Lámání hesel pomocí pravděpodobnostních gramatik

Technická zpráva FIT VUT v Brně

Radek Hranický, Lukáš Zobal, Dávid Mikuš

Technická zpráva č. FIT-TR-2019-03
Fakulta informačních technologií, Vysoké učení technické v Brně

Last modified: 14. července 2020

Lámání hesel pomocí pravděpodobnostních gramatik

Radek Hranický, Lukáš Zobal, Dávid Mikuš

Vysoké učení technické v Brně, email: ihranicky,izobal@fit.vutbr.cz, xmikus15@stud.fit.vutbr.cz

Abstrakt Pravděpodobnostní bezkontextové gramatiky jsou matematickým modelem, který lze použít pro popis znalostí o syntaxi existujících uživatelských hesel. Při lámání hesel pro přístup k zabezpečenému obsahu můžeme takovýchto znalostí využít k lepšímu cílení útoku. Tato technická zpráva vysvětluje metody použití gramatik pro popis struktury hesel. Dále vysvětluje možnosti generování hesel sekvenčně, paralalně a distribuovaně.

1 Úvod

Při tvorbě hesel pro přístup k zabezpečenému obsahu uživatelé často volí hesla, která jsou jednoduché k zapamatování [2]. Mnohé služby a systémy dnes již vynucují pravidla pro konkrétní podobu hesel, typu "použij alespoň jeden speciální znak", apod. Cílem takových přístupů je vést uživatele k tvorbě silnějších hesel [14,16]. Bezpečnostní incidenty v podobě rozsáhlých úniků uživatelských hesel v posledních letech však ukázaly, že uživatelé často tvoří hesla z existujících slov [4] a mnohdy používají stejné heslo napříč různými službami [3].

Tradiční způsoby lámání hesel představují útok hrubou silou a slovníkový útok. Při útoku *hrubou silou* se snažíme vyzkoušet všechny možné sekvence znaků nad konkrétní abecedou. Nevýhodou je obrovské množství kombinací, které navíc roste exponenciálně s délkou hesla. Mnohdy proto není výpočetně zvládnutelné vyzkoušet všechny možnosti v přijatelném čase. Při *slovníkovém útoku* zkoušíme všechna hesla obsažená v předem určeném textovém souboru, tzv. slovníku. Takový slovník má vždy konečnou velikost a typicky pokrývá pouze omezenou část možností. Při existujících znalostech o uživatelských heslech však může výrazným způsobem pomoci zapojení principů z oblasti pravděpodobnosti a statistiky [13,18,12].

Jednou z možností je použití *Markovových řetězců*, kterými můžeme modelovat podmíněnou pravděpodobnost $P(A|B)$, že znak A bude v hesle následovat bezprostředně po znaku B [13]. Pravděpodobnosti pro všechny znaky A, B můžeme uložit do matice získané na základě analýzy existujícího slovníku hesel. Markovovy řetězce pro útok hrubou silou používá např. nástroj *hashcat*[1], který zavádí mimojiné tzv. *masky* hesel, které definují, na jaké pozici se může nacházet

[1] https://hashcat.net/

který znak. Tento přístup však pracuje pouze se znaky samotnými a neuvažuje větší celky jako digramy, trigramy, aj.

Pro práci s většími fragmenty hesel ukazují Weir a kol. možnost použití *pravděpodobnostních bezkontextových gramatik*, angl. probabilistic context-free grammars (PCFG) [18]. Uvažujme slovník existujících hesel získaný např. na základě některého ze zmíněných bezpečnostních úniků. Pomocí gramatiky můžeme popsat strukturu jednotlivých hesel na úrovni sekvencí písmen, číslic a speciálních znaků. V gramatice tyto fragmenty modelují přepisovací pravidla, kde každé má přiřazenu pravděpodobnostní hodnotu odpovídající četnosti výskytů fragmentu v původním slovníku. Vytvořená gramatika pak umožňuje vytvořit nejen všechna hesla obsažená v původním slovníku, ale také řadu nových, která dosud neexistovala, ale jejich podoba stále zohledňuje uživatelské zvyklosti při tvorbě hesel. Princip PCFG umožňuje každému vygenerovanému heslu přiřadit pravděpodobnostní hodnotu, díky čemuž jsme schopni vygenerovat např. "1000 nejpravděpodobnějších hesel". Při vyzkoušení menšího počtu hesel tak tento způsob umožňuje dosáhonut vyšší úspěšnosti [18,17,6].

Tato technická zpráva popisuje techniky použití PCFG pro popis znalostí o uživatelských hesel a ukazuje možnosti generování nových kandidátů hesel sekvenčně, paralelně a distribuovaně.

2 Pravděpodobnostní bezkontextové gramatiky

Matematický model vychází z klasických *bezkontextových gramatik* [5], s tím rozdílem, že u PCFG je každému přepisovacímu pravidlu přiřazena pravděpodobnostní hodnota [1,11]. *Pravděpodobnostní bezkontextovou gramatiku* proto definujeme jako pětici:

$$G = (N, \Sigma, R, S, P), \tag{1}$$

kde:

- N je konečná množina nonterminálů,
- Σ je konečná množina terminálů, $N \cap \Sigma = \emptyset$,
- R je konečná množina přepisovacích pravidel ve tvaru $A \to \gamma$, pro která platí: $A \in N, \gamma \in (N \cup \Sigma)^*$,
- $S \in N$ je počáteční (startovací) symbol gramatiky,
- P je funkce pravděpodobnosti definována jako $P : R \to [0, 1]$, tj. každému přepisovacímu pravidlu je přiřazena pravděpodobnost v intervalu od 0 do 1. Dále platí, že součet pravděpodobností všech pravidel $A \to \gamma \in R$ se stejnou levou stranou A je roven 1. Tedy: $\forall A \in N, \sum_{A \to \gamma \in R}(P(A \to \gamma)) = 1$.

Gramatika se nazývá *pravděpodobnostní* díky pravděpodobnostní funkci P přiřazujícímu každému pravidlu pravděpodobnostní hodnotu. Gramatika se nazývá *bezkontextová*, neboť substituci pravé strany γ pravidla za nonterminál A lze provést bez ohledu na kontext, ve kterém je nonterminál A uložen.

2.1 Tvorba gramatiky

Gramatiku tvoříme na základě zpracování hesel z existujícího ("trénovacího") slovníku. Weir a kol. v původním návrhu uvažují rozdělení hesel na fragmenty v podobě souvislých posloupností písmen - letters (L), čísel - digits (D) a speciálních znaků (S) [18,17]. V pozdějších verzích nástroje *PCFG Cracker*[2], který tvorbu gramatiky i generování hesel implementuje, došlo k úpravě notace: písmena - alpha (A), čísla - digits (D), ostatní - others (O). Gramatika je ze slovníku tvořena následovně:

- Jednotlivá hesla rozdělíme na fragmenty typu A, D, či O různých délek. Každý fragment ukládáme pouze jednou, tzn. pokud jsme jej již dříve vytvořili v rámci aktuálního či předchozího hesla, nebudeme jej vytvářet podruhé.
- Pro každý fragment délky n sestrojíme přepisovací pravidlo $T_n \to f$, kde $T \in \{A, D, O\}$ je typ sady znaků, f je fragment samotný.
- Pravděpodobnost $P(T_n \to f)$ pravidla $T_n \to f$ bude určíme jako:

$$P(T_n \to f) = \frac{c_f}{c_{T_n}}, \tag{2}$$

  kde c_f je celkový počet výskytů fragmentu f ve slovníku a c_{T_n} je celkový počet výskytů všech fragmentů typu T a délky n [17].
- Pro každé heslo ze vstupního slovníku vytvoříme tzv. *bázovou strukturu* (base structure) B jako posloupnost nonterminálů $T^1_{n^1} \dots T^k_{n^k}$, kde k je počet nonterminálů v této struktuře. Bázová struktura je tedy posloupnost nonterminálů T_n popisující, z jakých fragmentů (typů a délek) se heslo skládá. Tedy např. pro heslo "hello!44mike" vznikne bázová struktura "$A_5O_1D_2A_4$", neboť heslo začíná pěti písmeny (A_5), následuje znak vykřičníku (O_1), dvě čísla (D_2) a nakonec 4 písmena (A_4).
- Pro každou bázovou strukturu B sestrojíme přepisovací pravidlo: $S \to B$, pokud toto pravidlo již v gramatice neexistuje.
- Pravděpodobnost $P(S \to B)$ určíme jako:

$$P(S \to B) = \frac{c_B}{c_{T_B}}, \tag{3}$$

  kde c_B je celkový počet výskytů hesel odpovídající bázové struktuře B a c_{T_B} všech bázových struktur [17].

Tabulka 1 ukazuje přepisovací pravidla gramatiky vytvoření na základě dvou hesel: "pass!word" a "love@love". Pro přepis nonterminálu S existuje pouze jedno pravidlo, neboť obě hesla jsou popsány stejnou bázovou strukturou $L_4S_1L_4$.

[2] https://github.com/lakiw/legacy-pcfg/tree/master/python_pcfg_cracker_version3

levá strana	→	pravá strana	pravděpodobnost
S	→	$L_4 S_1 L_4$	1
L_4	→	pass	0.25
L_4	→	word	0.25
L_4	→	love	0.5
S_1	→	@	0.5
S_1	→	!	0.5

Tabulka 1: Příklad přepisovacích pravidel v PCFG

2.2 Kapitalizace písmen

Způsob tvorby gramatiky popsaný v sekci 2.1 nijak neřeší rozdíly mezi malými a velkými písmeny. Analýzou existujících hesel jsme však zjistili, že není-li použití velkých písmen vyžadováno, uživatelé nejčastěji tvoří hesla pouze z písmen malých [9]. Jsou-li již velká písmena použita, pak nejčastěji na začátku slova či sousloví, např. hesla "Golf-Mike" či "HelloKitty!" z datové sady RockYou[3]. Tuto skutečnost můžeme využít při generování nových kandidátů hesel a s použitím pravděpodobnosti měnit velikost písmen ve fragmentech získaných z původního slovníku.

Weir a kol. proto dosavadní způsob upravují zavedením tzv. masek kapitalizace [17]. U přepisovacích pravidel pro fragmenty písmen, tj. $A_n \rightarrow f$, jsou u samotného fragmentu f písmena reprezentována jako malá. Pro každou délku n pak definuje jednu či více masek kapitalizace, přičemž každé přísluší určitá hodnota pravděpodobnosti. *Maska kapitalizace* M_n pro A_n je řetězec délky n složený z znaků U a L, udávající, na kterých pozicích ve slově se budou nacházet velká a malá písmena. Je-li na pozici p v masce znak U (uppercase), znamená to, že p-té písmeno v odpovídajícím slově bude velké. Je-li na pozici p v masce znak L (lowercase), znamená to, že p-té písmeno v odpovídajícím slově bude malé. Pravděpodobnostní hodnotu $P(M_n)$ masky M_n určíme jako:

$$P(M_n) = \frac{c_{M_n}}{c_{A_n}}, \tag{4}$$

kde c_{M_n} je počet všech výskytů písmenných fragmentů odpovídajících masce M_n a c_{A_n} je celkový počet výskytů všech písmenných fragmentů délky n. Součet pravděpodobností všech M_n délky n je roven 1. Tabulka 2 ukazuje příklad masek kapitalizace a jejich pravděpodobností pro A_5. U každé masky je uvedena také zkrácená zápisu masky, která je využívána v některé literatuře [17].

V nástroji PCFG Cracker jsou při tvorbě hesel nejprve vygenerovány fragmenty složené z písmen a až následně je "aplikována" kapitalizace. Z matematického pohledu však chceme u gramatiky zachovat definici bezkontextovosti [5]. Oproti původnímu přístupu tedy musíme každý fragment písmen délky n chápat jako jeden nonterminál. Pro každou masku M_n pak uvažujeme přepisovací pravidlo s pravděpodobností $P(M_n)$, které tento nonterminál přepíše na terminální

[3] http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2

maska kapitalizace	zkrácená verze	pravděpodobnost
LLLLL	L_5	0.928421
UUUUU	U_5	0.041223
ULLLL	U_1L_4	0.021047
UULLL	U_2L_3	0.006215
ULULU	$U_1L_1U_1L_1U_1$	0.003094

Tabulka 2: Příklad pravidel pro kapitalizaci písmen ve fragmentech délky 5

sekvenci sekvenci písmen o velikosti dané maskou kapitalizace. Celkový počet přepisovacích pravidel v gramatice nám tedy vzroste, ale získáme možnost tvořit dosud neexistující sekvence písmen při zachování uživatelských zvyklostí v psaní velkých a malých písmen.

3 Sekvenční generování hesel

Při útoku na zabezpečený obsah pomocí pravděpodobnostních bezkontextových gramatik využíváme gramatiku jako prostředek ke generování nových kandidátů hesel. *Kandidát hesla* (candidate password, též password guess) je řetězec generovaný gramatikou, který následně slouží jako vstup kryptografické hešovací funkce, na jejíž heš útočíme, popř. jako vstup funkce pro derivaci šifrovacího klíče, který chceme vyzkoušet. Pro pravděpodobnostní bezkontextovou gramatiku G množina všech kandidátů hesel odpovídá jazyku generovanému gramatikou [5]:

$$L(G) = \{w \mid S \Rightarrow^* w \wedge w \in \Sigma^*\}. \tag{5}$$

Každé heslo $p \in L(G)$ tedy vznikne ze startovacího nonterminálu S nějakou sekvencí derivačních kroků:

$$S \Rightarrow ... \Rightarrow p \tag{6}$$

při aplikaci konečné posloupnosti přepisovacích pravidel $r_1, r_2, \ldots, r_n \in R$. Nechť $R_p = \{r_1, r_2, \ldots, r_n\} \subseteq R$ je množina přepisovacích pravidel použitých k vygenerování hesla p. Pak pravděpodobnost $P(p)$ hesla p vypočítáme jako:

$$P(p) = \prod_{r \in R_p} P(r). \tag{7}$$

Pravděpodobnost hesla je tedy rovna součinu pravděpodobností všech přepisovacích pravidel použitých k jeho vytvoření.

Při použití PCFG vytvořených na základě větších slovníků mnohdy není výpočetně zvládnutelné vygenerovat a ověřit všechny kandidáty hesel [8]. Útok

tedy typicky omezujeme na počet n nejpravděpodobnějšího hesel. Hledáme tedy algoritmus, kterým taková hesla získáme.

Naivním způsobem je vygenerovat všechna přípustná hesla spolu s jejich pravděpodobnostmi, výsledek seřadit dle pravděpodobnosti a vybrat právě n hesel. Byť je tento způsob funkční, naráží opět na problém, který by měl řešit.

3.1 Preterminální struktury

Při provádění derivačních kroků ze startovacího nonterminálu dříve či později sestavíme větnou formu, u které další rozhodnutí o volbě přepisovacích pravidel neovlivní pravděpodobnost generovaného řetězce. Taková větná forma může být již samotným terminálním řetězcem, nebo na každý její nonterminál můžeme aplikovat pouze pravidla se stejnou pravděpodobností.

Definice 1 (Preterminální struktura) *Preterminální struktura je větná forma, ze které všechny přípustné posloupnosti derivačních kroků vytvoří řetězce se stejnou pravděpodobností.*

Pro každou preterminální strukturu jsme schopni jednoznačně určit pravděpodobnost všech hesel, která z ní mohou být vygenerována. Této skutečnosti můžeme využít a pomocí derivačních kroků tedy nejprve tvořit preterminální struktury a následně z nich generovat hesla samotná.

Nejjednodušším způsobem je vygenerovat nejprve všechny preterminální struktury, tyto seřadit dle pravděpodobnosti výsledných hesel a následně vybrat n nejpravděpodobnějších. Tento přístup však opět vyžaduje provedení značného množství výpočetních operací a zpracování velkého množství dat ještě než je možné vygenerovat první heslo. Generování preterminálních struktur a hesel samotných pak také není možné realizovat souběžně.

Dalším přístupem je využít techniku, kterou navrhli Narayanan a kol., kdy vygenerujeme pouze preterminální struktury s pravděpodobností nad stanovený limit. Podobný způsob také používá nástroj John the Ripper[4] v režimu "Markov". Tento způsob nicméně nezaručuje, že výsledná hesla budou vygenerována v pořadí dle pravděpodobnosti. Algoritmus pouze zajistí, že pravděpodobnost všech vygenerovaných kandidátů bude vyšší než stanovený limit, zatímco jejich pořadí není definováno [13].

Alternativou může být použití algoritmu *depth-first search*, tedy prohledávání do hloubky. Zde je však nutné nakonec projít všechny uzly [15]. S ohledem na velikosti derivačních stromů u PCFG vytvořených z reálných úniků hesel a počet operací *backtracking*, které by bylo nutno provést, se však tato metoda nejeví jako v praxi použitelná [17].

3.2 Pravděpodobnostní skupiny

Při tvorbě gramatiky z existujícího slovníku hesel vzniká množství přepisovacích pravidel, která tvoří množinu R. S výjimkou pravidel pro tvorbu bázových struktur, jsme z nonterminálu T_n na levé straně pravidla $r = T_n \rightarrow \gamma$ schopni určit

[4] https://www.openwall.com/john/

jeho typ T, tedy, zda bude tvořit fragmenty písmen (A), čísel (D), či ostatních znaků (O) a také délku n vytvořených fragmentů. Při zkoumání PCFG vytvořených na větších slovnících záhy zjistíme, že množství pravidel r se stejnou levou stranou T_n má stejnou pravděpodobnost $P(r)$.

Definice 2 (Skupina preterminálních pravidel) *Nechť skupina preterminálních pravidel $R_{T_n} \in R$ je množina všech přepisovacích pravidel $T_n \to \gamma$ se stejnou levou stranou T_n, kde $\gamma \in \Sigma^*$, tj. pravá strana obsahuje pouze terminální symboly.*

Definice 3 (Pravděpodobnostní skupina) *Nechť pravděpodobnostní skupina $R^p_{T_n} \subseteq R_{T_n}$ je skupina preterminálních pravidel, kde všechna mají stejnou pravděpodobnost p.*

levá strana	→	pravá strana	pravděpodobnost	skupina
A_{14}	→	siempreteamare	0.002379693	$R^{0.002379693}_{A_{14}}$
A_{14}	→	backstreetboy	0.002379693	
A_{14}	→	elamordemivida	0.002379693	
A_{14}	→	threedaysgrace	0.001322052	$R^{0.001322052}_{A_{14}}$
A_{14}	→	paralelepipedo	0.001322052	
A_{14}	→	glasgowrangers	0.001322052	
A_{14}	→	loveisintheair	0.001322052	
A_{14}	→	showmethemoney	0.001322052	
A_{14}	→	lordoftherings	0.001057641	$R^{0.001057641}_{A_{14}}$
A_{14}	→	iloveyousomuch	0.001057641	
A_{14}	→	jessemccartney	0.001057641	
A_{14}	→	ilovechocolate	0.001057641	
...	...	...	...	...

Tabulka 3: Příklad pravděpodobnostních skupin pravidel

Tabulka 3 ukazuje příklad sady přepisovacích pravidel pro A_{14}. Vidíme, že přepisovací pravidla tvoří skupiny se stejnou pravděpodobností. Pro jednoduchost zde neuvažujeme zavedení kapitalizace. Právě existence pravděpodobnostních skupin umožňuje efektivní běh algoritmu Next, který bude představen dále.

3.3 Funkce Next

Weir a kol. navrhli funkci *Next*, která na základě PCFG postupně vytváří preterminální struktury, ze kterých je možno generovat všechna možná hesla od nejpravděpodobnějšího po nejméně pravděpodobné. Její implementace používá prioritní frontu, kde prioritu určuje právě pravděpodobnost. Tato fronta slouží k ukládání jednotlivých preterminálních struktur, přičemž její implementace zajišťuje, že jsou zde řazeny dle pravděpodobnosti. Metoda *pop()* tedy z fronty

vybere vždy strukturu s nejvyšší pravděpodobností. V prvku prioritní fronty je kromě samotné preterminální struktury uložena také její pravděpodobnost, dále pak bázová struktura (viz sekci 2.1), ze které vznikla, velikost bázové struktury v podobě počtu po sobě jdoucích nonterminálů typu A, D, O a nakonec hodnotu tzv. pivotu (*pivot value*). Pivot zajišťuje, že nikdy nevygenerujeme dvě totožné preterminální struktury a že každému vytvořenému heslu bude odpovídat právě jeden derivační strom.

Vstupem algoritmu je gramatika v takové reprezentaci, že přepisovací pravidla pro nonterminály typu A, D, O jsou seřazena v pořadí od nejpravděpodobnějších po nejméně pravděpodobné. Algoritmus vytváří preterminální struktury tak, že postupně přepisuje nonterminály bázové struktury pomocí přepisovacích pravidel v pořadí od nejpravděpodobnějších po nejméně pravděpodobné. Je-li k dispozici více pravidel se stejnou pravděpodobností, je zde "dosazena" celá pravděpodobnostní skupina (viz sekci 3.2) - přesněji, je naznačeno, že ve fázi generování hesel budou pro přepis daného nonterminálu použita všechna pravidla z odpovídající pravděpodobnostní skupiny. Pro účely demonstrace toto budeme ilustrovat naznačením terminálů, které je možné dosadit, tj. např. místo $4A_3\$\$$ zapíšeme $4\{jan, pes\}\$\$$, čímž rozhodneme, že nonterminál A_3 bude při generování postupně přepsán právě na terminály "jan" a "pes".

Princip fungování funkce Next ukazuje algoritmus 1. Identifikátorem PT označujme preterminální stukturu. Algoritmus postupně vkládá prvky s PT do fronty, vybírá je a tvoří z nich nové - tak, že na původní bázovou strukturu aplikuje jiná, méně pravděpodobná přepisovací pravidla. Algoritmus dále používá tři pomocné funkce:

- *spocitej_pravdepodobnost(PT)* vypočítá celkovou pravděpodobnostní hodnotu dané preterminální struktury vynásobením pravděpodobností všech použitých přepisovacích pravidel;
- *generuj_hesla(PT)* slouží k vygenerování všech hesel z PT;
- *dekrementuj(PT, i)* pak z dané PT vytvoří novou způsobem, kdy na pozici i aplikuje další (méně pravděpodobné) přepisovací pravidlo, popř. naznačí použití pravidel z další (méně pravděpodobné) skupiny.

Chování algoritmu Next ilustrujeme na příkladu. Uvažujme PCFG s přepisovacími pravidly, které obsahuje tabulka tabulka 4. Algoritmus nejprve pro každou bázovou strukturu z gramatiky vytvoří nejpravděpodobnější PT a tuto vloží do fronty spolu s její velikostí, pravděpodobností a hodnotou pivotu nastavenou na 0. V prvotní fázi bude tedy fronta obsahovat tolik prvků, kolik je v gramatice bázových struktur. Gramatika v příkladu obsahuje dvě bázové struktury, proto v této fázi bude mít prioritní fronta právě dva prvky, jak ilustruje tabulka 1. Protože přepisovací pravidla $A_3 \rightarrow jan$ a $A_3 \rightarrow pes$ mají stejnou pravděpodobnost - je zde k nahrazení použita celá pravděpodobnostní skupina $\{jan, pes\}$. Finální přepsání provede až funkce *generuj_hesla()*.

Jednotlivé pozice nonterminálů v každé bázové struktuře označíme zleva indexy od 0. Pro $A_3D_1O_1$, bude nonterminál A_3 mít index 0, D_1 index 1 a O_1 index 2. Následuje fáze, kdy postupně vybíráme prvky z fronty a vytváříme z nich nové aplikováním dosud nepoužitých (skupin) přepisovacích pravidel na

Algoritmus 1: Algoritmus funkce Next [17]

```
Data: fronta, PCFG
// Pro každou bázovou strukturu base chceme nejpravděpodobnější PT
foreach base in PCFG do
    prvek.struktura = PT z base, která má nejvyšší pravděpodobnost
    prvek.pivot = 0
    prvek.num_strings = velikost base
    prvek.p = spocitej_pravdepodobnost(prvek.struktura)
    push(fronta, prvek)
prvek = pop(fronta)
while prvek != NULL do
    generuj_hesla(prvek) // všechna hesla z dané PT
    for i = prvek.pivot; i < prvek.num_strings; i++ do
        // Na pozici i pouzij dalsi pravidlo
        novy.struktura = dekrementuj(prvek.struktura, i)
        if novy.struktura != NULL then
            novy.pivot = i
            novy.p = spocitej_pravdepodobnost(novy.struktura)
            novy.num_strings = prvek.num_strings
            push(fronta, novy)
    prvek = pop(fronta)
```

nonterminály v odpovídající bázové struktuře. Vždy však nahrazujeme pouze ty nonterminály, jejichž index pozice je vyšší než hodnota pivotu.

Uvažujme příklady fronty dle tabulky 1. Při prvním volání operace *pop()* je vybrán prvek s nejvyšší pravděpodobností, tedy $4\{jan, pes\}\$\$$. Z tohoto se vygenerují hesla "4jan$$" a "4pes$$". Dále se vytvoří dva nové prvky s hodnotami pivotu 0 a 2 a vloží se do fronty. Obsah fronty po této operaci znázorňuje tabulka 2. V bázové struktuře došlo k použití odlišných přepisovacích pravidel pro D_1 a O_2. Protože pravidla $D_1 \rightarrow 5$ a $D_1 \rightarrow 6$ mají stejnou pravděpodobnost, opět byla použita celá pravděpodobnostní skupina. Proces pokračuje dále dle algoritmu 1, dokud nejsou všechny prvky vyčerpány, nebo není nalezeno správné heslo.

bázová struktura	**preterminální struktura**	**pravděpodobnost**	**pivot**
$D_1A_3O_2$	$4\{jan, pes\}\$\$$	0.1575	0
$A_3D_1O_1$	$\{jan, pes\}4!$	0.04875	0

Obrázek 1: Příklad prioritní fronty na začátku výpočtu

Weir a kol. provedli důkaz korektnosti algoritmu Next [18], který potvrzuje, že:

levá strana	→	pravá strana	pravděpodobnost
S	→	$D_1A_3O_2D_1$	0.75
S	→	$A_3D_1O_1$	0.25
A_3	→	jan	0.5
A_3	→	pes	0.5
D_1	→	4	0.6
D_1	→	5	0.2
D_1	→	6	0.2
O_1	→	!	0.65
O_1	→	%	0.3
O_1	→	#	0.05
O_2	→	$$	0.7
O_2	→	**	0.3

Tabulka 4: Pravidla PCFG pro ilustraci algoritmu Next

bázová struktura	**preterminální struktura**	**pravděpodobnost**	**pivot**
$D_1A_3O_2$	$4\{jan, pes\}**$	0.1575	2
$D_1A_3O_2$	$\{5, 6\}\{jan, pes\}\$\$$	0.1575	0
$A_3D_1O_1$	$\{jan, pes\}4!$	0.04875	0

Obrázek 2: Příklad prioritní fronty po rozgenerování prvního prvku

- Jsou vytvořeny všechny preterminální struktury, které z dané pravděpodobnostní bezkontextové gramatiky vytvořit lze.
- Žádná preterminální struktura není vytvořena více než jednou.
- Preterminální struktury na výstupu algoritmu jsou v nerostoucím pořadí dle pravděpodobnosti.

Algoritmus Next poskytuje pro každou PCFG jeden unikátní derivační strom. Tedy, pro každý řetězec generovaný gramatikou (kandidáta hesla) existuje právě jedna podmnožina množiny přepisovacích pravidel, která jej generuje. To je zajištěno právě použitím hodnoty pivotu.

3.4 Algoritmus Deadbeat dad

Byť algoritmus Next lze použít pro tvorbu preterminálních struktur, při generování z komplexnějších PCFG vykazuje extrémně vysoké paměťové nároky. Tento problém řeší algoritmus *Deadbeat dad*, který používá stejnou prioritní frontu, nicméně jeho implementace značně redukuje její velikost [17].

4 Paralelní generování hesel

Jak bylo vysvětleno v sekci 4, generovaná hesla tvoří množství skupin se stejnou pravděpodobností. V průběhu výpočtu je každá taková skupina popsána tzv.

preterminální strukturou - preterminal structure (PT). Tyto struktury jsou generovány pomocí algoritmu Next [18], popř. jeho zdokonalené verze Deadbeat dad [17]. Generování samotných hesel je tak možno logicky oddělit od generování PT. Obrázek 3a ilustruje původní nástroj PCFG Manager, který Weir a kol. implementovali v jazyce Python. Řešení používá zmíněnou prioritní frontu a celkem tři procesy: jeden, který generuje PT a umisťuje je do prioritní fronty, druhý proces, který z PT generuje samotná hesla a třetí pro zálohování stavu výpočtu. Žádnou další paralelizaci nástroj nepodporuje a nemusí tedy docházet k efektivnímu využití dostupných výpočetních prostředků, tj. jader procesoru.

Pro lepší využití výpočeních prostředků jsme řešení reimplementovali[5] s využitím jazyka Go[6], který umožňuje přímou kompilaci do strojového kódu. Pouhou reimplementací bylo dosaženo zhruba 4x rychlejšího generování hesel oproti původnímu řešení [8]. Použitá metodologie generování hesel však stále poskytovala prostor pro další optimalizace.

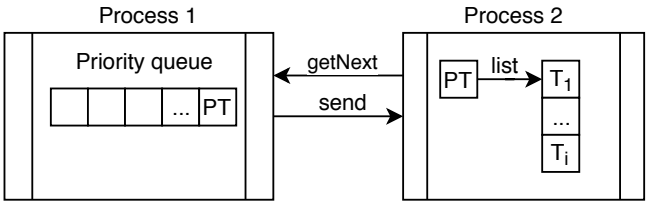
Ze všech kroků, které PCFG Manager provádí, je nejnáročnější právě generování kandidátu hesel z vytvořených PT [18,17]. Protože mezi jednotlivými PT není vzájemná závislost, rozhodli jsme se tuto část procesu paralelizovat. Naše řešení využívá jednu *goroutine* (odlehčené vlákno v jazyce Go) pro generování PT a plnění prioritní fronty. Kromě něj pak 1 až n gouroutines pro paralelní generování hesel. Hodnota n udává míru paralelizace, volí ji uživatel a měla by zohledňovat možnosti použitého procesoru. Výsledné řešení ilustruje obrázek 3b.

Pro synchronizaci a vzájemnou komunikaci mezi gouroutines je použito mechanismu *kanálů* (channels), které v Go pracují jako fronty typu FIFO. Jedna gouroutine může do kanálu zasílat data, nebo je z něj číst. Kanály implicitně nevyužívají vyrovnávací paměť a operace zápisu i čtení je blokující. V našem řešení nicméně používáme kanál s vyrovnávací pamětí (buffered channel) velikosti n, kde zápis do kanálu je blokován pouze je-li kanál plný, tj. obsahuje n prvků. Každý prvek v kanálu reprezentuje jednu PT. Hlavní goroutine (Goroutine M) generuje PT za použití algoritmu Deadbeat dad [17] a umisťuje je do prioritní fronty. Při každém vygenerování PT je tato vložena do kanálu, je-li v něm volný prstor. Pokud je kanál plný, výpočet hlavní gouroutine je automaticky pozastaven díky blokující operaci zápisu. Vycházíme zde z myšlenky, že nemá smysl generovat další PT v době, kdy nemohou být zpracovány. Oproti původnímu řešení není zaručeno generování hesel na výstupu v přesném pořadí dle pravděpodobnosti. Toto zde však nevadí, neboť cílem navrženého řešení je vygenerovat předem stanovený počet nejpravděpodobnějších hesel, který z dané PCFG vygenerovat lze, což je splněno.

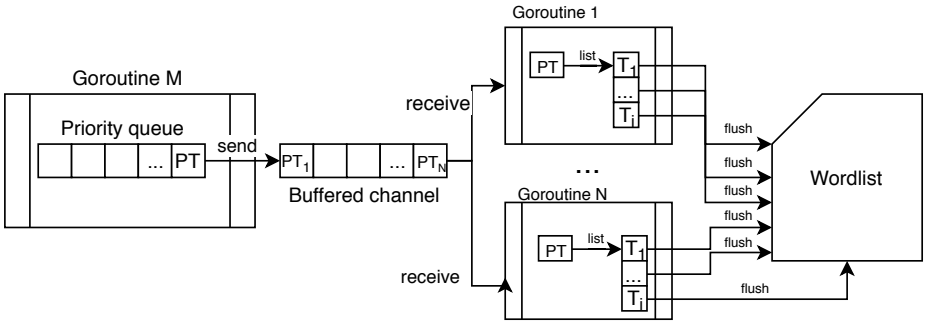
Analýzou výkonnosti jsme detekovali, že po paralelizaci je kritickým místem výstup programu, kde operace zápisu hesel zpomaluje celkový výpočet. Tento problém vyřešilo přidání dodatečné vyrovnávací paměti ke každe gouroutine. Každá gouroutine tedy může ihned zapsat hesla do paměti a nemusí čekat na bezprostřední moment, kdy bude výstup volný. Finální podobu paralelního generátoru ilustruje obrázek 3c. Řešení bylo podrobeno sadě experimentů, jejichž

[5] https://github.com/Dasio/pcfg-manager
[6] https://golang.org/

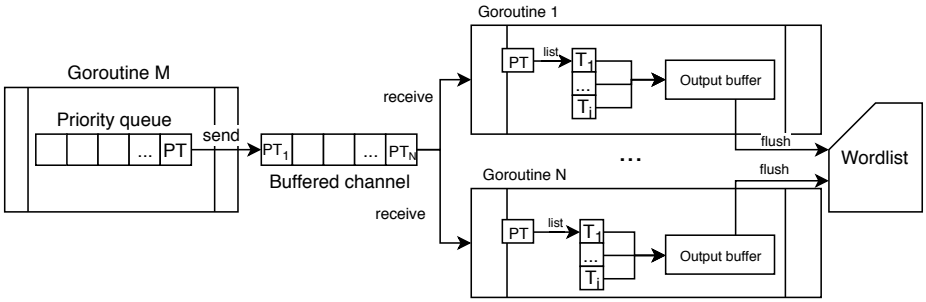
výsledky ukazují 8,14x až 40,43x rychlejší generování hesel oproti původnímu stavu. Experimentům se věnuje samostatný článek [8].



(a) Python PCFG Manager (Weir a kol. [18])



(b) Paralelní PCFG Manager v jazyce Go



(c) Paralelní PCFG Manager v jazyce Go - vylepšená verze

Obrázek 3: Architektura nástroje PCFG Manager
(PT - preterminální struktura, T - terminální struktura - heslo)

5 Distribuované generování hesel

Při distribuovaném lámání hesel uvažujeme sadu výpočetních uzlů, které jsou schopny ověřovat jednotlivé kandidáty hesel. Naivní řešení distribuovaného zpracování je generovat hesla na jednom řídicím uzlu a odesílat je na ostatní, které provádí jejich ověřování. Nevýhodou je vysoká náročnost na operační paměť řídicího uzlu a přenosovou síť. Navržené řešení je proto inspirováno paralelním řešením, popsaným v sekci 4. Řídicí uzel distribuuje pouze preterminální struktury (PT) a finální hesla se generují až na samotných výpočetních uzlech. Kromě prostého výpisu hesel naše řešení umožňuje také přímo spouštět proces lámání pomocí propojení s nástrojem hashcat[7], tedy provádět ověřování generovaných hesel vůči konkrétní sadě kryptografických hešů. Distribuce funguje na principu klient-server, kde server jako službu poskytuje přidělní výpočetní úlohy. Klientské uzly pak řeší samotné ověřování hesel.

5.1 Komunikační protokol

Řešení využívá principu vzdáleného volání procedur s využitím frameworku gRPC[8]. Pro popis struktury přenášených dat a jejich automatizovanou serializaci je použito technologie Protocol buffers[9].

Server obhospodařuje požadavky klientských uzlů a chová se podobně jako Gouroutine M z paralelní verze - generuje PT a z těchto následně tvoří úlohy, které přiděluje klientům k řešení. Jedna výpočetní úloha (*chunk*), tedy zahrnuje jednu nebo více PT. Jak ukazuje 1.1, server poskytuje klientským uzlům aplikační rozhraní v podobě čtyř metod.

```
1 service PCFG {
2     rpc Connect (Empty) returns (ConnectResponse) {}
3     rpc Disconnect(Empty) returns (Empty);
4     rpc GetNextItems(Empty) returns (Items) {}
5     rpc SendResult(CrackingResponse) returns (↩
      ResultResponse);
6 }
```

Výpis 1.1: Aplikační rozhraní serveru

Metodou `Connect()` se klient připojuje k serveru a volá ji ihned po svém spuštění. Server odpovídá zprávou `ConnectResponse`, která obsahuje PCFG v serializované kompaktní podobě. Je-li žádoucí přímo útočit na konkrétní sadu hešů pomocí nástroje hashcat, pak obsahuje zpráva také seznam hešů k prolomení -

[7] https://hashcat.net/hashcat/
[8] https://grpc.io/
[9] https://developers.google.com/protocol-buffers

tzv. *hashlist* a číslo, které označuje typ heše - tzv. *hash mode*[10]. Přehled jednotlivých zpráv ukazuje výpis 1.2.

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

message Items {
  repeated TreeItem preTerminals = 1;
}

message ResultResponse {
  bool end = 1;
}

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

Výpis 1.2: Zprávy mezi klientem a serverem

Po připojení klient požádá metodou `GetNextItems()` o přidělení PT, ze kterých bude následně generovat hesla. Server mu v odpovědi `Items` přidělí 1 až N preterminálních struktur. Po dokončení úlohy klient odešle metodou `SendResult()` na server výsledek v podobě zprávy `CrackingResponse`. Pokud bylo prováděno ověřování hesel vůči seznamu hešů a nějaký heš byl úspěšně prolomen, obsahuje zpráva asociativní pole prolomených hesel k jednotlivým hešům. V opačném případě se toto pole odešle prázdné. Server v odpovědi `ResultResponse` jen uvědomí klienta, zda se má ukončit, či má žádat o další úlohu - tj. znovu volat metodu `GetNextItems()`.

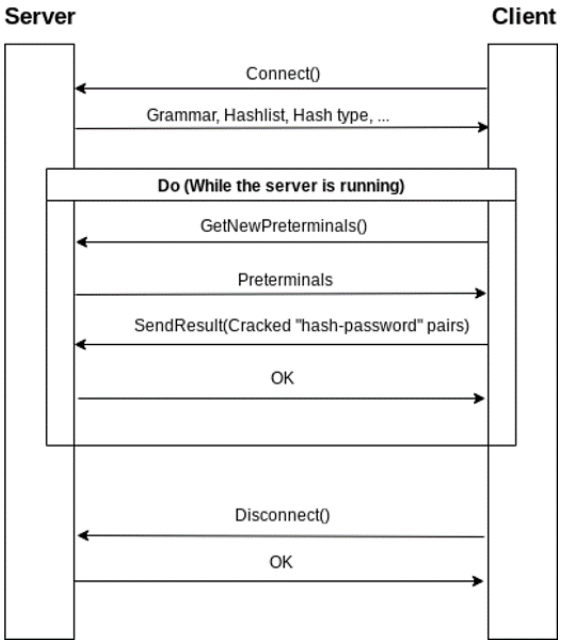
Poslední zpráva je `Disconnect()`, pomocí které klient oznamuje ukončení své činnosti a server tedy může adekvátně zareagovat. Např. pokud měl klient přidělenu úlohu a neoznámil serveru její výsledek. Tok jednotlivých zpráv znázorňuje diagram na obrázku 4.

5.2 Server

Server funguje jako řídicí bod výpočetní sítě. Z pohledu výpočtu uchovává tyto nejdůležitější datové struktury:

- **Prioritní fronta** (viz sekci 3) - tuto frontu používá algoritmus Deadbeat dad [17], pomocí kterého server generuje PT.

[10] https://hashcat.net/wiki/doku.php?id=example_hashes



Obrázek 4: Komunikační protokol distribuované verze

- **Vyrovnávací kanál** - vyrovnávací paměť v podobě kanálu jazyka Go. Tento kanál uchovává již vygenerované PT, ze kterých se následně tvoří úlohy pro klientské uzly.
- **Informace o klientech** - ke každému připojenému klientskému uzlu si server ukládá jeho IP adresu, aktuální výkon, celkový počet hesel, které uzel vyzkoušel a informace o poslední klientem dokončené úloze (počet hesel, čas zahájení a dokončení). Pokud uzel aktuálně řeší nějakou úlohu, je zde uložen její identifikátor (sekvenční číslo), jednotlivé PT a počet hesel, který z nich lze vygenerovat.
- **Seznam nedokončených úloh** - zde se ukládají úlohy, které byly již přiděleny nějakému uzlu, ale ten se odpojil ještě před ohlášením výsledku. Díky jejich uložení mohou být následně znovupřiděleny jinému uzlu.
- **Seznam dosud neprolomených hešů** - v režimu lámání hesel slouží k uložení dosud neprolomených hešů. V režimu generování se nepoužívá.
- **Seznam prolomených hešů** - v režimu lámání hesel slouží k uložení již prolomených hešů a odpovídajících hesel. V režimu generování se nepoužívá.

Po spuštění server načte gramatiku ve formátu, který používá nástroj PCFG Trainer[11]. Dále zkontroluje, zda má běh fungovat v *režimu lámání*, tedy jestli uživatel vybral příslušný seznam hešů a jejich typ. Pokud ano, uloží všechny do

[11] https://github.com/lakiw/legacy-pcfg/blob/master/python_pcfg_cracker_version3/pcfg_trainer.py

seznamu dosud neprolomených hešů. Pokud ne, bude fungovat v *režimu generování* a seznamy hešů zůstanou prázdné. Následně server alokuje vyrovnávací pamět v podobě kanálu, jehož velikost lze stanovit spouštěcím parametrem. Následně začne tvořit PT a umisťovat je do tohoto kanálu. Ke každé PT si server také ukládá, kolik kandidátů hesel je z ní možné vygenerovat. Server v tvorbě PT pokračuje, dokud kanál není plný a je co generovat. Pokud se kanál zaplní, generování je pozastaveno, dokud se kanál opět neuvolní.

Po připojení klientského uzlu si server informace o něm uloží a zašle mu načtenou gramatiku. V případě, že lámeme konkrétní kryptografické heše, zašle server klientovi také jejich seznam spolu s číselným označením typu, jak ukazuje obrázek 4.

Při žádosti o zaslání nové úlohy server vytvoří úlohu obsahující jednu nebo více PT, které postupně odebírá z kanálu. Kolik PT je použito záleží na počtu hesel (*keyspace*), které je z nich možné vygenerovat. Podobně jako plánovač v systému Fitcrack [7,10], se server snaží přizpůsobit velikost úlohy aktuálnímu výkonu klienta v počtu vyzkoušených hesel za vteřinu. Výkon je vypočten na základě velikosti a doby řešení poslední klientem dokončené úlohy. Velikost nové úlohy pak záleží na tomto výkonu a požadované době trvání jedné dílčí úlohy, kterou lze specifikovat parametrem `chunk-duration`. Velikost nové úlohy je tedy určena následujícími vzorci:

$$vykon\_klienta = \frac{velikost\_ulohy}{cas\_dokonceni - cas\_zahajeni} \tag{8}$$

$$velikost\_nove\_ulohy = vykon\_klienta * doba\_trvani. \tag{9}$$

Z kanálu se tedy odebere tolik PT, aby celkový počet hesel byl roven minimálně této vypočítané velikosti. Pokud uzel dosud žádnou úlohu nevyřešil, je velikost úlohy určena předem definovanou konstantou. Takto vytvořenou úlohu jednak pošle klientovi, jednak aktualizuje informace o klientech, aby věděl, že uzel danou úlohu řeší.

Výjimka nastává, pokud se některý z klientů odpojí v průběhu výpočtu - tedy bez oznámení výsledku. Jeho úlohu server uloží na speciální seznam nedokončených úloh, ze kterého je přidělena jinému uzlu při dalším volání `GetNextItems()`. Úlohy v tomto seznamu mají absolutní prioritu nad těmi, které jsou vytvářeny standardním způsobem.

Jakmile klient zašle serveru výsledek úlohy pomocí metody `SendResult()`, server ve svých datových strukturách aktualizuje informace o poslední dokončené úloze a odstraní dokončenou úlohu. Na základě výsledku pak ze seznamu neprolomených hešů odstraní všechny prolomené a tyto společně s hesly umístí do seznamu prolomených hešů. Jsou-li prolomeny všechny požadované heše, server vypíše nalezená hesla a ukončí svou činnost. V režimu generování se server neukončuje a pokračuje dále, dokud je možné generovat další hesla. Totéž platí pro režim lámání, pokud nebyly prolomeny všechny heše.

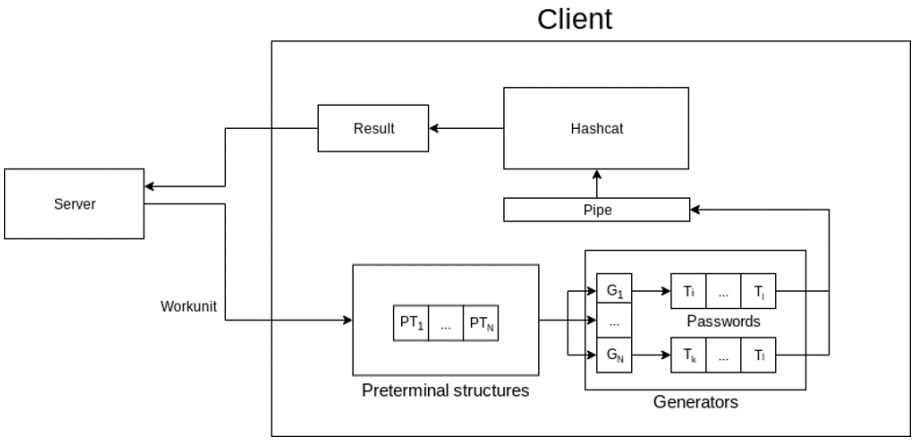
Nakonec, zavolá-li klient metodu `Disconnect()`, server odstraní uložené informace o tomto klientovi. Pokud měl přidělenu nějakou úlohu, je uložena na seznam nedokončených úloh, aby mohla být použita k dalšímu přidělení.

5.3 Klient

Po spuštění se klient připojí k serveru metodou `Connect()` a získá od něj gramatiku společně se seznamem (či prázdným seznamem) hešů, které mají být prolomeny. Následně zavolá metodu `GetNextItems()`, na základě které mu server přidělí úlohu obsahující PT, ze kterých bude tvořit hesla. Podobně jako server, také klient může být spuštěn v režimu lámání, či v režimu generování. V režimu generování prochází postupně všechny PT, které obdržel v rámci zadání úlohy. Z PT generuje hesla a vypisuje je na standardní výstup.

Pro režim lámání je nutné mít na straně klienta k dispozici zkompilovanou verzi program hashcat - cestu k programu je možno specifikovat formou spouštěcího parametru. V režimu lámání klient spustí nástroj hashcat s parametry (hashlist, hash mode) odpovídajícími typu řešené úlohy. Hashcat je spouštěn v režimu slovníkového útoku bez specifikace konkrétního slovníku, kdy přijímá hesla ze standardního vstupu. Klient proto vytvoří rouru (pipe), skrze kterou následně na vstup programu hashcat zasílá jednotlivá vygenerovaná hesla. Hashcat z každého hesla vytvoří kryptografický heš a kontroluje, zda se tento nenachází v seznamu hešů, které chceme prolomit. Po vygenerování všech hesel klient rouru uzavře a čeká na program hashcat, dokud neskončí. Po jeho ukončení se zkontroluje návratová hodnota a v případě úspěchu také seznam prolomených hešů.

O dokončení úlohy klient informuje server metodou `SendResult()`. Pokud byly v rámci úlohy prolomeny nějaké heše, zašle je serveru společně s odpovídajícími hesly jako součást zprávy. Architekturu klientské části a propojení s nástrojem hashcat ilustruje obrázek 5.



Obrázek 5: Architektura klientské části programu

6 Závěr

Pravděpodobnostní bezkontextové gramatiky představují prostředek, kterým můžeme modelovat uživatelské zvyklosti při tvorbě hesel a lépe tak cílit útoky na zabezpečený obsah. Automatizovaným zpracováním existujících slov lze vytvořit gramatiku, jejíž přepisovací pravidla zohledňují strukturu hesel na úrovni fragmentů složených z písmen, číslic a speciálních znaků. Zavedením pravděpodobnosti je možné reflektovat četnosti výskytů jednotlivých fragmentů v původní sadě slov.

V této technické zprávě byly popsány metody použitelné pro automatizovanou tvorbu takových gramatik, společně s algoritmy pro generování hesel z již existující gramatiky. Dále byla vysvětlena práce s větnými formami na úrovni tzv. preterminálních struktur, u kterých máme jistotu, že všechny z ní generované řetězce mají stejnou pravděpodobnost. Tato skutečnost umožňuje paralelizovat náročné části výpočtu a také generování hesel i lámání kryptografických hešů realizovat distribuovaně.

Reference

1. Baker, J. K.: Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, ročník 65, č. S1, 1979: s. S132–S132.
2. Bishop, M.; Klein, D. V.: Improving system security via proactive password checking. *Computers & Security*, ročník 14, č. 3, 1995: s. 233–249.
3. Das, A.; Bonneau, J.; Caesar, M.; aj.: The Tangled Web of Password Reuse. In *NDSS*, ročník 14, 2014, s. 23–26.
4. Florencio, D.; Herley, C.: A Large-scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-654-7, s. 657–666, doi:10.1145/1242572.1242661.
5. Ginsburg, S.: *The Mathematical Theory of Context Free Languages*. McGraw-Hill Book Company, 1966.
6. Houshmand, S.; Aggarwal, S.; Flood, R.: Next Gen PCFG Password Cracking. *IEEE Trans. Information Forensics and Security*, ročník 10, č. 8, 2015: s. 1776–1791.
7. Hranický, R.; Holkovič, M.; Matoušek, P.; aj.: On Efficiency of Distributed Password Recovery. *The Journal of Digital Forensics, Security and Law*, ročník 11, č. 2, 2016: s. 79–96, ISSN 1558-7215.
   URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11276
8. Hranický, R.; Lištiak, F.; Mikuš, D.; aj.: On Practical Aspects of PCFG Password Cracking. In *Data and Applications Security and Privacy*, ročník 33, Springer Verlag, 2019, ISBN 978-3-030-22478-3, ISSN 0302-9743, s. 43–60, doi:10.1007/978-3-030-22479-0_3.
   URL https://www.fit.vut.cz/research/publication/11955
9. Hranický, R.; Matoušek, P.; Ryšavý, O.; aj.: Experimental Evaluation of Password Recovery in Encrypted Documents. In *Proceedings of ICISSP 2016*,

SciTePress - Science and Technology Publications, 2016, ISBN 978-989-758-167-0, s. 299–306.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11052

10. Hranický, R.; Zobal, L.; Ryšavý, O.; aj.: Distributed password cracking with BOINC and hashcat. *Digital Investigation*, ročník 2019, č. 30, 2019: s. 161–172, ISSN 1742-2876, doi:10.1016/j.diin.2019.08.001.
URL https://www.fit.vut.cz/research/publication/11961
11. Jelinek, F.; Lafferty, J. D.; Mercer, R. L.: Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, Springer, 1992, s. 345–360.
12. Ma, J.; Yang, W.; Luo, M.; aj.: A Study of Probabilistic Password Models. In *2014 IEEE Symposium on Security and Privacy*, May 2014, ISSN 1081-6011, s. 689–704, doi:10.1109/SP.2014.50.
13. 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.
14. Proctor, R. W.; Lien, M.-C.; Vu, K.-P. L.; aj.: Improving computer security for authentication of users: Influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers*, ročník 34, č. 2, 2002: s. 163–169.
15. Tarjan, R.: Depth-first search and linear graph algorithms. *SIAM journal on computing*, ročník 1, č. 2, 1972: s. 146–160.
16. Vu, K.-P. L.; Proctor, R. W.; Bhargav-Spantzel, A.; aj.: Improving password security and memorability to protect personal and organizational information. *International Journal of Human-Computer Studies*, ročník 65, č. 8, 2007: s. 744 – 757, ISSN 1071-5819, doi:10.1016/j.ijhcs.2007.03.007.
17. Weir, C. M.: *Using probabilistic techniques to aid in password cracking attacks*. Dizertační práce, Florida State University, 2010.
18. Weir, M.; Aggarwal, S.; d. Medeiros, B.; aj.: Password Cracking Using Probabilistic Context-Free Grammars. In *2009 30th IEEE Symposium on Security and Privacy*, May 2009, ISSN 1081-6011, s. 391–405, doi:10.1109/SP.2009.8.