Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Metódy útoku hrubou silou na TrueCrypt

Diplomová Práca

2016 **Bc. 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 **Bc. Martin Strapko**

Poďakovanie

Abstrakt

Táto práca sa vo všeobecnosti zaoberá optimalizáciou útokov hrubou silou. Špeciálne rozoberá možnosti útoku na program TrueCrypt. Sústredí sa na to ako využiť znalosti o často používaných heslách na vylepšenie efektívnosti útoku. Hlavným cieľom práce je napísať program, ktorý sa snaží pomocou útoku hrubou silou nájsť čo najefektívnejšie heslo pre dešifrovanie disku zašifrovaného pomocou programu TrueCrypt.

KĽÚČOVÉ SLOVÁ: TrueCrypt, útok 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 to effectively recover password for disk encrypted with TrueCrypt

KEYWORDS: TrueCrypt, brute-force attack, context-free grammar

Obsah

Zoznam listingov

Zoznam obrázkov

Úvod

V dnešnom svete fungujúcom na elektronických dátach, ktoré pre nás majú obrovskú cenu, sa ľudia snažia ochrániť ich dôvernosť. Tieto dáta sa dajú jednoducho ochrániť, ak k ním bude mať fyzický prístup len majiteľ. Toto avšak nie je najpoužiteľnejšie riešenie, keďže takmer každý počítač je pripojený do nejakej siete. Iná možnosť je mať dáta uložené vo forme, ktorá bude dávať zmysel len oprávneným osobám, aj keď fyzický prístup k ním môžu mať aj iný ľudia. Na tento účel primárne slúži šifrovanie.

Bezpečnosť šifrovania veľmi záleží na utajený kľúča. Preto je dôležité aby nebol ľahko uhádnuteľný. Keďže počítače priniesli so sebou obrovskú výpočtovú silu, sú schopné robiť až niekoľko desiatok tisíc pokusov za sekundu snažiac sa uhádnuť tento kľúč [13]. 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 z ľubovolných znakov ASCII tabuľky.

Takéto hľadanie kľúča preberaním všetkých možností sa nazýva útok hrubou silou. Jeho podstatou je postupné generovanie možných kľúčov a následne overenie ich správnosti. V našej práci sa zameriame na útok pri ktorom má útočník fyzický prístup k súboru s hešmi. Postup tohto útoku je následovný:

- Útočník vygeneruje kandidáta na overenie
- Ak existuje, tak pripojí náhodný reťazec spojený s týmto heslom ku kandidátovi. Tieto reťazce zabezpečujú ochranu pred útokom pomocou predpočítaných hešov
- Zahešuje kandidáta pomocou zvoleného hešovacieho algoritmu
- Porovná novovzniknutý heš s hešmi nachádzajúcimi sa súbore z ktorého sa snaží nájsť heslá

Nakoľko je tento algoritmus na overovanie kandidátov dobre paralelizovateľný, keďže overenie jedného kandidáta nezávisí na žiadnom inom overení, jednou z optimalizácií tohto procesu bude počítanie týchto hešov pomocou grafických kariet. Nakoľko existujú práce [11] podrobne sa venujúce sa týmto optimalizáciám, nebudeme vrámci tejto práce implementovať program na overenie správnosti kandidátov.

V tejto práci sa budeme zaoberať skúmaním a implementáciou algoritmov na generovanie týchto kandidátov. Výstupom našej práce bude program, ktorý bude generovať zoznam hesiel, ktoré sa dajú následne použiť ako kandidáti v niektorom z voľne dostupných programov na skúšanie takýchto hesiel ako napríklad *hashCat*, *John The Ripper* alebo *PasswordsPro*.

Tieto programy podporujú viaceré spôsoby útokov hrubou silou na veľké množstvo známych a často používaných hešovacích funkcií. Tieto typy útokov zahŕňajú použitie predom vytvorených slovníkov, spájanie slov z rôznych slovníkov ako aj postupné generovanie všetkých možných reťazcov. V našej práci sa snažíme vyvinúť a implementovať ďalší spôsob generovania kandidátov použitím pravdepodobnostných bezkontextových gramatík.

Efektívnosť nami navrhnutého riešenia budeme porovnávať s algoritmom používajúcim Markovovské zdroje, ktoré sú používane aj vyššie spomenutými programami pri postupnom generovaní kandidátov. Tento algoritmus sme si vybrali, keďže jeho základný princíp priraďovania pravdepodobnosti jednotlivým znakom na základe predošlého stavu je najbližší k tomu, ktorý používame v našom algoritmus využívajúcom bezkontextové gramatiky.

KAPITOLA 1

TrueCrypt

TrueCrypt je šifrovací program umožňujúci používateľovi na základe ním zvoleného hesla vytvoriť šifrovaný disk. Taktiež používateľovi umožôuje vytvorenie virtuálného šifrovaného disku, ktorý bude následne uložený v súbore je fyzickom disku. 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 jeho základnom návrhu.

V septembri 2015 prišiel James Forshaw s informáciou o 2 chybách vo Windows drivery, ktorý používa program 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 písmená diskov [4]. Obe tieto chyby sa dokážu prejaviť až počas behu samotného programu. Preto ich v tejto práci používať nebudeme, nakoľko pre náš algoritmus nie je potrebné aby TrueCrypt bežal.

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 blokové šifry v XTS móde 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 partície prebieha vo viacerých fázach:

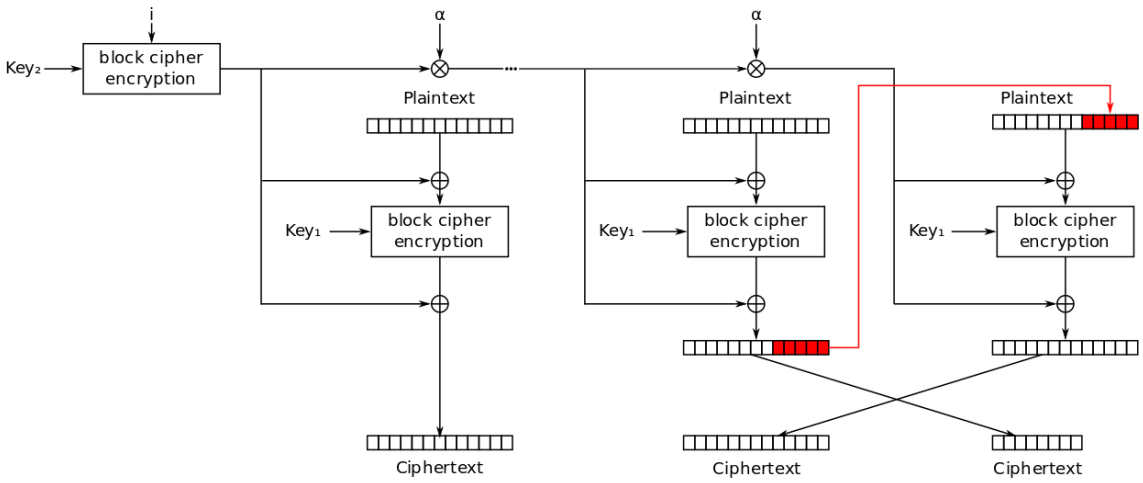
- Vygeneruje sa náhodný kľúč vhodný na šifrovanie pomocou zvoleného algoritmu.
  - V prípade XTS módu veľkosť kľúča zdvojnásobíme. Dôvodom je použitie 2 kľúčov pri šifrovaní v XTS móde ako je znázornené na obrázku 1.1

- Pomocou tohto kľúča sa zašifruje celá požadovaná partícia
- Vytvorí sa hlavička pre túto partíciu obsahujúca:
  - Verziu programu TrueCrypt
  - Verziu hlavičky
  - ASCII reťazec 'TRUE'
  - Vygenerovaný kľúč použitý na zašifrovanie dát
  - CRC-32 kontrolná suma kľúča
  - CRC-32 kontrolná suma zvyšku hlavičky
- Táto hlavička sa zašifruje pomocou kľúča vygenerovaného na základe používateľského hesla, ktoré nemusí byť vhodné na použitie ako šifrovací kľúč
  - K heslu sa pripojí náhodný 512 bitový reťazec - kryptografická soľ
  - Takto upravené heslo sa dá na vstup algoritmu PBKDF2
  - Transformáciami pomocou hešovacích funkcií vznikne vhodný kľúč na šifrovanie
- Vyššie vygenerovaná kryptografická soľ sa nezašifrovaná pridá pred hlavičku partície

Takto zašifrovaná partícia je nakoniec uložená na fyzickom disku v súboru obsahujúcom:

- Kryptografická soľ v nešifrovanej forme
- Šifrovaná hlavička partície obsahujúca kľúč použitý na šifrovanie dát
- Používateľom zvolené dáta zašifrované programom TrueCrypt

XTS mód Na obrázku 1.1 vidíme schému módu XTS pre blokové šifry. Základ tvorí mód XEX, ktorý bol navrhnutý na efektívne spracovanie za sebou nasledujúcich blokov vrámci dátového bloku napríklad diskového sektoru. Na schéme môžme vidieť použitie 2 rôznych kľúčov Key_1 a Key_2. Kvôli použitiu týchto kľúčov sa často krát generuje šifrovací kľúč dvojnásobnej dĺžky kedy sa následne prvá polovica použije ako Key_1 a druhá ako Key_2. Hodnota i vyjadruje číslo diskového sektoru, ktorý sa práve šifruje, zatiaľ čo α je prvok z konečného poľa $GF(2^128)$. Pre každý blok šifry sa spraví α^j, kde j je číslo bloku vrámci sektoru. Mód XTS prináša módu XEX podporu pre sektory veľkosti nedeliteľnej veľkosťou bloku šifry. Ak posledný blok otvoreného textu nemá dostatočnú veľkosť, pridá sa k nemu potrebný počet bajtov predošlého zašifrovaného bloku. Po zašifrovaní takto upraveného otvoreného textu sa novo zašifrovaný blok vymení so zvyškom predposledného šifrovaného bloku, ako je znázornené na schéme.



Obr. 1.1: Schéma módu XTS

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. Našim cieľom pri tomto hľadaní bolo nájdenie časti kódu, ktorá je zodpovedná za prijatie používateľského vstupu a jeho následne spracovanie. Toto spracovanie zahŕňa transformáciu tohto vstupu na kľúč, ktorým sa program pokúsi rozšifrovať hlavičku partície 1.1. Následne sa program posnaží rozšifrovať hlavičku pomocou tohto kľúču a overí, či dáta v nej dávajú zmysel. Toto zahŕňa overenie reťazca *'TRUE'*, verzie hlavičky, minimálnej verzie programu a CRC-32 súm kľúčov a ostatných položiek 1.2. Bohužiaľ vrámci tejto práce sme nemali čas izolovať minimálny kód potrebný na úspešne fungovanie tohto overenia.

Zbytok tejto práce sa venuje generovaniu slovníka z ktorého budeme brať kandidátov na hľadané heslo.

```
switch (pkcs5_prf)
{
  case RIPEMD160:
    derive_key_ripemd160 (keyInfo.userKey, keyInfo.keyLength,
            keyInfo.salt, PKCS5_SALT_SIZE,
            keyInfo.noIterations, dk, GetMaxPkcs5OutSize());
    break;

  case SHA512:
    derive_key_sha512 (keyInfo.userKey, keyInfo.keyLength,
            keyInfo.salt, PKCS5_SALT_SIZE,
            keyInfo.noIterations, dk, GetMaxPkcs5OutSize());
    break;

  case SHA1:
    // Deprecated/legacy
    derive_key_sha1 (keyInfo.userKey, keyInfo.keyLength,
            keyInfo.salt, PKCS5_SALT_SIZE,
            keyInfo.noIterations, dk, GetMaxPkcs5OutSize());
    break;

  case WHIRLPOOL:
    derive_key_whirlpool (keyInfo.userKey, keyInfo.keyLength,
            keyInfo.salt, PKCS5_SALT_SIZE,
            keyInfo.noIterations, dk, GetMaxPkcs5OutSize());
    break;

  default:
    // Unknown/wrong ID
    TC_THROW_FATAL_EXCEPTION;
}
```

Zdrojový kód 1.1: Ukážka kódu transformácie hesla na šifrovací kľúč

```
// Copy the header for decryption
memcpy (header, encryptedHeader, sizeof (header));

// Try to decrypt header
DecryptBuffer (header + HEADER_ENCRYPTED_DATA_OFFSET,
      HEADER_ENCRYPTED_DATA_SIZE, cryptoInfo);

// Magic 'TRUE'
if (GetHeaderField32 (header, TC_HEADER_OFFSET_MAGIC) != 0x54525545)
  continue;

// Header version
headerVersion = GetHeaderField16 (header, TC_HEADER_OFFSET_VERSION);

if (headerVersion > VOLUME_HEADER_VERSION)
{
  status = ERR_NEW_VERSION_REQUIRED;
  goto err;
}

// Check CRC of the header fields
if (!ReadVolumeHeaderRecoveryMode
  && headerVersion >= 4
  && GetHeaderField32 (header, TC_HEADER_OFFSET_HEADER_CRC) !=
    GetCrc32 (header + TC_HEADER_OFFSET_MAGIC,
    TC_HEADER_OFFSET_HEADER_CRC - TC_HEADER_OFFSET_MAGIC))
  continue;

// Required program version
cryptoInfo->RequiredProgramVersion = GetHeaderField16 (header,
              TC_HEADER_OFFSET_REQUIRED_VERSION);
cryptoInfo->LegacyVolume = cryptoInfo->RequiredProgramVersion < 0x600;

// Check CRC of the key set
if (!ReadVolumeHeaderRecoveryMode
  && GetHeaderField32 (header, TC_HEADER_OFFSET_KEY_AREA_CRC) !=
  GetCrc32 (header + HEADER_MASTER_KEYDATA_OFFSET,
      MASTER_KEYDATA_SIZE))
  continue;
```

Zdrojový kód 1.2: Ukážka kódu overenia správnosti hesla dešifrovaním hlavičky

KAPITOLA 2

Útoky hrubou silou

Základným princípom útokov hrubou silou je 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 odtlačku vyrobiť pôvodný reťazec, musíme pre vyskúšanie kandidáta zahešovať tohto kandidáta a následne porovnať výsledný odtlačok s tým od správneho hesla. Metód akými sa dajú títo kandidáti generovať je mnoho a nižšie si predstavíme niektoré z nich.

Všetky v praxi používané algoritmy používajú kľúče dĺžky aspoň 256 bitov, čo je ekvivalent 32 znakového reťazca zloženého z ľubovolných znakov ASCII tabuľky. V praxi je veľmi nepravdepodobné, že používateľ bude mať takto dlhé heslo založené nad tak veľkou abecedou. 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 (na $2.18953 * 10^{-45}$% pôvodnej veľkosti).

2.1 Inkrementálny útok

Inkrementálna metóda patrí medzi najzákladnejšie útoky hrubou silou a často krát sa práve ona myslí pod pojmom útok 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% úspešnosť. Jej problém avšak spočíva v množstve reťazcov, ktoré je potrebné vyskúšať. Vo väčšine prípadov 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 vyskúšanie všetkých trvalo približne $8.92 * 10^{13}$ rokov pri skúšaní 1 miliardy reťazcov za sekundu.

2.2 Slovníkový útok

Častokrát existuje ešte menšia množina reťazcov, heslá z ktorej majú omnoho väčšiu šancu, že medzi nimi bude hľadaný výraz. Toto je pravda špeciálne pri hľadaní používateľských hesiel, nakoľko používatelia volia heslá tak, 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 nájde heslo rýchlejšie 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 zoznamov 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 zoznamu 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 partícii zašifrovanej programom TrueCrypt. Keďže chceme toto heslo nájsť v ľubovoľne 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 pravdepodobnosti vygenerovania jednotlivých reťazcov aby následne mohol na výstup dávať heslá od najpravdepodobnejšieho po najmenej pravdepodobné.

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 konjuktívnom normálnom tvare, pre ktorý sa daný algoritmus snaží nájsť také ohodnotenie boolovských premenných, 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 konjuktívnej normálnej formy. Ako výstup tohto algoritmu je reťazec takmer náhodných bitov, ktoré dopredu poznáme, pretože tieto sú fyzický uložené na disku. Neznámymi v tomto boolovskom výraze sú vstupné dáta a kľúč pomocou ktorého boli tieto dáta zašifrované. Našim 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 túto metódu nevenovali hlbšiu neanalyzovali, keďže existuje viacero prác, ktoré do podrobna 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.

KAPITOLA 3

Používateľské heslá

Aj napriek tomu, ako sú používateľské heslá vo všeobecnosti ľahko prelomiteľné, sú dnes 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, tak by používali heslá s najväčšou entropiu. Tie by boli najdlhšie možné povolené systémom, zložené z náhodne vybraných znakov povolených týmto systémom 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 prispôsobená 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 sa skladať zo známych kusov informácie ako sú napríklad slová. Nakoniec ľudská myseľ funguje veľmi dobre vďaka redundancii informácie, čiže človeku sa ľahšie pamätajú veci, ktoré si vie odvodiť z viacerých iných kusov informácii.

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 sústredí 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, číslice 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 číslic 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 fráze, 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.

KAPITOLA 4

Učenie

Ako sme spomínali, budeme sa zaoberať útokom, ktorý dostane na vstupe slovník a na základe tohto slovníka bude generovať heslá zoradené podľa pravdepodobnosti. Kvalita výsledného zoznamu bude závisieť od schopnosti algoritmu správne sa naučiť ohodnotiť pravdepodobnosti 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 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 Σ je 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 daného slova. Je možné aby v takomto strome existovali 2 rôzne cesty odvodenia, ktoré nakoniec vygenerujú rovnakú vetnú formu. Tejto vlastnosti bezkontextových gramatík sa budeme snažiť vyhnúť vytvorením pravidiel tak, aby ľubovolná terminálna vetná forma mala práve jeden spôsob odvodenia 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 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 Markovovský zdroj vypočíta pravdepodobnosti následujúcich znakov.

Pomocou základnej implementácie Markovovho zdroja dokáže algoritmus používajúci 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 kratšie ako zadaná dĺžka. Na vyriešenie tohto problému sme sa rozhodli definovať pravdepodobnosti pre kombinácie znakov, ktoré sa na vstupe nevyskytli.

KAPITOLA 5

Implementácia

5.1 Pravdepodobnostná bezkontextová gramatika

Pravdepodobnostná bezkontextová gramatika je bezkontextová gramatika, ktorej pravidlá majú priradenú pravdepodobnosť. Pravdepodobnostná bezkontextová gramatika G je pätica $G = (M, T, R, S, P)$, kde:

- $M = N^i : i = 1, ..., n$ je množina neterminálov
- $T = w^k : k = 1, ..., V$ je množina terminálov
- $R = N^i \rightarrow \zeta^j : \zeta^j \epsilon (M \cup T)^*$ je množina pravidiel
- $S = N^1$ je počiatočný neterminál
- P je množina pravdepodobností pravidiel, kde platí $\forall i \sum_i P(N^i \rightarrow \zeta^j) = 1$

5.1.1 Tvorba gramatiky

Pri tvorbe gramatiky sme si dali za cieľ zaistiť aby gramatika spĺňala určité podmienky:

- Generovať všetky reťazce zo vstupnej abecedy kratšie ako používateľom zadaná maximálna dĺžka
- Vygenerovať každý reťazec práve raz

V nami vytvorenej gramatiky bude kategorizovať slová podľa ich zloženie z jednotlivých typov znakov. Pre každý takýto typ znakov vygenerujeme všetky možné reťazce zložené zo znakov tohto typu. Následne vytvoríme všetky možné predpisy zložené z kombinácií týchto typov. Uvažujme, že typ 1 obsahuje t_1 znakov a typ 2 obsahuje t_2

znakov a budeme vytvárať reťazce dĺžky 2, kde každý znak patrí do iného typu. Všetkých takýchto reťazcov je $t_1 * t_2 * 2$. Naša gramatika si bude pamätať len $t_1 + t_2 + 2$ položiek, z ktorých dokáže spájaním vygenerovať všetkých $t_1 * t_2 * 2$.

Jednoduché neterminály Prvý typ neterminálov, ktoré budeme nazývať jednoduché, obsahujú pravidlá, ktoré majú na pravej strane pravidla iba terminálne symboly. Keďže naša vstupná abeceda obsahuje okolo 70 znakov, medzi ktoré 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 reťazcov terminálnych znakov dĺžky k nad abecedou veľkosti n je n^k rozhodli sme sa zadefinovať maximálnu veľkosť jednoduchého neterminálu, vyjadrujúcu maximálné povolené k. Na základe testov zo sekcií 6.1 a 6.2 sa ako vhodná hodnota ukázalo $k = 5$. Avšak v prípade generovania malého počtu hesiel sa kvôli optimalizácií času oplatí použiť hodnotu $k = 4$.

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, tieto skupiný budeme nazývať zložené neterminály. Tieto neterminály vyjadrujú vždy jeden možný predpis pre terminálne slovo. Napríklad zložený neterminál $U_1L_3D_4$ vyjadruje všetky terminálne slová začínajúce na veľké písmeno nasledované tromi malými písmena, ukončené štvoricou cifier.

Kvôli dodržaniu jednoznačnosti generovania 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 zložené neterminály vyjadrujúce všetky možné predpisy slov kratších ako zadaná maximálna dĺžka.

5.1.2 Počítanie pravdepodobností

Aby sme vedeli čo najlepšie vyhovieť potrebám používateľa, potrebujeme im prispôsobiť našu gramatiku. Tu začnú zohrávať rolu pravdepodobnosti jednotlivých pravidiel našej gramatiky. Našim cieľom je nastaviť našu gramatiku tak, aby generovala heslá podľa pravdepodobnosti použitia daným používateľom. Úspešnosť tohto učenia gramatiky bude 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, ako sme už v úvode spomínali, 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 pravidlá tejto gramatiky sa budú meniť len pri zmene maximálnej dĺžky hesla. Preto používanie nášho algoritmu s rôznymi slovníkmi nevyžaduje vyrábanie novej gramatiky až do momentu kedy sa rozhodneme generovať heslá s inou maximálnou dĺžkou. Pravdepodobnosti 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.

```
# ideme po pismenach slova zo slovnika
for i in range(1, len(word)):
  # ak sa zmenil typ znaku na male pismeno
  if (word[i] in lower) and (currentNet != 'L'):
    # k zlozenemu neterminalu pridame jednoduchy posledneho
    # videneho typu a velkosti
    rule += currentNet + str(i-startI)
    # pripocitame pocet vyskytov retazca daneho posledneho
    # jednoducheho neterminalu
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    # pripocitame pocet vyskytov daneho jednoducheho neterminalu
    ruleCount[currentNet + str(i-startI)] += occ
    # dalsi typ zacina na i-tej pozicii
    startI = i
    # je to retazec malych pismen
    currentNet = 'L'
    # budujeme od zaciatku
    currentSubstring=''
  # ak sa zmenil typ znaku na velke pismeno
  elif (word[i] in upper) and (currentNet != 'U'):
    # k zlozenemu neterminalu pridame jednoduchy posledneho
    # videneho typu a velkosti
    rule += currentNet + str(i-startI)
    # pripocitame pocet vyskytov retazca daneho posledneho
    # jednoducheho neterminalu
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    # pripocitame pocet vyskytov daneho jednoducheho neterminalu
    ruleCount[currentNet + str(i-startI)] += occ
    # dalsi typ zacina na i-tej pozicii
    startI = i
    # je to retazec velkych pismen
    currentNet = 'U'
    # budujeme od zaciatku
    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 je 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. Existuje taktiež viacero spôsobov ako počítať výskyty zložených neterminálov.

- Do výskytov počítame len výskyty hesiel ktoré sú presne reprezentované daným

Tabuľka 5.1: Ukážka počítania pravdepodobností - slovník

Slovník
abc
1bc
12
1b1
b
113
b2c
bca
a
11

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 pre každú zmenu typu neterminálu pripočítanie výskytov k doteraz vytvorenému zloženému neterminálu, ukážka takto upraveného kódu je v 5.3.

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.

Vplyv výberu niektoré z vyššie uvedených spôsobov na kvalitu gramatiky a rýchlosť jej generovania potrebne skúmali v sekciách a . Na základe týchto testov

TODO

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

Tabuľka 5.2: Ukážka počítania pravdepodobností - gramatika

Pravidlo	$p_zkladna$	$p_podreazce$	$p_rekurzvne$
$L1 \to a$	2	2	2
$L1 \to b$	3	3	3
$L1 \to c$	2	2	2
$L2 \to ab$	1	1	1
$L2 \to bc$	2	2	2
$L2 \to ca$	1	1	1
$L2 \to aa\|ac\|ba\|bb\|cb\|cc$	0	0	0
$D1 \to 1$	3	3	3
$D1 \to 2$	1	1	1
$D1 \to 3$	1	1	1
$D2 \to 11$	2	2	2
$D2 \to 12$	1	1	1
$D2 \to 13\|21\|22\|23\|31\|32\|33$	0	0	0
Pravidlo	$p_zkladna$	$p_podreazce$	$p_rekurzvne$
$Z \to L1$	2	3	2
$Z \to D1$	0	2	0
$Z \to L2$	0	2	0
$Z \to D2$	2	3	2
$Z \to D1L1$	0	1	5
$Z \to L1D1$	0	1	5
$Z \to D1L2$	1	1	0
$Z \to L2L1$	2	2	2
$Z \to D2D1$	1	1	?
$Z \to L1D1L1$	1	1	?
$Z \to D1L1D1$	1	1	?
$Z \to L1D2\|L2D1\|D2L1$	0	0	?

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 os-

Tabuľka 5.3: 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

tatné 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, tak platí $p[t+1] = p[t]/p_i[t] * p_i[t+1]$. 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 rádom 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 repisovať neterminály od najľavejšieho a vďaka tomu nebudeme generovať duplikáty, ktoré by sa od seba líšili len v poradí v akom sme rozpísali neterminály na terminály.

Vo tabuľke 5.3 demonštrujeme dva kroky nášho algoritmu na generovanie hesiel. V tomto príklade sa sústredíme na generovanie rôznych terminálnych slov zo zloženého neterminálu $U_1L_3D_2$. V ľavej časti tabuľky môžme vidieť zadefinované 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 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 heuristiky, ktoré by zmenšili pamäťovú náročnosť a čo najlepšie uchovali poradie hesiel.

V zdrojovom kóde 5.6 môžme vidieť časť kódu zodpovednú za napĺňanie prioritnej fronty ďalšími kandidátmi na najbližšie vygenerované heslo. Premenná *task* je usporiadaná dvojica (rád, vektor). Algoritmus prejde od člena určeného rádom vektora až po koniec vektora (riadok 1) a pre každý prvok posunie index ukazujúci na aktuálne použitý prvok (riadok 4). Taktiež vypočíta pravdepodobnosť hesla reprezentovaného novým stavom vektora (riadok 3 a 7). Túto pravdepodobnosť spolu s usporiadanou dvojicou obsahujúcou zmenený rád vektora a samotný vektor vloží do prioritnej fronty (riadok 8 a 9).

Veľkým nedostatkom použitia bezkontextových gramatík je vo veľkosti pamäte, ktorú potrebuje. Samotný zápis gramatiky na disku v tvare JSON mal pri gramatike generujúcej 12 znakové heslá okolo 1 gigabajtu. Avšak táto veľkosť není 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 za taký 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 implementácií bezkontextových gramatík bola možnosť prerušovaného generovania. Vtedy používateľ má možnosť generovať požadované heslá po ľubovoľné veľkých častiach.

5.2 Markovovský 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 Markovovský zdroj. Ako sme písali v predošlej kapitole, jedná sa o náhodný proces, ktorý spĺňa podmienku bezpamäťovosti. 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 dali za cieľ pri bezkontextových gramatikách:

- Generovať všetky reťazce zo vstupnej abecedy kratšie ako používateľom zadaná maximálna dĺžka
  - Pravdepodobnosti jednotlivých znakov sú inicializované na 0, Markovovský zdroj ich nikdy nevygeneruje
- Vygenerovať každý reťazec práve raz
  - Keďže tomuto zdroju nič nebráni v tom vygenerovať viac krát počas behu to isté slovo

Aj keď druhú podmienku nevedia Markovovské zdroje splniť už priamo z definície, s prvou sme sa pokúsili niečo vymyslieť. Najprv sme sa pokúšali inicializovať pravdepodobností všetkých znakov na nenulovú hodnotu. Toto však spôsobilo, že sa zdroj relatívne ľahko dostal medzi prefixy, ktoré neboli definované. Pri takýchto prefixoch majú všetky znaky rovnakú pravdepodobnosť. Dôsledkom tohto nastávalo cyklenie sa v týchto neznámych stavoch, čo spôsobovalo generovanie dlhých nezmyselných reťazcov 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 jeden 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 nás 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 hodnotu $\varepsilon > 0$. 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 ε mala byť niekoľko násobne menšia ako najnižšia známa pravdepodobnosť pre daný prefix. Vplyv tejto konštanty na výkon Markovovského zdroja je znázornený na grafe 6.8.

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 ε ak sa jedná o znak pomocou ktorého v ďalšom kroku algoritmu vznikne videný prefix. V prípade, že znak dostane náš algoritmus do stavu s iným neznámym prefixom, nastavíme tomuto znaku pravdepodobnosť δ, ktorá je niekoľko krát menšia ako ε. Opäť vplyv tejto konštanty na výsledky algoritmu je znázornený na grafe 6.8. Použitím hodnôt ε a δ by sme mali dostať algoritmus používajúci Markovovský zdroj do stavu, kedy v konečnom čase dokáže vygenerovať ľubovolný počet hesiel.

Vrámci implementácie sme algoritmus používajúci Markovovský zdroj upravili tak, aby nepoužíval žiadnu pamäť navyše oproti pôvodnému návrhu. Jediná informácia, ktorú si tento algoritmus pamätá je tabuľka pravdepodobností nasledovania znakov 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 ε ak sa vygenerovaním tohto znaku dostane do známeho prefixu alebo pravdepodobnosť δ ak pridanie tohto znaku vedie do ďalšie stavu s neznámym prefixom.

Rozdiely vo veľkosti ε oproti najmenšej nenulovej pravdepodobnosti pre ten prefix a δ od ε 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 ε a δ sú znázornené v kapitole Testy.

V tomto prípade sme zvažovali aj použitie možnosti zmeny týchto pravdepodobností počas behu programu podobne ako 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 z dôvodu časovej tiesne sme nenašli priestor na implementáciu a preskúmanie takto upraveného algoritmu.

```
  # ak sa zmenil typ znaku na cislicu
  elif (word[i] in digit) and (current != 'D'):
    # k zlozenemu neterminalu pridame jednoduchy posledneho
    # videneho typu a velkosti
    rule += currentNet + str(i-startI)
    # pripocitame pocet vyskytov retazca daneho posledneho
    # jednoducheho neterminalu
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    # pripocitame pocet vyskytov daneho jednoducheho neterminalu
    ruleCount[currentNet + str(i-startI)] += occ
    # dalsi typ zacina na i-tej pozicii
    startI = i
    # je to retazec cislic
    currentNet = 'D'
    # budujeme od zaciatku
    currentSubstring=''
  # ak sa zmenil typ znaku na symbol
  elif (word[i] in special) and (current != 'S'):
    # k zlozenemu neterminalu pridame jednoduchy posledneho
    # videneho typu a velkosti
    rule += currentNet + str(i-startI)
    # pripocitame pocet vyskytov retazca daneho posledneho
    # jednoducheho neterminalu
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    # pripocitame pocet vyskytov daneho jednoducheho neterminalu
    ruleCount[currentNet + str(i-startI)] += occ
    # dalsi typ zacina na i-tej pozicii
    startI = i
    # je to retazec symbolov
    currentNet = 'S'
    # budujeme od zaciatku
    currentSubstring=''
  # presiahli sme velkost jednoducheho neterminalu
  elif len(currentSubstring) >= maxNetSize:
    # k zlozenemu neterminalu pridame jednoduchy posledneho
    # videneho typu a velkosti
    rule += currentNet + str(i-startI)
    # pripocitame pocet vyskytov retazca daneho posledneho
    # jednoducheho neterminalu
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    # pripocitame pocet vyskytov jednoducheho daneho neterminalu
    ruleCount[currentNet + str(i-startI)] += occ
    # dalsi typ zacina na i-tej pozicii
    startI = i
    # budujeme od zaciatku
    currentSubstring=''
  currentSubstring += word[i]
...
```

Zdrojový kód 5.2: Úprava pravidiel na základe vstupného slova - Pokračovanie

```
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
    # este sme takyto zlozeny neterminal nevideli
    if not rule in rulez['Z']:
      rulez['Z'][rule] = 1
    # pripocitame pocet vyskytov tohto zlozeneho netermialu
    rulez['Z'][rule] += occ
    # pripocitame pocet vyskytov nejakeho neterminalu
    ruleCount['Z'] += 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
    # este sme takyto zlozeny neterminal nevideli
    if not rule in rulez['Z']:
      rulez['Z'][rule] = 1
    # pripocitame pocet vyskytov tohto zlozeneho netermialu
    rulez['Z'][rule] += occ
    # pripocitame pocet vyskytov nejakeho neterminalu
    ruleCount['Z'] += occ
    startI = i
    currentNet = 'U'
    currentSubstring=''
```

Zdrojový kód 5.3: Pripočítanie výskytov k podmnožinám zložených neterminálov

```
  elif (word[i] in digit) and (current != 'D'):
    rule += currentNet + str(i-startI)
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    ruleCount[currentNet + str(i-startI)] += occ
    # este sme takyto zlozeny neterminal nevideli
    if not rule in rulez['Z']:
      rulez['Z'][rule] = 1
    # pripocitame pocet vyskytov tohto zlozeneho netermialu
    rulez['Z'][rule] += occ
    # pripocitame pocet vyskytov nejakeho neterminalu
    ruleCount['Z'] += occ
    # dalsi typ zacina na i-tej pozicii
    startI = i
    # je to retazec cislic
    currentNet = 'D'
    # budujeme od zaciatku
    currentSubstring=''
  # ak sa zmenil typ znaku na symbol
  elif (word[i] in special) and (current != 'S'):
    # k zlozenemu neterminalu pridame jednoduchy posledneho
    # videneho typu a velkosti
    rule += currentNet + str(i-startI)
    # pripocitame pocet vyskytov retazca daneho posledneho
    # jednoducheho neterminalu
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    # pripocitame pocet vyskytov daneho jednoducheho neterminalu
    ruleCount[currentNet + str(i-startI)] += occ
    # este sme takyto zlozeny neterminal nevideli
    if not rule in rulez['Z']:
      rulez['Z'][rule] = 1
    # pripocitame pocet vyskytov tohto zlozeneho netermialu
    rulez['Z'][rule] += occ
    # pripocitame pocet vyskytov nejakeho neterminalu
    ruleCount['Z'] += occ
    startI = i
    currentNet = 'S'
    currentSubstring=''
```

Zdrojový kód 5.4: Pripočítanie výskytov k podmnožinám zložených neterminálov - Pokračovanie

```
  elif len(currentSubstring) >= maxNetSize:
    rule += currentNet + str(i-startI)
    rulez[currentNet + str(i-startI)][str(currentSubstring)] += occ
    ruleCount[currentNet + str(i-startI)] += occ
    # ak uz mame zlozeny neterminal
    if len(rule) > 2:
      # este sme taky nevideli
      if not rule in rulez['Z']:
        rulez['Z'][rule] = 1
      # pripocitame pocet vyskytov tohto zlozeneho netermialu
      rulez['Z'][rule] += occ
      # pripocitame pocet vyskytov nejakeho neterminalu
      ruleCount['Z'] += occ
    startI = i
    currentSubstring=''
  currentSubstring += word[i]
```

Zdrojový kód 5.5: Pripočítanie výskytov k podmnožinám zložených neterminálov - Pokračovanie

```
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 5.6: Generovanie všetkých susedných vektorov

KAPITOLA 6

Evaluácia výsledkov

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 merali 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

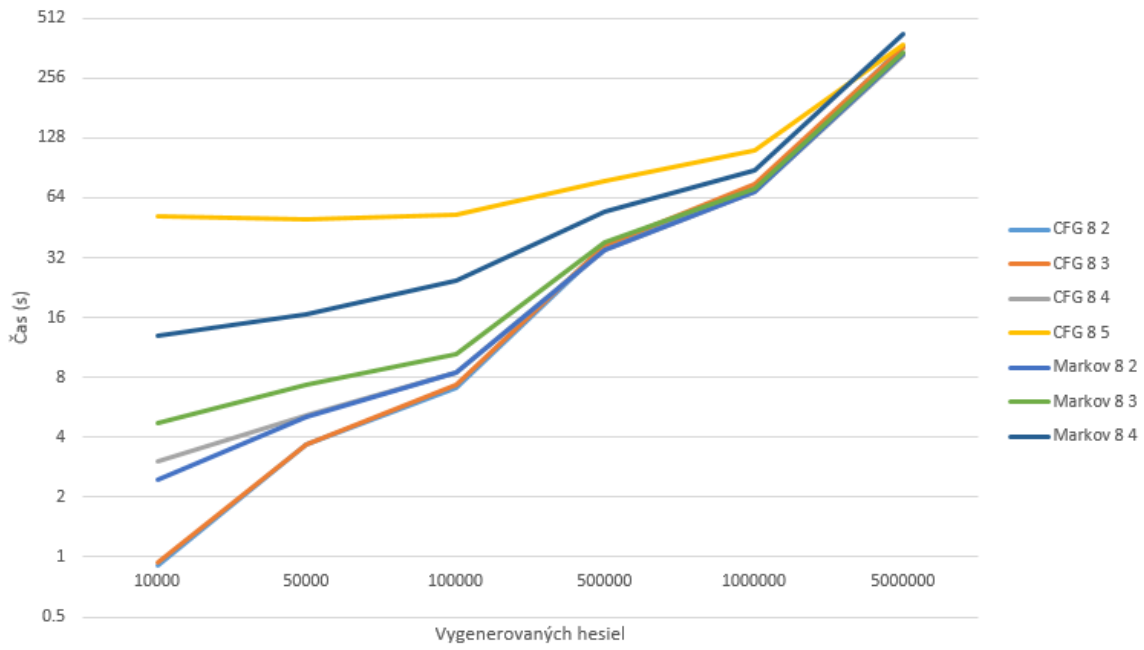
	d	p	GEN	10000	50000	100000	500000	1000000	5000000
CFG	6	2	0.103	0.754	3.504	7.097	35.109	71.129	369.057
CFG	6	3	0.32	2.696	5.441	9.028	37.121	72.181	354.763
CFG	6	4	6.053	2.621	5.53	8.96	37.515	73.744	367.655
CFG	6	5	156.141	47.595	50.767	54.054	81.957	116.699	397.522
Markov	6	2	—	1.284	3.936	7.469	34.533	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 nevypovedajú o tom ako sa budú jednotlivé algoritmy správať ak by boli použité v praxi.

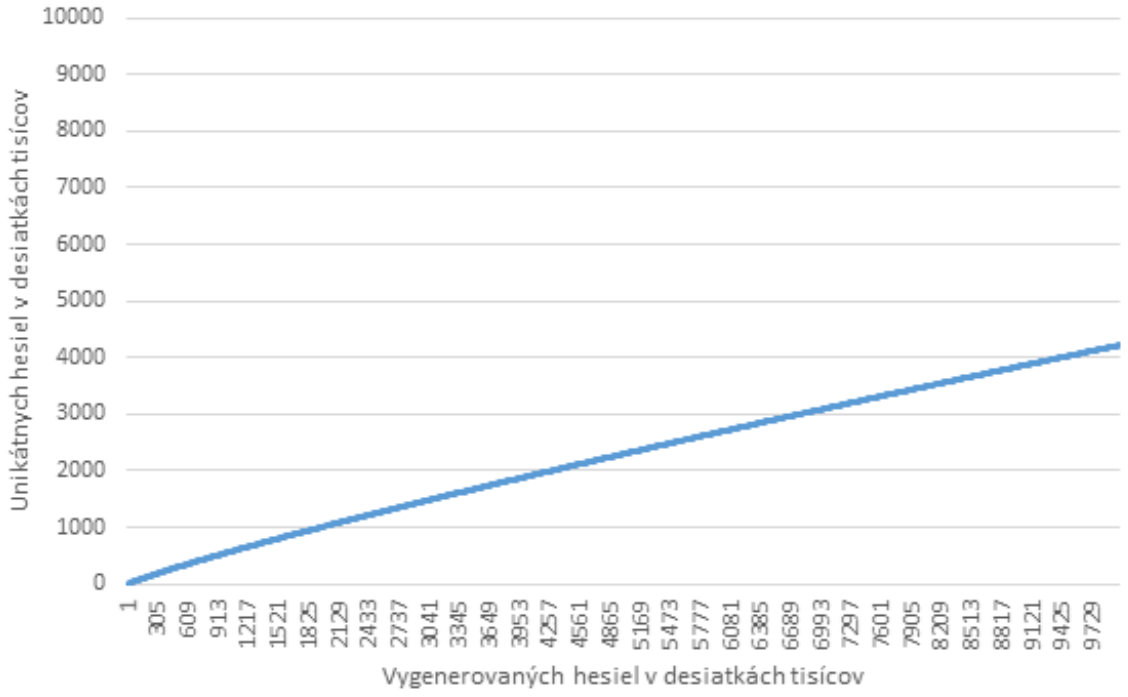
6.2.1 Heslá zo vstupného slovníka

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 v tisícoch 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.



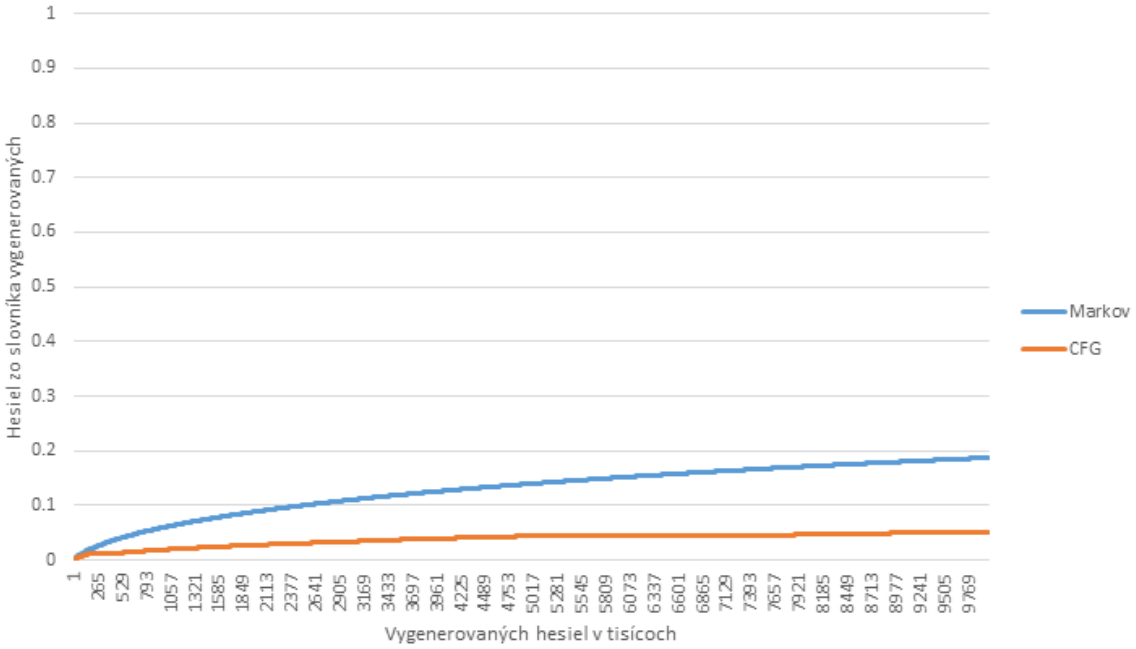
Obr. 6.1: Čas generovania hesiel

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. 6.2: Počet unikátnych hesiel

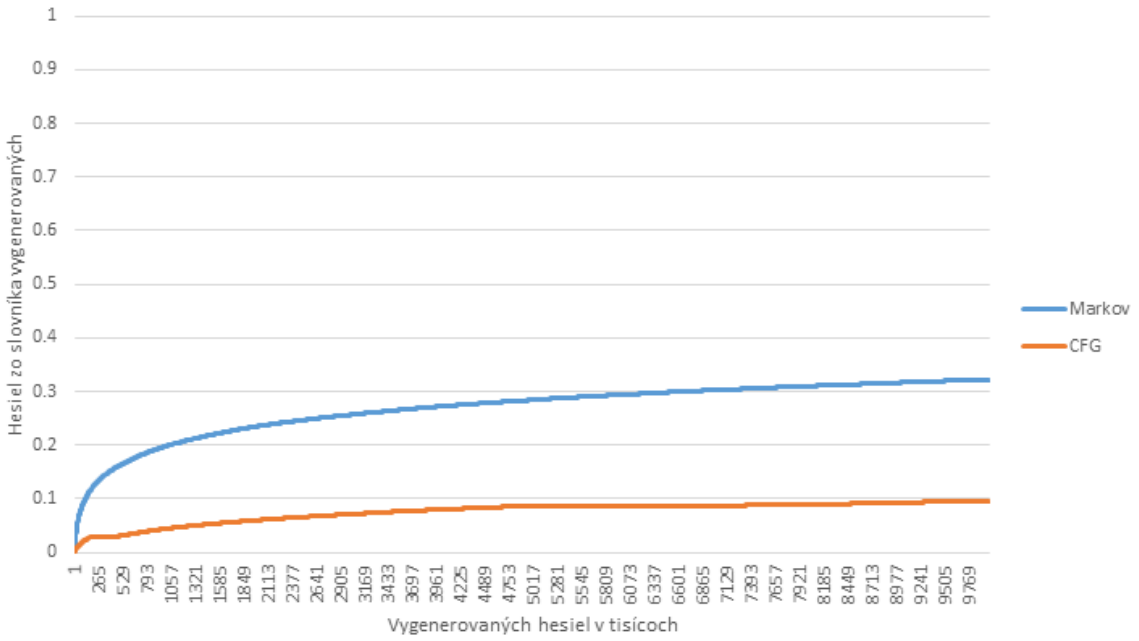
Heslá zo vstupného slovníka Na 6.3 vidíme priebeh hodnôt, kde heslá boli porovnávané so vstupným slonvíkom. Vidíme, že nami definované a implementované riešenie pomocou bezkontextových gramatík má na rozdiel od Markovovského zdroja omnoho pomalší rast počtu hesiel patriacich do slovníka. Pri 100 miliónoch 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.



Obr. 6.3: Pomer vygenerovaných hesiel zo vstupného slovníka

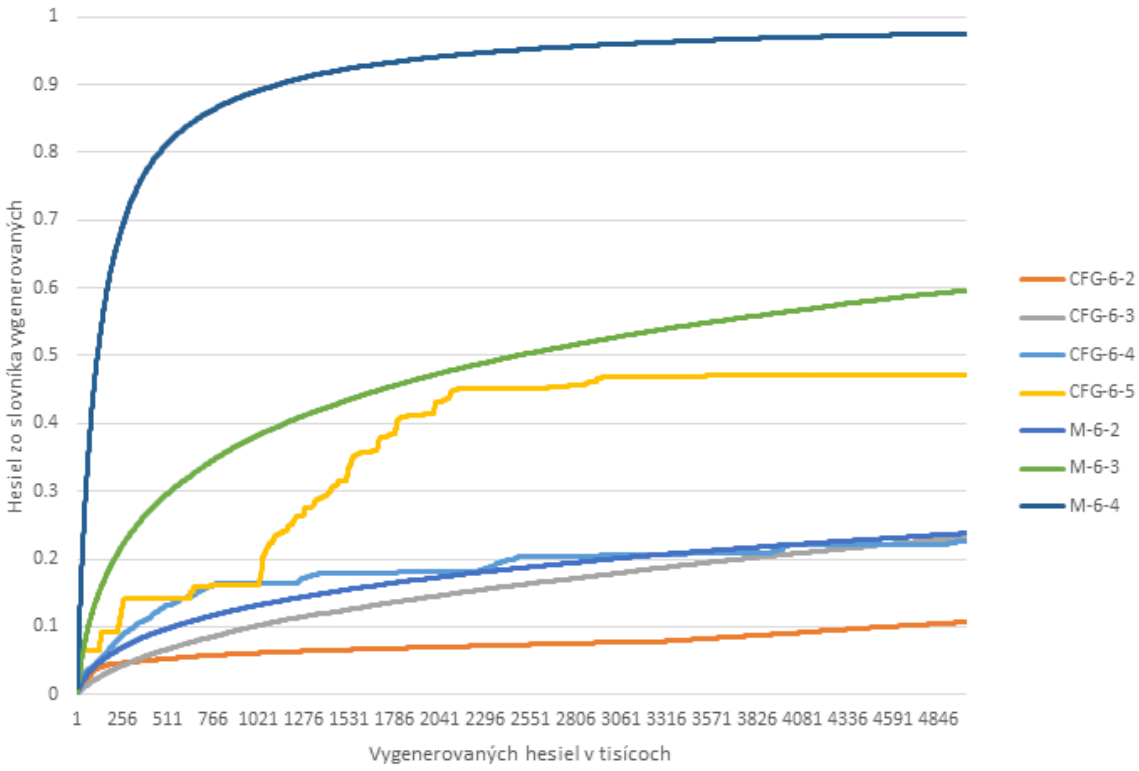
Heslá z nezávislého slovníka Graf 6.4 znázorňuje hodnoty po porovnaní vygenerovaných hesiel s iným nezávislým slovníkom. Pri tomto teste sme zobrali heslá vygenerované našimi algoritmami, ktoré na vstupe dostali slovník *rockyou*. Následne sme spravili testy počtu vygenerovaných hesiel, tentokrát avšak z iného ako vstupného slovníka. V tomto prípade sme použili slovník *phpbb*. Týmto by sme chceli ukázať schopnosť nášho algoritmu vygenerovať slovník veľmi podobný tým používaným v praxi.

Ďalej sme taktiež skúmali ako sa správajú nami implementované algoritmy ma menších dátach. Na obrázku 6.5 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 Markovovské zdroje sa toto číslo počíta z 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 Markovovské zdroje je v tejto metriky opäť lepší



Obr. 6.4: Pomer vygenerovaných hesiel z nezávislého slovníka

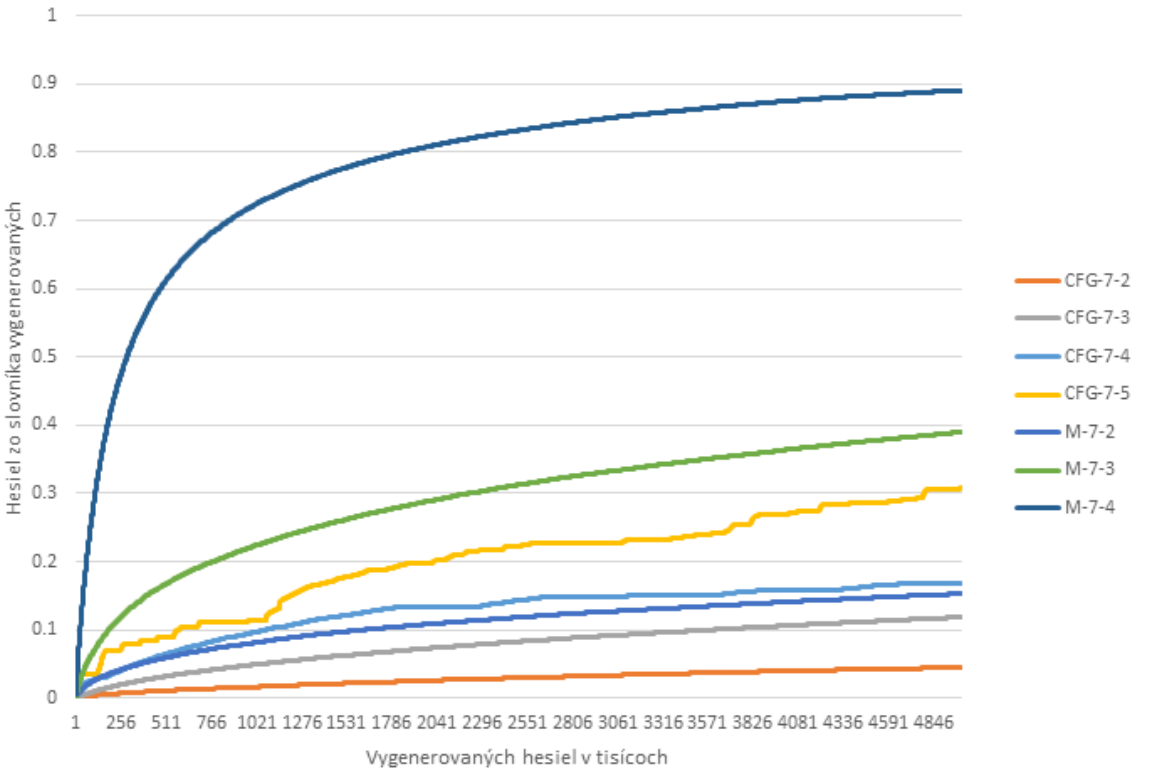
ako algoritmus používajúci pravdepodobnostné bezkontextové gramatiky. Použili sme slovník *phpbb* stiahnutý z [2], ktorý sme upravili aby všetky heslá mali dĺžku najviac 6 znakov. Takto upravený slovník mal nakoniec 55 744 rôznych hesiel.



Obr. 6.5: Počet vygenerovaných hesiel zo slovníka - dĺžka 6

Po zhliadnutí rovnakého grafu pre algoritmy pustené pre generovanie hesiel dĺžky

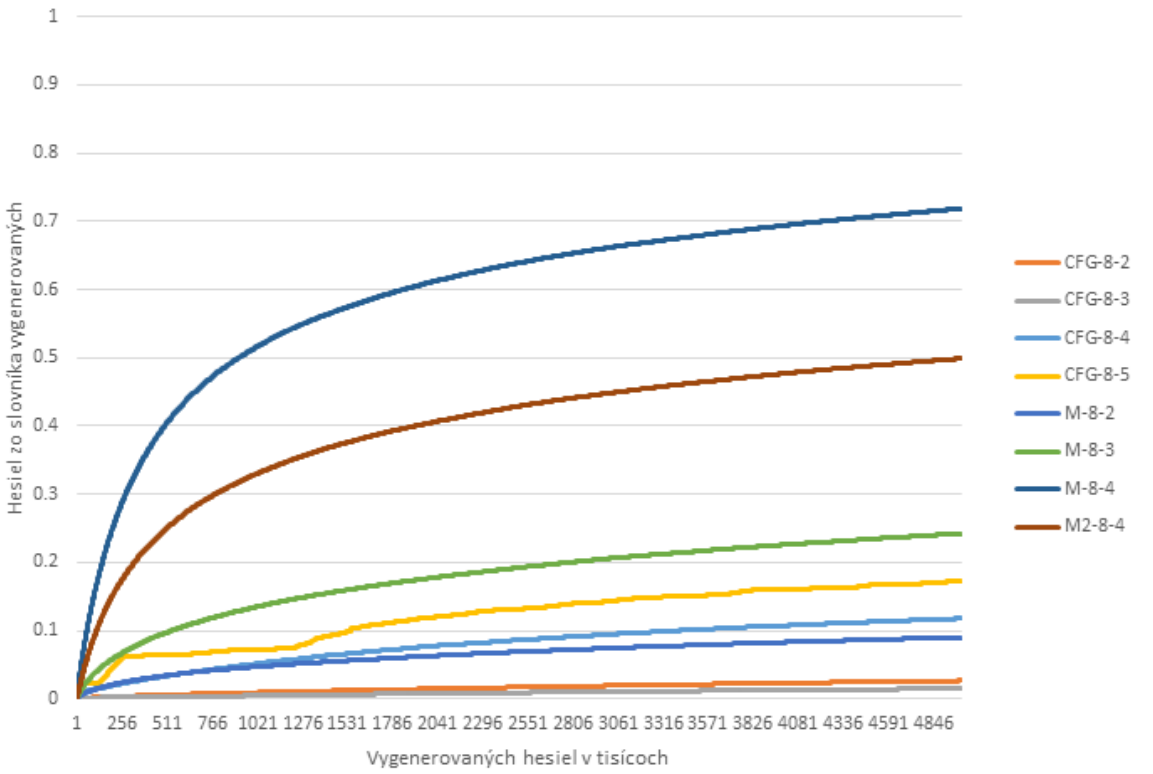
7 a s upraveným slovníkom phpbb obsahujúcim heslá najviac dĺžky 7 sme si všimli, že množstvo hesiel zo vstupného slovníka generovaných našimi algoritmami sa nezmenilo až na Markovovské zdroje s prefixom 4. V tomto teste bola veľkosť vstupného slovníka 88 416 hesiel.



Obr. 6.6: Počet vygenerovaných hesiel zo slovníka - dĺžka 7

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 Markovovské zdroje s prefixom nastaveným na dĺžku 4. V grafe zobrazujúci tieto dáta sme zobrazili výsledky tohto testu aj pre nami upravenú verziu Markovovských zdrojov, ktorá by mala byt schopná v konečnom čase vygenerovať heslá z celého priestoru možností. Označili sme ju ako *M2-8-4* keďže sa jedná o druhú verziu Markovovských zdrojov použitých v tejto práci.

V prípade nami upraveného Markovovského zdroja a algoritmu, ktorý ho používa pri generovaní hesiel sme taktiež testovali vplyv nastavenie konštánt δ a ε. V následujúcom grafe 6.8 zobrazujeme rozdiel v počte vygenerovaných hesiel zo vstupného slovníka pre jednotlivé parametre označené v legende grafu ako $\delta - \varepsilon$. Vidíme, že tieto parametre nemajú žiaden vplyv na prvých 5 miliónov hesiel vygenerovaných týmto algoritmom.



Obr. 6.7: Počet vygenerovaných hesiel zo slovníka - dĺžka 8

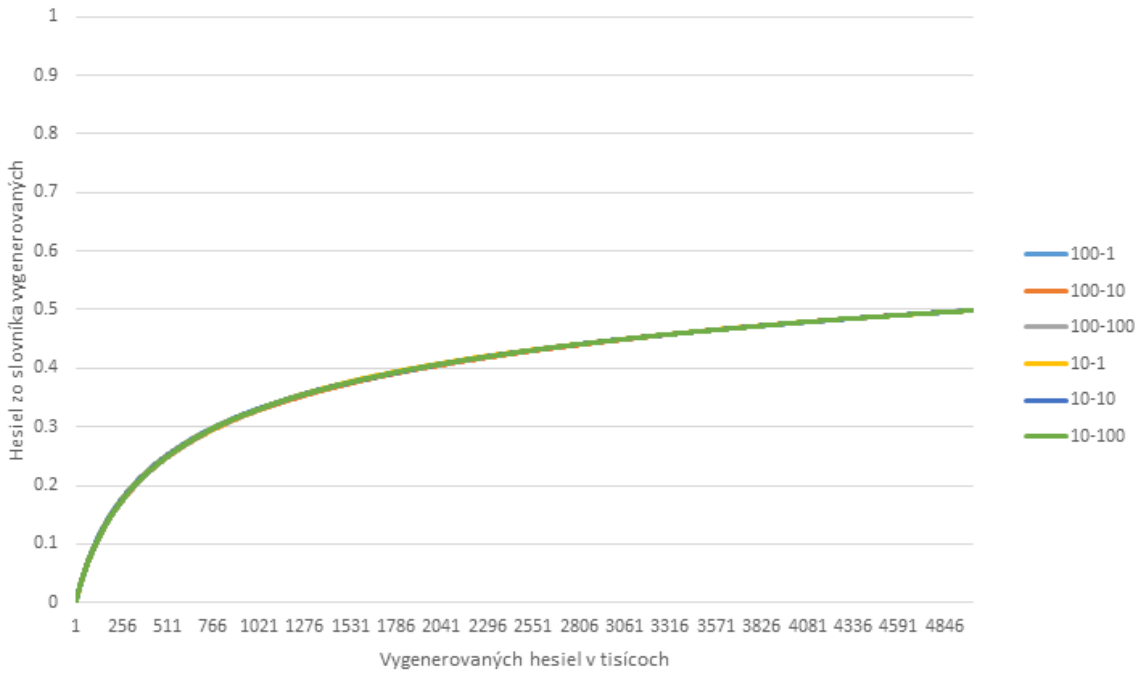
Na základe 6.3 a 6.4 vidíme, že nami navrhnutá metóda pomocou bezkontextových gramatík síce negeneruje veľa hesiel zo vstupného slovníka počas prvých miliónov vygenerovaných hesiel. Tento problém by sa dal vyriešiť, tým že by sme vždy ako prvé na výstup poslali všetky heslá zo slovníka, keďže ten býva zanedbateľne malý oproti veľkosti priestoru hesiel, ktorý musíme prehľadať aby sme definitívne našli hľadané heslo.

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. 6.8: 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
- *OPSV* - Odchýlka Pozície v Slovníku a na Výstupe - Absolútna hodnota rozdielu v pozícií na vstupe a na výstupe algoritmu prenásobená 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.

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

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

Tabuľka 6.2: Miery presnosti

	d	p	PPS	PPS %	RPSV	OPSV
CFG	6	2	31328	56.1	36.567	36.671
CFG	6	3	26358	47.2	37.749	37.762
CFG	6	4	31033	55.6	19.673	19.725
CFG	6	5	28554	51.2	21.397	21.484
CFG	7	2	47454	53.6	26.489	26.531
CFG	7	3	40343	45.6	25.831	25.848
CFG	7	4	46739	52.8	16.648	16.700
CFG	7	5	46718	52.8	21.715	21.804
CFG	8	2	88521	61.6	16.139	16.199
CFG	8	3	61113	42.5	22.384	22.436
CFG	8	4	70500	49.0	15.028	15.071
CFG	8	5	75570	52.5	14.749	14.865
Markov	6	2	25219	45.2	28.307	28.332
Markov	6	3	25959	46.5	14.815	14.840
Markov	6	4	27705	49.7	3.649	3.706
Markov	7	2	39090	44.2	21.289	21.323
Markov	7	3	40283	45.5	13.321	13.353
Markov	7	4	43232	48.8	4.700	4.757
Markov	8	2	61688	42.9	15.745	15.792
Markov	8	3	66401	46.2	9.701	9.748
Markov	8	4	68291	47.5	4.596	4.658
CFG	12	4	3781788	28.3	5.879	5.925
Markov	12	4	4609712	34.5	2.191	2.224

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 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 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 kedy algoritmus je odmenený znížením skóre ak sa mu podarí vygenerovať niektoré heslo skôr ako sa nachádza vo vstupnom slovníku. Posledná miera *odchýlka pozície v slovníku* má vyjadrovať absolútny rozdiel pozícií oproti vstupnému slovníku bez ohľadu na to, či heslo bolo vygenerované skôr ako vo vstupnom slovníku. Malý rozdiel týchto hodnôt naznačuje, že väčšina hesiel, ktoré boli generované našimi algoritmami bola vygenerovaná neskôr ako bol jej výskyt vo vstupnom slovníku.

KAPITOLA 7

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 implementácií 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 nedostatok vyžaduje vývoj algoritmu, ktorý dokáže generovať terminálne vetné formy gramatiky s použitím malého množstvá 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 δ a ε, 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 zbytku existujúcich hesiel. Toto avšak zahŕňa výskum, ktorý by sa dal rozobrať v samostatnej práci.

KAPITOLA 8

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 možností útokov hrubou silou, ktoré by sme mohli pri riešení nášho problému použiť.

Pri implementácií nášho riešenia používajúceho bezkontextové gramatiky sme používali znalostí 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. 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] T. Murakami, R. Kasahara, and T. Saito. An implementation and its evaluation of password cracking tool parallelized on gpgpu. In *Communications and Information Technologies (ISCIT), 2010 International Symposium on*, pages 534–538, Oct 2010.

[12] Eugene H Spafford. Observing reusable password choices. 1992.

[13] 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.

[14] Moshe Zviran and William J. Haga. A comparison of password techniques for multilevel authentication mechanisms. *The Computer Journal*, 36(3):227–237, 1993.