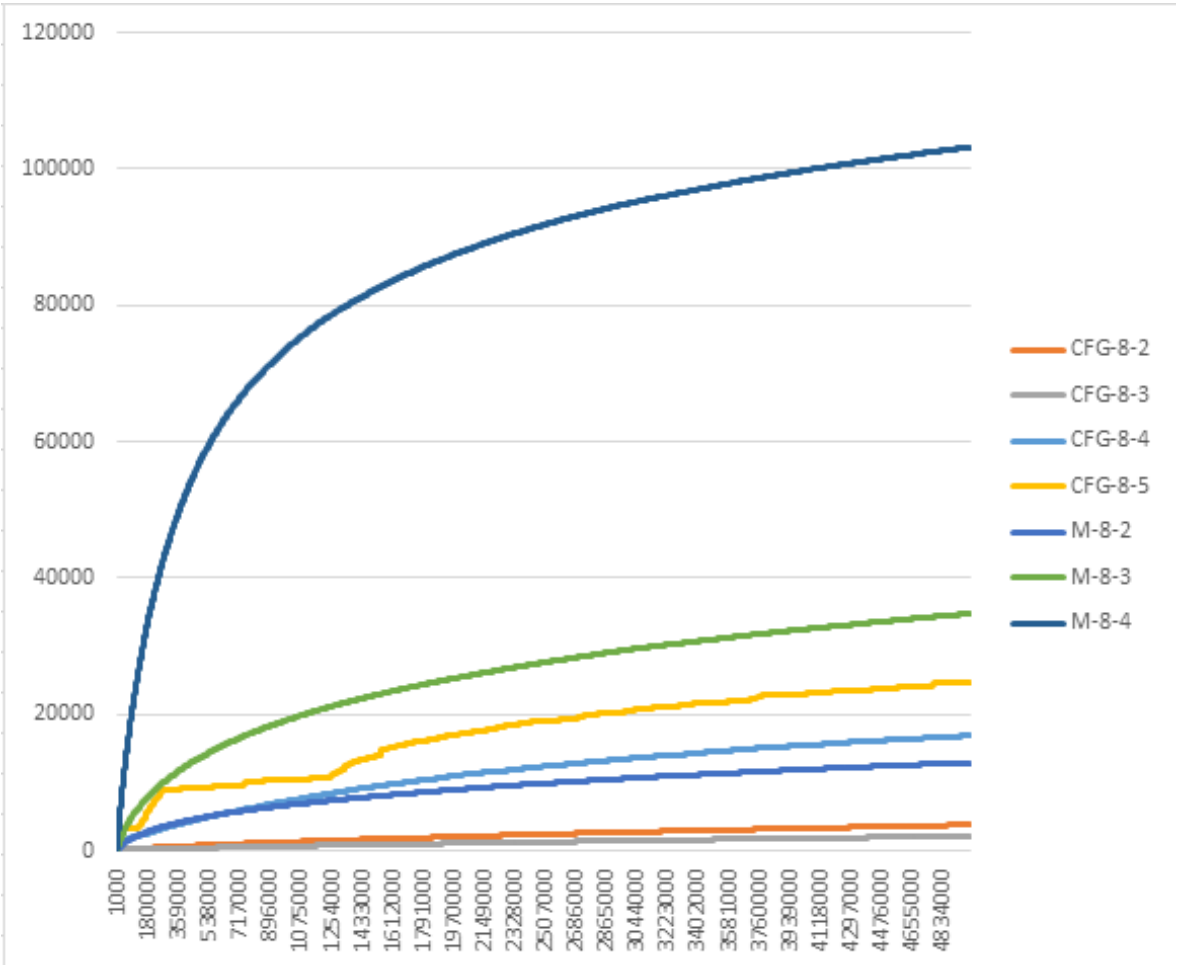
ceho Markovský zdroj negeneruje celý priestor hesiel určenej dĺžky.

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 prenásobený 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 vrámci generovaného slovníka. Hodnota $indS_i$ vyjadruje tú istú hodnotu pre vstupný slovník. Hodnoty v_i sú počty výskytov hesiel zadefinované vo vstupnom slovníku.

Priemerná pozícia v slovníku ukazuje ako pravdepodobné heslá v priemernom prípade generuje náš algoritmus. Keďže toto číslo sa často krát zdá veľké, prikladáme k nemu v druhom stĺpci jeho percentuálnu hodnotu. Hodnoty d a p vyjadrujú maximálnu dĺžku hesiel v slovníku a dĺžku použitých prefixov v Markovskom zdroji. Pre hodnoty

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

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

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. Gehringer. 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.