VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DISTRIBUOVANÉ GENEROVANIE HESIEL POMOCOU PRAVDEPODOBNOSTNÝCH GRAMATÍK

DISTRIBUTED PASSWORD GENERATION USING PROBABILISTIC GRAMMARS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DÁVID MIKUŠ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK HRANICKÝ**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Mikuš Dávid, Bc.**
Program: Informační technologie Obor: Počítačové sítě a komunikace
Název: **Distribuované generování hesel pomocí pravděpodobnostních gramatik**
Distributed Password Generation Using Probabilistic Grammars
Kategorie: Paralelní a distribuované výpočty

Zadání:

1. Seznamte se s možnostmi využití pravděpodobnostních bezkontextových gramatik pro účely lámání hesel a existujícím řešením PCFG Cracker (M. Weir).
2. Navrhněte způsob, jak generování hesel na základě existující gramatiky distribuovat na více výpočetních uzlů.
3. Navrhněte rozšíření existujícího nástroje pro distribuované lámání o vámi navržený způsob útoku.
4. Navržené rozšíření implementujte a srovnejte jej s naivním řešení, kdy jsou hesla z gramatiky generována do formy slovníku na jediném uzlu.
5. Zhodnoťte dosažené výsledky.

Literatura:

- WEIR, M., AGGARVAL, S., DE MEDEIROS, B., GLODEK, B. Password Cracking Using Probabilistic Context-Free Grammars. In: *30th IEEE Symposium on Security and Privacy*. Berkeley (CA): IEEE 2009. s. 391-405. ISBN 978-0-7695-3633-0.
- S. Houshmand, S. Aggarwal and R. Flood, "Next Gen PCFG Password Cracking," in *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 8, s. 1776-1791, Aug. 2015. ISSN 1556-6013.
- MA, J., YANG, W., LUO, M., LI, N. A Study of Probabilistic Password Models. In: *IEEE Symposium on Security and Privacy (SP)*. San Jose (CA): IEEE 2014. s. 689-704. ISSN 1081-6011.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Hranický Radek, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 30. října 2018

Abstrakt

Táto práca sa zaoberá procesom lámania hesiel, popisom existujúcich útokov a generovania hesiel na základe pravdepodobnostnej gramatiky. Táto gramatika sa dá použiť na útok, ktorý funguje na základe trénovania nad existujúcim zoznamom hesiel a následným generovaním hesiel pomocou zostrojenej bezkontextovej gramatiky z procesu trénovania. Jadrom práce je návrh a implementácia distribuovaného riešenia pre tento typ útoku. Implementácia zahŕňa prepísanie existujúceho riešenia a optimalizáciu využitia všetkých dostupných zdrojov.

Abstract

This thesis describes a process of password cracking, existing types of attacks and generating passwords using probabilistic grammar. This grammar can be used as an attack that works on the basis of learning from an existing list of passwords and generating them by using constructed context-free grammar from the learning phase. The core of this thesis is the design and implementation of distributed solution for this type of attack. Implementation includes refactoring of existing solution and optimization to maximize use of every available resource.

Kľúčové slová

heslo, lámanie, distribúcia, gramatika, pcfg

Keywords

password, cracking, distribution, grammar, pcfg

Citácia

MIKUŠ, Dávid. *Distribuované generovanie hesiel pomocou pravdepodobnostných gramatík.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Hranický

Distribuované generovanie hesiel pomocou pravdepodobnostných gramatík

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Radka Hranického. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Dávid Mikuš
20. mája 2019

Poďakovanie

Rád by som poďakoval svojmu vedúcemu Ing. Radkovi Hranickému za odborné vedenia a pomoc pri písaní tejto práce a kolegovi Bc. Filipovi Lištiakovi za vzájomné zdieľanie znalosti v problematike generovania hesiel.

Obsah

Kapitola 1

Úvod

Lámanie hesiel je proces, v ktorom sa útočník snaží prelomiť heslo užívateľa, poprípade sa užívateľ snaží obnoviť svoje zabudnuté heslo. Taktiež to využívajú vládne agentúry pri forenznej analýze zabavených elektronických prístrojov podozrivého [5]. Väčšinou sa jedná o zašifrované súbory. Prelomenie modernej šifry je takmer nemožné, preto sa útočník radšej snaží uhádnuť heslo užívateľa.

Ludia si väčšinou vyberajú heslá podľa vzorov, napríklad použitie veľkého písmena na začiatku hesla alebo pridanie číslic na koniec. Preto je výhodné uprednostniť skúšanie takýchto typov hesiel voči ostatným. Analýzou existujúcich hesiel je možné predikovať pravdepodobné heslá. Tento prístup sa snaží uhádnuť vzory, ktoré užívateľ mohol zvoliť. Jeden z prístupov je použitie pravdepodobnostnej bezkontextovej gramatiky, ktorý je veľmi úspešný. Existuje riešenie využívajúce tento prístup, ktoré vytvoril M. Weir a kol [16]. Toto riešenie má ale niekoľko nedostatkov, ako rýchlosť generovania hesiel a absencia distribuovaného výpočtu. Jedným z cieľov práce je odstrániť niektoré nedostatky tohto riešenia.

Proces lámania hesla je výkonovo náročný. Využívajú sa všetky dostupné prostriedky zariadenia ako CPU a GPU. Ale aj tieto prostriedky na jednom zariadení dosahujú svoj limit. Preto sa začali používať viaceré uzly, ktoré si tento proces rozdelia a tým to celé urýchlia.

Táto práca sa zaoberá návrhom a implementáciou distribuovaného generovania hesiel pomocou pravdepodobnostných bezkontextových gramatík. Pri použití jedného zariadenia sa naráža na limity, ktoré nejdú ďalej prekonať ako nedostatok pamäte a výpočtového výkonu. Preto sa začalo používať viacero výpočtových uzlov, aby bol proces lámania hesiel rýchlejší. Celý proces je nutné inteligentne rozložiť na menšie celky, ktoré budu riešiť jednotlivé uzly, aby každý uzol bol čo najefektívnejšie využitý.

Kapitola 2 rozoberá proces lámania hesiel, aktuálne typy útokov, ako sa rieši distribúcia hesiel a existujúce nástroje. V kapitole 3 sa rieši generátor hesiel na základe bezkontextovej pravdepodobnostnej gramatiky, ako funguje trénovanie, generovanie hesiel a použité algoritmy. Kapitola 4 rozoberá možné prístupy k distribúcii hesiel, výhody, nevýhody, porovnania jednotlivých riešení a zvolenie najlepšieho riešenia. Kapitola 5 popisuje chovanie reimplementovaného generátoru hesiel z gramatiky. V kapitole 6 je vysvetlená implementácia distribuovaného generátoru. A na záver v kapitole 7 sú popísane jednotlivé experimenty a zhodnotené výsledky.

Kapitola 2

Lámanie hesiel

Motiváciou lámať heslá je získať prístup k informáciám, ktoré patria obeti. Môže sa jednať o osobný účet alebo zašifrovaný disk, súbor. Využívajú to najmä vládne agentúry pri vyšetrovaní v oblasti forenznej analýzy, napríklad pri zabavení osobných zariadení obvineného a hľadaním dôkazov [5]. Taktiež to ale môže použiť vlastník, ktorý heslo zabudol alebo aj útočník pre zisk neautorizovaného prístupu. Rozlišujeme 2 typy: online a off-line.

Online útok je snaha o zisk prístupu k zariadeniu ktorý, nie je útočníkovi fyzický prístupný. Pri tomto type sa stále vyskytujú obranné mechanizmy, medzi najznámejšie patrí obmedzení počet pokusov o prihlásenie, kedy po prekročení je zablokovaný prístup útočníkovi napríklad pomocou IP adresy.

Pri off-line útoku je fyzický prístupne zariadenie kde, sa vyskytuje zašifrovaný obsah. Útočník už nie je limitovaný žiadnou dodatočnou ochranou ako sa môže vyskytovať pri online útoku. Nie je už obmedzený počtom pokusov prihlásení ale len, samotným výkonom vlastných prostriedkov. Pri tomto útoku sa ale nevykonáva útok priamo na šifrovací algoritmus, ale na samotné heslo. Šifrovanie (viď obr. 2.1) prebieha väčšinou zahešovaním hesla, ktoré ďalej vstupuje do šifrovacieho algoritmu ktorým sa zašifruje súbor.

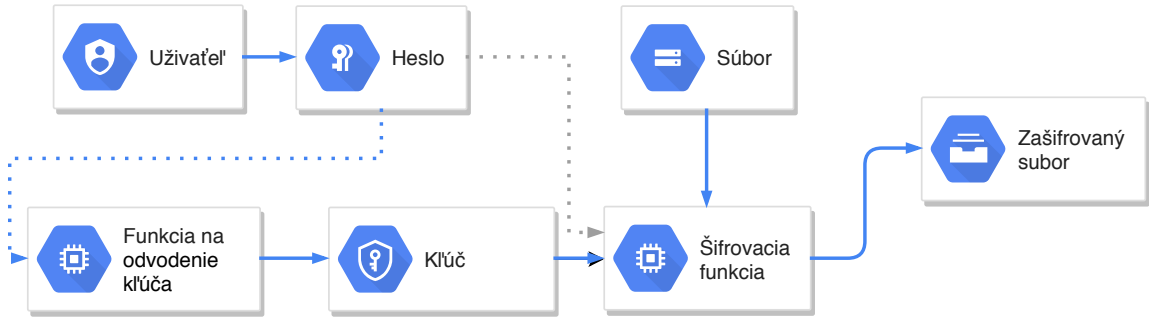
Zahešované heslo[12] je výstup z kryptografickej hešovacej funkcie. Táto funkcia dostane na vstup reťazec ľubovolnej dĺžky. Výstup je reťazec, ktorý ma vždy rovnakú dĺžku. Funkcia ma niekoľko dôležitých vlastnosti:

- odolnosť voči získania predlohy - je obtiažne nájsť vstup m aby platilo $h = hash(m)$,
- odolnosť voči získania predlohy - je obtiažne nájsť pre vstup x také y aby platilo $hash(x) = hash(y)$,
- odolnosť voči nájdeniu kolízie - je obtiažne nájsť 2 rôzne vstupy x, y, aby platilo $hash(x) = hash(y)$.

Pre zvýšenie času potrebného pre prelomenie hesla sa ďalej používa opakované použitie heš funkcie. Stanoví sa fixný počet kôl, v prvom kole na vstupe je heslo a jeho výstup znova putuje ako vstup. Týmto prístupom sa jednoducho spomalí proces overenia hesla bez nutnosti zmeny heš funkcie.

Útok spočíva skúšaním vybraných hesiel a overením či heslo je správne. To sa vykoná buď len zahešovaním hesla a overenie hešu ak je heš dostupný alebo následne ešte aj dešifrovaním súboru a overenie či obsah dát je validný. Aby sa nemusel validovať obsah dát, tak sa do zašifrovaného súbory prikladá výsledný heš na porovnanie. Do šifrovacej funkcie môže vstupovať buď samotné heslo alebo, výstup z funkcie na odvodenie kľúča *(key derivation function)* - KDF. KDF slúži pre generovanie kryptografických kľúčov z tajného reťazca

(napr. heslo) [1]. Takáto funkcia môže prijímať aj ďalšie vstupy ako napríklad soľ *(salt)*, alebo počet iterácii ktoré má vykonať. Výstup by mal byť nerozoznateľný od náhodného binárneho reťazca. Cieľ je získať silný kľúč pre šifrovaciu funkciu.



Obr. 2.1: Diagram šifrovania súboru

2.1 Typy útokov

Existuje viacero typov útokov podľa čoho vyberať heslá. Ak útočník má aspoň minimálnu znalosť o hesle, tak na základe toho môže prispôsobiť typ útoku. Napríklad pri snahe zistiť heslo anglický hovoriacej osoby bude pravdepodobnejšie, že heslo sa bude skladať z anglických slov a vďaka tohto poznatku môže útočník zmenšiť množstvo hesiel, ktoré bude skúšať.

Hrubá sila (Bruteforce)

Najjednoduchší ale najúspešnejší typ. V tomto útoku sa skúša každá kombinácia z vybranej znakovej sady. Nevýhodou je veľké množstvo hesiel, ktoré je nutné skúsiť a to s dĺžkou hesla rastie. Toto množstvo vyjadruje rovnica 2.1 kde S predstavuje veľkosť znakovej sady a max je maximálna dĺžka hesla. Tento útok sa používa len do určitej dĺžky a ďalej sa kombinuje s iným typom.

$$\sum_{i=1}^{max} S^i \tag{2.1}$$

Pri znakovej sade malých latinských písmen `a-z` pri zameraní na 5 znakové heslo bude skúšať heslá od `aaaaa` po `zzzzz`. Záleži na vybranom nástroji či bude zvyšovať najľavejší znak (`aaaaa -> baaaa`) alebo najpravejší znak (`aaaaa -> aaaab`), čo môže ovplyvniť čas prelomenie hesla.

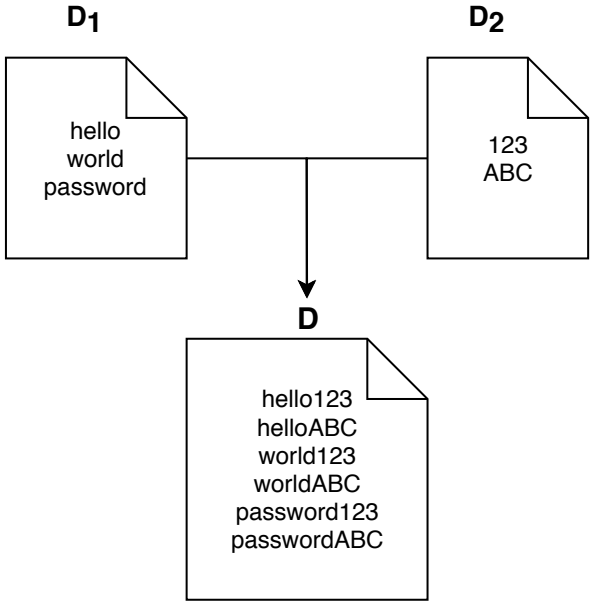
Slovníkový útok (Dictionary attack)

Slovníkový útok je podobný útoku s hrubou silou, ale namiesto toho aby systematický generoval heslá zo zvolenej znakovej sady, tak má k dispozícii slovník hesiel z ktorého čerpá v ktorom sa nachádzajú bežné slová alebo, často používane heslá napríklad z uniknutých a prelomených databáz. Tieto slová sa získavajú z bežných lexikálnych slovníkov. Útok je omnoho rýchlejší ako útok hrubou silou kvôli menšiemu množstvu hesiel, ktoré závisí len na veľkosti slovníka.

Pre zrýchlenie existuje tzv. *Rainbow table*, pred útokom sa zostrojí tabuľka, do ktorej sa uloží heslo v čistej forme a jeho heš. Výhodou je, že pri útoku stačí len porovnávať samotný heš a netreba ho vypočítavať na úkor pamäti, kde je nutné tabuľku uložiť. Obranou je použitie soli (*salt*), ktorá je uložená spolu s heslom a spolu s ním vstupuje aj do hešovacej funkcie, tým sa zabraní efektívnom použití tabuľky. Pretože vstup už nebude len samotné heslo ale aj soľ, ktorú útočník nepozná pokým sa k zašifrovanému súboru nedostane. Pri použití soli už nejde pred-vypočítať tabuľku pre každú kombináciu hesla a soli, bolo by to príliš výpočtovo aj pamäťovo náročné.

Ďalšia forma je kombinovanie slovníkov, kde sa kombinujú heslá z jednotlivých slovníkoch v danom poradí. Celkový počet hesiel viď 2.2 je p ktoré vznikne ako súčin mohutnosti slovníkov, ukážka viď 2.2. Kde D_i je slovník s indexom i a n udáva počet slovníkov.

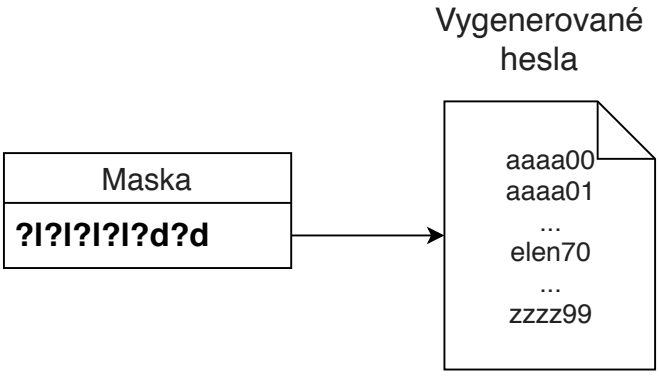
$$p = \prod_{i=1}^{n} |D_i| \tag{2.2}$$



Obr. 2.2: Ukážka kombinovania slovníkov D1 a D2

Útok pomocou masky (Mask attack)

Využíva sa maska zložená zo zástupných znakov, ktoré definujú znaky za ktoré sa nahradia. Ak definujeme `?l` ako zástupný znak pre malé písmena a `?d` pre číslice, tak sekvencia `?l?l?l?l?d?d` bude generovať heslá, ktoré sú 6 znakové, na prvých 4 miestach bude malé písmeno nasledované 2 číslicami. Výhodou je možnosť definovať aj útok hrubou silou, len pre každú dĺžku treba zadať masku. Analýza hesiel môže poskytnúť vzory ktoré sa následne dajú definovať maskou. Napríklad, jeden vzor môžu byť nazačiatku písmena nasledované 4 číslicami, čo môže predstavovať meno a rok narodenia.



Obr. 2.3: Príklad masky

Markovský model

Markovské modely [10] sa zvyčajne používajú v spracovaní prirodzeného jazyka a sú základom pre systémy na rozpoznávanie reči. Tieto modely sa začali používať aj pre generovaní hesiel, tak že jednotlivé znaky predstavujú skrytý Markovský model. Pri hesle `x`, `n-tý` znak hesla je daný pravdepodobnosťou znaku na pozícii `n-1`. Pravdepodobnosť výskytu hesla je potom daná ako súčin jednotlivých znakov, ktoré vyjadruje vzorec 2.3:

$$P(x_1x_2...x_n) = \upsilon(x_1)\prod_{i=1}^{n-1}\upsilon(x_{i+1}|x_1), \tag{2.3}$$

kde P(.) je Markovská pravdepodobnosť distribúcie na reťazci, x_i sú jednotlivé znaky a υ je funkcia, ktorá vyjadruje frekvenciu jednotlivých znakov a digramov v texte. Frekvencia je získania z frekvenčnej analýzy databázy prelomených hesiel, alebo slovníka. Pri takomto type útoku je použitá väčšinou databáza prelomených hesiel, pretože obsahuje aj čísla a špeciálne znaky.

Pre jedno vrstvový model sa zostaví 2D matica ktorá, obsahuje na prvej pozícii riadku znak, za ktorým nasledujú ďalšie znaky, ktoré sú zoradené podľa pravdepodobnosti a predstavujú aký znak môže nasledovať. Na príklade ukázanom v tabuľke 2.4 to znamená, že najväčšia pravdepodobnosť výskytu znaku po znaku `a` bude znak `n` a po ňom znak `l`, po znaku `b` to bude znak `o`.

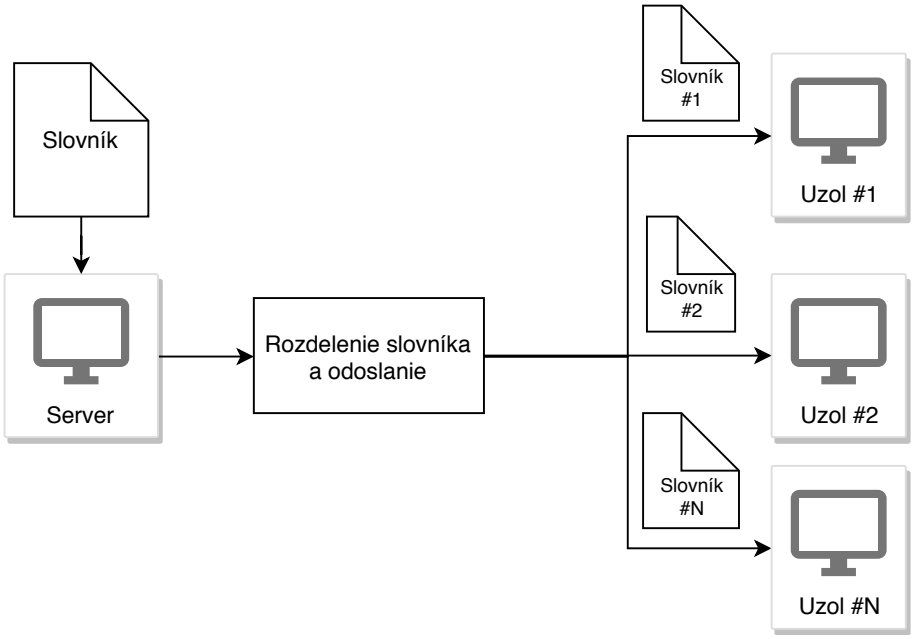
Existujú aj viac vrstvové modely, kde sa využíva pravdepodobnosť viacero predchádzajúcich znakov, tvorba takéhoto modelu je ale pomalá a vyžaduje väčší slovník pre frekvenčnú analýzu [15].

ε	a	n	p	...
a	n	l	t	...
b	o	e	a	...
c	h	i	e	...
...	...	...	...	...
z	a	e	o	...

Obr. 2.4: Príklad tabuľky pre Markov model prvého radu

2.2 Distribúcia

Distribúciou hesiel je možné docieliť zrýchlenia procesu lámania pomocou využitia výkonu viacero výpočetných uzlov. Existujú viaceré techniky ako to docieliť efektívne a závisí to aj od typu útoku. Najjednoduchšia technika je fixne rozdeliť sadu medzi jednotlivé uzly na kúsky (*chunks*), ktoré sa rozpošlú a následne sa už len čaká na na spätnú správu o úspechu či neúspechu, pri úspechu uzol vráti nájdene heslo. Pri slovníkovom útoku by to znamenalo rozdeliť slovník na rovnaké časti viď 2.5.



Obr. 2.5: Rozdelenie uzlov pri slovníkovom útoku

Táto jednoduchá technika má ale obmedzenia. Predpokladá, že každý uzol je rovnako výkonný, čo nemusí byt vždy pravda a musí mať konečný stavový priestor hesiel, čo pri útoku hrubou silou môže byť problém, keďže je nekonečný. Riešením by bolo priestor orezať stanovením maximálnej dĺžky hesla.

Ďalšou technikou je postupne rozdeľovanie úloh. Úloha je postupné rozdeľovaná a odoslaná na uzly [8]. Uzol prevezme svoj kúsok úlohy a po jeho skončení obdrží ďalší, dokým sa stavový priestor nevyčerpá, alebo jeden z uzlov heslo nenájde. Výhoda tohto prístupu je možnosť rozdeľovať úlohu nerovnomerne, tak aby každý uzol spracoval svoj kúsok za rovnaký čas ako ostatné. Jeden spôsob ako to docieliť je sledovať, za aký čas uzol spracoval svoj kúsok a následne pri prideľovaní ďalšieho, mu môže server poslať väčší či menši kúsok.

Ďalšou výhodou je väčšia odolnosť voči chybám, ak vypadne jeden z uzlov tak, server sa to po čase dozvie a odošle jeho kúsok inému uzlu. Rozdeliť úlohu na kúsky sa dá na základe indexu hesla [6] použitá v nástroji FITCrack (viď 2.3.3).

Definujme p ako heslo nad abecedou Σ, $p \in \Sigma^*$. Množinu $P \subset \Sigma^*$ označuje stavový priestor, podoba tejto množiny závisí na konkrétnom typu útoku. Pre náš účel bude P vždy konečná usporiadaná množina. Definujeme si funkciu *generátor hesiel* ako $g(i) : N \rightarrow P, i \in < 0, |P| - 1 >$ a i je index hesla.

Pri útoku hrubou silou nad abecedou $\Sigma = \{a, b, .., z, 0, .., 9\}$ pri rozsahu hesla 1 až 2 by to znamenalo:

$$\begin{aligned} g(0) = a, \quad g(25) = z, \quad g(35) = 9 \\ g(36) = aa, \quad g(61) = az, \quad g(1331) = 99 \end{aligned} \tag{2.4}$$

Následne sa dá úloha jednoducho rozdeliť na základe rozsahu indexov. Pri slovníkovom útoku by to znamenalo index hesla v slovníku.

2.3 Nástroje

Existuje niekoľko nástrojov, ktoré automatizujú proces lámania hesiel. Líšia sa podporou hešov, ktoré sú schopné prelomiť, ďalej rýchlosťou čo zahrňuje či mimo CPU podporujú aj GPU alebo FPGA. Jedná významná vlasnosť je schopnosť distribúcie na viacero uzlov, čo umožňuje zvýšiť rýchlosť lámania použitím viacero výpočetných prostriedkov.

2.3.1 John the Ripper

John the Ripper[1] je jeden z najstarších udržiavaných nástrojov. Pôvodne bol určený len na unixove *crypt* heše. Aktuálne podporuje viacero typov hešov, aj keď nie také množstvo ako spomínaný Hashcat. Je to open-source projekt, takže sa dá jednoducho pozrieť ako lámanie presne vykonáva. Obsahuje grafickú nadstavbu, ktorá zjednodušuje jeho použitie pre začiatočníkov.

Mimo útokov hrubou silou a slovníkového podporuje pravidlá pre modifikáciu hesiel. Obsahuje množstvo pravidiel[2], ale taktiež si užívateľ môže vytvoriť vlastné. Pravidlá sa používajú pri slovníkovom útoku a slúžia pre prispôsobenie vstupného slovníka. Každé pravidlo modifikuje heslo zmenou, orezaním alebo pridaním reťazca. Príklad pravidla je kapitalizácia, ktorá nastaví prvý znak na veľké písmeno a zvyšné na malé.

2.3.2 Hashcat

Hashcat[3] je open-source nástroj ktorý obsahuje veľké množstvo funkcii. Podporuje populárne operačne systémy (Windows, Linux, MAC), platformy (CPU, GPU, FPGA, ...), rôzne typy hešov (MD5, SHA2, SHA3, ...). Je to najmä CLI nástroj, počas útoku zobrazuje niekoľko výpisov o priebehu (viď obrázok 2.6): rýchlosť lámania na jednotlivých zariadení, aktuálne lámane heslo a ďalšie veci ako teplota zariadení. Vďaka licencii MIT a verejným zdrojovým kódom, vývoj napreduje začlenením komunity. Mimo zdokonaľovania samotného Hashcatu, vznikajú ďalšie nástroje postavené nad tým.

Medzi ďalšie vlastnosti patrí meranie výkonu (*benchmark*), vďaka ktorému je možné zistiť približnú rýchlosť obnovy hesla pre dané zariadenie a vybraný heš. Umožňuje pozastavenie procesu lámania hesla a vytvorenie bodu obnovy, od ktorého sa dá neskôr znova začať. Je považovaný za najlepší nástroj pre obnovu hesiel. Dôvod je najmä jeho rýchlosť, ktorú poskytuje pre overovanie jednotlivých hesiel. Používa OpenCL kernely, pomocou ktorých celý proces overovania paralelizuje na grafických kartách. Mimo toho poskytuje viaceré generátory hesiel a typy útokov.

Hashcat disponuje parametrami `skip` a `limit`, ktoré preskočia časť slovníka a obmedzí počet hesiel. Vďaka tomuto sa dá implementovať jednoduchá distribúcia, kde každý uzol

[1] https://www.openwall.com/john
[2] https://charlesreid1.com/wiki/John_the_Ripper/Rules
[3] https://hashcat.net/

dostane slovník a následne parametrami `skip` a `limit` sa určí časť slovníka, ktorú bude používať pre lámanie.

Nevýhoda je absencia grafického rozhrania, ale existujú neoficiálne nadstavby ako HashKiller[4]. Ostatné nástroje v porovnaní s hashcatom nepodporujú také veľké množstvo kryptografických hešov a ani nedosahujú jeho rýchlosti.

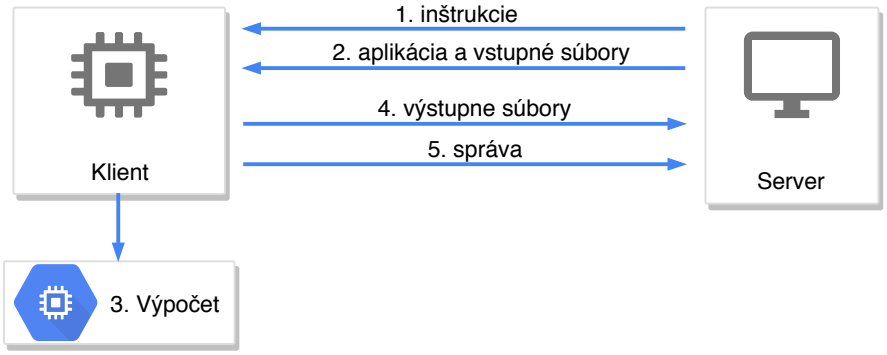
```
hashcat (v5.0.0) starting...

OpenCL Platform #1: NVIDIA Corporation
======================================
* Device #1: GeForce GTX 1080, 2028/8112 MB allocatable, 20MCU
* Device #2: GeForce GTX 1080, 2029/8119 MB allocatable, 20MCU
* Device #3: GeForce GTX 1080, 2029/8119 MB allocatable, 20MCU
* Device #4: GeForce GTX 1080, 2029/8119 MB allocatable, 20MCU

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates

Applicable optimizers:
* Optimized-Kernel
* Zero-Byte
* Single-Hash
* Single-Salt
* Brute-Force

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 55

Watchdog: Temperature abort trigger set to 90c

Session..........: hashcat (Brain Session/Attack:0xc054fc8f/0x6e7dc2f0)
Status...........: Running
Hash.Type........: phpass, WordPress (MD5), phpBB3 (MD5), Joomla (MD5)
Hash.Target......: $H$Js5boZ2wsUlgl2tI6b5PrRoADzYfXD1
Time.Started.....: Sun Oct 28 17:02:05 2018 (11 secs)
Time.Estimated...: Fri Nov 21 04:22:41 9862 (7844 years, 23 days)
Guess.Mask.......: ?a?a?a?a?a?a?a?a [8]
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     6684 H/s (94.69ms) @ Accel:256 Loops:1024 Thr:256 Vec:1
Speed.#2.........:     6653 H/s (95.15ms) @ Accel:256 Loops:1024 Thr:256 Vec:1
Speed.#3.........:     6746 H/s (93.82ms) @ Accel:256 Loops:1024 Thr:256 Vec:1
Speed.#4.........:     6720 H/s (94.20ms) @ Accel:256 Loops:1024 Thr:256 Vec:1
Speed.#*.........:    26809 H/s
Recovered........: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.........: 0/66342043128906625 (0.00%)
Rejected.........: 0/0 (0.00%)
Brain.Link.#1....: RX: 1.3 MB (0.00 Mbps), TX: 10.5 MB (0.00 Mbps), idle
Brain.Link.#2....: RX: 1.3 MB (0.00 Mbps), TX: 10.5 MB (0.00 Mbps), idle
Brain.Link.#3....: RX: 1.3 MB (0.00 Mbps), TX: 10.5 MB (0.00 Mbps), idle
Brain.Link.#4....: RX: 1.3 MB (0.00 Mbps), TX: 10.5 MB (0.00 Mbps), idle
Restore.Point....: 0/66342043128906625 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:102400-103424
Restore.Sub.#2...: Salt:0 Amplifier:0-1 Iteration:103424-104448
Restore.Sub.#3...: Salt:0 Amplifier:0-1 Iteration:105472-106496
Restore.Sub.#4...: Salt:0 Amplifier:0-1 Iteration:106496-107520
Candidates.#1....: sarierin -> b2*12312
Candidates.#2....: ahLIERIN -> jURRIESS
Candidates.#3....: hNherane -> iQTRIESS
Candidates.#4....: d&serane -> 2$712312
Hardware.Mon.#1..: Temp: 56c Fan: 32% Util:100% Core:1822MHz Mem:4513MHz Bus:1
Hardware.Mon.#2..: Temp: 58c Fan: 34% Util:100% Core:1809MHz Mem:4513MHz Bus:1
Hardware.Mon.#3..: Temp: 54c Fan: 31% Util:100% Core:1847MHz Mem:4513MHz Bus:1
Hardware.Mon.#4..: Temp: 59c Fan: 35% Util:100% Core:1835MHz Mem:4513MHz Bus:1

[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit =>
```

Obr. 2.6: Rozhranie nástroja hashcat pri lámaní hesla

2.3.3 Fitcrack

Fitcrack[5] [6] je distribuovaný systém pre obnovu hesiel šifrovaných médií a lámania kryptografických hešov. Využíva nástroj Hashcat, vďaka čomu dosahuje veľkého množstva podporovaných formátov a rýchlosti pri využitia dostupného hardware. Využíva technológie

[4] https://hashkiller.co.uk/
[5] https://fitcrack.fit.vutbr.cz/



Obr. 2.7: Komunikácia v systéme BOINC

OpenCL a open-source framework Berkeley Open Infrastrucute for Network Computing (BOINC), ktorý slúži pre automatizáciu riadenia distribuovaného výpočtu.

Výpočetnu sieť tvorí server, poprípade servery a množstvo klientov. Server riadi, priraďuje a plánuje úlohy na výpočet, pričom jednotlivé klienti vykonávajú samotný výpočet a jeho výsledok informuje server. Sieť funguje na klient-server architektúre (viď 2.7), klient sa pripája na server a žiada o úlohu. Server na základe aktuálneho stavu priradí kúsok úlohy ktoré sú tvorené na mieru konkrétneho uzlu.

2.3.4 Hashtopolis

Hashtopolis[6] je multiplatformový nástroj pre distribúciu úloh z hashcatu na viacero zariadení. Hlavným cieľom je prenositeľnosť, robustnosť a podpora viacero užívateľov zároveň. Nástroj je založený na klient-server architektúre a skladá sa z 2 častí:

1. Agent - podpora viacero klientov (C#, Python),
2. Server - PHP aplikácia, ktorá operuje na dvoch koncových uzlov, sprostredkúvajúc Adminské rozhranie a prípojný bod pre Agentov.

Hashtopolis komunikuje pomocou HTTP(S) použitím serializačného formátu JSON (*JavaScript Object Notation*). Výhoda tohto prístupu je vysoká použiteľnosť aj v obmedzených sieťach. Príklad správy je znázornený vo výpise 2.1, ktorou Agent oznamuje serveru prelomený heš s heslom `Happy123`.

```
{
    "action":"solve",
    "token":"ceLJOah9YB",
    "chunk":76,
    "progress":1,
    "total":1,
    "state":5,
    "cracks":[
        "$DCC2$10240#USER3#e098509e3489df1a73f1d2d997efc6f6:Happy123"
    ]
}
```

Výpis 2.1: Správa o prelomenom heši z nástroja Hashtopolis

[6] https://github.com/s3inlc/hashtopolis

Medzi výhody patrí: dostupnosť cez webové rozhranie (viď obrázok 2.9), serverová architektúra kompatibilná s bežnými poskytovateľmi webových služieb (PHP + MySQL), manažment slovníkov a pravidiel, automatická aktualizácia hashtopolisu a Hashcatu, štatistika bežiacich úloh, vizuálna reprezentácia distribúcie úloh.

Pôvodne bol hashtopolis určený len pre distribúciu úloh na agentov používajúcich Hashcat. Neskôr bol doplnený o podporu pre vlastné lámacie binárne súbory, ktoré môžu byť použité paralelne na rozličných úloh. Pre každú úlohu je zvolený lámací softvér. Klient si následne stiahne špecifický lámací softvér. Okrem toho, podporuje aj verzovanie vďaka čomu je možné zvoliť si verziu softvéru pri vytváraní úlohy. Výber verzie dovoľuje si vybrať špecifickú verziu, ktorá má najlepší výkon pre lámaný heš. Ďalšia vlastnosť je posielanie notifikácii. Užívateľ si vytvorí notifikáciu zloženú z:

1. Spúšť - definuje typ akcii, ktoré spustia notifikáciu. Jedná sa napríklad o prelomenie hešu alebo, či bola vytvorená nová úloha,

2. Typ - udáva, kde sa notifikácia odošle, umožňuje zadať email, alebo poslanie správy cez cURL programom ako Slack alebo Mattermost. Mimo toho je možnosť si vytvoriť aj vlastný typ.

2.3.5 Hashstack

Hashstack[7] je komerčná služba pre lámanie hesiel od firmy Terahash. Poskytuje predaj hardvéru spolu s vlastným softvérovým riešením založeným na Hashcate. V rámci softvéru obsahuje aj webového rozhranie pre správu distribúcie.

Oproti ostatným nástrojom, ktoré poskytujú GPU akceleráciu len pre malú množinu dostupných hešovacích formátov. Hashstack podporuje GPU akceleráciu pre viac ako 375 hešovacích formátov. Udáva výkon lepší o 35 % oproti ostatným nástrojom využívajúc podobný hardvér. Hashstack pracuje na niekoľkých hešov paralelne a aj v rámci úloh, čo umožňuje nájsť väčšinu hesiel v najkratšom možnom čase.

Poskytuje horizontálnu škálovateľnosť pridávaním ďalších zariadení. Zákazník si môže kúpiť ľubovolné množstvo zariadení, aký mu rozpočet dovolí. Hashstack všetky tieto zariadenia dokáže využiť pre distribuovaný proces lámania hesiel. Informácie o priebehu lámania posktuje webového rozhranie viď obrázok 2.10. Najväčšou výhodou nie je využitie čistého výpočtového výkonu, ale schopnosť pracovať na 200 úlohách paralelne.

Zaručuje aj vysokú odolnosť voči chybám, ak vypadne zariadenie z výpočtu, neznamená to, že úlohy zlyhajú. Hashstack poruchové zariadenie vyradí z výpočtu, dokým sa problémy nevyriešia. Takáto elastickosť umožňuje zároveň pridávanie ďalších zariadení za behu lámania. Jednoducho sa pripojí do distribuovaného výpočtu a začne pracovať na aktívnych úlohách v rade.

Nevýhodou je nedostupnosť zdrojových súborov. Softvérové riešenie je na mieru prispôsobené pre ich hardvér. Vysoká cena zariadení, ktorá sa pohybuje od 15 950 $ za najlacnejšie až po 31 700 $.

2.3.6 PassWare Kit

PasswareKit[8] je nástroj, ktorý objaví všetky zašifrované súbory na zariadení a dešifruje ich. Práca s nástrojom je sprostredkovaná pomocou grafického rozhrania viď obŕazok 2.11.

[7] https://terahash.com/
[8] https://www.passware.com/kit-forensic/

Podporuje viac ako 280 súborových formátov (MS Office, PDF, Zip, RAR, ...). Vie pracovať so zálohami a dátami z cloudových služieb (Apple iCloud, MS OneDrive, Dropbox). Ďalšia vlasnosť je analýza pamäti, hibernačných súborov a extrakcia kľúčov pre FileVault2, TrueCrypt, VeraCrypt, BitLocker, účty pre Windows a Mac. Okrem toho umožňuje aj lámanie hesiel pre iPhone/iPad a Androidové zálohy.

Na lámanie hesiel okrem využitia CPU, vie aj akcelerovať výpočet pomocou viacero grafických kariet, čo vďaka jednej GPU môže narásť rýchlosť obnovy až 400-násobne. Využíva *rainbow tables* pri formátoch, kde je to možné, vďaka čomu sa nemusí lámať heslo, ale len sa z tabuľky pre daný heš vytiahne heslo.

Softvérove riešenie obsahuje aj tzv. Passware Kit Agent, ktorý slúži pre distribuované lámanie hesla po sieti. Podporuje Windows (64-bit), Linux (64-bit) ale aj cloudovú instanciu EC2 od Amazonu. Škálovatelnosť výkonu pridaním ďalších agentov je lineárna. Pridaním druhého agenta s výkonom rovnakým ako prvý, zdvojnásobí rýchlosť lámania.

2.3.7 Cain & Abel

Cain & Abel[9] je jeden z najstarších nástrojov pre obnovu hesiel pre operačný systém Windows. Pre lámanie hesiel podporuje útok hrubou silou, slovníkový a kryptoanalytický útok. Okrem lámania hesiel poskytuje aj odpočúvanie sieti, nahrávanie VoIP konverzácii, obnovu Wi-Fi kľúčov, analýzu smerovacích protokolov, MITM (*Man in the middle*) útok pomocou ARP. V rámci odpočúvaní sieti vie analyzovať šifrované protokoly ako SSH-1 a HTTPS. Obsahuje filtre pre zachytávanie prihlasovacích údajov z rozličných autentizačných mechanizmov.

Posledná vydaná verzia bola v roku 2014 a už nie je v aktívnom vývoji. Poskytuje síce množstvo funkcii mimo lámania hesiel, ale v tomto ohľade nevyniká oproti ostatným nástrojom. Podporuje len niekoľko typov hešov a hlavne nevie využiť GPU pre akceleráciu výpočtu.

2.3.8 Password recovery toolkit

Password recovery toolkit je komerčný nástroj od firmy AccessData[10] využívaný vladnými agentúrami. Podporuje klasické typy útokov ako slovníkový a útok hrubou silou. Analyzuje obsah súborov pre zistenie typu šifrovania. Pred každým lámaním súboru sa pre súbor vytvorí heš, ktorý slúži na zistenie či pri procese lámania sa obsah súboru nezmenil. Nástroj poskytuje *Rainbow tables*, ktoré obsahujú predvypočítane heše, vďaka ktorým sa nemusí heš počítať ale nájde sa v tabuľke. Týmto sa ušetrí množstvo výpočtového výkonu, ak sa heslo nachádza v tabuľke.

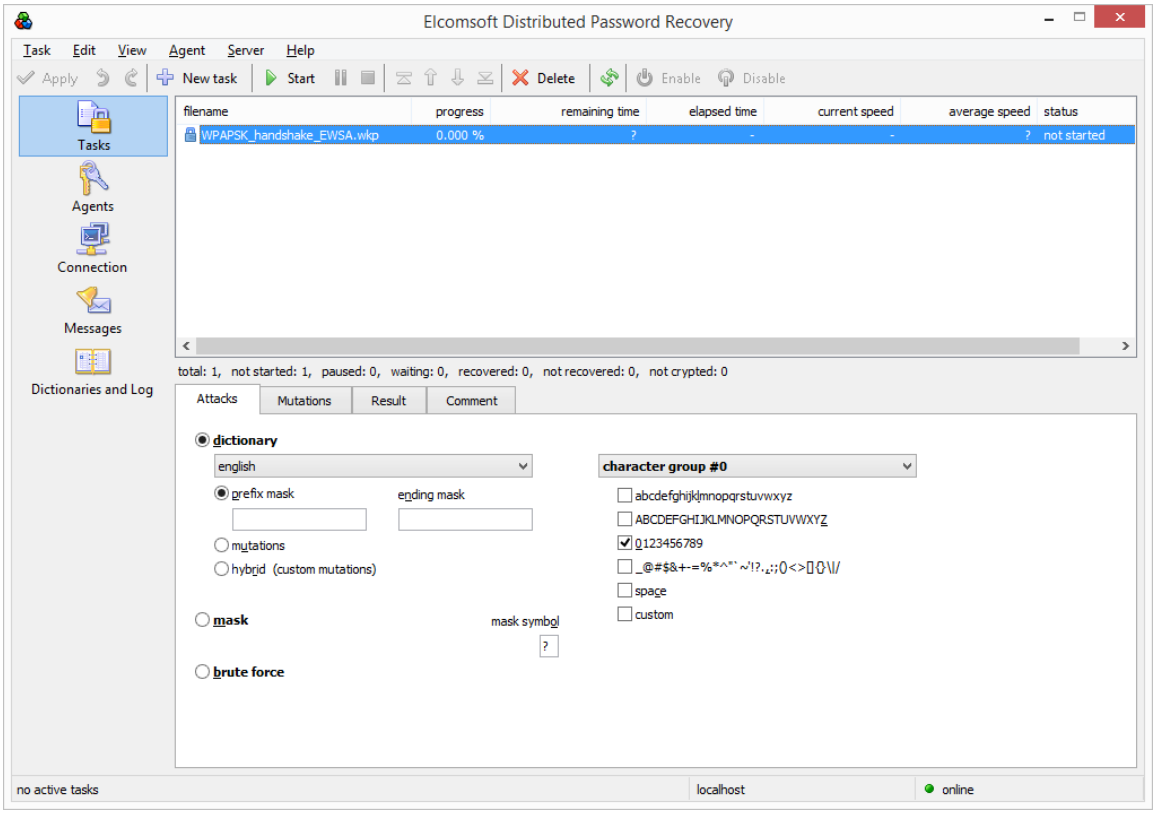
Podporuje využitie GPU pre akceleráciu výpočtu, ktoré je možné použiť ale len na operačnom systéme Windows s grafickými kartami NVIDIA. Nástroj je schopný využiť GPU len na pár vybraných formátov a to hlavne šifrované súbory od firmy Microsoft (Office, Onenote, Project, Access) a WinZip9.

[9] http://www.oxid.it/cain.html
[10] https://accessdata.com

2.3.9 Elcomsoft

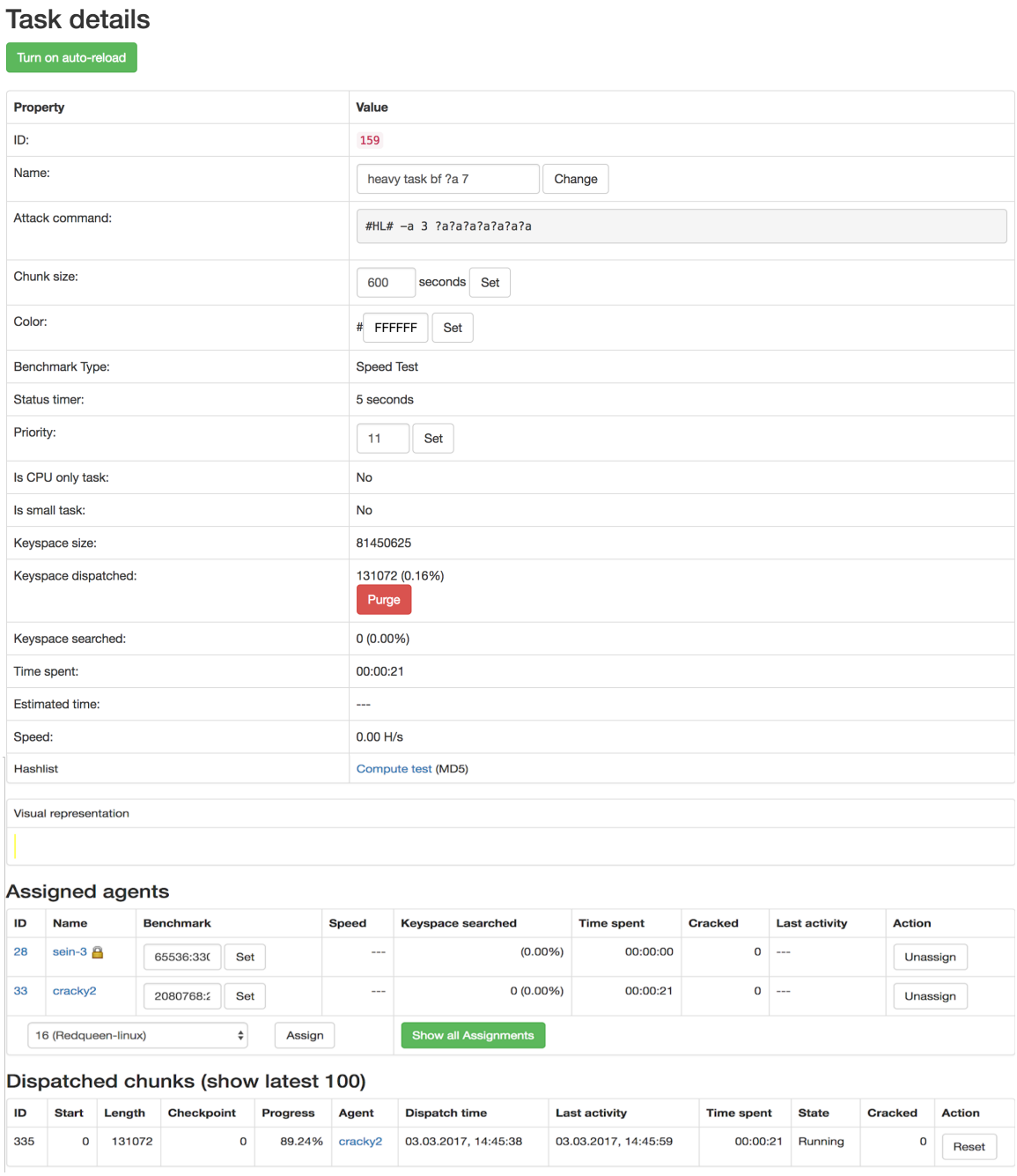
Elcomsoft[11] nie je nástroj ale firma, ktorá poskytuje sadu nástrojov. Jednotlivé nástroje dokážu dešifrovať súbory na nechránenom disku, prelomiť šifrované zálohy na mobilných zariadeniach, dešifrovať celé disky a schopné distribuovať lámanie.

Podporuje viac ako 300 formátov medzi ktoré patria Microsoft Office, OpenOffice, ZIP, 7zip, RAR, PDF. Pre akceleráciu procesu dokáže využiť grafické karty od NVIDIA, AMD ale aj Intel. Ale nie pre každý formát, napríklad pre ZIP a RAR archívy vie použit len grafické karty od NVIDIA. Udáva linearnú škálovatelnosť až pre 10 000 zariadení pri distribuovanom výpočte. Nástroj sa ovláda pomocoui grafického rozhrania viď obrázok 2.8.

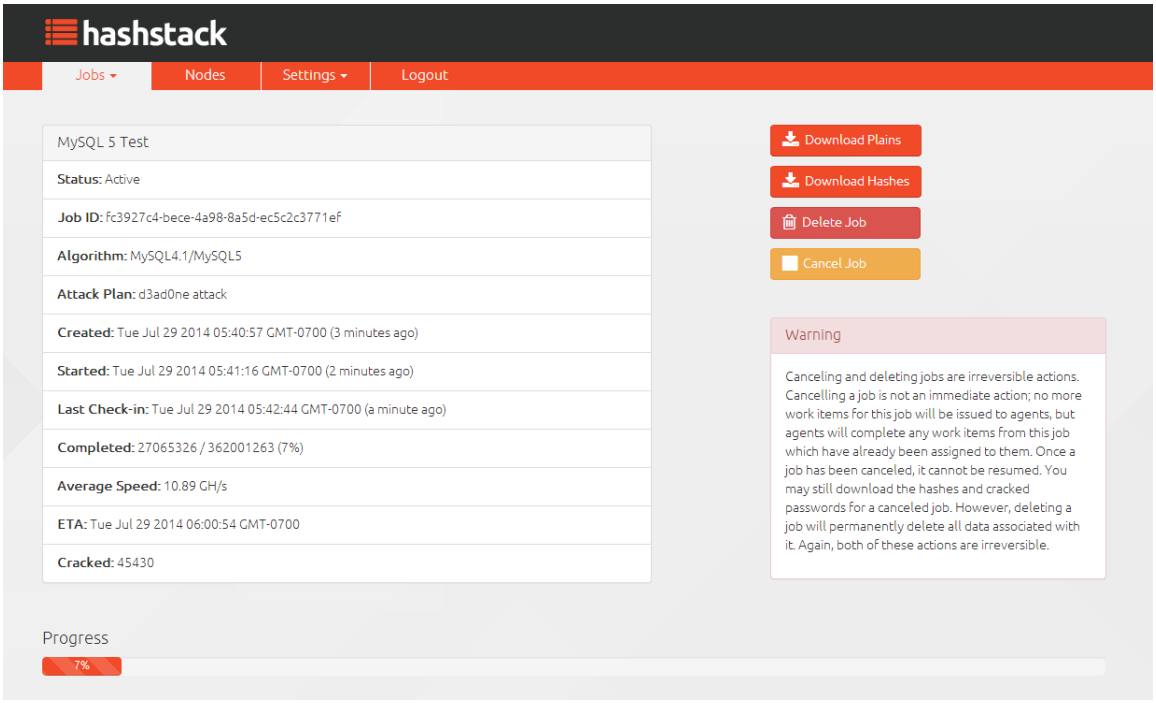


Obr. 2.8: Rozhranie nástroja ElcomSoft Distributed Password Recovery

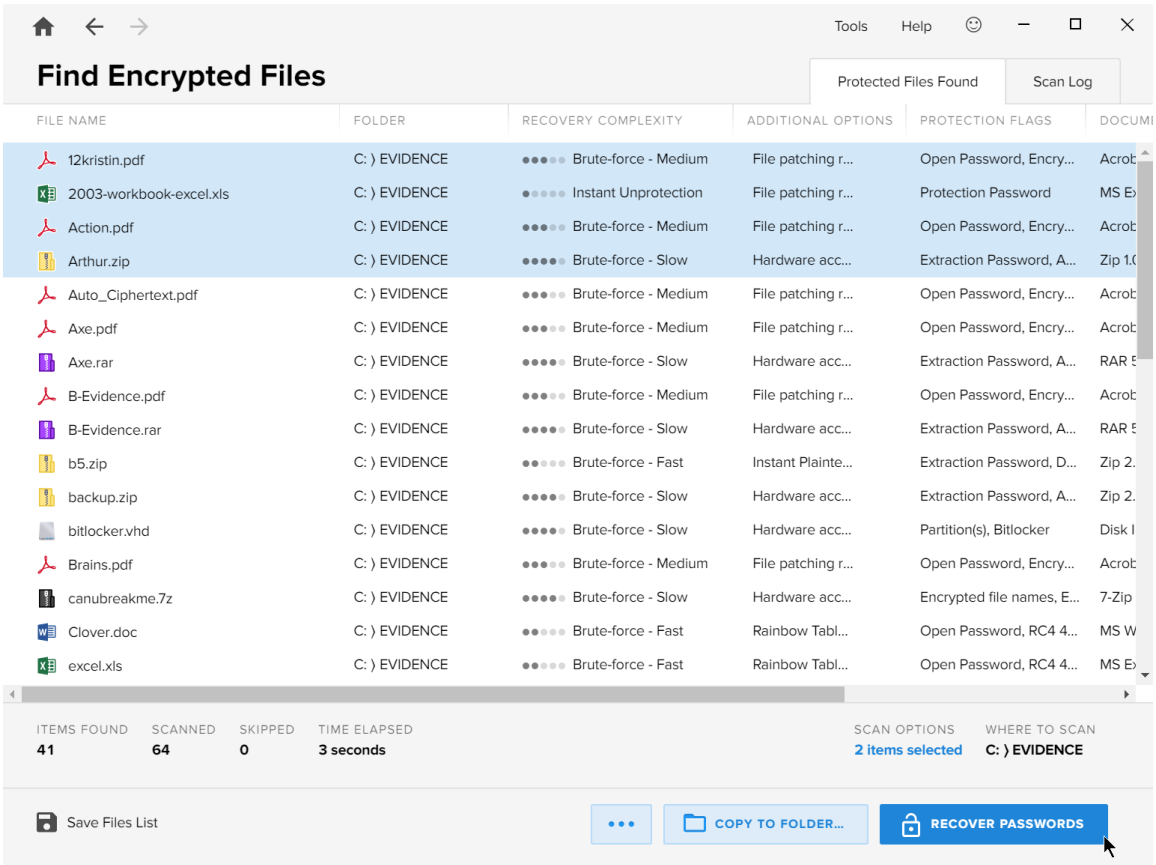
[11] https://www.elcomsoft.com



Obr. 2.9: Webového rozhranie nástroja Hashtopolis



Obr. 2.10: Webového rozhranie nástroja Hashstack



Obr. 2.11: Rozhranie nástroja Passware Kit

Kapitola 3

PCFG

Pravdepodobnostná bezkontextová gramatika *Probabilistic Context-Free Grammar* [17] je matematický model. Používa sa pri útoku, kde sa na trénovanej sade vytvorí bez-kontextová gramatika s pravdepodobnostnými prechodmi a na základe nej sa generujú hesla od najvyššej pravdepodobnosti. Základom je, že niektoré odhady hesla majú väčšiu pravdepodobnosť sa trafiť do lámaného hesla.

3.1 Trénovanie

Pred samotným útokom je nutné vytvoriť gramatiku na základe vstupnej sady. V tejto fázy sa počítajú frekvencie určitých vzorov v reťazci zo vstupu a výstup je gramatika. Výhoda tohto prístupu je, že gramatika po vygenerovaní môže byť ďalej distribuovaná bez zverejnenia pôvodného zoznamu hesiel, ktorý bol použitý na trénovanie. Definujme si zástupne znaky, ktoré budu reprezentovať množinu znakov vyskytujúcich sa za sebou:

- `L` - znaky z abecedy,
- `D` - číslice,
- `S` - špeciálne znaky.

Reťazec `heslo@123` by definovala jednoduchá štruktúra LSD. Definujme si základnú štruktúru, ktorá je definovaná podobné ako jednoduchá ale zaznamenáva aj informáciu o dĺžke podreťazca, v našom príklade by to znamenalo $L_5S_1D_3$. Príklady každého typu štruktúry viď tabuľka 3.1.

Štruktúra	**Príklad**
Jednoduchá	LSD
Základná	$L_5S_1D_3$
Preterminálna	L_5@123
Terminálna (heslo)	*heslo*@123

Tabuľka 3.1: Jednotlivé vyskytujúce sa štruktúry

Pre každý typ reťazca (znaky z abecedy, číslice, špeciálne znaky) vypočítame pravdepodobnosť výskytu na základe dĺžky reťazca a množstva výskytu daného typu podľa 3.1:

$$Prob(x) = \frac{x_n}{k_n} \tag{3.1}$$

kde x_n je počet výskytov reťazca x v trénovanej sade a k_n je počet výskytov reťazcov daného typu.

Preterminály sú reťazce, ktoré obsahujú len pseudo terminály alebo terminály. Preterminálne štruktúry sú dôležité pretože pravdepodobnosť tejto štruktúry sa už nezmení žiadnou ďalšou náhradou a tým pádom je pravdepodobnosť rovnaká ako terminálu.

Najprv sa získajú pravdepodobnosti (viď tabuľka 3.2) jednotlivých reťazcov zložených len z číslic, alebo len zo špecialných znakov, tieto pravdepodobnosti sú nezávisle voči základnej štruktúry v ktorej sa objavili. Reťazec je definovaný najdlhšou dĺžkou opakovaných znakov daného typu, to znamená že `"79"` bude zaradené do D_2 a jednotlivé číslice 7 a 9 nebudú počítane do D_1.

1 číslica	**Počet výskytov**	**%**
1	12788	50,7803
2	2789	11,0749
3	2094	8,32308
4	1708	6,78235
7	1245	4,94381
5	1039	4,1258
0	1009	4,00667
6	899	3,56987
8	898	3,5659
9	712	2,8273

2 číslice	**Počet výskytov**	**%**
12	1084	5,99425
13	771	4,26344
11	747	4,13072
69	734	4,05884
06	595	3,2902
22	567	3,13537
21	538	2,97501
23	533	2,94736
14	481	2,65981
10	467	2,58239

Tabuľka 3.2: Príklad vypočítaných pravdepodobnosti číslic [17]

Ďalším krokom je analýza základných štruktúr a ich pravdepodobnosti, ktorá sa počíta podobne ako pri jednotlivých číslic a špecialných znakov:

$$P(x) = \frac{x_n}{k_{total}}, \tag{3.2}$$

kde x_n je počet výskytov danej základnej štruktúry a k_{total} je počet analyzovaných základných štruktúr, ktoré by mali zodpovedať počtu hesiel v trénovanej sade. Ako príklad tabuľka 3.3 zobrazuje prvých 10 najpravdepodobnejších základných štruktúr natrénovaných na uniknutých heslách z MySpace. Väčšina užívateľov volilo heslo zložené zo znakov abecedy nasledované jednou, alebo dvoma číslicami. Výber hesla ovplyvňuje samozrejme rôzne faktory, ako osobné charakteristiky (vek, krajina, ...), ale aj podmienky pre vytvorenie hesla.

Rozlíšenie veľkosti písmen

Pre rozlíšenie veľkosti sa používajú ďalšie prechody v gramatike. Analyzujú sa všetky výskyty reťazcov zložené z písmen, nezávisle na základnej štruktúre a extrahuje sa maska [16]. Veľké písmena sú označené znakom `U` (*uppercase letters*) a malé znakom `N` (*non-uppercase*). Ako príklad reťazec `TajneHeslo` by bolo reprezentované maskou $U_1N_4U_1N_4$. Pravdepodobnosti sú potom pridelené každej maske na základe počtu výskytu masky s danou dĺžkou pre každý reťazec. Príklad tabuľka 3.4, kde vidno že pokiaľ nie je nutné mať veľké písmeno v hesle, tak užívatelia ho nepoužijú.

Proces trénovania končí uložením jednotlivých pravdepodobnosti pre základne štruktúry, číselne reťazce a reťazce zložené zo špeciálnych znakov.

Základná štruktúra	Počet výskytov	Celková pravdepodobnosť(%)
L_6D_1	2382	7,09752
L_6D_2	2181	6,49861
L_7D_1	2100	6,25726
L_8D_1	1746	5,20246
L_5D_2	1557	4,63931
L_9D_1	1366	4,07020
L_7D_1	1328	3,95697
L_5D_1	1283	3,82288
L_4D_2	1206	3,59345
L_8D_2	1152	3,43255

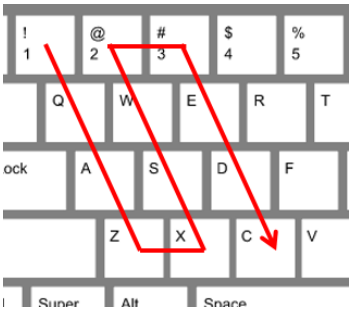
Tabuľka 3.3: 10 najpravdepodobnejších základných štruktúr netrénovaných na uniknutých heslách z MySpace [16]

Maska	Počet výskytov	Celková pravdepodobnosť(%)
N_6	7080	93,206
U_1N_5	241	3,1727
U_6	222	2,9225
N_3U_3	8	0,1053
$U_1N_4U_1$	6	0,0078

Tabuľka 3.4: 5 najpravdepodobnejších masiek pre 6 znakový reťazec [16]

Klávesové vzory

Klávesový vzor je sekvencia kláves na klávesnici. Neberie ohľad na samotné znaky, ale namiesto toho vytvára fyzický tvar (viď obrázok 3.1), ktorý sa dá ľahko zapamätať. Typickým príkladom je `qwerty`, ktorý začína písmenom `q` a nasleduje za ním 5 písmen v sekvencii doprava. Tento vzor je často kombinovaný s ďalšími komponentami, ako napríklad pridanie číslic (`qwerty2000`).



Obr. 3.1: Klávesový vzor 1qaz2wsx3edc

Bol zavedený nový preterminál `K`, ktorý označuje takýto vzor. Napríklad pre štruktúru $K_4S_1D_1$ je terminál `qw34!99`. Pri zavedení klávesového preterminálu nastáva problém s nejednoznačnosťou gramatiky. Gramatika je nejednoznačná, ak pre terminál existuje viac ako jeden derivačný strom. Nasledujúca gramatika je nejednoznačná, pretože pre terminál

`alice123` existujú 2 derivačné stromy, jeden z L_5D_3, druhý z L_5K_3.

$$\begin{aligned} S &\rightarrow L_5D_3 \mid L_5K_3 \\ L_5 &\rightarrow alice \\ D_3 &\rightarrow 123 \mid 456 \\ K_3 &\rightarrow 123 \mid asd \end{aligned}$$

Heslo	**Originálna základna štruktúra**	**Štruktúra s klávesovým vzorom**
$asdf$	L_4	K_4
$q1q1$	$LDLD$	K_4
$asd1234qw$	$L_3D_4L_2$	$K_3D_4L_2$

Tabuľka 3.5: Základne štruktúry s klávesovými preterminálmi

Nejednoznačnosť gramatiky je vyriešená počítaním nájdením základných štruktúr a vzorov. V tabuľke 3.5 sú 2 heslá, ktoré vedú k preterminálu K_4 a pre toto pravidlo je počítadlo zväčšené o 2 a nevedie ku zvýšeniu počítadla pre pravidlá $S \rightarrow L_4$ a $S \rightarrow LDLD$. Keby v trénovanej sade sa ocitlo heslo `john`, počítadlo originálnej základnej štruktúry by bolo zväčšené o 1, ale nie je k tomu žiaden odpovedajúci klávesový vzor.

Ak štruktúra obsahuje číslice alebo špeciálne znaky, klasifikujú sa ako D alebo S. Akýkoľvek pod-reťazec zložený aspoň z 3 znakov je klasifikovaný ako preterminál K ak spĺňa klávesový vzor a je maximálnej dĺžky. Napríklad terminál $qwerty7800$ je klasifikovaný ako K_8D_2 a nie ako L_6D_4 alebo K_6D_4.

3.2 Generovanie

Pre generovanie hesiel sa používa pravdepodobnostná bezkontextová gramatika, ktorá vychádza z bezkontextovej. Oproti pôvodnej štvorici [18], obsahuje naviac množinu s pravdepodobnosťami R, kde pre každé pravidlo A má definovanú pravdepodobnosť. Súčet pravdepodobnosti pre jednotlivé pravidlá musí byť 1.

Definícia 1 *Pravdepodobnostná bezkontextová gramatika je pätica $G = (N, \Sigma, P, S, R)$, kde N je konečná množina neterminálov, Σ je konečná množina terminálov, S je štartovací neterminál, R je množina pravdepodobností a P je konečná množina pravidiel v tvare:*

$$A \rightarrow \alpha, A \in N, \alpha \in (N \cup \Sigma)^*$$

Reťazec odvodený zo štartovacieho symbolu je nazývaný vetná forma, pravdepodobnosť vetnej formy je súčin jednotlivých pravdepodobností derivácii. Príklad gramatiky, ktorá vznikne z trénovanej fázy je zobrazená v tabuľke 3.6. ĽS znamená ľavá strana pravidla, v ktorom vždy bude jeden neterminál. PS je pravá strana pravidlá, ktorá už mimo neterminálov môže obsahovať aj terminály .Jedna z množných preterminalných štruktúr ktorá vznikne po derivovaní:

$$S \rightarrow L_3D_1S_1 \rightarrow L_34S_1 \rightarrow L_34!$$

ktorého pravdepodobnosť je 0.0975.

Tento prístup je vhodný pre distribuované prostredie, server môže vypočítať preterminál podľa vzostupnej pravdepodobnosti a predať ho ďalej klientskému uzlu, ktorý dosadí slová zo slovníka a následne overí či vygenerovaný terminál je hľadané heslo.

ĽS	PS	Pravdepodobnosť	ĽS	PS	Pravdepodobnosť
S	$D_1L_3S_2D_1$	0,75	S	$L_3D_1S_1$	0,25
D_1	4	0,60	D_1	5	0,20
D_1	6	0,20	S_1	!	0,65
S_1	%	0,30	S_1	#	0,05
S_2	$$	0,70	S_2	**	0,30

Tabuľka 3.6: Príklad pravdepodobnostnej bezkontextovej gramatiky [17]

3.2.1 Efektívna funkcia *Next*

Funkcia *Next* berie vygenerovanú gramatiku ako vstup a generuje hesla na výstup vzostupnom poradí na základne pravdepodobnosti. Vygenerovať najpravdepodobnejšie heslo je jednoduché, stačí nahradiť základne štruktúry s najpravdepodobnejšími terminálmi a vybrať preterminál s najväčšou pravdepodobnosťou, problém nastáva, keď chceme ďalšie v poradí a aby to bolo efektívne.

Jeden zo spôsobov je získať všetky preterminály a vypočítať ich pravdepodobnosť a následne zoradiť. Tento spôsob ale nie je vhodný do distribuovaného prostredia, pretože je nutné vygenerovať veľké množstvo dat a zoradiť, čo zaberie značný čas ešte predtým, ako je schopné vygenerovať prvé heslo.

Ďalší spôsob je vygenerovať preterminály len s pravdepodobnosťou nad stanovený limit [10]. Problém je ale, že tento algoritmus nevygeneruje hesla vzostupne podľa pravdepodobnosti, ale sú v náhodnom poradí, jedine čo zaručuje je že pravdepodobnosť terminálov spadá nad stanovený limit.

Najefektívnejší spôsob používa prioritnú frontu, kde prvý záznam vo fronte obsahuje najpravdepodobnejší terminál, jeho pravdepodobnosť, základnú štruktúru a pivot [16] . Hodnota pivotu je skontrolovaná, keď preterminál je vybraný z fronty, pivot určuje ktorý preterminál môže byť vložený do fronty ako ďalší. Význam pivotu je zabezpečiť, aby všetky preterminály vychádzajúce z rovnakej základnej štruktúry sú vložené do fronty bez duplicít. Vo výsledku je vygenerovaný len jeden derivačný strom zo všetkých možných.

Jednotlivé premenné v základnej štruktúre označíme indexami zľava. Pri $L_3D_1S_1$, premenná L_3 bude mať index 0, D_1 - 1 a S_1 - 2. Všetky hodnoty terminálu budú zoradené podľa pravdepodobnosti daného typu, tým sa zaručí rýchle nájdenie ďalšej najpravdepodobnejšej hodnoty terminálu. Pivot značí, že po vybraní záznamu z fronty sa nahradí premenná s indexom rovnému alebo väčšiemu ako je hodnota pivotu.

Pri prvotnom stave fronty viď tabuľka 3.7 sa vyberie preterminál $4\{jan, pes\}\$\$$ a vygeneruje sa 2 nové s pivotnými hodnotami 0 a 2 viď tabuľka 3.8. Proces lámania hesla pokračuje dokým sa fronta nevyčerpá alebo nie je heslo prelomené.

Základna štruktúra	Preterminál	Pravdepodobnosť)	Pivot
$D_1L_3S_2$	$4\{jan, pes\}\$\$$	0,1575	0
$L_3D_1S_1$	$\{jan, pes\}4!$	0,04875	0

Tabuľka 3.7: Príklad prioritnej fronty

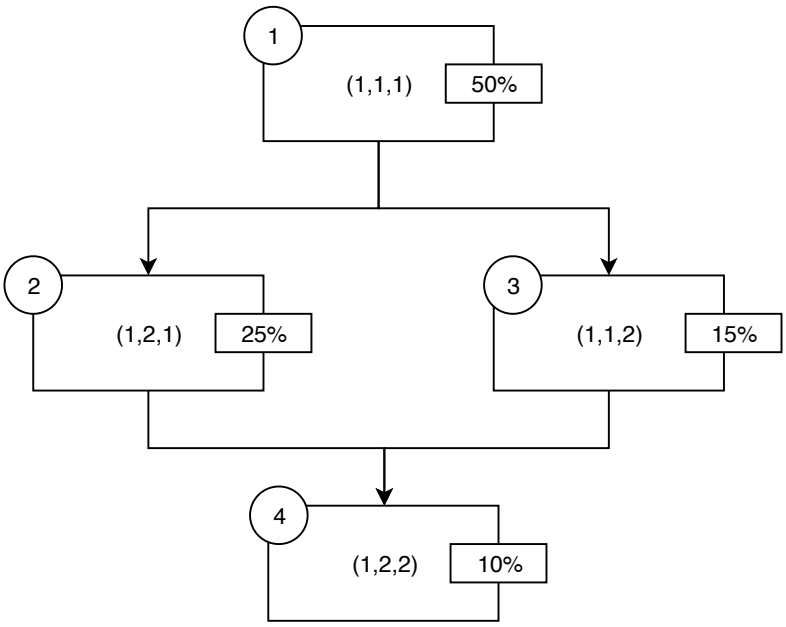
Základna štruktúra	Preterminál	Pravdepodobnosť)	Pivot
$D_1L_3S_2$	$4\{jan, pes\} * *$	0,1575	2
$D_1L_3S_2$	$\{5, 6\}\{jan, pes\}\$\$$	0,1575	0
$L_3D_1S_1$	$\{jan, pes\}4!$	0,04875	0

Tabuľka 3.8: Príklad prioritnej fronty po vybraní prvého prvku z 3.7

3.2.2 Algoritmus Deadbeat dad

Pri generovaní nových záznamov sa používal pivot, ktorý zaručuje, aby pre každé dieťa v derivačnom strome existoval presne jeden rodič, ktorý ho vygeneruje. Prioritná fronta zaručuje, aby boli reťazce vygenerované v poradí podľa pravdepodobnosti. Spolieha sa teda na ukladanie záznamov do prioritnej fronty, kvôli tomuto využíva veľké množstvo pamäti. Pre zníženie pamäťovej náročnosti bol vyvinutý algoritmus *Deadbeat dad* [16].

Kľúčom pre zníženie pamäťovej náročnosti, je čo najviac oddialiť vloženie záznamov do fronty. Predošlým algoritmom *Next*, akonáhle bol rodičovský uzol vybraný z fronty, tak všetky jeho podriadené uzly boli vložené do fronty na základe pivotu. Cieľ algoritmu *Deadbeat dad* je niekedy zahodiť podriadené uzly, je to dôsledok nepoužitia pivotu a tým pádom môžu vzniknúť uzly, ktoré vygenerujú rovnaký terminál, viď obrázok 3.2.



Obr. 3.2: Viacero rodičov pre uzol 4

Pôvodná funkcia *next* vyberie uzol 1 a vloží uzly 2 a 3 do fronty. Následne je vybraný uzol 2, pretože má v daný moment najväčšiu pravdepodobnosť a vloží uzol 4. Pravdepodobnosť podriadeného uzlu bude vždy menšia oproti rodičovskému uzlu, čiže uzol 4 nikdy nebude vybraný pred uzlom 3, preto by ideálne mal vložiť uzol 4 až uzol 3 a nie uzol 2.

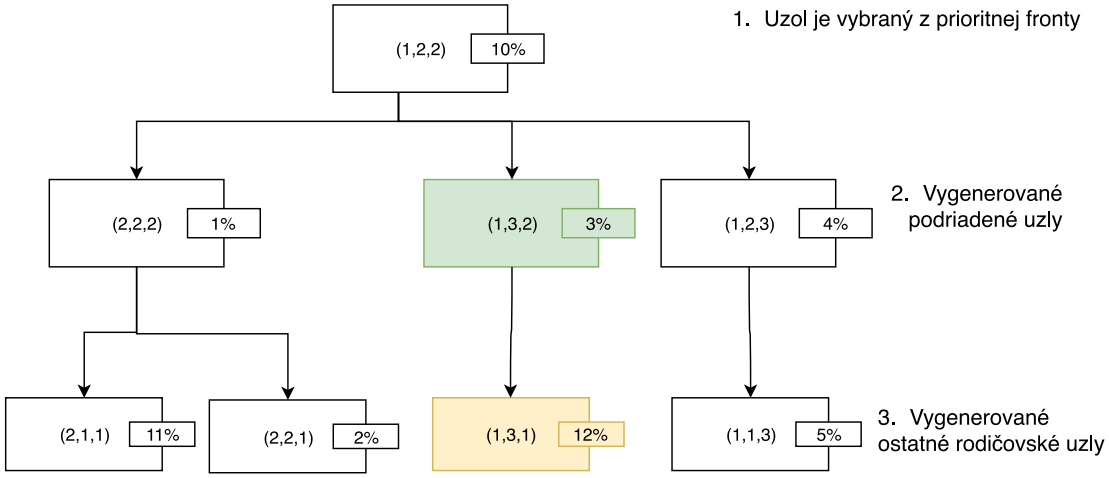
Podriadené uzly nikdy nebudú vybrané pred rodičovskými uzlami a tie sú vyberané podľa pravdepodobnosti, ideálne by podriadený uzol mal vkladať rodičovský uzol s najmenšou pravdepodobnosťou.

Tento algoritmus používa tiež prioritnú frontu ale pri vybraní uzlu su vygenerované všetky podriadené uzly. Pre každý tento uzol sa vygenerujú všetky možné rodičovské uzly

a kontroluje pravdepodobnosť oproti aktuálnemu rodičovskému uzlu. Ak aspoň jeden takto vygenerovaný rodičovský uzol ma menšiu pravdepodobnosť ako aktuálny, tak aktuálny rodičovský uzol sa tohto podriadeného uzlu vzdá a zahodí, pretože ho neskôr vygeneruje iný uzol.

Na príklade zobrazený na obrázku 3.3 je vybraný uzol, vygenerujú sa podriadené uzly, pre každý tento uzol sú vygenerované ostatné rodičovské uzly. Pôvodne vybraný rodičovský uzol má pravdepodobnosť 10 %+. Prvý podriadený uzol má 2 ďalšie rodičovské uzly, jeden z nich má pravdepodobnosť 2 %, čiže tento rodič sa o tento podriadený uzol neskôr postará a tým pádom tento uzol vybraný rodičovský uzol nevloží do fronty. To isté sa stane aj pri 3. podriadenom uzle. Druhý podriadený uzol má už len jedného ďalšieho rodiča, ktorý má ale väčšiu pravdepodobnosť (12 %) ako vybraný. Tým pádom je podriadený uzol $(1, 3, 2)$ vložený do prioritnej fronty.

Týmto algoritmom je značne zmenšená pamäťová náročnosť na úkor výpočtu generovania rodičovských uzlov. Ale pri zmenšenej veľkosti prioritnej fronty, klesá aj čas, ktorý je potrebný pre vloženie záznamu do prioritnej fronty.



Obr. 3.3: Správanie algoritmu deadbeat dad pri vybraní uzlu

Kapitola 4

Návrh distribuovaného generátoru

Aby generovanie hesiel pomocou pravdepodobnostnej bezkontextovej gramatiky bolo použiteľné v praxi musí podporovať distribúciu lámania hesiel na viacero výpočetných uzlov, to znamená aj samotné generovanie hesiel. Distribúcia je dôležitý faktor, aby sa dal proces lámania hesla čo najviac zrýchliť a paralelizovať využitím všetkých dostupných prostriedkov.

Pri návrhu treba brať v úvahe, aby heslá boli vygenerované v poradí, ktoré sa čo najviac blíži zoradeniu podľa pravdepodobnosti. Ak by sa hesla generovali mimo poradia, tak by tento typ útoku nebol efektívny a nemalo by zmysel ho distribuovať. Distribuovaný generátor musí rozposielať jednotlivé kúsky na generovania čo najrovnomernejšie alebo na základe výkonu jednotlivého uzlu. Každý uzol by mal byť schopný vygenerovať rovnaké množstvo hesiel. Úloha by nemala byť priveľká, aby v prípade zlyhania uzlu nedošlo k veľkej strate výpočtového času. Ale ani primalá, aby čas strávený pri komunikácii medzi serverom a klientom bol čo najmenší. Pri tomto útoku nastáva problém, nie je možné jednoducho rozdeliť úlohu na menšie fixné časti, ako je to napríklad pri slovníkovom útoku.

Existuje viacero možnosti ako toho docieliť, ale každý z uvedených možností vyžaduje, aby každý uzol mal dostupnú celú gramatiku na základe ktorej bude generovať svoj kúsok hesiel. Samozrejme môže sa o všetko starať len jeden uzol, ktorý bude generovať heslá a následne rozposielať ďalším ako v prípade slovníkového útoku, ale tento spôsob je síce jednoduchý ale značne neefektívny [7]. Po sieti sa bude prenášať veľké množstvo dát (hesiel) a ďalšie uzly budú musieť čakať na hlavný uzol, ak rýchlosť linky nebude stíhať. To môže spôsobiť mrhanie výkonu uzlov, ak uzly budu lámať heslá rýchlejšie ako ich hlavný uzol bude schopný rozposielať.

4.1 Generovanie preterminálnych štruktúr na serveri

Jedna z možností ako distribuovať, je mať jeden hlavný server a ďalšie klientské uzly. Server sa bude starať o:

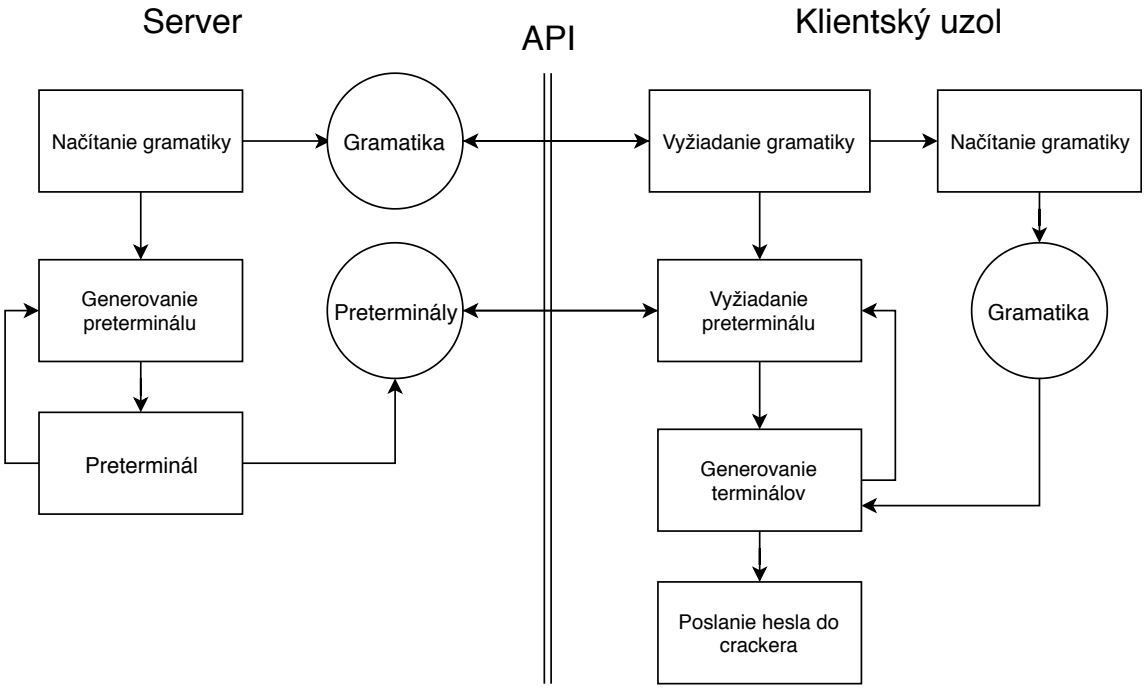
- načítanie gramatiky,
- správu prioritnej fronty,
- rozposielanie preterminálnych štruktúr uzlom.

Server je jediný, ktorý sa bude starať o deriváciu preterminálnych štruktúr a rozposielať ďalším uzlom. Každá preterminálná štruktúra môže reprezentovať tisícky [16] hesiel. Týmto

spôsobom by sa oproti naivnému riešeniu ušetrila veľká časť sieťovej prevádzky. Využil by sa výkon uzlu a nemusel by čakať na jednotlivé heslá od serveru, ale len na preterminálne štruktúry. Týmto prístupom budú heslá stále vygenerované vzostupne podľa ich pravdepodobnosti, keďže heslá ktoré vzniknú z jednej preterminálnej štruktúry máju vždy rovnakú pravdepodobnosť. Toto platí ale len pre daný klientský uzol.

Nevýhoda tohto prístupu je stále závislosť na hlavnom uzle, ktorý musí spravovať prioritnú frontu a starať sa o generovanie preterminálnych štruktúr. Pri veľkom množstve uzlov môže stále nastať problém, kedy bude klientský uzol dlhšie čakať na preterminálnu štruktúru ako samotné lámanie hesla. Ale zároveň je to výhoda pri implementácii, kde existuje len jedna prioritná fronta na hlavnom uzle ktorú treba implementovať.

Klientský uzol po pripojení k výpočtu musí získať najprv gramatiku od servera. Následne si vypýta úlohu čo predstavuje preterminálnu štruktúru. Server generuje preterminálne štruktúry a ukladá si ich do fronty. Server pri požiadavke od klienta vyjme preterminálnu štruktúru z fronty a odošle ho klientovi, toto sa opakuje do doby, pokým ma server dostupné preterminálne štruktúry alebo heslo nebolo nájdene na žiadnom z klientských uzlov. Proces je znázornený na obrázku 4.1.



Obr. 4.1: Návrh generovania preterminálnych štruktúr na serveri

Server poskytuje klientovi jednoduché API[1]:

- `GetGrammar()` - vráti načítanú gramatiku zo súboru,
- `GetNextPreterminal()` - vráti klientovi preterminálnu štruktúru, ktorá je prvá vo fronte,
- `SendPassword(result)` - klient odošle správne nájdene heslo.

[1] Application programming interface

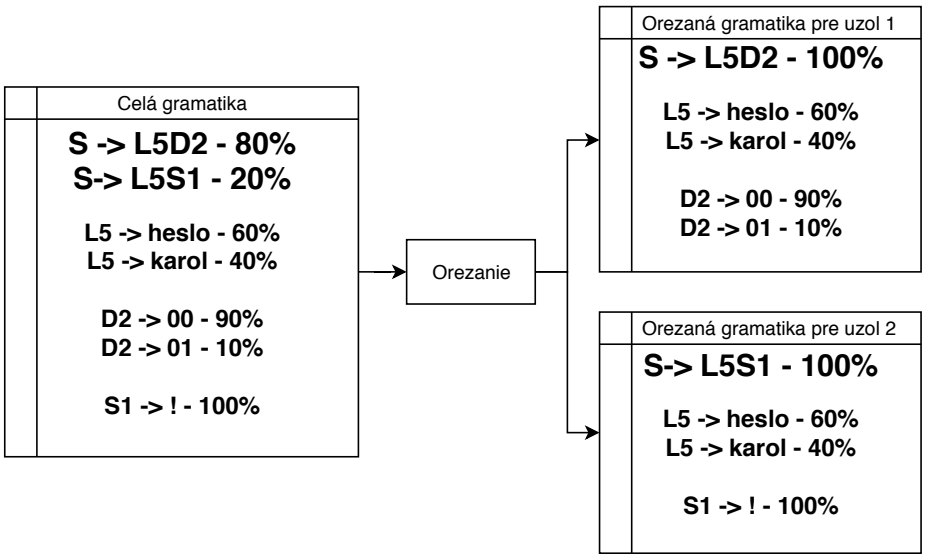
4.2 Generovanie preterminálnych štruktúr na klientovi

Ďalšou možnosťou je generovať už aj preterminálne štruktúry na klientských uzloch. Server pošle časť gramatiky klientskému uzlu, ktorý sa postará o zvyšok. Klientský uzol bude nezávislý čo znamená, že každý bude mať:

- vlastnú časť gramatiky,
- prioritnú frontu o ktorú sa bude starať,
- generátor preterminálnych štruktúr,
- generátor terminálov.

Pri tomto prístupe sa využije celý výkon klientského uzlu, keďže si sám bude spravovať všetky spomenuté veci. Server viď obrázok 4.3 sa stará len o orezávanie gramatiky na časti a ich rozposielanie. Problém nastáva pri generovaní, kde sa už nedá zaručiť generovanie terminálov v poradí podľa pravdepodobnosti. Poradie bude zachované len v rámci danej orezanej gramatike.

Ďalej je treba navrhnúť ako orezať gramatiku. Jedna varianta je posielať postupne zoradené podľa pravdepodobnosti pravidlá, ktoré vychádzajú zo štartovacieho symbolu. Týmto prístupom uzol vždy vygeneruje heslo podľa určitej masky.



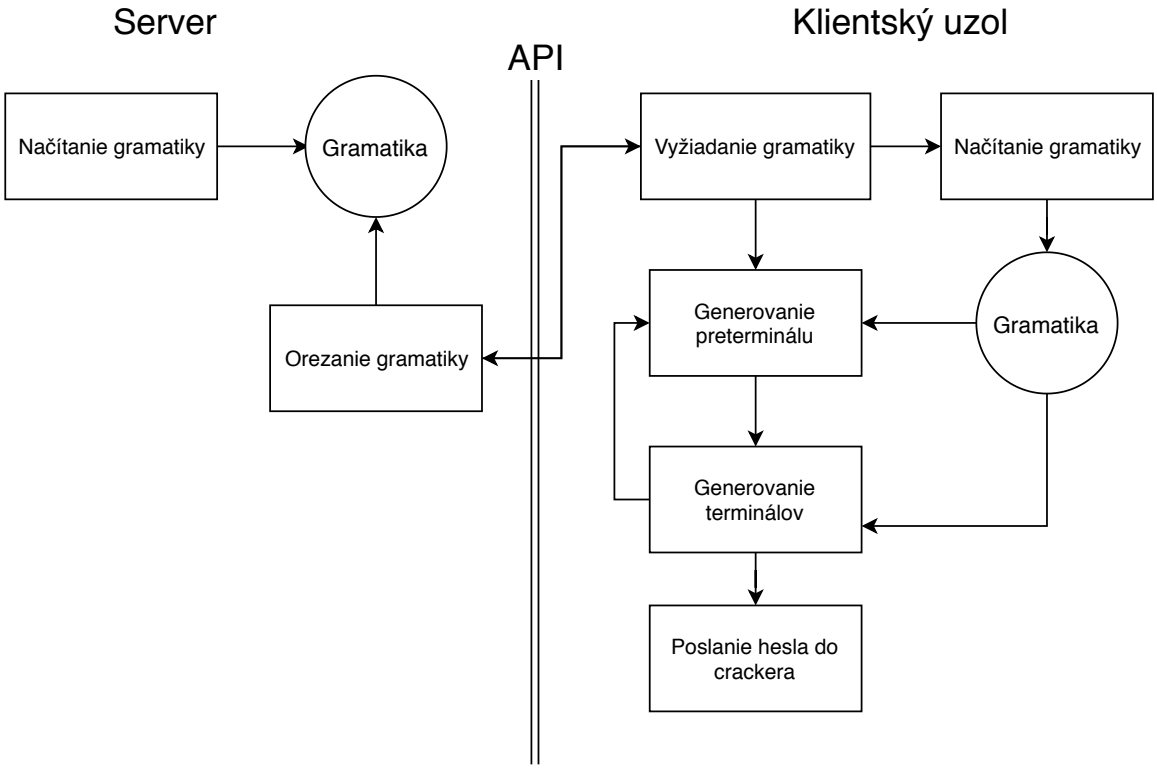
Obr. 4.2: Návrh orezania gramatiky

Pri použití tohto typu orezávanie by stačilo poslať na začiatku celú gramatiku, okrem pravidiel vychádzajúcich zo štartovacieho symbolu. A následne posielať len základne štruktúry (pravidla vychádzajúce zo štartovacieho symbolu). Týmto prístupom by sa zbytočne neposielali neterminály, ktoré už klient mohol mať dostupné.

Na príklade časti gramatiky viď rovnica 4.1, je 5 pravidiel vychádzajúcich zo štartovacieho symbolu zoradených podľa ich pravdepodobnosti. Pri poslaní prvého pravidlá uzol vygeneruje všetky terminály o dĺžke 6 v ktorých sa vyskytujú len písmena. Druhé pravidlo spraví to isté ale o dĺžke 7.

$$
\begin{aligned}
S &\to A6 \quad && p = 0,1661498 \\
S &\to A7 \quad && p = 0,1045314 \\
S &\to A8 \quad && p = 0,08162449 \\
S &\to A6D1 \quad && p = 0,08132440 \\
S &\to A7D1 \quad && p = 0,06401921
\end{aligned} \tag{4.1}
$$

Nastane problém, že bude generovať aj tie najmenej pravdepodobné heslá ku ktorým by sa použitím metódy popísanej v 4.1 nedostalo, pretože by vygenerovalo napríklad najprv heslá z $S \to A6D1$, ktoré by mali väčšiu pravdepodobnosť ako najmenej pravdepodobný terminál z neterminálu $A6$. Tento problém sa dá z časti riešiť stanoveným prahu pravdepodobnosti pre jednotlivé neterminály.



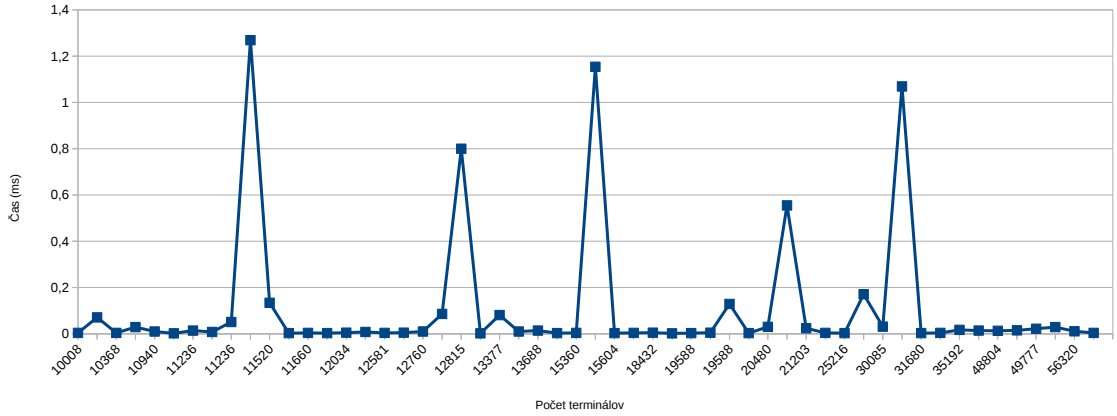
Obr. 4.3: Návrh generovania preterminálnych štruktúr na klientskom uzle

4.3 Výber metódy generovania

Metóda popísaná v sekcii 4.1 má lepšie vlastnosti pre generovanie, oproti metóde popísanej v 4.2, pretože zachováva poradie generovania na základe pravdepodobnosti. Ak generovanie preterminálnych štruktúr je znateľne rýchlejšie ako generovanie terminálov, tak táto metóda bude efektívnejšia.

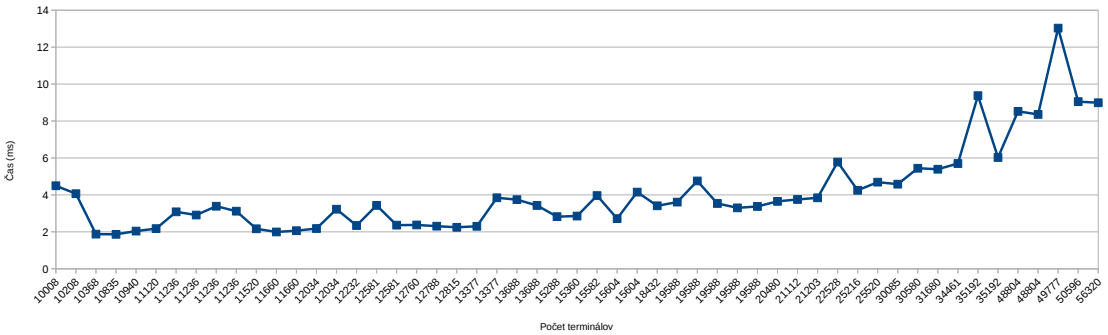
Pre tento účel boli vykonané experimenty, v ktorých sa sledovala rýchlosť generovania preterminálnych štruktúr a následne rýchlosť vygenerovania terminálov z nich. Graf 4.4 zobrazuje, ako dlho trvá vygenerovať novú preterminálnu štruktúru v závislosti na počtu terminálov, ktoré sa z nej vygenerujú. Na jednej konkrétnej gramatike sa spustilo gene-

rovanie a sledovalo sa, ako dlho trvá funkcia *Next* a boli zaznamenané len preterminálne štruktúry, ktoré vygenerovali viac ako 10 000 terminálov. Generovanie trvá skoro vždy pod 0,1ms. Výkyvy na grafe sú spôsobené modifikáciou prioritnej fronty, ktorá je implementovaná pomocou haldy a haldu je nutné opätovne preusporiadať. Na grafe vidno, že počet vygenerovaných preterminálnych štruktúr neovplyvňuje počet terminálov, ktoré sú z nich vygenerované.



Obr. 4.4: Rýchlosť generovania preterminálnych štruktúr

Graf 4.5 zobrazuje ako dlho trvá vygenerovať terminály z preterminálnej štruktúry. Vychádza z toho istého experimentu, ktorý bol použitý na graf 4.5. Na grafe vidno, že čas potrebný na vygenerovanie rastie s počtom terminálov, ktoré sú vygenerované. Generovanie 10 000 terminálov zaberie cca 5 ms a 50 000 terminálov cca 10 ms.



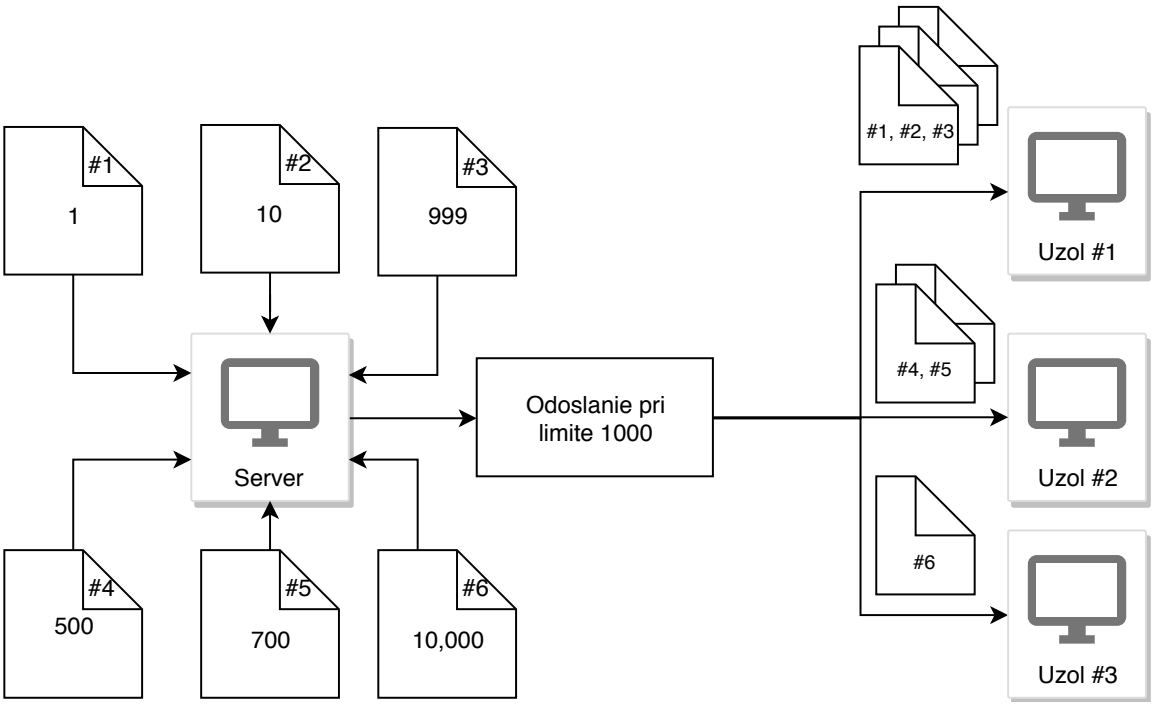
Obr. 4.5: Rýchlosť generovania terminálov

Vo výsledku generovanie preterminálnych štruktúr oproti terminálov je 20 až 2000 krát rýchlejšie, čo je dostatočné pre naše účely, klientské uzly mimo generovania strávia čas samotným lámaním hesla, čo zvyčajne zaberie viac času ako samotné generovanie. Toto ale platí len preterminálne štruktúry, ktoré vygenerujú tisícky terminálov. Môže sa stať, že preterminálna štruktúra vygeneruje len jeden terminál a len samotná sieťová komunikácia by zbytočne brzdila proces lámania, ak by sa posielali heslá po jednom.

Riešením tohto problému je posielať viac, ako jednu preterminálnu štruktúru. Server spočíta koľko terminálov preterminálna štruktúra vygeneruje. Ak je to malé množstvo, tak spočíta pre ďalšiu preterminálnu štruktúru a to bude opakovať až dokým počet spoločne

vygenerovaných terminálov neprekročí stanovený limit. Následne odošle zhluk preterminálnych štruktúr uzlu, ktorý z nich postupne vygeneruje terminály.

Na diagrame 4.6 je znázornene, ako by prebiehalo posielanie preterminálnych štruktúr na uzly pri stanovenom limite 1000. Server ma vygenerovaných 6 preterminálnych štruktúr v počte terminálov nasledujúci podľa poradia: 1, 10, 999, 500, 700, 10000. Prvému uzlu pošle prvé 3 preterminálne štruktúry, pretože ich súčet presahuje limit 1000. Druhému uzlu podobne, samotnú 4. preterminálnu štruktúru nepošle, pretože nepresahuje limit a až následne s piatym presiahne, čiže sú mu odoslané preterminálne štruktúry s označením #4 a #5. Šiesta preterminálna štruktúra bude odoslaná už osobitne, pretože sama o sebe presahuje limit.



Obr. 4.6: Odosielanie zhlukov preterminálných štruktúr

Na základe experimentu, ktorého výsledky zobrazujú grafy 4.4 a 4.5, som sa rozhodol použiť metódu generovania preterminálnych štruktúr na serveri popísanú v sekcii 4.1. Pretože vlastnosti tejto metódy sú lepšie, zachováva poradie generovania podľa pravdepodobnosti, čo u druhej metódy obecne neplatí. Experiment vyvrátil domnienku, že by klienti museli čakať na vygenerovanie preterminálnej štruktúry. Generovanie preterminálnych štruktúr je radovo v desiatkach až tisíckach rýchlejšie ako následne generovanie terminálu.

4.4 Rozšírenie pomocou nástroja Hashcat

Okrem generovania je nutné vygenerované heslá aj skúsiť na prelomenie hešu. Keďže je Hashcat považovaný za elitu v procese obnovy hesla, bol vybraný práve tento nástroj. Umožňuje predávať heslá cez štandardný vstup. Namiesto toho, aby sa rozšíril priamo Hashcat, čo by bolo implementačné náročne, tak sa použije práve funkcionalita predávania hesiel cez vstup. Generátor bude generovať heslá a posúvať ich ďalej cez rúru (*pipe*) nástroju, ktorý si ich bude postupne brať a skúšať vygenerovanými heslami prelomiť zadané heše.

Aktuálny generátor implementovaný Weirrom v jazyku Python ale nevyužíva efektívne prostriedky zariadenia. Python je interpretovaný jazyk, ktorý väčšinou býva pomalší ako kompilované jazyky. Aby proces lámania hesla nespomaľovalo samotné generovanie hesiel, tak je vhodné prepísať existujúce riešenie. Vybraný bol jazyk Go, ktorý je statický kompilovaný jazyk od firmy Google. Medzi najväčšie výhody jazyka je jeho jednoduchosť, rýchlosť a podpora paralelizácie. Na stránke[2] je porovnanie rýchlosti rozličných algoritmov, kde Go nad Pythonom vyhráva.

Okrem reimplementácie je možné paralelizovať samotné generovanie. Podobne ako navrhnutý návrh 4.1. Každú preterminálnu štruktúru je možné generovať ďalej samostatne. Jeden hlavný proces bude generovať preterminálne štruktúry a zvyšné z nich vygenerujú terminály. Týmto by sa mohlo docieliť ďalšieho zrýchlenia. Celý proces generovania hesiel bude prebiehať na CPU a samotné lámanie hesiel na GPU prostredníctvom Hashcatu. Týmto sa efektívne využije výkon ako CPU tak aj GPU zariadenia.

Proces trénovania z existujúcich hesiel a tvorenie gramatiky bude naďalej vykonávať program od Weirra, ktorý je dostatočne rýchly a hlavne je to len jednorázová záležitosť. Problém je, že takéto gramatiky generujú neskutočne veľké množstvo hesiel. Ako ďalšie vhodné rozšírenie generátora je obmedziť počet vygenerovaných hesiel. Reimplementovaný generátor bude mať možnosť stanoviť maximálny počet vygenerovaných hesiel, aby generátor v rozumnom čase mohol skončiť.

Pri zrýchlení generovania, sa nemusia heslá posielať ďalej do nástroja Hashcat, ale môže to slúžiť aj pre rýchle generovanie slovníkov. Takéto slovníky môžu slúžiť napríklad pre *honeywords*, ktoré sa môžu použit pre zvýšenie bezpečnosti [9]. Jedná sa o falošné heslá, ktoré útočník nemôže rozlíšiť. Každý užívateľ bude mať jedno legitímne heslo a niekoľko falošných (*honeywords*). Pri použití falošného hesle, systém zašle alarm.

[2] https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/go-python3.html

Kapitola 5

Reimplementácia generátoru

Pôvodná implementácia generovania napísaná Weirrom nepodporuje distribúciu hesiel. Okrem toho je napísaná v jazyku Python a rýchlosť generovania nie je dostatočné rýchla pre online lámanie hesiel. Jadro programu bolo prepísané do jazyku Go[1], ktorý je oproti Pythonu statický typovaný jazyk. Prvým krokom bolo prepísanie pôvodnej implementácie, aby funkčnosť bola zachovaná a produkovala rovnaké výsledky. Najľahším spôsobom je snaha prepisu 1:1, čo nebolo vždy možné kvôli odlišnostiam jazykov.

5.1 Gramatika

Textovú reprezentáciu gramatiky, ktorá je výstup programu `pcfg-trainer` treba načítať a uložiť do vnútornej reprezentácie definovaná ako štruktúra `Grammar`, ktorá má nasledujúcu formu:

```
type GrammarMapping map[string]map[string]int32
type Grammar struct {
    Sections []*Section
    Mapping GrammarMapping
}
type Section struct {
    Type string
    Name string
    Replacements []*Replacement
}
type Replacement struct {
    Probability float64
    IsTerminal bool
    Values []string
    Function string
    Pos []int32
}
```

Gramatika obsahuje sekcie a mapovanie, ktoré slúži pre ľahší pohyb v jednotlivých sekciách. Každá sekcia má náhrady (*replacements*), meno a svoj typ:

- START - obsahuje základne štruktúry gramatiky,

[1] https://golang.org/

- BASE_A - znaky,
- BASE_D - číslice,
- BASE_O - špeciálne znaky,
- CAPITALIZATION - určuje veľkosť písmen, ktorá je aplikovaná na sekciu `A`

Meno označuje dlžku reťazca. Príklad pre sekciu `BASE_A` s menom `2`:

```
{Probability:0.2307692 Values:[ne]}
{Probability:0.1538462 Values:[go ou]}
{Probability:0.0769231 Values:[rd me ub ib aa if]}
```

Najväčšiu pravdepodobnosť pre 2 znakový reťazec je `ne`, na 2. mieste sú 2 hodnoty s rovnakou pravdepodobnosťou `go, out`. Typ `START` obsahuje prvotné prechody zo štartovacieho symbolu.

```
{Probability:0.7 Values:[A2D1]
{Probability:0.3 Values:[A2]
```

Tu môžeme vidieť pravdepodobnosť 70 % základnej štruktúry `A2D1`, ktorá vygeneruje 2 znaky a následne jednu číslicu.

5.2 Generátor hesiel (*GuessGeneration*)

Generátor hesiel sa stará o generovanie terminálov z preterminálnej štruktúry a gramatiky. Pri vytvorení si z gramatiky vytiahne všetky potrebné náhrady a vytvorí z nich pole `GuessIndex` viz výpis 5.1. GuessIndex generuje výsledne terminály na základe typu funkcie:

- Copy, Shadow - len kopirujú terminály na výstup,
- Capitalization - mení veľkosť písmen.

```
type GuessIndex struct {
    replacement *Replacement
    function string
    topIndex int
    guessPointer int
    Reset func([]string, bool) ([]string, bool)
    Next func([]string, bool) ([]string, bool)
}

type GuessGeneration struct {
    grammar *Grammar
    guess []string
    structures []*GuessIndex
}
```

Výpis 5.1: Štruktúra typov GuessIndex a GuessGeneration

Na základe typu funkcie sa nastavia funkcie `Reset` a `Next`. Reset slúži pre vytvorenie prvotného kúsku terminálu a Next pre ďalšie. `topIndex`, udáva ktorú aktuálne hodnotu z náhrady použije ako ďalšiu pri zavolaní funkcii Next. `guessPointer` si uchováva na ktorý index pola `guesses` sa uloží ďalší kúsok hesla. Rozdiel medzi týmito dvoma premennými môžme vidieť pri funkcii na kapitalizáciu.

```
func (g *GuessIndex) nextCap(guess []string, new bool) ([]string, bool) {
    g.topIndex++
    if g.topIndex >= len(g.replacement.Values) {
        return []string{}, false
    }
    rule := g.replacement.Values[g.topIndex]
    var tmpString strings.Builder
    baseWord := guess[g.guessPointer]
    tmpString.Grow(len(baseWord))
    lPos := 0
    for _, ch := range baseWord {
        if rule[lPos] == 'U' {
            tmpString.WriteRune(unicode.ToUpper(ch))
        } else {
            tmpString.WriteRune(unicode.ToLower(ch))
        }
        lPos++
    }
    guess[g.guessPointer] = tmpString.String()
    return guess, true
}
```

Na začiatku sa zväčší `topIndex` a skontroluje či môže ešte niečo vygenerovať, inak vracia `false`. Pomocou `topIndex` si vytiahne pravidlo v tvare ULL. Toto pravidlo aplikuje na aktuálny kúsok hesla, na ktorý ukazuje `guessPointer` a prepíše ho. `Next` funkcia vracia pole aktuálnych kúskov a pravdivostnú hodnotu, či bolo niečo nové vygenerované.

`GuessGeneration` pre každý vytvorený `GuessIndex` volá ich prisluchajúcu `Next` funkciu a jednotlivé kúsky heslá spojí do terminálu, ktorý následne vráti. Príklad pri generovaní zo základnej štruktúry `A5D1A1D2`. Nižšie sú ukazané volania jednotlivých funkcii a stav pola, ktoré obsahuje kúsky hesla.

1. Shadow [citron] - pridal sa prvý prvok `citron`,
2. Capitalization [Citron] - aplikovaná maska $ULLLL$,
3. Copy [Citron, 4] - nakopírovaná číslica 4,
4. Shadow [Citron, 4, d] - pridaný reťazec `d`,
5. Capitalization [Citron, 4, d] - aplikovaná maska L,
6. Copy [Citron, 4, d, 42] - nakopírovaná číslica 42

Výsledne pole sa zoberie a spojí, vznikne prvý terminál `Citron4d42`.

5.3 Prioritná fronta

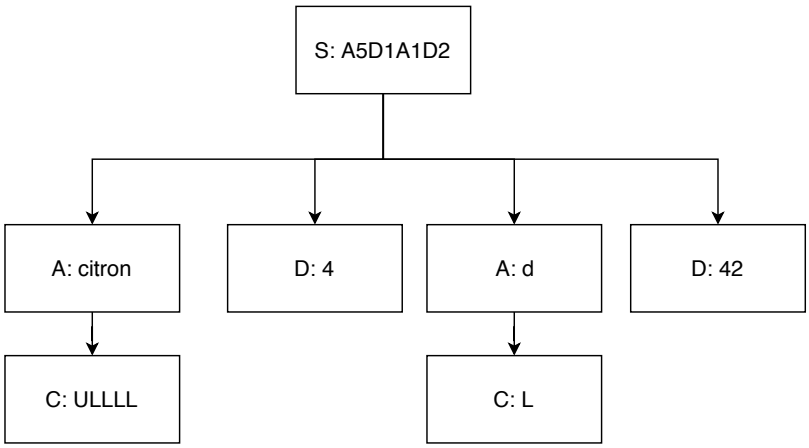
Prioritná fronta je jadrom generovania. Stará sa, aby vyberané prvky boli podľa najvyššej pravdepodobnosti. Pri inicializácii sa vloží ako prvý prvok štartovací symbol. Pri vkladaní sa vypočíta a priradí pravdepodobnosť. Prvok sa nachádza v stromovej štruktúre a indexuje sa do gramatiky. Ak nie je koreňový prvok, tak sa pri počítaní pravdepodobnosti násobí

pravdepodobnosť jednotlivých dcérskych prvkov. Na výpise 5.2 je štruktúra, ktorá je uložená v prioritnej fronte. `Index` ukazuje do sekcii v gramatiky a `Transition` do náhrady v sekcii.

```
type TreeItem struct {
    Index int
    Transition int
    Childrens []*TreeItem
}
```

Výpis 5.2: Štruktúra TreeItem

Pri vyberaní z fronty sa vyberie prvok s najväčšou pravdepodobnosťou. Prvku sa nájdu jednotlivé dcérske prvky pomocou algoritmu *DeadbeatDad* viď sekcia 3.2.2. Každý prvok je znova zaradený do prioritnej fronty. Prvok je vrátený generátoru iba v prípade, že je to už preterminálna štruktúra, z ktorej sa budú generovať terminály. Na obrázku 5.1 môžeme vidieť príklad takéhoto prvku (S - štartovací symbol, A - reťazec z písmen, D - číslica, C - kapitalizácia). Tento prvok vznikol pôvodne z prvku, ktorý obsahoval len koreňový uzol zo štartovacieho symbolu. Postupne vznikajú ďalšie a väčšie prvky, ktoré sa zaraďujú do prioritnej fronty.

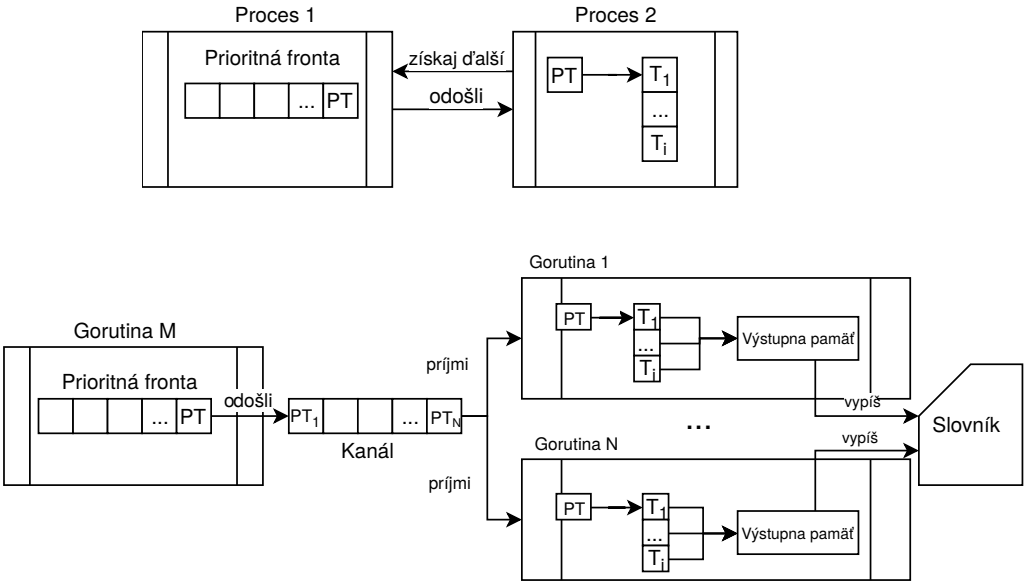


Obr. 5.1: Príklad prvku, ktorý bude generovať terminály

5.4 Manažér generátoru

Manažér generátoru používa prioritnú frontu a vyberá z nej postupne prvky. Z každého vybraného prvku vypíše terminály pomocou generátoru hesiel. V pôvodnej Pythonovej verzii viď obrázok 5.2 (PT - preterminálna štruktúra, T - terminál) jeden proces generuje preterminálne štruktúry a druhý terminály. V Go verzii je generátor rozdelený na niekoľko gorutin.

Gorutinu [2] môžeme chápať ako odľahčene vlákno o ktoré sa stará plánovač samotného Go. Na zásobníku oproti vláknam zaberajú len pár KB a veľkosť sa môže zväčšovať/zmenšovať podla potreby aplikácie. Gorutiny su multiplexované na menšie množstvo vlákien operačného systému. Môže existovať len jedno vlákno, ktoré obsluhuje tisícku gorutin. Ak jedna z gorutin vo vlákne blokuje napríklad na vstup užívateľa, tak zvyšné gorutiny su presunuté do nového vlákna. Komunikujú prostredníctvom kanálov. Kanál je rúra, ktorá



Obr. 5.2: Porovnanie generovania v Pythone oproti Go

spája súbežne gorutiny. Zamedzuje *race conditions* pri prístupu k zdielanej pamäti. Jedna gorutina môže do kanálu posielať hodnoty a druhá prijímať. Predvolené je prijímanie a odosielanie blokujúca operácia pokým nie je prijímateľ aj pripravený. Existujú vyrovnávacie kanály (*buffered channels*), do ktorých je možné odoslať viacero hodnôt naraz, bez toho aby odosielanie bolo blokované, pokým hodnotu niekto nepríjme.

Hlavná gorutina sa stará o prioritnú frontu, generovanie preterminálnych štruktúr a ich odosielanie do vyrovnávacieho kanálu, ktorého veľkosť je daná počtom gorutín. Ostatné gorutiny prijímajú preterminálne štruktúry z kanálu. Každá gorutina generuje terminály a vypisuje ich na výstup. Pri vygenerovaní je zvýšení čítač počtu vygenerovaných terminálov. Čítač je použitý pre možnosť definovania maximálneho počtu terminálov, čo v pôvodnej verzii nebolo možné. Táto možnosť ale nie je presná. čítač sa kontroluje pred vytiahnutím ďalšieho prvku z prioritnej fronty. To znamená, že ak obmedzíme počet vygenerovaných terminálov na 10 a máme už 5 vygenerovaných, vytiahneme z fronty ďalšiu preterminálnu štruktúru, ktorá ale môže vygenerovať viac ako 5 terminálov.

5.5 Porovnanie Python vs Go

Po implementácii bolo overené, či nová verzia produkuje rovnaké výsledky ako pôvodná. Testy boli vykonané na niekoľkých gramatikách a výstup porovnaný. Generátor bol vždy ukončený po 15 minútach, pretože konca behu generátora na vybraných gramatík by sme sa nedočkali. Go verzia vygenerovala viac hesiel. Zobral sa počet hesiel vygenerovaných z Python verzie a slovník vygenerovaný z go verzie bol orezaný na tento počet. Obe verzie vygenerovali rovnaký výsledný slovník. Jediný rozdiel, bol že v niektorých prípadoch heslá neboli vygenerované v rovnakom poradí. Ide ale len o rozdiel pár pozícii, ktoré je spôsobené ak preterminálna štruktúra má rovnakú pravdepodobnosť ako iná.

Testovacia zostava:

- **OS:** Ubuntu 16.04 64-bit
- **CPU:** Intel i7 4700-HQ
- **RAM:** 8GB - DDR3L 1600 MHz
- **SSD:** Samsung 850EVO
- **HDD:** 1TB HDD 5400 RPM

Testy boli robené na 2 rozličných gramatikách:

- Default - prevzaté z `https://github.com/lakiw/pcfg_cracker`,
- rockyou-75 - natrénovaná gramatika z TOP75% uniknutých hesiel z rockyou.

V tabuľke 5.1 môžeme vidieť porovnanie rýchlostí jednotlivých verzii. Pri gramatike `default` go verzia bola rýchlejšia 1,75 krát, pri gramatike `rockyou` 3,8 krát. Ale počet gorutin mal minimálny efekt na výslednú rýchlosť, rýchlosť sa zvýšila len o 5-10%.

	Python	**Go - 1 gorutina**	**Go - 16 gorutin**
Default	172,625	284 048	300 601
Rockyou	75,342	265 173	291 743

Tabuľka 5.1: Rýchlosť generovania hesiel za sekundu

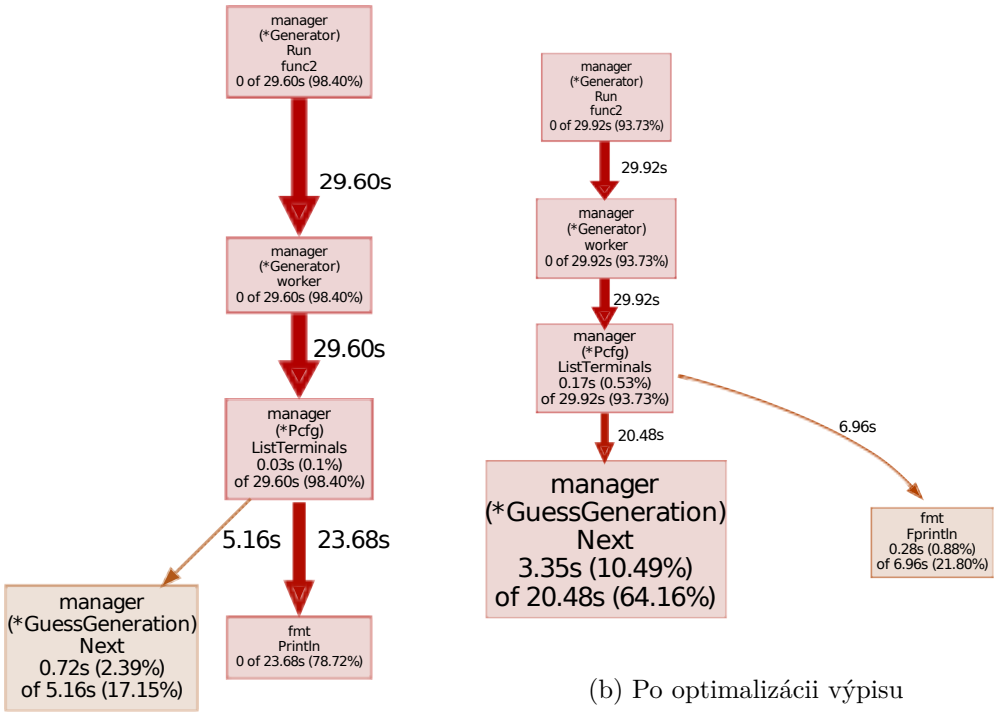
Následne sa vykonalo profilovanie generátoru pomocou nástroja `go tool pprof` na vzorke 30 sekundového generovania. Výstup je orezaný graf volaní funkcii a čas ich trvania. Na obrázku 5.3a môžeme vidieť, že väčšinu času zabrala funkcia `fmt.Println`, ktorá len vypisuje heslo na výstup. Pri každom výpise hesla je zavolané systémové volanie. To aj vysvetľuje, prečo väčší počet gorutin príliš nezrýchlil generovanie.

Riešením tohto problému je vyrovnávací zápis. Namiesto toho, aby sa každý terminál ihneď vypísal na výstup cez systémove volanie, tak sa bude ukladať do pamäte. Pre každú preterminálnu štruktúru sa vytvorí vyrovnávacia pamäť, do ktorej sa budu postupne zapisovať heslá. Akonáhle sa vygeneruje posledné heslo zo štruktúry, tak sa heslá vypíšu na výstup. Po tejto optimalizácii sa vykonalo znova profilovanie. Na obrázku 5.3b môžeme vidieť ze pomer medzi logikou generovania a samotným výpisom sa vymenil.

Po optimalizácii sa znova otestovala rýchlosť generovania. Tentokrát sa testovalo na HDD aj na SSD. Výsledky teraz boli znateľne, na grafoch 5.4 a 5.5 vidíme porovnanie koľko hesiel bolo vygenerovaných za 3 minúty behu. Pri pôvodnej verzii SSD nehrá takú veľkú rolu ako pri Go verzii, kde pri gramatike Rockyou použitie SSD skoro zdvojnásobilo rýchlosť generovania viď tabuľka 5.2. Vo výsledku nová Go verzia oproti pôvodnej Python verzii dosahuje až 13 - 40 násobne zrýchlenie.

	Python	Go	Zrýchlenie
Default (HDD)	164 520	2 254 555	13,7×
Rockyou (HDD)	102 236	2 725 752	26,7×
Default (SSD)	180 013	2 695 802	15,0×
Rockyou (SSD)	115 797	4 681 641	40,3×

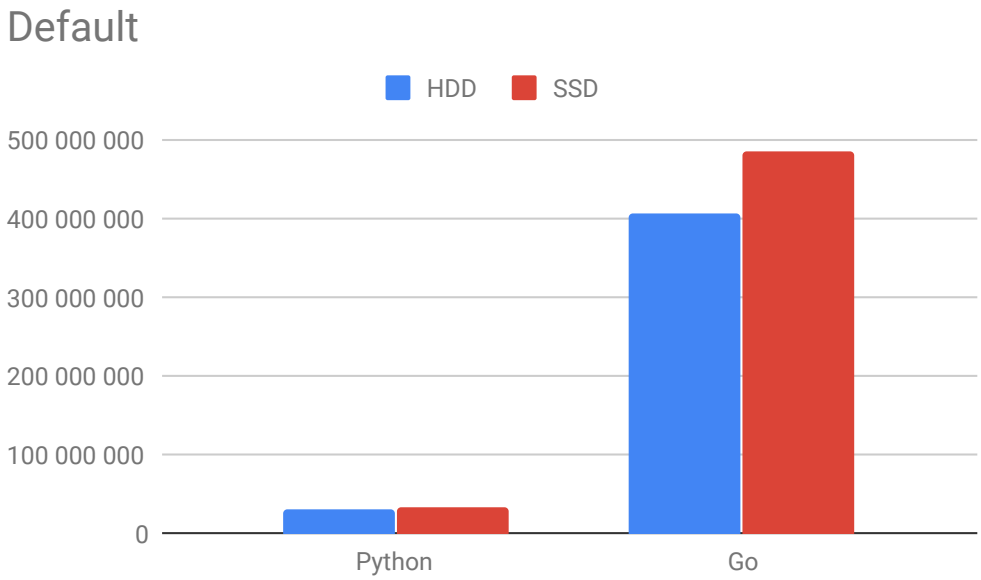
Tabuľka 5.2: Rýchlosť generovania hesiel za sekundu po optimalizácii



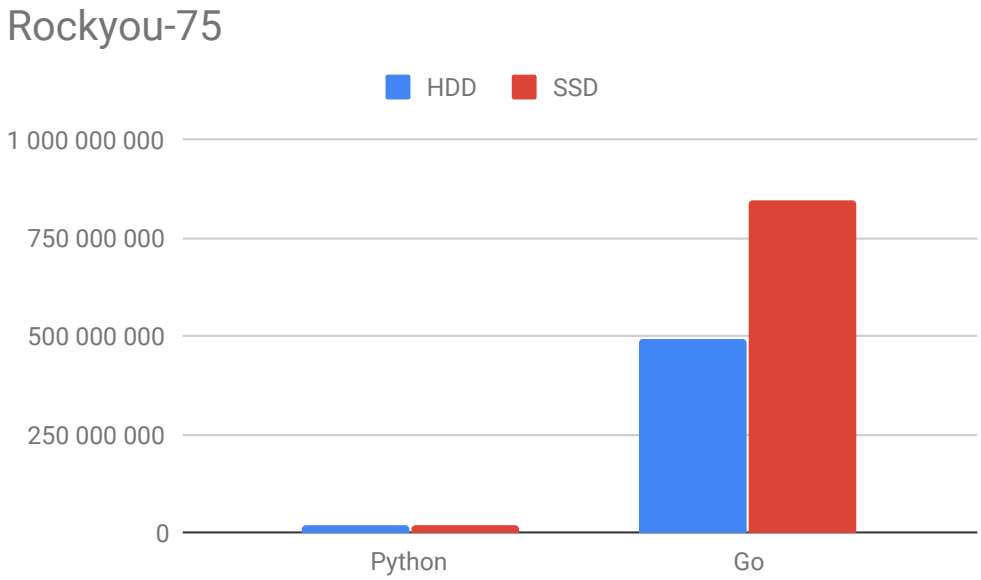
(a) Pred optimalizáciou

(b) Po optimalizácii výpisu

Obr. 5.3: Profil generovania



Obr. 5.4: Počet vygenerovaných hesiel za 3min pri gramatike Default



Obr. 5.5: Počet vygenerovaných hesiel za 3min pri gramatike Rockyou

Kapitola 6

Implementácia distribuovaného generátoru

Táto kapitola sa zaoberá samotnou implementáciou, ktorej návrh je popísaný v kapitole 4.2. Implementácia rozširuje a používa reimplementovaný generátor z kapitoly 5.

6.1 Architektúra

Distribúcia je založená na architektúre klient-server, v ktorej server poskytuje služby a klient sa na ne aktívne dotazuje [11]. Predstavuje medziprocesovú komunikáciu, pretože sa vymieňajú dáta medzi serverom a klientom, kde každý z nich plní inú úlohu. Klient nemusí vedieť ako server plní svoju úlohu, stačí mu aby rozumel odpovedi zo servera stanovená na vopred definovanom komunikačnom protokole. Všetky protokoly operujú na aplikačnej vrstve. Pre formalizáciu výmeny dát server implementuje programové aplikačné rozhranie (API - *application programming interface*), ktorý slúži ako abstraktné rozhranie pre prístup ku službe. Výhody tohto prístupu sú:

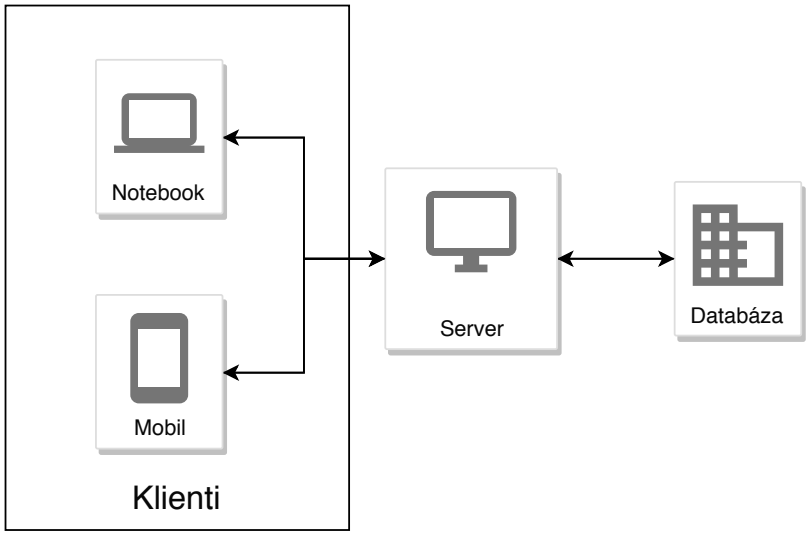
- centralizovaná kontrola, server môže napríklad odoprieť prístup klientovi,
- rozdelenie záťaže výpočtu na niekoľko výpočetných strojov,
- znižuje replikáciu dát, kde sa dáta ukladajú len na serveri.

Medzi najznámejší príklad využívajúc tejto architektúry je HTTP (*hypertext transfer protocol*) protokol [4]. Prehliadač (klient) žiada o multimediálne súbory ako obrázky, text, javascriptové súbory a server ich ako odpoveď vracia. Klienta nezaujíma odkiaľ a ako ich dostane. Server môže mať centrálnu databázu viď obrázok 6.1, brať súbory z iného úložiska alebo ich generovať pri klientskej žiadosti, o čom samotný klient nevie.

6.2 Aplikačné rozhranie

Aplikačné rozhranie je definované cez protocol buffer. Server poskytuje službu PCFG so 4 metódami viď výpis 6.1.

Metódu `Connect` volá klient ihneď pri spustení. Server odpovedá správou `ConnectResponse` viď výpis 6.2, ktorá obsahuje gramatiku, zoznam hešov a mód pre hashcat. Gramatika je nutná pre samotné generovanie hesiel. Zoznam hešov sú heše, ktoré hashcat sa bude snažit prelomiť a mód hashcatu je typ zaslaných hešov.



Obr. 6.1: Príklad klient-server architektúry

```
service PCFG {
    rpc Connect (Empty) returns (ConnectResponse) {}
    rpc Disconnect(Empty) returns (Empty);
    rpc GetNextItems(Empty) returns (TreeItems) {}
    rpc SendResult(CrackingResponse) returns (ResultResponse);
}
```

Výpis 6.1: Aplikačné rozhranie pre distribúciu generovania

```
message ConnectResponse {
    Grammar grammar = 1;
    repeated string hashList = 2;
    string hashcatMode = 3;
}
```

Výpis 6.2: Správa ConnectResponse

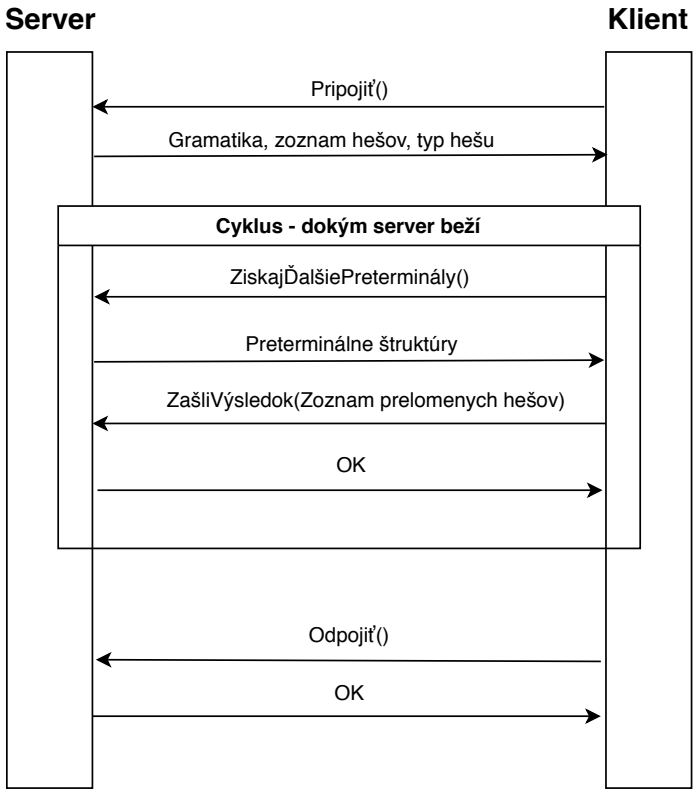
Po pripojení nasleduje klientska žiadosť pomocou metódy `GetNextItems` o preterminálne štruktúry, z ktorej bude klient generovať terminály. Klient neposiela žiadne argumenty na server a server mu pridelí úlohu s `N` preterminálnymi štruktúrami.

Po skončení generovania a lámania hesiel, klient odošle výsledok na server. Výsledok obsahuje mapu prelomených hešov (viď výpis 6.3) asociovaných k heslu. Ak klient neprelomil žiaden heš, tak odošle prázdnu mapu. Server ako odpoveď posiela len príznak označujúci, či sa má klient ukončiť.

```
message ResultResponse {
    bool end = 1;
}
message CrackingResponse {
    map<string, string> hashes = 1;
}
```

Výpis 6.3: Správy pri žiadosti GetNextItems

Posledná správa je `Disconnect`, ktorou klient serveru len oznamuje svoje ukončenie a ten môže na to adekvátne zareagovať. Napríklad, ak klient mal pridelenú úlohu a neoznámil jej výsledok a klient sa ukončí, tak server zaradí úlohu do fronty pre ďalšie uzly. Tok správ je znázornený na obrázku 6.2.



Obr. 6.2: Komunikácia medzi serverom a klientom generátora

6.3 gRPC

gRPC (*Remote Procedure Calls*)[1] je otvorený (*open source*) aplikačný rámec (*framework*) pre vzdialene volanie procedúr. Slúži pre komunikáciu klientov so serverom. Vyvinutý Googlom, ktorý to používa na prepojenie veľkého množstva mikroslužieb. Medzi hlavné výhody patrí:

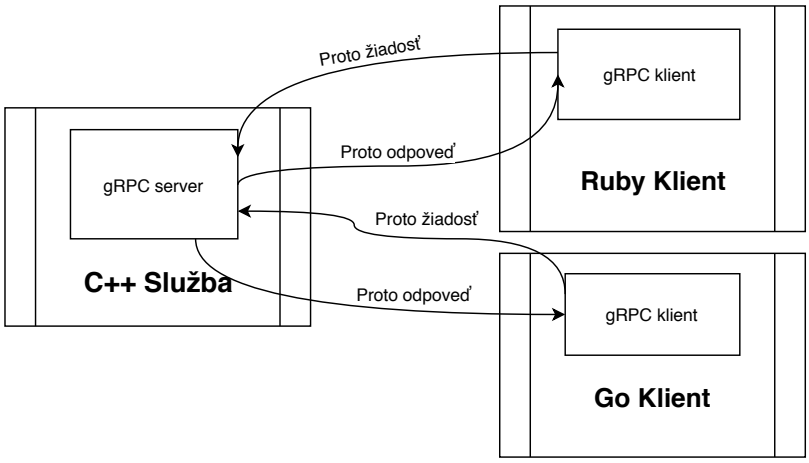
- nízka latencia,
- vysoká škálovatelnosť,
- podpora množstva programovacích jazykov (C++, Java, Python, Ruby, Go, Node, Objective-C).

Používa *protocol buffers* ako jazyk popisujúce rozhranie (*Interface Definition Language*) ako aj základny formát pre výmenu správ.

[1] https://grpc.io/docs/guides/index.html

Klientská aplikácia volá metódy na serverovej aplikácii, ako keby to boli lokálne objekty, čo zľahčuje vývoj distribuovaných aplikácii a služieb. gRPC je založený na myšlienke definovania služby, špecifikuje metódy, ktoré môžu byť volané a ich parametre s návratovými hodnotami. Na serverovej aplikácii sa implementuje rozhranie a spúšťa sa gRPC server pre spracovanie klientských volaní. Na klientskej strane je vygenerovaný klient, ktorý poskytuje rovnaké metódy ako server. Klient sa následne len pripojí na server a dané metódy volá.

Klienti a serveri môžu bežať a komunikovať navzájom v rôznych prostrediach. Každá časť môže byť napísaná v rôznom podporovanom programovacom jazyku. Server môže byť napísaný v C++ a 2 rôzny klienti napríklad v Ruby a Go viď obrázok 6.3.



Obr. 6.3: Komunikácia v gRPC v rozličných programovacích jazykov

Pomocou protocol buffers gRPC definuje služby. Na výpise 6.4 je definovaná jedna služba `HelloService` s metódou `SayHello`, ktorá prijíma správu `HelloRequest` a vracia správu `HelloResponse`. Existuju 4 typy služieb:

1. unárne (*Unary*) - klient odošle jednú správu na server, ktorý odpovedá jednou odpoveďou ako normálne volanie funkcie,

2. streamovanie zo servera (*Server streaming*)- server na správu od klienta odosiela tok dát.

3. streamovanie z klienta (*Client streaming*) - klient odosiela tok sekvenciu správ, keď skončí, tak čaká kým ich server prečíta a vráti odpoveď,

4. obojsmerné streamovanie (*Bidirectional streaming*) - obe strany odosielajú a čítaju správy. Oba toky dát fungujú nezávisle, takže môžu čítať a zapisovať v poradí v akom chcú.

gRPC funguje na novom HTTP2, ktoré poskytuje binárne rámce a kompresiu. HTTP2 prinieslo asynchronitu, multiplexovanie žiadosti pomocou streamov, ktoré GRPC využíva. GRPC podporuje 2 autorizačné mechanizmy:

1. SSL/TLS - používa sa na autorizáciu servera a na šifrovanie správ medzi klientom a serverom,

2. autorizáciu s googlom pomocou tokenu - používa sa pre prístup k Google API.

Medzi ďalšie možnosťami patrí definovanie času pre vypršanie žiadosti (*timeout*), prerušenie žiadosti.

```
service HelloService {
    rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
    string greeting = 1;
}

message HelloResponse {
    string reply = 1;
}
```

Výpis 6.4: Príklad služby v protocol buffer

Protocol Buffers

Protocol buffers[2] sú jazykovo a na platforme nezávisle mechanizmy pre serializovanie štrukturovaných dát. Definujú ako dáta majú byť štrukturované a následne je vygenerovaný kód pre zadaný jazyk, ktorý umožňuje ľahko, rýchlo čítať a zapisovať do dát.

Definujú sa v súbore `.proto`, ktorý obsahuje správy (*messages*) a tie sú zložené z skalárnych typov, ďalších správ, vymenovania alebo pole skalárnych typov či správ. Medzi skalárne typy patria double, float, int32, int64, uint32, uint64, sint32, sint64, fixed42, fixed64, bool, string, bytes. Skalárne typy su po vygenerovaní kódu pre daný programovací jazyk prevedený na ich ekvivalent. Napríklad proto typ `bytes` je v C++ prevedený na typ `string` a v Jave na `ByteString`. Na výpise 6.5 je príklad definovania správ. Správa `Foo` obsahuje 2 polia (*fields*). Jedno je `number` skalárneho typu `int`. Druhé je vnorené s názvom `embeddedBar` obsahujúce nám definovanú správu `Bar`. Správa `Bar` obsahuje pole čísiel a definuje vymenovanie, ktoré je aj použité ako druhé pole. Každé vymenovania musí mať prvok s hodnotou 0.

```
message Foo {
    int number = 1;
    Bar nestedBar = 2;
}
message Bar {
    repeated int numbers = 1;
    enum Status {
        OK = 0;
        ERROR = 1;
    }
    Status status = 2;
}
```

Výpis 6.5: Príklad správy v protocol buffer

Každé prvok má v rámci správy priradené unikátne číslo. Tieto čísla slúžia pre označenie prvku v rámci binárneho formátu a nemalo by byť menené. Čísla od 1 do 15 zaberajú len jeden bajt a mali by byť použité pre frekventované prvky. Prvky ktoré majú nulovú hodnotu (0 pre `int`, prázdny reťazec ‘‘ pre `string`, `false` pre `bool`) nie sú vôbec zaslané a prijímateľ

[2] https://developers.google.com/protocol-buffers/docs/overview

pri deserializácii doplní nulovú hodnotu. Toto treba brať v úvahe pri definovaní správ, kedy napríklad nejde rozlíšiť, či bola zaslaná hodnota `false`, alebo nebola zaslaná vôbec.

Aktualizovanie správ je jednoduché. Pri pridaní nového prvku do správy sa nerozbije kompatibilita so starým vygenerovaním kódom. Pre zachovanie kompatibility je nutné zachovať priradené čísla. Prvky môžu byť zo správy vymazané, pokiaľ jeho číslo nebude znova použité.

6.4 Server

Server je centrálny uzol pre distribúciu generovania. Načítava gramatiku, generuje preterminálne štruktúry, ktoré ďalej rozposiela pripojeným klientom. Spúšťa sa ako ďalší príkaz samotného generátora: `pcfg-manager server`. Obsahuje niekoľko prepínačov:

- `-r` - meno gramatiky,
- `--hashlist` - cesta k súboru, v ktorom sa nachádza zoznam hešov na prelomenie,
- `--hashcat-mode` - určenie typu hešu,
- `--port` - určuje na ktorom porte bude gRPC server počúvať,
- `--term-que-size` - určuje veľkosť kanálu v ktorom sa uchovávajú preterminálne štruktury, ktoré vedú ku generovaniu terminálov, štandardná hodnota je 10 000,
- `--chunk-start-size` - určuje koľko preterminálných štruktúr bude zaslaných pri prvom pripojení klienta, štandardná hodnota je 10 000,
- `--chunk-duration` - určuje ako dlho by malo lámanie hesla u klienta trvať a na základe rýchlosti sa bude prispôsobovať veľkosť zaslaných preterminalných štruktúr, štandardná hodnota je 30 sekúnd.

Pri spustení si server načíta zoznam hešov a vytvorí si manažér pre generovanie hesiel z gramatiky. Spustí sa nová gorutina pre generovanie hesiel do vyrovnávacieho kanálu o veľkosti určené prepínačom `term-que-size`. Server bude generovať preterminálne štruktúry, ktoré sa ďalej generujú na terminály, pokiaľ kanál nebude plný. Týmto prístupom sú docielené 2 veci:

1. server nebude zbytočne generovať preterminálne štruktúry, pokiaľ ich nemá kto ďalej generovať,
2. kanál slúži ako fronta, keď si klient vypýta ďalšiu časť úlohy, tak nemusí čakať kým server vygeneruje nové preterminálne štruktúry, ale vyberie ich rovno z kanála.

Po spustení generátora sa spustí gRPC server, ktorý sprostredkováva komunikáciu s klientmi. Server si uchováva informácie o pripojených klientov v internej mape `clients`, do ktorej sa pristupuje adresou klienta a obsahuje `ClientInfo` viz výpis 6.6. Obsahuje gRPC adresu klienta, aktuálnu úlohu ak mu už bola pridelená, čas pridelenia úlohy a čas skončenia poslednej úlohy. Okrem toho obsahuje počet terminálov z predošlej úlohy, táto sa používa pre prispôsobovanie veľkosti úlohy pre klienta.

```
type ClientInfo struct {
    Addr string
    ActualChunk Chunk
    StartTime time.Time
```

```
    EndTime time.Time
    PreviousTerminals uint64
}

type Chunk struct {
    Id uint32
    Items []*pb.TreeItem
    TerminalsCount uint64
}
```

Výpis 6.6: Informácie o klientovi

Po pripojení klienta gRPC metódou `Connect()`, si server z gRPC pripojenia extrahuje adresu klienta a uloží si informáciu o klientovi do internej mapy. Následne je klientovi zaslaný zoznam neprelomených hešov, spolu s gramatikou a typom hešu.

Pri žiadosti o novú úlohu metódou `GetNextItems()`, si server vytiahne údaje o klientovi. Ak je to prvá úloha pre klienta, tak mu je zaslaná o veľkosti, ktorú definuje prepínač `chunk-start-size`. V opačnom prípade sa spočíta rýchlosť predchádzajúcej úlohy, ktorá je daná v sekundách.

$$rychlost = PredosleTerminaly/(KoncovyCas - ZaciatocnyCas)$$

Nakoniec sa určí veľkosť úlohy pomocou rýchlosti a dĺžky úlohy, ktorá je definovaná prepínačom `chunk-duration`. Na základe veľkosti úlohy je vytvorená samotná úloha pomocou funkcie `GetNextChunk`.

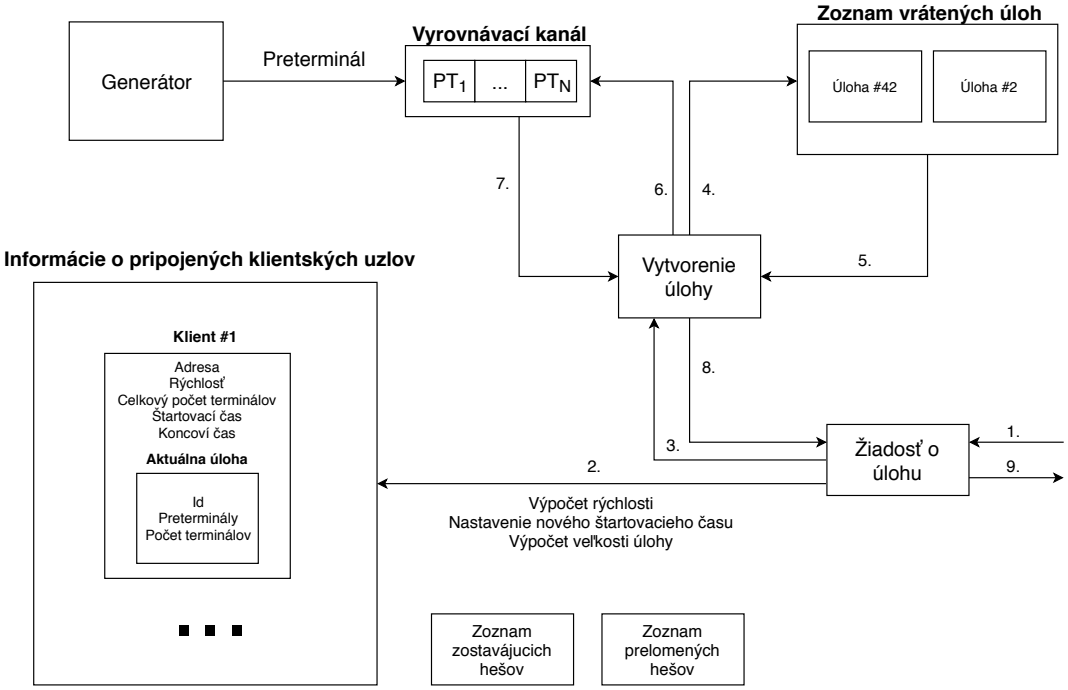
$$velkostUlohy = rychlost * dlzkaUlohy$$

Vytvorenie úlohy

Funkcia `GetNextChunk(size uint64)` vytvára novú úlohu. Ako parameter berie veľkosť, ktorá udáva aké množstvo terminálov by mala byť približne schopná vygenerovať. Z vyrovnávacieho kanála si po jednom berie preterminálné štruktúry. Spočíta koľko terminálov bude z neho vygenerovaných a pripočíta toto množstvo do počítadla. Prevedie internú štruktúru preterminálnej štruktúry do gRPC správy a pridá ju do úlohy. Toto sa opakuje do tej doby pokým nenastane jedno z nasledujúceho:

1. Počítadlo nie je väčšie ako požadovaná veľkosť,
2. Do 2 sekúnd z kanála neprišla nová preterminálna štruktúra,

Tok informácii a architektúra servera je znázornená na obrázku 6.4. Výnimka tohto generovania je, ak sú vo fronte vrátené úlohy. Vrátené úlohy vznikajú, keď sa klient odpojil s pridelenou úlohou a neposlal výsledok. V tomto prípade vrátené úlohy majú prioritu a je znovu priradená novému klientovi. Každá úloha je inkrementálne očíslovaná a vrátená ako výstup z funkcie. Nižšie číslo úlohy znamená, že vygeneruje terminály, ktoré majú väčšiu pravdepodobnosť ako vygenerované terminály z úlohy s väčším číslom. Toto je priestor na zlepšenie, akým spôsobom znova priraďovať vrátené úlohy. V aktuálnej implementácii, sa zoberie prvá vrátená úloha, ktorej veľkosť je menšia ako 1.1-násobok požadovanej veľkosti. Na ďalšom zvážení je, či by nebolo výhodnejšie vrátenú úlohu znova rozdeliť na jednotlivé preterminálné štruktúry, z ktorých by zostrojovali nové úlohy rovnakým spôsobom ako je popísaný vyššie. Pseudokód algoritmu viď 1 je znázornený nizšie.



Obr. 6.4: Architektúra servera a priebeh toku údajov pri získavaní novej úlohy

Po tom ako klient skončí generovanie a lámanie hesiel, odošle metódou `SendResult` výsledok na server. Server si vytiahne informácie o klientovi, odstráni priradenú úlohu, aktualizuje koncový čas a počet predošlých terminálov. Následne z výsledku prejde všetky heše, zo zoznamu zostávajúcich hešov odstráni prelomené heše a tieto heše spolu s prelomeným heslom uloží do zoznamu prelomených hešov. Ak nezostáva žiaden heš na prelomenie, tak sa gRPC server ukončí a sú vypísané prelomené heslá.

Posledná metóda týkajúca sa komunikácii s klientom je `Disconnect`. Odstraňuje informácie o klientovi zo zoznamu pripojených klientov. Okrem toho, kontroluje či mal priradenú úlohu. Ak áno, táto úloha je priradená na koniec zoznamu vrátených úloh. Problém nastáva ak sa klient ukončí a neoznámi to. Tu je priestor na zlepšenie, kde by bolo vhodné pridať kontrolu, ak klient nedokončí úlohu do stanoveného času, označiť ju za nedokončenú a priradiť ju znova do zoznamu vrátených úloh.

6.5 Klient

Klient sa stará o generovanie hesiel z preterminálnych štruktúr a ich lámania. Spúšťa sa ako ďalší príkaz samotného generátora: `pcfg-manager client`. Obsahuje niekoľko prepínačov:

- `--server` - definuje adresu a port servera, štandardne localhost:50051,
- `--hashcat-folder` - určuje cestu k zložke nástroja Hashcat, štandardne ./hashcat ,
- `--generate-only` - pri použití sa heslá len generujú a nepoužíva sa hashcat pre lámanie.

Po spustení sa klient ihneď pripojí na server cez gRPC. Zavolá metódu `Connect`, cez ktorú obdrží gramatiku a zoznam hešov pre prelomenie. Po pripojení beží klient v cykle.

Algoritmus 1 Vytvorenie úlohy

```
function GetNextChunk(size)
    for chunk ← returnedChunks.Front(); chunk ≠ nil; chunk ← ch.Next() do
        if chunk.terminalsCount < size * 1.1 then
            return chunk
        end if
    end for
    chunk.terminalsCount ← 0
    chunk.items ← []
    while chunks.terminalsCount < size do
        if timeout(PT ← GeneratorChannel, 2s) then
            break
        end if
        Push(chunk.items, PT)
        chunk.terminalsCount ← chunk.terminalsCount + CountTerminals(PT)
    end while
    chunk.id ← lastChunkID + +
    return chunk
end function
```

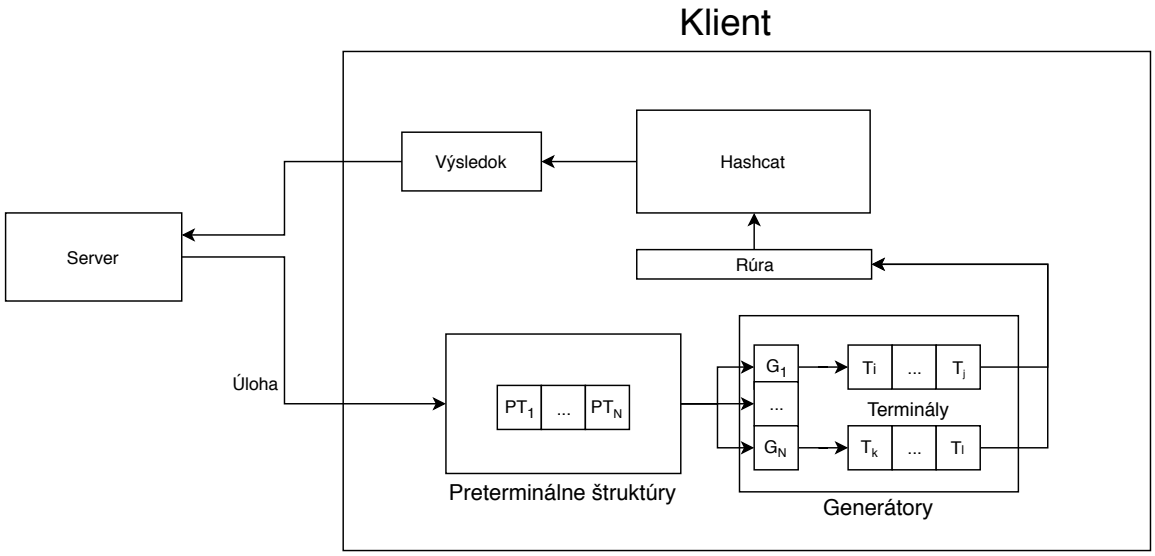
Metódou `GetNextItems` získa preterminálne štruktúry z ktorých bude generovať terminály. Následne nastane jedna z nasledujúcich situácii na základe parametru spusteného klienta:

1. klient bude heslá len generovať,
2. klient heslá generuje a posúva ich do nástroja Hashcat.

Pri prvej možnosti, prechádza po jednom každú preterminálnu štruktúru a vypisuje na štandardný výstup terminály. Pri vypisovaní je použitá vyrovnávacia pamäť. Terminály z jednej preterminalnej štruktúry sú najprv ukladané do pamäti. Na konci je táto pamäť vypísaná na výstup. Týmto prístupom sa odľahčí zápis na disk, kedy by sa pri každom hesle volalo systémove volanie pre zápis a spomaľovalo by to generovanie.

Pri druhej možnosti sa mimo generovania spúšťa aj Hashcat pre lámanie hešov. Pri prijatí úlohy sa spustí Hashcat a vytvorí sa rúra na štandardný vstup, cez ktorú sa ďalej budú posielať terminály znázornený na obrázku 6.5. Pri generovaní terminálov z každej preterminálnej štruktúry sa terminály zapisujú do vytvorenej rúry. Týmto prístupom sú postupne nástroju Hashcat predávané heslá, ktorý ich bude skúšať na zozname hešov prijatého zo servera. Po vygenerovaní všetkých terminálov sa ukončí rúra a čaká sa na skončenie Hashcatu. Po skončení sa skontroluje návratová hodnota a následne sa získajú výsledky.

Po úspešnom generovaní sa výsledky zašlú serveru, kde pri prvej možnosti sa vždy pošle prázdny zoznamov prelomených hešov. Pri použití Hashcatu, sa otvorí súbor s výsledkami a pretransformuje ho do gRPC správy. Ak server už nemá ďalšiu úlohu pre klienta, tak pomocou metódy `Disconnect` sa odpája a končí. Klient reaguje na signály `SIGINT` a `SIGTERM`, kedy ukončí generovanie a odpojí sa od servera. Server na to môže zareagovať a prideliť nedokončenú úlohu klienta inému klientskému uzlu.



Obr. 6.5: Generovanie a lámanie hesiel na klientskom uzle

6.6 Možnosti vylepšenia

V rámci implementácie sú stále možnosti na zlepšenie. Jeden z problémov reimplementovaného programu je strata úlohy, keď sa klient nečakane odpojí, bez toho aby dal serveru vedieť. Toto môže nastať ak klient je nasilu ukončený, čo môže zapríčiniť výpadok energie alebo nedostupnosť pripojenia. Riešenie tohto problému môže vyriešiť kontrolovanie času pridelenia úlohy s parametrom `chunk-duration`. Server by v stanovenom intervale kontroloval každého pripojeného klienta. Ak by rozdiel aktuálneho času s časom pridelenia úlohy bol väčší ako 2× `chunk-duration`, tak by sa klient prehlásil za odpojeného a jeho úloha by sa zaradila medzi vrátené úlohy. Tieto úlohy by boli následne pridelené ostatným aktívnym klientom a žiadna úloha by nebola stratená.

Reimplementovaný generátor nepodporuje Markovské modely, ktoré pôvodný generátor podporoval. Preto je nutné gramatiku trénovať s prepínačom `--coverage 1`, ktorý spôsobí že vygenerovaná gramatika nebude obsahovať Markovské modely. Modely slúžia pre doplnenie generovania hesiel, ktoré sa v trénovanej sade vôbec nevyskytovali.

Ďalším problémom je autorizácia klientov, ktorá nie je vôbec riešená. Ak je server vystavený verejne internetu, tak ktokoľvek so znalosťou aplikačného protokolu a adresy servera sa môže pripojiť. Útočník môže zahltiť server správami o pridelenie úlohy. Úlohu by ihneď odoslal ako ukončenú bez nájdeného hesla. Vypočítaná rýchlosť klienta serverom by bola neskutočné veľká, čo by spôsobilo snahu o vygenerovania veľkej úlohy. Nie len že by sa takto heslá strácali, ale spôsobilo by to aj neschopnosť odpovedať ostatným klientským uzlom. Na koniec by serveru pravdepodobne došla pamäť a celý distribuovaný výpočet by stroskotal. Riešenia, ktoré by z časti mohli vyriešiť tento problém:

1. stanovenie maximálnej veľkosti úlohy - zredukovalo by šancu nedostatku pamäte, ale nerieši stratené heslá,

2. posielanie rovnakej úlohy viacerým klientom - rovnaká úloha bude odoslaná viacerým klientským uzlom čím správnosť dokončenej úlohy potvrdí viacero uzlov, čo ale vo výsledku spomalí cely proces,

3. použitie gRPC certifikátov pre klientské uzly ktorými sa autorizujú,

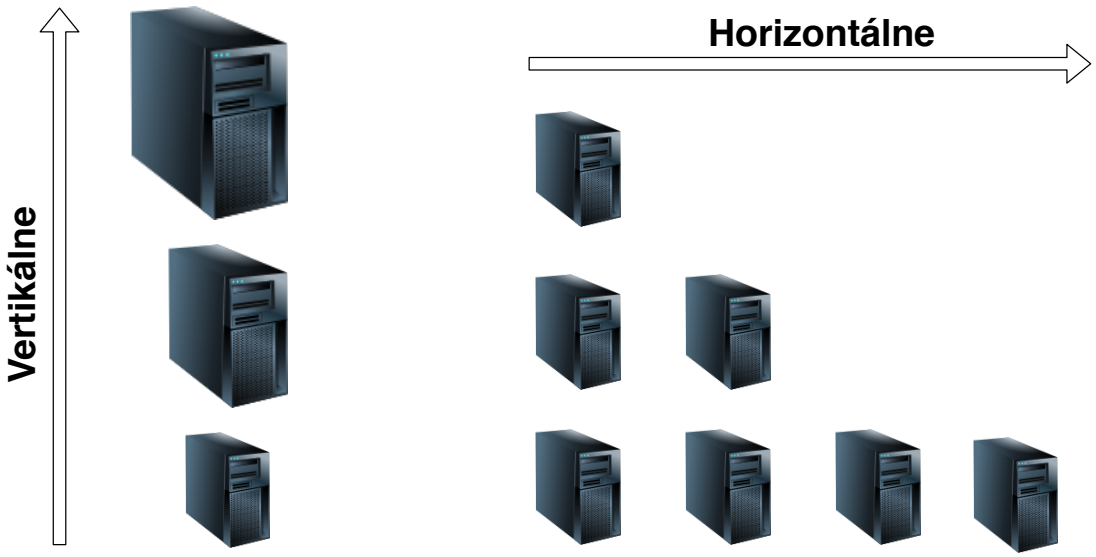
4. VPN (*Virtual Private Network*) - všetky klientský uzly a server budú v súkromnej virtuálnej sieti

Ďalší nedostatok je nemožnosť horizontálnej škálovateľnosti servera. Horizontálna škálovatelnosť [3] (viď obrázok 6.6) je pridávanie zariadení za účelom zvýšenia výkonu. Vertikálna je zvyšovanie výkonu jedného zariadenia (lepšie CPU, väčšia pamäť, ...). Rýchlosť generovania terminálov sa dá veľmi ľahko škálovať pridaným ďalších klientov do distribuovaného výpočtu. Lenže po určitom množstve server nemusí stíhať generovať preterminálné štruktúry. A tým pádom klienti budú zbytočne čakať na úlohu. Server pri aktuálnej implementácii nie je možné jednoducho škálovať. Jadrom generovania preterminálných štruktúr je prioritná fronta, ktorá sa stále mení počas generovania štruktúr. Sú 2 možné riešenia:

1. Jeden uzol bude obsahovať prioritnú frontu, ostatné serveri budú k fronte pristupovať (odoberať a pridávať prvky). Najviac času sa trávi pri generovaní nových prvkov pomocou algoritmu *Deadbeat dad* viď sekcia 3.2.2. Týmto by sa záťaž výpočtu algoritmu rozdelila na viaceré uzly. Nové prvky po vygenerovaní budú zaradené do jednej spoločnej prioritnej fronty. Vo výsledku by úlohy už ale nemuseli byť priradené presne podľa zostupnej pravdepodobnosti.

2. Každý server bude obsahovať časť gramatiky a bude mať vlastnú prioritnú frontu. Jedná sa o podobný prístup ako je popísaný v sekcii 4.2. Problém nastáva ako pri prvom riešení, heslá nebudú generované presne podľa zostupnej pravdepodobnosti. Tu by záležalo ako efektívne by sa orezala gramatika, inak má prvé riešenie v tomto ohľade výhodu.

V oboch prípadoch je nutný uzol, ktorý sa bude starať buď o prioritnú frontu, alebo o rozdeľovanie gramatiky. V druhom prípade stačí len na začiatku rozdeliť gramatiku na pevné celky a priradiť serverom bez nutnosti hlavného uzlu, prišli by sme ale o možnosť dynamického pripájania serverov k výpočtu.



Obr. 6.6: Vertikálne a horizontálne škálovanie

Kapitola 7

Experimenty

V tejto kapitole budú vykonané experimenty s novým implementovaným distribuovaným generátorom, ktorý pre lámanie hesiel využíva nástroj Hashcat. Meranie prebieha na zariadeniach s nasledujúcimi parametrami:

- **CPU** - Intel Core i5-3570K,
- **GPU** - NVIDIA GTX 1050Ti,
- **RAM** - DDR3 8GB,
- **základná doska** - Intel DB75EN.

Bude sa merať najmä rýchlosť samotného generovania a škálovateľnosť pridávaním ďalších zariadení. Či zaznamenáme lineárnu škálovateľnosť. Následne sa porovná rýchlosť generovania na jednom uzle oproti serveru, ktorý bude generovať preterminálné štruktúry a 1 uzlom, ktorý bude z nich následne generovať terminály. A nakoniec porovnanie rýchlosti spolu s lámaním hesiel. Bude sa znova sledovať škálovateľnosť, ale najmä či bude rozdiel medzi generovaním a zasielaním preterminálnych štruktúr oproti riešeniu, kde budú posielané samotné heslá. Týmto bude simulované generovanie slovníka a jeho distribúcia.

7.1 Generovanie na jednom zariadení oproti serveru s klientským uzlom

V tomto experimente sa spustí generátor bez distribúcie na jednom zariadení, ktorý bude generovať zároveň preterminálne štruktúry ale aj výsledne terminály. Porovnávať sa bude s distribuovaným riešením pozostávajúci zo servera, ktorý generuje preterminálne štruktury a jedného klientského uzlu, ktorý z nich následne vygeneruje terminály. Experiment bol vykonaný na gramatike Rockyou-65 s obmedzením počtom vygenerovaných terminálov na 50 000 000. Server bol spustený s parametrom `-chunk-duration 5s`.

V tabuľke 7.1 sú uvedené výsledky. Ako sa dalo predpokladať distribuované riešenie je o niečo rýchlejšie. Na zvolenej gramatike došlo ku 35% zrýchleniu.

7.2 Škálovateľnosť generovania

Pri tomto experimente sa bude sledovať rýchlosť samotného generovania hesiel. Experiment bude uskutočnený na 1 serveri a postupne na 1, 2 až 4 klientských uzloch. Použitá je

	Čas (s)	Rýchlosť (hes/s)
1 zariadenie	34,46	1 450 593
Server + Klient	25,51	1 960 803

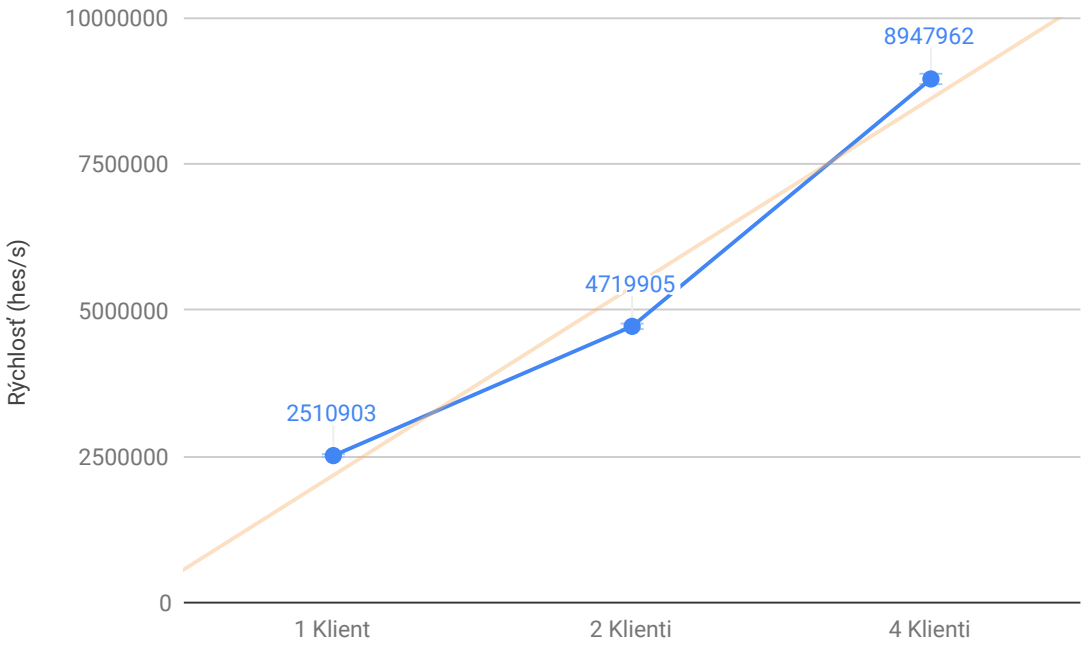
Tabuľka 7.1: Rýchlosť a čas lámanie generovanie hesiel gramatiky Rockyou-65

gramatika Rockyou-65 s obmedzením množstvom vygenerovaných terminálov na 500 000 000 (500M) a parametrom `-chunk-duration 5s`. V tabuľke 7.2 sú zobrazené výsledky experimenty.

	Čas (s)	Rýchlosť (hes/s)
1 Klient	200,74	2 510 903
2 Klienti	106,79	4 719 905
4 Klienti	56,33	8 947 962

Tabuľka 7.2: Rýchlosť generovania 500M hesiel

Na grafe 7.1 vidno takmer lineárnu škálovateľnosť vďaka pridávaním ďalších uzlov. Použitie 2 uzlov zlepšilo rýchlosť o 88% oproti jednému uzlu. Štyri uzly zaznamenali nárast rýchlosti o 89.6% oproti dvom a oproti jednému uzlu je to 3,56 násobne zrýchlenie.



Obr. 7.1: Škálovateľnosť

Tento experiment potvrdil škálovateľnosť distribúcie samotného generovania. Týmto prístupom sa dajú generovať slovníky z jednej gramatiky omnoho rýchlejšie, ako pri použití jedného uzlu. Z tohto vyplýva, že generovanie hesiel by nemalo brzdiť proces lámania hesiel ani pri vyššom počte klientských uzlov.

7.3 Lámanie hesiel

Pri tomto experimente sa bude sledovať rýchlosť lámania hesiel pomocou vygenerovaných hesiel z generátora. Ďalej sa bude sledovať škálovateľnosť. A nakoniec porovnanie rýchlosti pri generovaní preterminálnych štruktúr na serveri oproti generovania terminálov rovno na serveri a ich zasielanie na klientské uzly. Generovanie terminálov na serveri je simulácia naivného prístupu pri generovaní, kedy by sa na začiatku vytváral vopred slovník.

Nasledujúce testy boli vykonávane na gramatike Rockyou-65 a bcrypt heši [13]. Distribúcia bola spustená pokiaľ sa nepokúsilo prelomiť 502 327 hesiel. V tabuľke 7.3 sú výsledky 3 experimentov, kde sa lámali heslá na 1 serveri spolu s 1,2 a 4 klientskými uzlami. Ukazuje sa takmer lineárna škálovateľnosť. Dva klientské uzly sú o 1,95× výkonnejšie oproti jednému. Štyri uzly sú o 1,91× výkonnejšie oproti dvom uzlom a o 3,73× oproti jednému uzlu.

	Čas (s)	**Rýchlosť (hes/s)**
1 Klient	181.35	2769
2 Klienti	92.9	5407
4 Klienti	48.57	10342

Tabuľka 7.3: Rýchlosť a čas lámanie bcrypt hešu pri 502 327 heslách

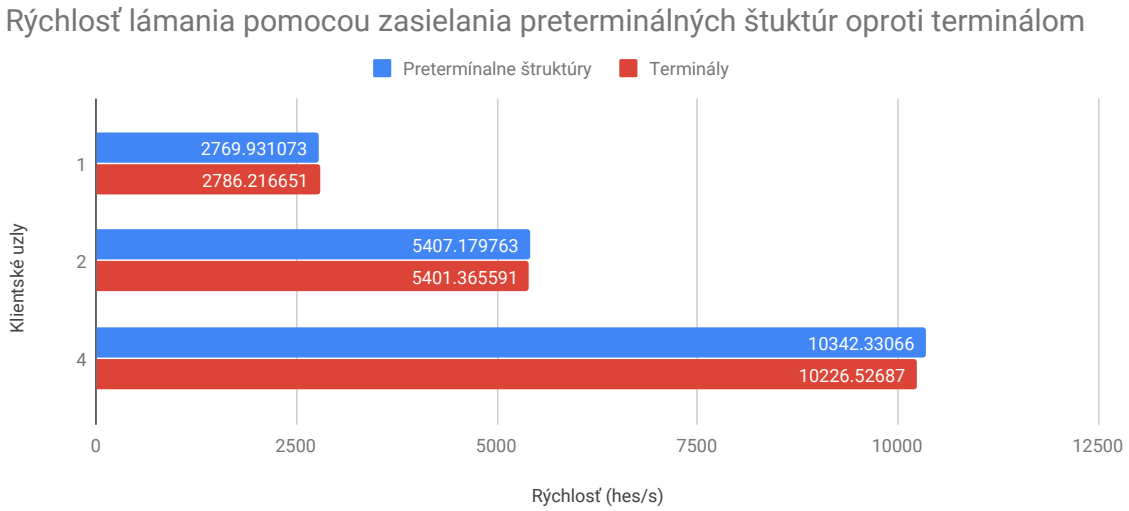
Následne sa spravil rovnaký experiment, kedy sa namiesto preterminálnych štruktúr posielali priamo terminály. Server teda generoval preterminálne štruktúry a z nich aj terminály, ktoré klientom zasielal. Výsledky vidno v tabuľke 7.4. Rýchlosti sú takmer totožné s predošlým experimentom, kde sa posielali preterminálne štruktúry. Porovnanie vidno na grafe 7.2. Rýchlosť lámania hešu je značne pomalšia oproti samotnému generovaniu hesiel a preto sa výsledky výrazne nelíšia. Server je schopný dostatočné rýchlo vygenerovať terminály pre klienta. A veľkosť správ odosielaní po sieti nie je tak obrovský rozličný, aby ovplyvnilo rýchlosť. Pri posielaní preterminálnych štruktúr sa po sieti odoslalo `401KB` a pri terminálnoch `4950kB`.

	Čas (s)	**Rýchlosť (hes/s)**
1 Klient	180,29	2786
2 Klienti	93	5401
4 Klienti	49,12	10226

Tabuľka 7.4: Rýchlosť a čas lámanie bcrypt hešu pri 502 327 heslách pri zasielaní terminálov

Rýchlosť distribúcie pomocou preterminálnych štruktúr sa oproti posielania terminálov síce nezlepšila. Ale stále má tento prístup výhody oproti naivnému generovaniu slovníka a jeho distribuovaniu. Pri naivnom riešení by sa musel slovník vopred vygenerovať. Otázka je ale aký veľký. Reálna netrénovaná gramatika je schopná vygenerovať gigabajtové slovníky. A generovanie takejto gramatiky väčšinou v reálnom čase ani neskončí. Je možné obmedziť počet vygenerovaných terminálov. To ale môže znamenať, že sa vygeneruje nedostatočne veľký slovník a klientské uzly by skončili proces lámania hesiel pred tým, ako by sa reálne heslo mohlo ďalej v gramatike vyskytovať. Pri vygenerovaní veľkého slovníka je zasa možné zbytočne mrhanie zdrojov zariadenia. Mohol by sa generovať slovník v rade hodinách a zaberať zbytočne veľa miesta na disku, ak by sa heslo vyskytovalo na začiatku gramatiky.

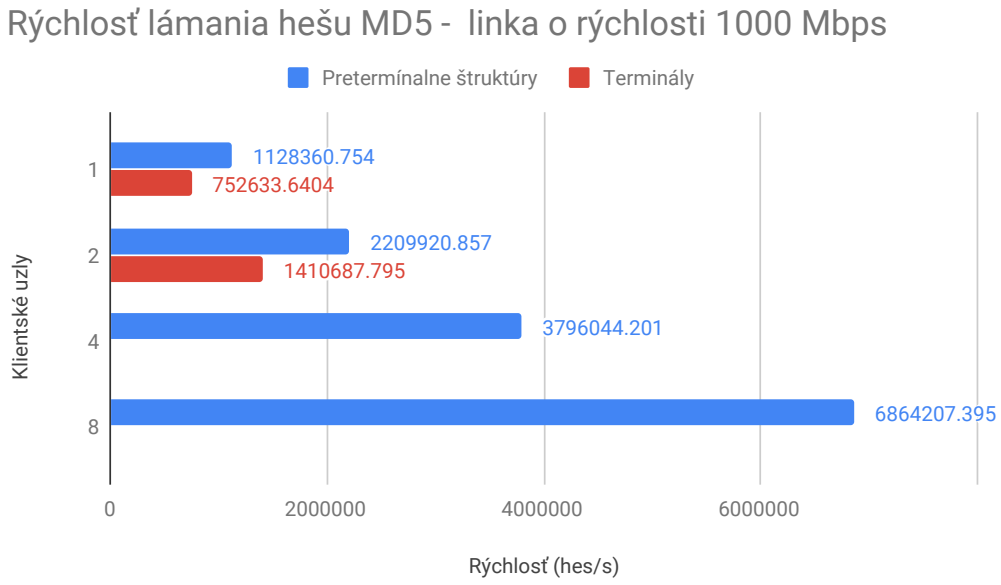
Výhoda generovania preterminálnych štruktúr je teda najmä vďaka generovaniu za chodu. Preterminálne štruktúry a aj terminály sa generujú len keď sú potrebné. Týmto prístupom sa zbytočne nemrhá výkon procesoru na generovanie.



Obr. 7.2: Rýchlosť lámania hesiel pri 500 000 heslách a hešu bcrypt

Rýchlejšie heše

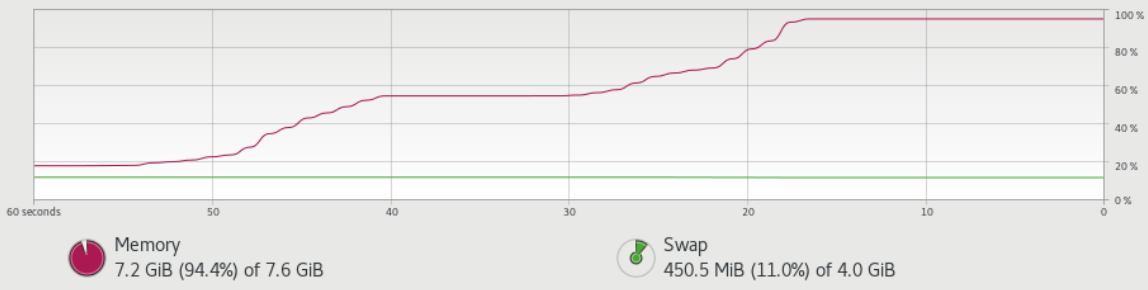
V ďalších experimentoch zvolíme rýchlejší heš pre prelomenie a to MD5 [14]. Gramatika zostane rovnaká a obmedzíme počet vygenerovaných hesiel na 500 000 000 (500M). Experimenty budú vykonávane na rôzne rýchlych linkách a to: 1000 Mbps, 100 Mbps a 10 Mbps.



Obr. 7.3: Rýchlosť lámania hesiel pri 500M heslách a MD5 hešu pri 1000 Mbps linke

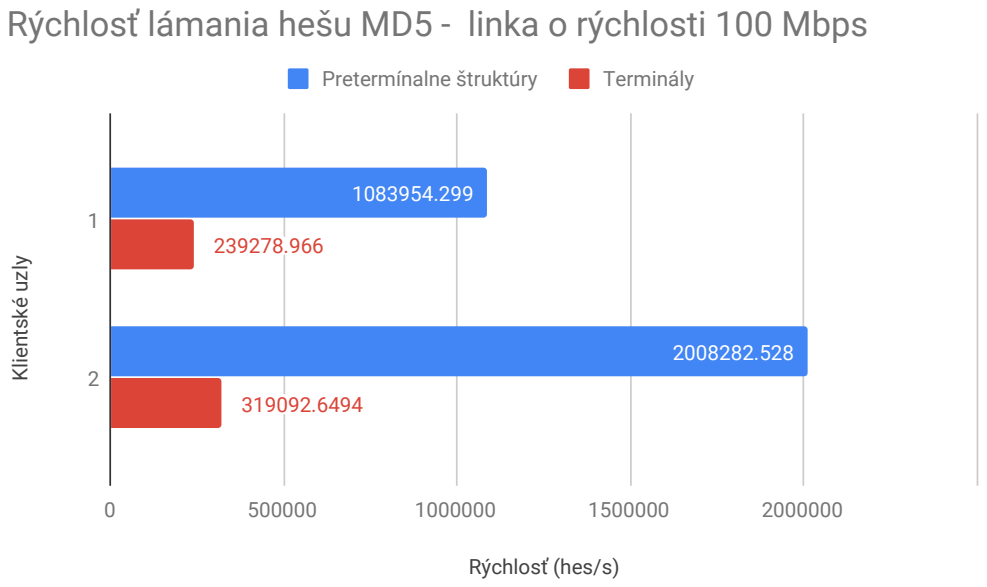
Na grafe 7.3 vidíme rýchlosti lámania prvého experimentu, v ktorom pre server bola použitá 1000Mbps linka. Pri jednom klientskom uzle je posielanie preterminálnych štruktúr `1.5x` rýchlejšie oproti posielaní terminálov. Pri dvoch uzloch je rýchlosť `1.57x` väčšia. Pri posielaní preterminalných štruktúr bolo po sieti prenesených 101.6MB dát a pri posielaní terminálov bolo prenesených 6136MB, čo je až 60 násobný rozdiel.

Na grafe ďalej vidno škálovateľnosť pri rozposielaní preterminálných štruktúr. Pridávaním ďalších klientov, zvyšuje rýchlosť celého procesu lámania hesiel. Pri rozposielaní terminálov pri väčšom počte klientských uzlov nastal problém s nedostatkom pamäti servera, ktorý následne zamrzol. Na obrázku 7.4 vidíme ako sa postupne zaplňovala pamäť servera. Pri poslaní 16. až 20. úlohy sa pamäť vyšplhala na 100 %, server zamrzol a proces lámania bol nasilu ukončený.



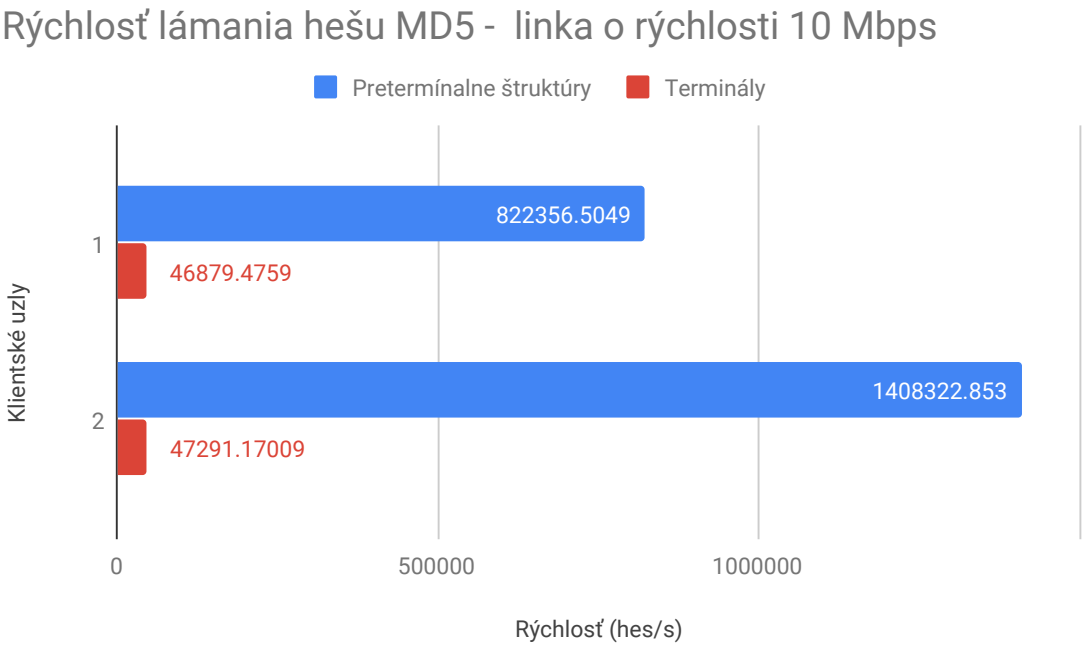
Obr. 7.4: Nedostatok pamäte servera pri posielaní terminálov štyrom uzlom

V ďalšom experimente obmedzíme linku servera na 100 Mbps. Na grafe 7.5 vidíme značný rozdiel v rýchlostiach. Posielanie preterminálnych štruktúr oproti terminálom je na 1 uzle 4,53 násobne rýchlejšie a pri 2 uzloch 6,3 násobne. Pridaním druhého uzlu pri posielaní terminálov sa zväčšila rýchlosť len 1,33 násobne. Oproti tomu pri preterminálnych štruktúrach je to 1,85 násobne zrýchlenie.



Obr. 7.5: Rýchlosť lámania hesiel pri 500M heslách a hešu MD5 pri 100 Mbps linke

V poslednom experimente je linka obmedzená na 10Mbps. Na grafe 7.6 vidno znateľný rozdiel rýchlosti v prístupoch rozposielania. Rozposielanie preterminálnych štruktúr je 17,5 násobne rýchlejšie na 1 uzle oproti posielaním terminálom a 29,78 násobne rýchlejšie pri 2 uzloch. Pridaním druhého uzlu pri posielaní terminálov výkon nezlepší. V tomto prípade je rýchlosť linky najväčšie obmedzenie. Nárast rýchlosti nebolo ani o 1%, oproti tomu posielanie preterminálnych štruktúr stále vykazuje škálovatelnosť aj na pomalšej linke.



Obr. 7.6: Rýchlosť lámania hesiel pri 500M heslách a hešu MD5 pri 10 Mbps linke

V tabuľke 7.5 sú časy jednotlivých experimentov pri lámaní MD5 heš a vygenerovaním 500M hesiel. PT značí rozposielanie preterminálnych štruktur a T označuje odosielanie terminálov. Na časoch vidno výhodu rozposielania preterminálnych štruktúr oproti terminálom pri lámaní slabšieho (rýchlejšieho) hešu. Aj pri rýchlej linke, tento prístup vykazuje lepšiu rýchlosť. Pri pomalších linkách je už rozdiel znateľný, kedy sa nevyplatí posielať samotné terminály.

		1000 Mbps	**100 Mbps**	**10 Mbps**
1 Klient	**PT**	7m 26,78s	7m 45s	10m 12,93s
	T	11m 9,70s	35m 6,49s	2h 59m 11,8s
2 Klienti	**PT**	3m 48,08s	4m 10,99s	5m 57,90s
	T	5m 57,34s	26m 19.6s	2h 57m 38,2s
4 Klienti	**PT**	2m 14,78s	2m 21,19s	4m 0,87s
8 Klientov	**PT**	1m 18,44s	1m 25,75s	3m 53,16s

Tabuľka 7.5: Čas lámania MD5 hešu skúšaním 500M hesiel

7.4 Zhodnotenie výsledkov

Na základe výsledkov z experimentov vieme zhodnotiť niekoľko vecí. Navrhnutý a implementovaný spôsob distribúcie vykazuje lineárnu škálovatelnosť. Pridávaním ďalších klientských uzlov sa zvyšuje rýchlosť generovania hesiel.

Implementovaný generátor funguje v spolupráci s nástrojom Hashcat, ktorému generátor poskytuje hesla na lámanie. Aj pri procese lámania hesla sa pridávaním klientských uzlov zvýšila rýchlosť samotného lámania. To znamená, že generátor hesiel nebrzdí proces.

Rozposielanie preterminálných štruktúr je výhodnejšie oproti posielaniu samotným terminálom. Po sieti je prenesené menšie množstvo dát. A server je odľahčený od generovania terminálov, ktoré si klientský uzol sám vygeneruje. Posielaním menšieho množstvo dát sa prejavilo najmä pri lámanie rýchlejších hešov, kde rýchlosť bola približne $1.5\times$ rýchlejšia.

Najväčší prínos sa ukázal pri pomalých linkách. Posielanie samotných terminálov brzdilo celý proces lámania hesiel len kvôli prenášaniu hesiel. Zdroje klientských uzlov neboli dostatočne využité, pretože klienti strávili väčšinu času čakaním na heslá. Tento problém pri posielaní preterminálnych štruktúr nebol až taký znateľný a naďalej vykazovalo lineárnu škalovatelnosť. Pri posielaní terminálov pridaním ďalšieho klientského uzlu na pomalej linke nenastalo znateľne zrýchlenie.

V experimentoch bola naivná distribúcia slovníkov simulovaná rozposielaním terminálov. Ale nie je to úplne to iste. Terminály z gramatiky nemusí byť možne v reálnom čase všetky vygenerovať. Výhoda implementovanej distribúcie oproti generovaniu slovníka vopred a jeho rozposielaní je najmä vďaka generovaniu terminálov za chodu. Terminály sú vygenerované až v momente, keď sú potrebné. Tým odpadá nutnosť si špecifikovať veľkosť slovníka, ktorý by sa mal vygenerovať vopred.

Kapitola 8

Záver

Použitie pravdepodobnostnej bezkontextovej gramatiky je ďalší z prístupov, ako generovať heslá oproti klasickým spôsobom ako je slovníkový útok alebo útok hrubou silou. Tento prístup funguje na základe trénovania nad existujúcim zoznamom hesiel, sledujúc vzory, ktoré užívateľ volí. Na základe toho zostrojí gramatiku, ktorá vygeneruje heslá.

Distribúcia takého typu útoku nie je taká jednoduchá, ako pri slovníkovom útoku, alebo útoku hrubou silou, pretože generovanie hesiel nie je možné rozdeliť na rovnomerné časti, ale dá sa k tomu priblížiť. Existujúce riešenie, ktoré využíva len jedno zariadenie, generuje heslá ako je popísané v kapitole 3, ale rýchlosť nie je dostatočná pre praktické použitie. Generátor bol prepísaný do jazyka Go a následne vykonané optimalizácie, vďaka čomu sa dosiahlo 8 až 40 násobné zrýchlenie.

V práci boli navrhnuté dve metódy ako distribuovať generovanie hesiel na základe pravdepodobnostnej gramatiky. Experimentálne bola zvolená najlepšia metóda, ktorá je schopná efektívne využiť každé výpočtové zariadenie v sieti a následne bola implementovaná. Experimenty ukázali lineárnu škálovateľnosť pridávaním ďalších klientských uzlov.

Implementovaný distribuovaný generátor bol prepojený s nástrojom Hashcat pre podporu lámania hesiel. V rámci experimentov bol ukázaný prínos posielania preterminálnych štruktúr z gramatiky. Tento prístup je oproti posielaní samotných hesiel rýchlejší najmä pri lámaní hešov, ktoré sú rýchle na prelomenie. Rýchlejšie je to najmä vďaka menšej dátovej vyťaženosti sieťovej linky a aj vďaka prevedením práce generovania terminálov zo serveru na klientské uzly.

Ďalšou výhodou mimo rýchlejšieho generovania a lámania hesiel je generovanie hesiel za chodu. Heslá sú vygenerované až keď sú potrebné a neplytvá sa výkonom procesoru. Pri slovníkovom útoku by bolo nutné najprv slovník vygenerovať a následne distribuovať. Zvyčajná gramatika je ale schopná vygenerovať nespočetné množstvo hesiel. Generovanie by bolo nutné po nejakom čase zastaviť. Pri posielaní preterminálnych štruktúr tento problém nenastáva.

Do budúcna by bolo vhodné navrhnúť spôsob distribúcie generovania preterminálnych štruktúr, ktoré môže brzdiť generovanie hesiel pri veľkom počte klientských uzlov. V implementácii nie je vyriešená autorizácia klientov, aj keď bolo navrhnutých niekoľko riešení. Strata úlohy od klienta je len z časti vyriešená.

Literatúra

[1] Chuah, C. W.; Dawson, E.; Simpson, L.: Key Derivation Function: The SCKDF Scheme. In *Security and Privacy Protection in Information Processing Systems*, editácia L. J. Janczewski; H. B. Wolfe; S. Shenoi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-39218-4, s. 125–138.

[2] Donovan, A.: *The Go programming language*. New York: Addison-Wesley, 2015, ISBN 978-0134190440.

[3] Dutta, S.; Gera, S.; Verma, A.; aj.: SmartScale: Automatic Application Scaling in Enterprise Clouds. In *2012 IEEE Fifth International Conference on Cloud Computing*, June 2012, ISSN 2159-6190, s. 221–228, doi:10.1109/CLOUD.2012.12.

[4] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. Technická Správa 2616, Jún 1999, obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.
URL http://www.ietf.org/rfc/rfc2616.txt

[5] Garfinkel, S. L.: Digital forensics research: The next 10 years. *Digital Investigation*, ročník 7, 2010: s. S64 – S73, ISSN 1742-2876, doi:https://doi.org/10.1016/j.diin.2010.05.009, the Proceedings of the Tenth Annual DFRWS Conference.
URL http://www.sciencedirect.com/science/article/pii/S1742287610000368

[6] Hranický, R.; Zobal, L.; Večeřa, V.: Distribuovaná obnova hesel. Technická správa, FIT-TR-2017-04, CZ, 2017.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11568

[7] Hranický, R.; Zobal, L.; Večeřa, V.; aj.: Distributed Password Cracking in a Hybrid Environment. In *Proceedings of SPI 2017*, University of defence in Brno, 2017, ISBN 978-80-7231-414-0, s. 75–90.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11358

[8] Hranický, R.; Zobal, L.; Večeřa, V.; aj.: Distribuce výpočtů pro nástroj hashcat. Technická správa, FIT-TR-2018-04, Brno, CZ, 2018.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11884

[9] Juels, A.; Rivest, R. L.: Honeywords: Making Password-cracking Detectable. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-2477-9, s. 145–160, doi:10.1145/2508859.2516671.
URL http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/2508859.2516671

[10] Narayanan, A.; Shmatikov, V.: Fast Dictionary Attacks on Passwords Using Time-space Tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-226-7, s. 364–372, doi:10.1145/1102120.1102168.
URL http://doi.acm.org/10.1145/1102120.1102168

[11] Oluwatosin, H. S.: Client-server model. *IOSRJ Comput. Eng*, ročník 16, č. 1, 2014: s. 2278–8727.

[12] PRENEEL, B.: *Analysis and Design of Cryptographic Hash Functions*. Dizertačná práca, Citeseer, 2003.

[13] Provos, N.; Mazieres, D.: A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, 1999, s. 81–91.

[14] Rivest, R.: The MD5 Message-Digest Algorithm. Technická Správa 1321, Apríl 1992, updated by RFC 6151.
URL http://www.ietf.org/rfc/rfc1321.txt

[15] Tansey, W.: Improved models for password guessing. *University of Texas, Tech. Rep*, 2011.

[16] Weir, C. M.: *Using probabilistic techniques to aid in password cracking attacks*. Dizertačná práca, The Florida State University, 2010.

[17] Weir, M.; Aggarwal, S.; de Medeiros, B.; aj.: Password Cracking Using Probabilistic Context-Free Grammars. 05 2009, s. 391–405, doi:10.1109/SP.2009.8.

[18] Češka, M.; Vojnar, T.; Smrčka, A.; aj.: Teoretická informatika - Studijní text. 2018, str. 53.
URL https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf

Príloha A

Obsah přiloženého paměťového média

Na médiu sa nachádzaju nasledujúce zložky:

- `tex` - obsahuje zdrojové súbory tejto práce napísane v Latexu
- `src` - zdrojové súbory distribuovaného generátora implementovaného v jazyku Go