VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVÁNÍ HESEL NA ZÁKLADĚ PRAVIDEL

RULE-BASED PASSWORD GENERATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **KAREL JIRÁNEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK HRANICKÝ**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19900/2016/xjiran00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Jiránek Karel**
Obor: Informační technologie
Téma: **Generování hesel na základě pravidel**
Rule-Based Password Generation
Kategorie: Bezpečnost

Pokyny:

1. Seznamte se s architekturou a implementací nástroje Fitcrack/Wrathion.
2. Nastudujte techniky generování hesel pomocí předem definovaných pravidel (regulární gramatiky, kombinace řetězců, apod.).
3. Navrhněte rozšiřující sadu generátorů hesel, které budou těchto technologií využívat.
4. Navrženou sadu generátorů implementujte (včetně kódu pro akceleraci pomocí GPU).
5. Porovnejte dosažené výsledky a Vaši implementaci porovnejte s existujícími nástroji.

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.
- 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.
- A další dle dohody s vedoucím.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hranický Radek, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá generováním hesel na základě pravidel, paralelním generováním hesel s využitím GPU a frameworku OpenCL. V práci je řešen návrh generátoru hesel pro nástroj Fitcrack. Modul pro generování využívá strojového učení na reálných heslech ke zkrácení času potřebného k nalezení správného hesla.

Abstract

The thesis describes password generation based on rules and parallel generation of passwords on GPU with OpenCL framework. The thesis addresses the design of a password generator for Fitcrack tool. Module for generation of passwords uses machine learning with real passwords to reduce time to find a correct password.

Klíčová slova

Generování hesel, Fitcrack, kryptografie, OpenCL

Keywords

Password generation, Fitcrack, cryptography, OpenCL

Citace

JIRÁNEK, Karel. *Generování hesel na základě pravidel*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Hranický Radek.

Generování hesel na základě pravidel

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Hranického. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Karel Jiránek
16. května 2017

Poděkování

Rád bych poděkoval Ing. Radku Hranickému za odborné vedení práce a cenné rady, vstřícnost a pomoc při vypracovávání této práce.

Obsah

Kapitola 1

Úvod

Problematikou ochrany soukromí a tajných informací se lidstvo zaobíralo od nepaměti. Již kolem roku 480 př. n. l. vznikly první sofistikovanější metody pro utajení obsahu zpráv, jako například Caesarova šifra. Postupem času se vyčlenil samostatný vědní obor s názvem kryptografie, která se zaobírá problematikou utajování smyslu a obsahu textu. Největšího rozvoje zažila kryptografie v období válek, kdy na utajení rozkazu či zprávy závisely životy vojáků i osudy celých říší a států. V dnešní době nás kryptografie a její prostředky obklopují na každém kroku. Ať už při internetové komunikaci v podobě přihlašování pomocí hesel, tak i v běžném životě, například při zamykání auta dálkovým ovládáním.

Mezi nejdůležitější vlastnosti, kterých chceme z pohledu kryptografie u utajovaného textu dosáhnout, patří důvěrnost, integrita a autenticita. Dosažením důvěrnosti zaručujeme, že zprávu si přečte pouze adresát, kterému byla určena. Integrita nám značí neporušenost zprávy – text dorazí v takovém znění, jak jej autor napsal. Prokázáním autenticity neboli identity nám protější strana potvrzuje, že je tím, za koho se vydává. Použitím různých šifrovacích algoritmů a pečetí je možné dosáhnout důvěrnosti a integrity. K dosažení autenticity je nutné využít jiné prostředky, jako jsou jedinečná přihlašovací jména, hesla a piny.

Heslo je slovo či řetězec znaků, jehož znalostí uživatel prokazuje svoji identitu v autentizačním procesu. V mnoha případech uživatelem nemusí být pouze fyzická osoba, ale i počítač, nebo jiné elektronické zařízení. Sílu a odolnost hesla proti prolomení lze posuzovat z mnoha pohledů – délky, struktury, četnosti výskytu speciálních znaků apod. Délku a strukturu hesla je nutné volit s ohledem na uživatele, který si jej bude muset pamatovat. Jestliže uživatelem je počítač, je možné volit či přidělovat hesla dlouhá a složitá. Odlišná situace nastává u člověka, který je schopen pamatovat si jen omezenou délku řetězce a atypické struktury se speciálními znaky a čísly mu činí problémy. Z toho vyvstává otázka, zdali člověk tvoří hesla podle nějakých pravidel či šablon, aby si hesla lehce zapamatoval.

Zapomenuté, či ztracené heslo trápí nejen běžné uživatele, ale i policii během objasňování trestné činnosti. Obnova hesla je časově velmi časově náročná činnost a vyžaduje specializované hardwarové prostředky a programy. Problémem dosavadních řešení využívaných k obnově hesel je čas potřebný k získání správného hesla.

Tato práce si dává za úkol rozšířit schopnosti prolamovacího nástroje Fitcrack o další generátor hesel, který bude vytvářet hesla v pravděpodobnostním pořadí, a tím potencionálně zkrátí čas potřebný k nalezení správného hesla. Generování hesel je akcelerováno s využitím paralelního zpracování na grafické kartě a standardu OpenCL. Kapitola 2 je zaměřena na představení technologie OpenCL, kterou používá nástroj Fitcrack. V kapitole 3 si představíme základní strukturu tohoto nástroje. V kapitole 4 jsou popsány aspekty ovlivňující proces vytváření hesla. Návrhu gramatiky a generátoru hesel se věnuje kapitola 5.

Kapitola 2

Architektura OpenCL

OpenCL je všeobecně uznávaný multiplatformní průmyslový standard pro programování aplikací určených pro heterogenní systémy. Koncept OpenCL umožňuje naplno využít potenciál paralelního zpracování na výpočetních komponentách, jako jsou procesory (CPU), grafické karty (GPU) a programovatelná hradlová pole (FPGA), a to bez ohledu na jejich hardwarovou architekturu [8].

První verze OpenCL (OpenCL 1.0) vznikla roku 2008 jako odpověď na neutěšenou situaci na poli paralelních výpočtů. Před příchodem toho standardu bylo nutné před vývojem aplikace detailně nastudovat specifikaci cílového zařízení, z důvodu rozdílného paměťového modulu komponenty a jejího přístupu k prvkům na hardwarové úrovni. Tento jev nejenže prodlužoval dobu vývoje aplikace a její cenu, ale i velmi znesnadňoval použití paralelismu ve větší míře.

Architekturu OpenCL lze rozdělit do čtyř modelů:

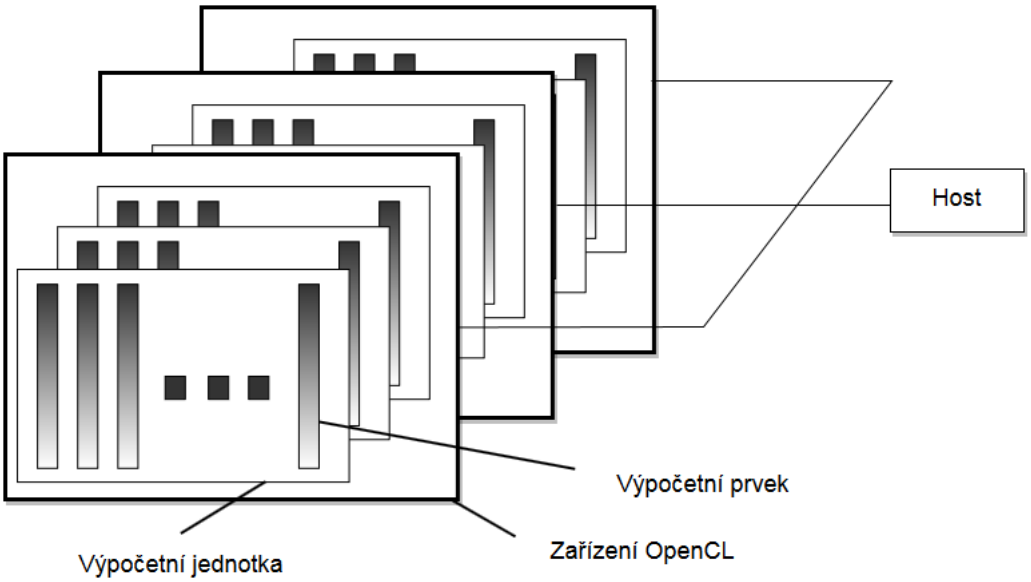
- model platformy,
- exekuční model,
- paměťový model,
- platformní model.

Model platformy

Model platformy je definován jako vysokoúrovňová reprezentace heterogenních systémů schopných nabízet služby OpenCL. Model se skládá vždy z jednoho hostitele a jednoho či více zařízení OpenCL (často nazývaných jako výpočetní jednotky). Hostitele si můžeme představit jako systém s kterým zařízení OpenCL komunikuje, například operační systém počítače. Kernel je sada funkcí OpenCL prováděná paralelně na každém z těchto zařízení. Hostitel komunikuje s prostředím mimo program OpenCL a se zařízeními OpenCL pomocí vstupně-výstupních operací [8].

Výpočetními jednotkami můžou být GPU, DSP[1] nebo jakékoliv komponenty s podporou OpenCL. Jednotlivá zařízení můžeme chápat jako kolekci výpočetních jednotek. Nejmenšími částmi, na které lze zařízení rozdělit, jsou výpočetní prvky. Schéma modelu platformy je vyobrazeno na obrázku číslo 2.1 [8].

[1] Digital signal processor – specializované zařízení pro výpočty s digitálními signály.



Obrázek 2.1: Model platformy v OpenCL

Exekuční model

Tento model popisuje provádění kernelu na zařízení OpenCL.

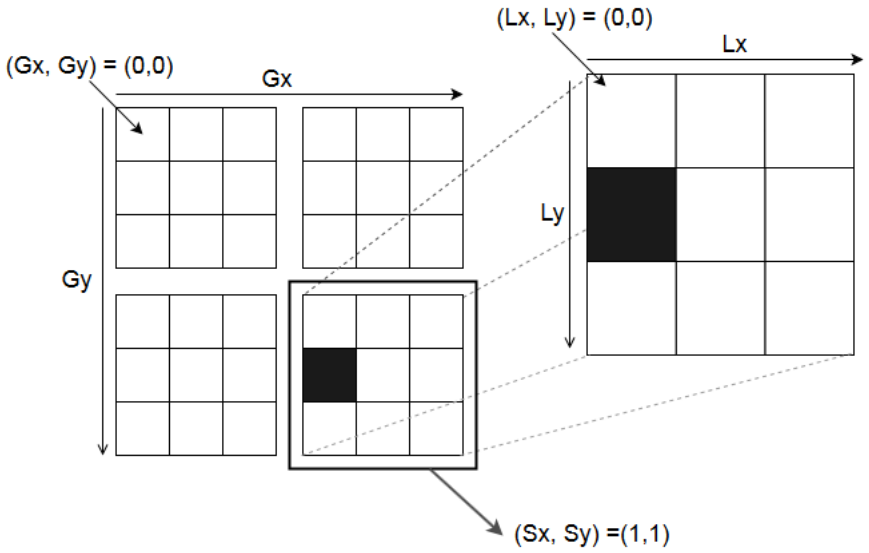
Ze všeho nejdříve je na hostiteli definován kernel, jakožto sada instrukcí. Následně je kernel pomocí příkazu nahrán na zařízení OpenCL, kde se vytvoří indexový prostor. Pro každý bod z tohoto prostoru je prováděna samostatná instance kernelu – **pracovní jednotka**. Všechny jednotky paralelně provádějí stejný kód, avšak jejich chování se může lišit v závislosti na vstupních datech [8].

Pracovní jednotky jsou organizovány do **pracovních skupin** s unikátním ID, jednotlivé jednotky mají svoje lokální ID, které je jednoznačně identifikuje v pracovní skupině. Globální index, identifikující pracovní jednotku v rámci všech jednotek, lze získat sečtením ID skupiny a lokálního ID vynásobeného velikostí skupiny. V příkladu na obrázku 2.2 má černé pole (pracovní jednotka) globální index G =(3,4), lokální index L =(0,1) a číslo pracovní skupiny S = (1,1). Dosazením do výše zmíněného vzorce se lze snadno přesvědčit, že jsme vyčetli indexy správně:

$$G = S \ * velikostskupiny + L$$
$$G = (1,1) * 3 + (0,1)$$
$$G = (3,4)$$

Před zahájením samotného provádění instrukcí musí hostitel ještě nastavit kontext kernelu. Volbu kontextu můžeme chápat jako volbu prostředí, zařízení pro provádění instrukcí. Kontext je definován jako kombinaci zdrojů:

- **Zařízení** – je kolekce OpenCL zařízení, na kterých chceme provádět kód.
- **Kernely** – jsou funkce určené pro běh na zařízeních.
- **Objekty programu** – chápeme jako implementace kernelů.
- **Objekty paměti** – proměnné kernelů, které explicitně definuje hostitel.



Obrázek 2.2: Příklad globálního a lokálního indexu výpočetní jednotky v dvojrozměrném poli. Gx a Gy udávají globální index jednotky, Sx a Sy jsou indexy celé pracovní skupiny. Lokální index je označen jako Lx a Ly.

Model paměti

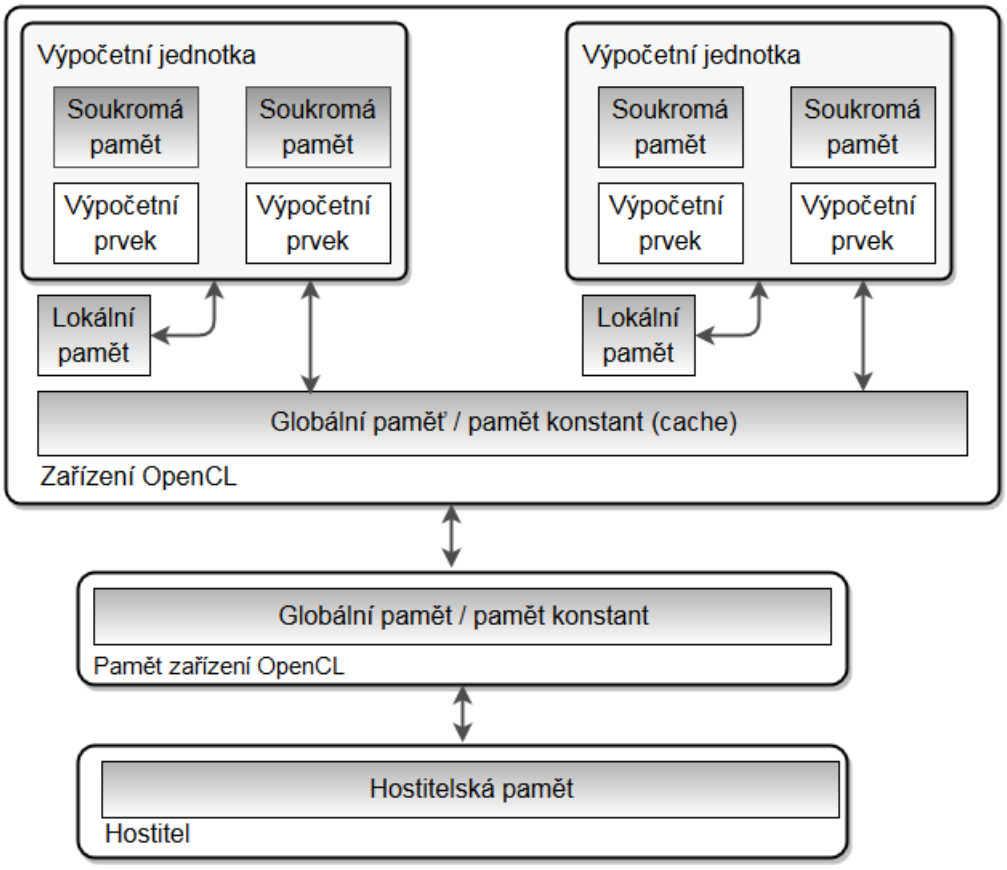
Model paměti přesně definuje strukturu, obsah a způsob manipulace s pamětí. Zde si musíme uvědomit, že operace s pamětí musí být univerzální a nezávislé na architektuře koncového zařízení.

OpenCL rozlišuje dva typy paměťových objektů: buffer objekty a obrazové objekty. Buffer objekt je kontinuální část paměti přístupná kernelům. Programátor může pracovat s touto pamětí pomocí ukazatelů a dle libosti do ní zanášet datové struktury. Naproti tomu obrazové objekty jsou striktně omezeny uchovávat pouze obrazy [8].

OpenCL rozděluje paměť na pět regionů, které jsou abstrakcí fyzické paměti:

- **Paměť hostitele** – je část paměti přístupná pouze hostiteli (OpenCL pouze definuje její interakci s objekty OpenCL).
- **Globální paměť** – je paměť, se kterou mohou manipulovat všechna zařízení OpenCL. Může být zálohovaná na zařízení v závislosti na vlastnostech zařízení.
- **Konstantní paměť** – je součástí globální paměti. Zůstává konstantní po celou dobu vykonávání kernelu – jednotlivé zařízení z ní mohou pouze číst. Inicializace probíhá ze strany hostitele ještě před vysláním startovacího příkazu kernelům.
- **Lokální paměť** – je paměť sdílená v rámci pracovní skupiny. Většinou se vyskytuje přímo v zařízení, v ojedinělých případech je mapována přímo do globální paměti (v případě, že zařízení nedisponuje prostorem pro lokální paměť).
- **Soukromá paměť** – paměť, ke které mají přístup pouze pracovní prvky.

Model paměti si můžeme prohlédnout na obrázku číslo 2.3.



Obrázek 2.3: Abstrakce paměti použitá v OpenCL

Programovací model

Posledním modelem je model programovací, který je především zaměřen na to, jak se aplikace OpenCL mapuje na výpočetní prvky, paměťové regiony a hostitele. Ze všech, již dříve zmíněných modelů, je tento model nejvíce zaměřen na hardwarovou podstatu.

Programovací model lze rozdělit na dvě skupiny – datový paralelismus a paralelismus úkolů. Datový paralelismus můžeme zjednodušeně popsat, jako vykonávání stejné sady instrukcí nad různými daty. Příkladem je násobení vektoru (prvků vektoru) konstantní hodnotou.

Paralelismus úkolů je používaný především v případech, kdy je třeba aplikovat vícero funkcí na stejná data. Jednotlivé jednotky mohou vykonávat velmi podobný kód lišící se pouze ve vstupních parametrech. Příkladem paralelismu je hledání minima a maxima v sadě čísel [8].

Kapitola 3

Nástroj Fitcrack

Fitcrack je nástroj pro obnovu hesel vyvíjený výzkumnou skupinou NES@FIT. Projekt vznikl na Fakultě informačních technologií, Vysoké učení technické v Brně, v návaznosti na předchozí úspěchy nástroje Wrathion [10]. V současné době se jedná spíše o prototyp řešení, než o produkt určený k distribuci. Fitcrack ve verzi 0.2 je schopen obnovovat hesla samostatně (na jedné pracovní stanici), ale i distribuovaně v počítačové síti s využitím platformy BOINC[1]. Podporované formáty[2] zahrnují pdf, zip, 7z, rar, doc(x), ppt a xls.

3.1 Architektura nástroje

Architektura nástroje, kterou si můžeme blíže prohlédnout na obrázku číslo 3.1, se skládá se tří částí:

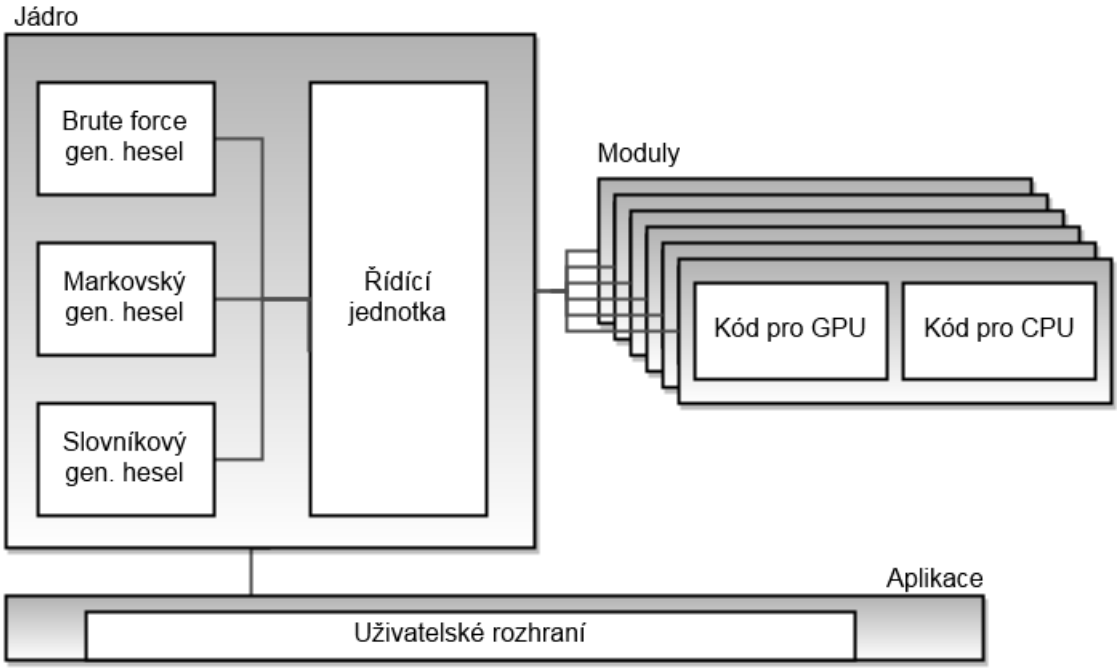
- **Jádro** – obsahuje funkcionalitu nástroje (generátory hesel a řídící algoritmy).
- **Moduly** – ověřují hesla pro jednotlivé formáty vstupního souboru.
- **Aplikace** – obsluhuje jádro a komunikuje s uživatelem textovou formou (v příští verzi Fitcracku již grafickým rozhraním)

3.2 Generátory hesel

Jednou z hlavních součástí nástroje Fitcrack jsou generátory hesel, které mají za úkol co v nejkratším čase vytvářet řetězce předem dané délky a struktury. Jestliže je prováděno generování paralelně, běží stejný kód generátoru na více jádrech současně, čímž dosahujeme většího počtu hesel za jednotku času. Jak už bylo zmíněno, generátory vykonávají stejný kód, ale liší se výchozím bodem. Například máme-li dva generátory hesel, generátor 1 hesla generuje hesla začínající písmenem a a generátor 2 vytváří hesla s počátečním písmenem b. Po dokončení procesu je heslo předáno crackeru, který ověřuje jeho správnost.

Fitcrack v aktuální verzi podporuje hned několik generátorů. Nejjednodušším z nich je generátor založený na technice známé jako **brute force**. Při použití této techniky se nepoužívají žádné heuristiky k omezení, popřípadě seřazení množiny hesel, tudíž je tato metoda velmi rychlá a na vývoj nenáročná. Generátor v každé iteraci obmění jeden znak

[1] https://boinc.berkeley.edu/
[2] k datu 6.12.2016



Obrázek 3.1: Architektura nástroje Fitcrack.

v řetězci, čím postupně zmenšuje množinu neprozkoumaných hesel. Na první pohled by se zdálo, že cesta *hrubé síly* bude jedinou cestou. Avšak jak si ukážeme později, existuje více přístupů ke generování hesel.

Hlavním nedostatek brute force generátoru je, že se zvětšující se délkou hesla a rostoucím počtu přípustných znaků diametrálně roste prozkoumávaná množina hesel. Například víme-li, že hledané heslo má přesně osm znaků a obsahuje pouze malá písmena anglické abecedy, dostáváme se na 26^8 (208 827 064 576) možností. Kdybychom vzali v potaz i velká písmena, speciální znaky a fakt, že délka hesla je většinou neznámá, nemuseli bychom se výsledku i s použitím paralelismu dočkat. Nehledě na to, že každé heslo musí být po vygenerování nezanedbatelnou dobu ověřováno pomocí šifrovacího algoritmu.

Generátor založený na Markovském modelu zohledňuje fakt, že člověk podvědomě tvoří slova (hesla) podle určitých vzorců. Idea **Markovského generátoru** je založená na tom, že jednotlivá písmena jsou ve slovech běžné řeči následována určitým písmenem s určitou pravděpodobností [9]. Konkrétní pravděpodobnosti pro dvojice znaků lze získat frekvenční analýzou slovníku daného jazyka. Generátor využívající Markovských modelů sice nikterak nezmenšuje množinu hesel, ale prvně vytváří pravděpodobnější (obvyklejší) řetězce.

Další možností jak zefektivnit hledání hesla je zvolit **Slovníkový generátor**, který používá předem připravené seznamy slov (hesel). Generátor postupně vybírá z předloženého slovníku hesla a posílá je dalším modulům k ověření. Příkladem vhodného slovníku je seznam nejčastějších hesel. Lidé často volí hesla velmi laxně a předvídatelně, notoricky známými příklady jsou hesla *password*, *123456* nebo *qwerty* [2]. Jiným příkladem slovníku mohou být seznamy slov v konkretním jazyce. Slovníkový generátor omezuje množinu hledaných hesel pouze na ta, která se vyskytují ve zvoleném slovníku. Efektivnost metody velmi ovlivňuje kvalita použitého slovníku a je zřejmé, že oproti předešlým dvěma typům generování, není možné, obsáhnout všechna hesla.

V praxi se ukazuje jako nejlepší kombinace všech zmíněným způsobů.

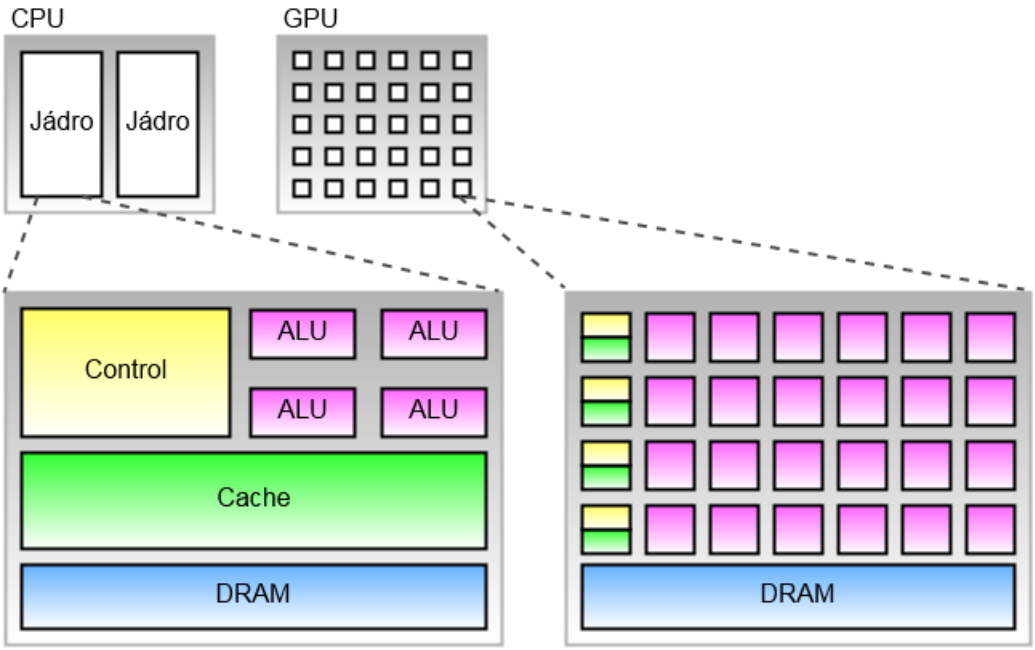
Tato práce si dává za úkol rozšířit množinu generátorů hesel o generátor využívající k vytváření řetězců předchozí zkušenosti s hesly v tréninkové množině.

3.3 Prolamování hesel

Obnovu hesel lze provádět Fitcrackem buď na procesoru (CPU), nebo grafické kartě (GPU). První zmíněná metoda má výhodu v tom, že nevyžaduje podporu žádného z frameworků (CUDA, OpenCL, ...). Avšak výpočetní výkon a míra paralelismu je u crackování na CPU nesrovnatelně menší, než je tomu u druhého způsobu.

Abychom získali alespoň obecné povědomí, proč tomuto tak je, musíme se podívat na strukturu čipu GPU a CPU. Dnešní běžné procesory obsahují řádově jednotky výpočetních jader, oproti tomu grafické karty bývají vybavovány desítkami až stovkami jader. Při bližším zkoumání architektury jádra zjistíme, že většinu fyzického prostoru v jádře zabírá kešovací paměť a oblast pro kontrolu toku. V jádře grafického čipu jsou tyto oblasti potlačeny na úkor navýšení počtu aritmeticko-logických jednotek (ALU). Rozdíl v architektuře můžeme vidět na obrázku 3.2.

Zmíněná fakta dělají z GPU vhodného kandidáta na akceleraci paralelního obnovování hesel.



Obrázek 3.2: Rozdíl architektur CPU (vlevo) a GPU (vpravo).

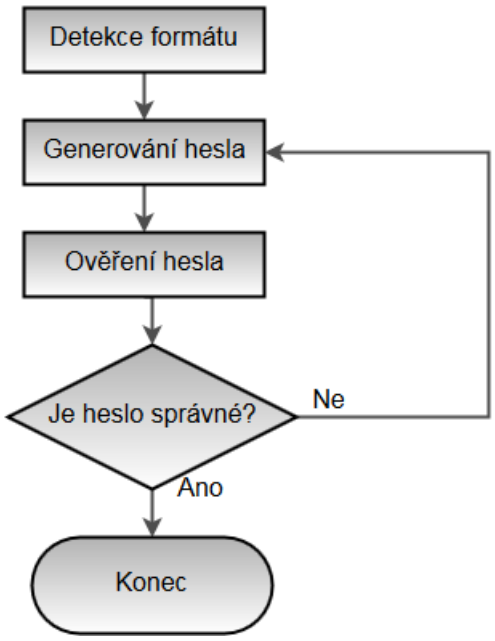
3.4 Postup při obnově hesel

Uživatel při spuštění předkládá zaheslovaný soubor nástroji v podobě argumentu příkazové řádky a soubor XML s metadaty potřebnými pro obnovu hesla. Dále je možné specifikovat počet znaků hesla, počet vláken pro crackování, mód obnovy hesel a v neposlední řadě volí generátor hesel.

Soubor s metadaty v podobě XML je možné získat pomocí podpůrného nástroje FitXtractor, který ze zaheslovaného souboru vyextrahuje potřebné informace (hashe, kon-

trolní součty apod.). Soubor, u kterého chceme obnovit heslo, musí být z podporované sady formátů. Kompletní výčet aktuálně podporovaných formátu lze nalézt na domovských stránkách Fitcracku v záložce Features[3].

Crackování probíhá v iteracích tak, že se prvně získá heslo z generátoru hesel. Následně se heslo pomocí zvoleného šifrovacího algoritmu vypočítán otisk (hash), ten je porovnán hashem v souboru s metadaty. Jestliže došlo ke shodě, nalezli jsme heslo pro odemčení souboru, jinak algoritmus požádá generátor o nové heslo a postup se opakuje. Generování hesel na CPU je zobrazeno na obrázku 3.3 Podobně probíhá i distribuované lámání hesel. Avšak s rozdílem, že koncové stanice provádějí crackování a zároveň komunikují s řídícím synchronizačním uzlem.



Obrázek 3.3: Vývojový diagram popisující generování hesel na CPU s využitím nástroje FitCrack.

[3]http://wrathion.fit.vutbr.cz/fitcrack/?i=features

Kapitola 4

Tvorba hesel

Jak jsme se přesvědčili v předchozí kapitole technika brute force — postupné zkoušení všech kombinací znaků v hesle — se stává pro delší hesla, kvůli expanzi počtu kombinací, nepoužitelnou. Proto je nutné, se co nejvíce zaměřit na omezení, nebo alespoň uspořádání, přípustných hesel. Klíčem k úspěchu je použití heuristik, mezi něž patří již zmíněný Markovský model. Důležitou roli hraje i zkoumání uživatelů a jejich návyků při tvorbě hesel.

4.1 Síla hesla

Sílu hesla můžeme definovat jako odolnost vůči prolomení, kolik pokusů budeme potřebovat k nalezení hesla. Je obtížné exaktně definovat předpis pro silná hesla, jelikož i dlouhá hesla mohou být potenciálně slabá díky své struktuře. Naopak kratší řetězce s neobvyklou strukturou nebo výskytem speciálního znaku mohou crackovací proces značně prodloužit.

Značný vliv na sílu hesla mají restriktivní pravidla použitá při vytváření hesel. Spousta serverů a aplikací uživatele nutí vytvářet potenciálně silná hesla tím, že vyžaduje minimální délku řetězce a použití číslic či speciálních znaků.

V experimentu prováděném vědeckými pracovníky na Carnegie Mellon University (CMU) se ukázalo, že vliv na potenciální sílu hesla mají i další faktory. Nepatrně lepší odolnost vůči prolomení vykazovala hesla tvořená ženami. V rámci mezioborového srovnání se nejlépe umístila hesla studentů informačních technologií, nejhůře dopadla fakulta podnikatelská. Rozdíl mezi těmito fakultami byl zhruba 10% v počtu uhodnutých hesel za jednotku času. Nezanedbatelný vliv měl i počet a rozmístění číslic a speciálních znaků v hesle [5].

Na univerzitě Carnegie Mellon byl ve stejnou dobu prováděn obdobný experiment, kterého se účastnili studenti napříč všemi fakultami. Úkolem bylo vytvořit heslo podle stanovené politiky. O 10 % lépe dopadla hesla studentů, kteří do dotazníku uvedli, že shledávají tvoření hesel zábavným [5].

4.2 Aspekty ovlivňující tvorbu hesel

Při zkoumání a prolamování hesel je nutné přihlédnout k jistým návykům a vlastnostem uživatelů, kteří hesla tvoří. Rozdílná hesla budou vytvářet programátoři, počítačoví technici a úřednice na dopravním magistrátu.

Jedním z faktorů, který značně ovlivňuje výstavbu hesla, je omezenost lidské paměti. Mozek přijímá každou sekundu 16 milionů bitů informací, přičemž vědomě zpracovává pouhých 50 [7]. Do vědomé paměti lze uložit jen nepatrný zlomek těchto přijatých dat, proto

se lidská mysl snaží ukládané informace propojovat a dávat jim určitý význam. Není pak tedy divu, že si uživatelé svá hesla personifikují. Často se mezi hesly objevují data narození, jména přátel a mazlíčku nebo názvy věcí úzce spjatých s uživatelem.

Další pomůckou, kterou člověk využívá k snadnějšímu zapamatování hesla, jsou takzvané klávesové vzory. Analogií v běžném životě je zadávání pinu platební karty. Prsty vykonávají naučený pohyb, aniž bychom více přemýšleli nad tím, jaká tlačítka na bankomatu mačkáme. Z podstaty věci by se mohlo zdát, že vzory jsou ideálním kompromisem mezi zapamatovatelností a sílou hesla, jelikož výsledná sekvence znaků se jeví jako náhodná. Ve většině případů tomu tak opravdu je, což dokazuje i práce *Next Gen PCFG Password Cracking* [3], ve které se mimo jiné hodnotí přínos naučených klávesových vzorů při lámání hesel. Avšak laxně zvolené vzory, jako jsou například řetězce s opakujícím se jedním znakem nebo řádkové vzory (např. sekvence *qwerty*), mohou naopak pravděpodobnost uhodnutí rapidně zvýšit.

Kapitola 5

Návrh generátoru hesel

Při návrhu rozšiřujícího generátoru budeme využívat pravděpodobnostních bezkontextových gramatik k vytvoření hesla.

Definice 1 *Gramatika je uspořádaná čtveřice ve tvaru $G = (N, T, P, S)$, kde N je konečná množina neterminálních symbolů, T konečná množina terminálních symbolů, P konečná množina přepisovacích pravidel a S počáteční symbol patřící do množiny N [6].*

V našem případě bude množina N obsahovat všechny zástupné znaky pro jednotlivá písmena ve slově. Množina terminálních symbolů T bude obsahovat fragmenty hesel. Přepisovací pravidla P definují, jakým fragmentem nahradit zástupný znak z množiny N. Počáteční symbol je závislý na prvním znaku v hesle.

Gramatiku označujeme jako pravděpodobností, když je ke každému přepisovacímu pravidlu přiřazena určitá pravděpodobnost [4]. V praxi to znamená, že se budeme snažit sestavit množinu obecných předpisů seřazených podle pravděpodobnosti výskytu v trénovací množině hesel. Jednotlivé předpisy budeme nadále označovat jako pre-terminální řetězce. Pre-terminální řetězec si můžeme představit jako šablonu pro generování hesla, do které budeme dosazovat postupně podle pravděpodobnosti seřazené útržky hesel z trénovací množiny. Úkolem, před kterým stojíme, je, jak takovouto gramatiku vytvořit, aby co nejvíce přispěla k rychlejšímu prolomení hesla.

5.1 Použitá gramatika

Gramatika použitá v generátoru vychází z práce doktoranda Matthewa Weira [11].

Prvně si musíme definovat množinu zástupných znaků, kterými budeme nahrazovat jednotlivé znaky v hesle. Všechna velká i malá písmena abecedy budeme v hesle ve fázi učení přepisovat na L (**L**etter – písmeno). Číslice nahradíme písmenem D (**D**igit – číslice). Všechny ostatní znaky, jako jsou například *@*, *#* a *$*, zaměníme za S (**S**pecial character – speciální znak). V tabulce 5.1 jsou shrnuta doposud zmíněná pravidla přepisování.

Nepřerušená sekvence stejných zástupných znaků je zkrácena na jeden zástupný znak, za který je přidána číslovka udávající délku nepřerušené sekvence. Jednoznakové sekvence jsou z důvodu dalšího zpracování řetězce doplněny o číslovku jedna. Například heslo *myAngles* přepíšeme na pre-terminální řetězec *LLLLLLLL*, který posléze zkrátíme na *L8* – osmipísemné heslo. V tabulce 5.2 jsou uvedeny další příklady přepisů hesel na pre-terminální symboly.

Typ znaku	Pre-terminální symbol	Symboly
Písmeno	L	abcdefghijklmnopqrstuvwxyz
Číslice	D	1234567890
Speciální znal	S	@$%&(){}+-.

Tabulka 5.1: Tabulka znázorňující typ znaku, jeho zástupný symbol (pre-terminální znak) a symboly, které budou tímto pre-terminálním znakem nahrazeny

Heslo	Přepis	Výsledná pre-terminální forma
car	LLL	L3
pass12	LLLLDD	L4D2
rock&roll	LLLLSLLLL	L4S1L4
P@triots	LSLLLLLL	L1S1L6
orange00!	LLLLLDDS	L5D2S1

Tabulka 5.2: Tabulka přepisů hesel na pre-terminální formy

Příklad gramatiky, která vznikne z hesel *pass@word* a *love!word*, můžeme vidět na následujícím příkladu. Hodnota v závorce značí pravděpodobnost použití přepisovacího pravidla a spočítáme ji jako součin pravděpodobností všech fragmentů na pravé straně přepisovacího pravidla. Fragment *word* má největší pravděpodobnost, jelikož se vyskytuje v heslech dvakrát.

$$
\begin{aligned}
G =& \{N, T, P, S\} \\
N =& \{L4, S1, L4'\} \\
T =& \{pass, word, love, @, !\} \\
P =& \{L4 \rightarrow word S1 L4' \ (50\%) \mid pass S1 L4' (25\%) \mid love S1 L4' \ (25\%), \\
& \quad S1 \rightarrow @ \ (50\%) \mid ! \ (50\%), \\
& \quad L4' \rightarrow word \ (50\%) \mid pass \ (25\%) \mid love \ (25\%) \ \} \\
S =& \{L4\}
\end{aligned}
$$

5.2 Fáze učení

Ve fázi učení je generátoru předložena trénovací množina hesel, z které si vytváří sadu pre-terminálních řetězců. V jednotlivých iteracích si program načte jedno heslo ze souboru a provede přepis na pre-terminální řetězec.

Vedle slovníku pre-terminálů je nezbytné vytvářet i slovníky písemných fragmentů hesel (struktur L_n), čísel a speciálních znaků. Fragmentem je myšlena souvislá část hesla obsahující pouze znaky, pre-terminální symboly, jednoho typu. Z hesla *p@sswod11* získáme čtyři fragmenty: *p*, *@*, *ssword* a *11*. Každý z takto vzniklých fragmentu si uložíme do vlastního souboru. Jestliže při přepisování narazíme na stejný fragment, či celý pre-terminální řetězec zvýšíme jeho počet výskytů.

Zpracovávání souboru končí, jakmile jsou přepsána všechna hesla z trénovací množiny a soubory vzestupně setříděny dle počtů výskytů jednotlivých struktur. Trénování musí proběhnout alespoň jednou před spuštěním generování hesel.

Výstupem trénovací fáze jsou následující soubory:

- **preTerm.dic:** Soubor obsahující všechny pre-terminální řetězce seřazené dle pravděpodobnosti výskytu.
- **alpha.dic** – setříděné soubory písemných fragmentů,
- **digit.dic** – setříděné soubory číselných fragmentů,
- **specChar.dic** – setříděné soubory fragmentů speciálních znaků.

5.3 Trénovací množiny

Jako trénovací množiny posloužily uniklé množiny hesel volně dostupné na internetu[1]. Trénování probíhalo na heslech z komunitního serveru Myspace (≈ 37 tisíc hesel), fóra phpBB (≈ 185 tisíc hesel) a zábavního portálu RockYou (≈ 15 milionů hesel).

Z analýzy souborů bylo zjištěno, že restriktivní pravidla použitá při tvorbě hesel byla velmi slabá. Uživatelé nebyli nuceni používat velká písmena, speciální znaky ani čísla. Dokonce nebyla vyžadována minimální délka hesla.

Zajímavostí je, že anglické slovo *love* (láska, milovat) se vyskytlo v množině hesel z RockYou přibližme 200 000 tisíckrát, ale slovo *hate* (nenávist, nenávidět) pouze 120 000 krát. Spojení těchto dvou řetězců *hatelove* (nenávist lásky, nenávidět lásku) 120 krát. Za povšimnutí stojí i to, že v sadě hesel RockYou můžeme v deseti případech najít i unikátní písmeno české abecedy *ř*.

5.4 Výsledky tréninku generátoru

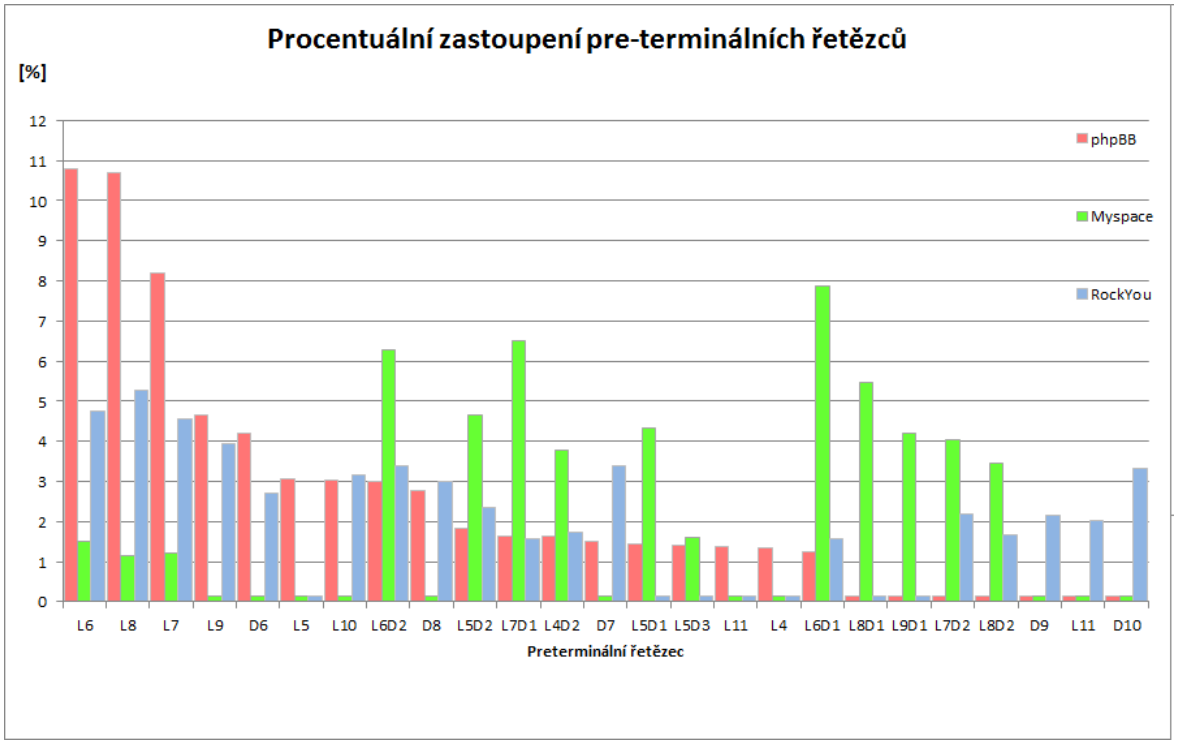
V grafu na obrázku číslo 5.1 můžeme vidět procentuální zastoupení jednotlivých preterminálních struktur napříč trénovacími sadami. Pro přehlednost jsou do grafu zaneseny jen struktury, u kterých bylo dosaženo aspoň dvouprocentního zastoupení v jednom ze tří slovníků.

Hesla složená z pěti až devíti písmen, a jejich kombinace s čísly na konci, dominovala ve všech sadách použitých ve fázi učení. Nejmarkantněji však v sadě phpBB, kde hesla délky 6 a 8 dokonce překročila hranici 10% všech výskytů. Z grafu je patrné, že uživatelé serveru Myspace dávali přednost kombinaci písmen následovaných jedním nebo více čísly před jednoduchými hesly bez čísel. Graf získaný při trénování na sadě hesel RockYou vykazuje nevyrovnanější výsledky ze všech tří testovaných množin.

Bez ohledu na množinu hesel se dle očekávání na první místa v počtu výskytů dostaly osmipísmenné fragmenty řetězce *password* a *Password*.

V trénovací fázi s hesly z phpBB, RockYou a Myspace se přes hranici 2% nedostala ani jedna struktura obsahující speciální znak. Možnou příčinou může být fakt, že běžní uživatelé speciální znaky (především speciální znaky @, #, $, %, ... vyskytující se v na klávesnici v horní řadě tlačítek) používají zřídkavě, a tudíž je pro ně jejich poloha na klávesnici neintuitivní.

[1] https://wiki.skullsecurity.org/index.php?title=Passwords.



Obrázek 5.1: Porovnání zastoupení pre-terminálních struktur v setech phpBB, Myspace a RockYou.

5.5 Jádro generátoru

Generování hesel probíhá tak, že se vyjme ze slovníků pre-terminálních řetězců (*preTerm.dic*) první pre-terminální řetězec (šablona). Do něj se postupně ze slovníků *alpha.dic*, *digit.dic* a *specChar.dic* doplňují fragmenty (terminály) patřičné délky.

Pro správné generování je nutné definovat pomocný index (pivot). Pivot vždy ukazuje na část pre-terminálního řetězce, kterou aktuálně obměňujeme (nahrazujeme fragmentem ze slovníku). Ostatní části řetězce jsou v této chvíli statické. Výchozím bodem pivotu je u nového pre-terminálního řetězce vždy první část, například u řetězce *L5S2D1* je první část *L5*. Pivot se posune na další část řetězce ve dvou případech – jsou-li vyčerpány všechny možnosti (fragmenty) z odpovídajícího slovníku, nebo je-li v pre-terminálním řetězci nenahrazená pre-terminální část. Postup posunu pivotu a generování hesel je zobrazen v algoritmu 1.

Jádro generátoru je společné pro obě implementace generátoru – CPU i GPU.

Algorithm 1 Algoritmus generování hesel

```
1: for each pre-term from preTerm.dic do
2:     pivot ← 1                                              ▷ reset pivot
3:     while pivot != 0 do                                    ▷ all possible passwords generated
4:         for each fragment from (frag).dic indexed by pivot do
5:             pre-term[pivot] ← fragment
6:             if all pre-terms replaced then
7:                 pass password to crackers
8:             else
9:                 pivot ← pivot + 1                          ▷ move pivot right
10:            end if
11:        end for
12:        pivot ← pivot − 1                                  ▷ move pivot left
13:        pre-term[pivot] ← pre-term part                    ▷ reset current part of pre-term
14:    end while
15: end for
```

Předpokládejme, že ve fázi trénování vznikly slovníky zachycené v tabulce 5.3. Generátor v první iteraci vyjme sekvenci *L5D1*. Nahradí *L5* prvním pětipísmenným fragmentem *passw* ze souboru *alpha.dic*. Obdobně je s přispěním souboru *digit.dic* přepsána i část *D2*. Jelikož byly nahrazeny všechny části pre-terminálního řetězce, je první iterace u konce. Vygenerovali jsme heslo *passw1*. V druhé iteraci ponecháme fragment *passw* na svém místě a sekci *D2* (číslo 11) zaměníme za 69 (další číselná sekvence ze souboru *digit.dic*). Dostáváme tak druhé heslo k ověření *passw69*. Další pre-terminální řetězec je načten ve chvíli, kdy byly do předchozího pre-terminálního řetězce dosazeny všechny kombinace fragmentů.

Crackry postupně přijmou od generátorů následující hesla k ověření: *passw11*, *passw69*, *alice11*, *alice69*, *passw@* a *alice@*.

preTerm.dic
L5D2
L5S1

alpha.dic
passw
alice

digit.dic
11
69

specChar.dic
@

Tabulka 5.3: Příklad souborů získaných z fáze generování

Kapitola 6

Implementace

Tato kapitola popisuje implementaci učícího skriptu a nového modulu pro nástroj Fitcrack na základě návrhu popsaného v kapitole 5. Kapitola je rozdělena na dvě části, přičemž první popisuje implementaci trénovacího skriptu a druhá reprezentuje implementaci generátoru.

6.1 Implementace trénovacího skriptu

Pro tvorbu skriptu jsem zvolil skriptovací jazyk Python. Hlavním důvodem pro volbu jazyka Python byla snadná práce s kolekcemi a strukturami založenými na mapování jednoho datového typu na druhý.

6.1.1 Vstupy a omezení

Skript na svém vstupu očekává jeden soubor s hesly. Podmínkou je, že jednotlivá hesla musí být oddělena alespoň jedním znakem nového řádku. Při nedodržení této podmínky, bude řádek s více hesly interpretován jako jedno heslo a bílé znaky budou brány jako speciální symboly.

K interpretování skriptu je nutné mít nainstalovaný překladač pro Python 3.0. Z důvodu omezení rozsahu bezznaménkového 64 bitového celého čísla (uint64_t) jsou všechny pre-terminální řetězce, který by generovaly hesla s pořadovým číslem 2^{64} a více během zápisu do souboru vypuštěny. Omezení bylo implementováno proto, že na začátku generování nástroj Fitcrack potřebuje znám index posledního hesla. Python 3.0 podporuje práci s extrémně velkými čísly, avšak standardní knihovna jazyka C++ nikoliv, takže by při načítání indexu posledního možného hesla mohlo dojít k přetečení datového typu uint_64.

6.1.2 Tělo skriptu

Po inicializaci pomocných proměnných a úspěšném otevření vstupního souboru jsou vytvořeny čtyři výstupní soubory. Pro každý typ fragmentu (znaky, čísla a speciální znaky) je založen nový soubor. Taktéž je otevřen soubor pro uložení pre-terminálních řetězců.

Segmentování hesel na fragmenty probíhá v hlavní smyčce programu. Vstupní soubor je čten řádek po řádku a pro každé heslo jsou provedeny separační operace. Oddělení fragmentů probíhá pomocí pomocné smyčky, která čte heslo znak po znaku. Jakmile se na aktuální pozici nachází znak jiného než stávajícího typu, je čtení přerušeno a doposud načtená znaková sekvence (fragment) je uložena do pomocné proměnné typu *defaultdic*, která nám dovoluje s řetězce asociovat hodnotu. Kolekce *defaultdic* nám zaručuje, že každý řetězec

bude uložen v kolekci pouze jednou, přičemž v našem případě je s fragmentem asociována i číselná hodnota značící počet výskytu fragmentu v trénovací množině hesel. S uložením fragmentu je spojená i operace substituce fragmentu za pre-terminální podobu. Zkracování a substituce řetězce je popsána v podkapitole 5.1.

Po dokončení substituce všech hesel, ukončení hlavní smyčky programu, jsou slovníky setříděny na základě asociované hodnoty *defaultdic* a délky hesla. Setříděním je docíleno, že na vrcholu slovníku (na prvním řádku) je fragment/pre-terminální řetězec s nejvyšším počtem výskytů ve trénovací množině hesel (největší pravděpodobnost) a nejkratší délkou. Jako poslední krok se uskuteční uložení fragmentů a pre-terminálních řetězců do souborů. Společně s fragmenty jsou do souboru uloženy meta informace o délce fragmentů. Na rozdíl od práce Matta Weira [11] je v našem případě uložen pouze fragment bez pravděpodobností hodnoty. Ukládat hodnotu je zbytečné, jelikož při stávajícím návrhu generátoru tuto hodnotu nevyužijeme. Nezahrnutím informace o pravděpodobnosti může dojít k situaci, kdy je vygenerováno heslo z fragmentů s nižší celkovou pravděpodobností výskytu dříve než heslo s celkovou vyšší pravděpodobností. Situace je zřejmější z následujícího příkladu.

Mějme pre-terminální řetězec *L3S2* a setříděné slovníky {*abc 0.05, def 0.04*} a {*$$ 0.06, @@ 0.02*}[1]. Při generování jsou hesla sestavena v následujícím pořadí *abc$$*, *abc@@*, *def$$* a *def@@*. Vypočtené pravděpodobnosti hesel jsou 0.003, 0.001, 0.0024 a 0.0008. Povšimněme si, že třetí hodnota pravděpodobnosti je vyšší než hodnota druhá. K tomuto jevu dochází proto, že nejprve je dosazováno do pravější části pre-terminálního řetězce. Až po vyčerpání fragmentů pro pravější část je obměněna část levější viz sekce 5.5, konkrétně posun pivotu v algoritmu 3.3.

Bez výpočtu celkové pravděpodobnosti nemá generátor šanci tento problém odhalit. Zanedbat tento fakt jsem se rozhodl především proto, že v praxi s reálnými hesly není rozdíl v pravděpodobnosti mezi hesly vygenerovanými ve špatném pořadí tak markantní. Chyba je částečně redukována paralelním generováním na GPU, kde je v jednom běhu vygenerováno až několik desítek hesel, takže chybně seřazená hesla jsou předána k ověření ve stejný okamžik (paralelně).

6.2 CPU generátor – implementace

Kód generátoru spouštěný na CPU je jako ostatní části nástroje Fitcrack implementován v jazyce C/C++. Na vstupu generátor očekává adresář se slovníky vytvořenými v trénovací fázi.

6.2.1 Třída TGenerator

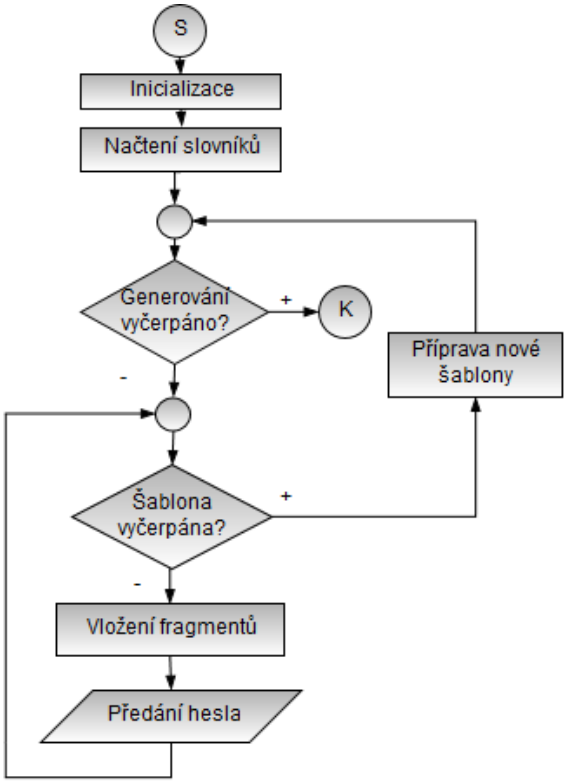
Rozhraním generátoru je třída TGenerator, která mimo komunikace s okolím, zabezpečuje funkcionalitu celého generátoru. Třída definuje několik vlastních datových typů, přičemž jedním z nejdůležitějších je kontejner *TDicMap*. Tento typ je interně reprezentován jako neuspořádaná mapa celočíselných hodnot na vektor řetězců – *std::unordered_map<int, std::vector<std::string>>*. Rozdíl oproti klasickému kontejneru *map* je v rychlosti přístupu k jednotlivým položkám [1]. Pod celočíselným klíčem kontejneru typu *TDicMap* si můžeme představit délku fragmentu. Hodnoty asociované s klíčem jsou jednotlivé fragmenty o délce klíče.

[1] Hodnota za fragmentem značí pravděpodobnost výskytu fragmentu v rámci slovníku daného typu.

Neméně důležitým typem je struktura *TFragTemplate*, sloužící jako šablona pro jednotlivé části pre-terminálního řetězce. Celočíselné hodnoty obsažené ve struktuře reprezentující typ slovníku pro výběr fragmentu, délku fragmentu, aktuální pozici ve slovníku a délku slovníku. Poslední dvě zmíněné hodnoty jsou spíš meta informace sloužící k minimalizaci přístupu do paměti, než proměnné nezbytné ke generování.

6.2.2 Generování

Před generováním je nutné načíst slovníky do paměti RAM a vytvořit šablonu *TFragTemplate* pro každou část pre-terminálního řetězce. O tyto kroky se starají funkce *loadDictionaries* a *makeTemplate*. Schéma generování je možné vidět na obrázku číslo 6.1



Obrázek 6.1: Schéma generování na CPU. Pro jednoduchost není na obrázku posun pivotu ani ošetření chybových stavů

6.2.3 Samostatné verze

CPU generátor je dostupný i jako stand alone verze. Tato verze generátoru je určena ke tvorbě slovníků, neprovádí ověřování hesel, pouze dosazuje do pre-terminálních řetězců fragmenty v pravděpodobnostním pořadí, více o generování v sekci 5.5. V tomto případě se může markantněji projevit chyba s pořadím hesel popsaná na příkladu v sekci 6.1.2. Výsledkem generování je jeden soubor obsahující hesla setříděná v pravděpodobnostním pořadí. Tento soubor může být využit při klasickém slovníkovém útoku.

Program na svém vstupu očekává jeden až dva argumenty. První argument je povinný a reprezentuje adresář se slovníky z fáze generování. Druhý argument určuje počet hesel ve

výsledném souboru s hesly. Při nezadání délky souboru (druhý argument) program generuje a ukládá všechna přípustná hesla. Je vhodné upozornit, že výsledná velikost souboru při neomezení počtu hesel může během několika sekund přesáhnout desítky až stovky MB[2].

6.3 GPU generátor – implementace

Před spuštěním nástroje Fitcrack, do kterého jsou obě verze generátoru (CPU i GPU) integrovány, může uživatel zvolit prolamování s využitím grafické karty. V tomto případě kód generátoru běží na grafické kartě.

Při návrhu GPU verze generátoru bylo nutné brát zřetel na to, že paměťové prostory grafické karty a RAM paměti počítače se se v mnohém liší. Kód GPU generátoru (kernel) je napsán ve frameworku OpenCL. Problematika architektury OpenCL je vysvětlena v kapitole 2.

6.3.1 Reprezentace datových struktur na GPU

V jazyce OpenCL není možné využívat datové typy a prostředky, které nám nabízí standardní knihovna jazyka C++. Z toho vyplývá, že hojně využívané datové typy jako například *std::vector* a *std::map* musí být reprezentovány v jazyce OpenCL jiným způsobem.

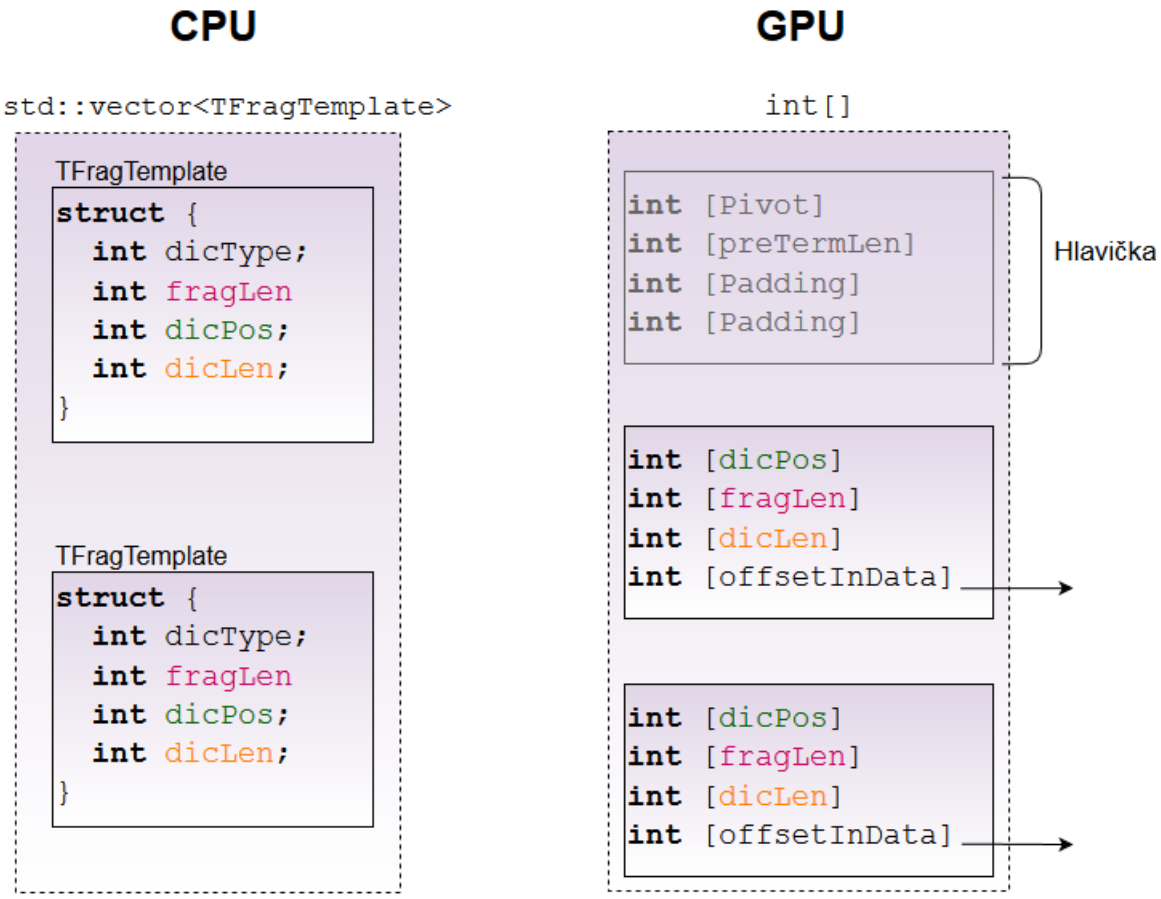
Slovníky typu *TDicMap*[3] obsahující jednotlivé fragmenty hesel jsou před zasláním grafické kartě serializovány funkcí *gpu_serializeDic* do jednoho velkého pole znaků (*char* data*). Jelikož by nebylo možné v takovémto poli určit, kde fragment začíná a končí, je nutné vytvořit pomocnou proměnnou, která bude uchovávat meta informace o jednotlivých fragmentech, potažmo slovnících. O uchování informací se stará pomocná proměnná *LenOffsetMaps* typu *std::map*, kde klíčem je délka fragmentu a hodnotou výchozí posunutí, neboli hodnota udávající pozici v poli *data*, od které jsou jednotlivé fragmenty o délce klíče uloženy. Tato proměnná není zasílána na grafickou kartu, takže ji není nutné převádět do formy přijímané OpenCL frameworkem.

Při vytváření hesla je zapotřebí při každé změně pre-terminálu znovu vytvořit pomocnou proměnnou *std::vector<TFragTemplate>*. Popis datového typu *std::vector<TFragTemplate>* lze nalézt v sekci 6.2.1. Tato šablona je pomocí funkce *gpu_initFragTemplates* nejprve převedena do podobny přijímané grafickou kartou, následně je na grafickou kartu zaslána funkcí *sendToGPU*. Srovnání reprezentace šablony na CPU a GPU si můžeme prohlédnout na obrázku 6.2. Na obrázku je také vyobrazena hlavička šablony, která uchovává informace, které jsou uloženy na CPU odděleně (mimo šablonu). Poslední dvě položky hlavičky jsou v aktuální verzi generátoru nevyužity a tvoří výplň, tak aby velikost hlavičky byla rovna velikosti jedné šablony.

6.3.2 Výchozí posunutí

Samotná logika generování na GPU a CPU je téměř stejná, liší se pouze reprezentací datových struktur. Dosazováním do pre-terminálních řetězců za pomocí indexů popsaných v předchozí sekci vznikají výsledná hesla. Na grafické kartě je zapotřebí také řešit problém paralelně běžících instancí kernelu – je nutné zajistit, aby kernely generovaly různá hesla. Z tohoto důvodu je nutné rozdělit prohledávaný prostor hesel, jinak řečeno přidělit každé instanci kernelu jiný výchozí index pro generování.

[2] Uvažujeme vstupní adresář se slovníky, které vznikly trénováním na reálných množinách hesel.
[3] Klíčem této datové struktury datové je délka fragmentu a hodnotou je vektor fragmentů.



Obrázek 6.2: Porovnání reprezentace datového typu TFragTemplate na CPU a GPU.

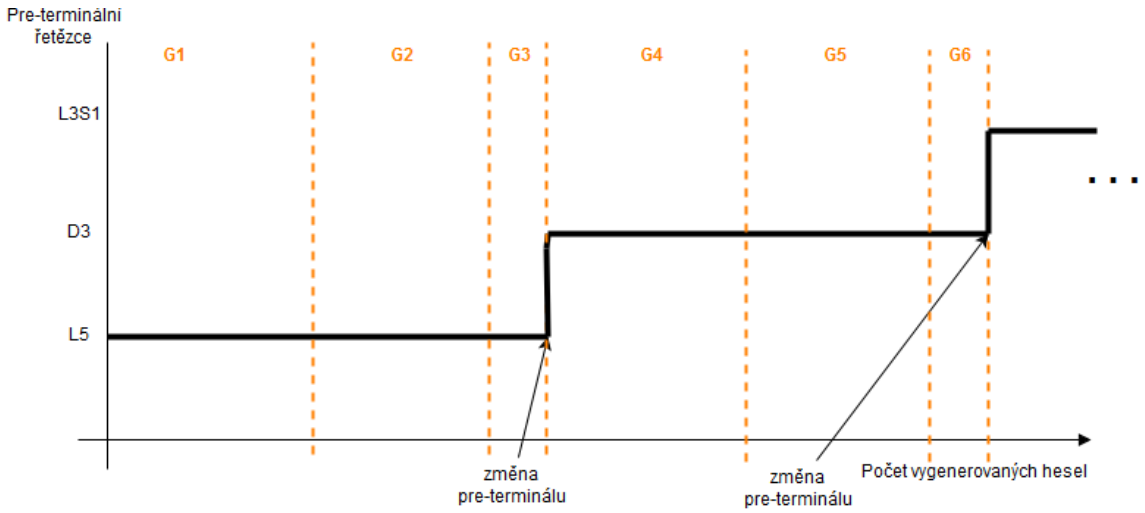
Představme si, že máme pre-terminální řetězec *L5D3*, slovníky pro *L5* a *D3*, přičemž každý obsahuje deset fragmentů, počet výpočetních jednotek GPU je stanoven na 4 a rozsah indexů hesel k vygenerování je 0 až 99. Z počtu fragmentů ve slovnících *L5* a *D3* jsme schopni dopočítat celkový počet vygenerovaných hesel, jako součin počtu fragmentů v *L5* a počtu fragmentů v *D3*. V našem případě je celkový možný počet hesel 100 ($10 * 10$). Podělíme-li velikosti rozsahu přidělených indexů (100) počtem výpočetních jednotek (4) dostaneme počet hesel k vygenerování pro jednu jednotku GPU (25). První výpočetní jednotka GPU (kernel) bude generovat hesla s indexem 0 až 24, druhá 25 až 49, třetí 50 až 74 a poslední čtvrtá 75 až 99[4]. V první generaci hesel tudíž budou vygenerována hesla s indexy 0, 25, 50, 75, v druhé generaci hesla s indexy 1, 26, 51, 76 atd. Paralelizace probíhá na úrovni jednotlivých pre-terminálů – na heslech z jednoho pre-terminálu současně pracují všechny výpočetní jednotky. Rozdělením indexů pro jednotlivé kernely jsme paralelizovali výpočet a také předešli překrývání vygenerovaných hesel. Nové indexy jsou přiřazeny po vygenerování všech možných hesel z aktuální šablony.

Problém nastává v případě, že počet přidělených hesel není násobkem počtu výpočetních jednotek GPU. Například máme-li počet výpočetních jednotek stanoven na 12 a počet přidělených hesel k vygenerování je 30, přidělíme každému z kernelů dvě hesla (30 modulo

[4]Indexujeme-li od nuly – první heslo je na indexu nula

12). Postupně kernely vytvoří hesla s indexy 0 až 23. V přidělené množině hesel nám zbývá 6 nevygenerovaných hesel (s indexy 24 až 29). Tato hesla jsou po vygenerování všech přidělených hesel opět přidělena kernelům. Avšak šest z dvanácti výpočetních jednotek bude během generování nevyužito.

Na obrázku 6.3 můžeme vidět postup generování na grafické kartě. Během jedné generace (označené písmenem G) generují všechny jednotky hesla z přiřazené množiny hesel. Povšimněme si generací s pořadovým číslem tři a šest. Během těchto generací jsou některé jednotky nevyužity – je třeba vygenerovat zbývající hesla z aktuálního pre-terminálu. V nejhorším případě se bude v těchto generacích generovat pouze jedno nové heslo. V nejlepším případě $P - 1$ hesel, kde P značí celkový počet výpočetních jednotek.



Obrázek 6.3: Průběh generování na grafické kartě. Oranžová čára odděluje jednotlivé generace hesel. Množina hesel mezi těmito čarami je vygenerována během jednoho spuštění kernelu. Během generací s pořadovým číslem tři a šest (*G3* a *G6*) jsou některé výpočetní jednotky grafické karty nevyužity z důvodu generování zbývajících hesel ze stávající šablony.

Kapitola 7

Experimenty

Tato kapitola se věnuje experimentování s generátorem hesel navrženým v kapitole 5. Sekce 7.1 až 7.3 jsou zaměřeny na popis experimentů se samotným generátorem a integrovanou verzí.

Hlavní výhodou generátoru založeného na pravděpodobnostních pravidlech je seřadit a omezit prohledávanou množinu hesel a tím také zkrátit dobu potřebnou pro nalezení hesla. Omezení množiny hesel je závislé na použitém slovníku ve fázi tréninku. Z logiky návrhu je patrné, že generátor nevyčerpá celou množinu přípustných hesel, tudíž některá hesla nebudou nikdy vygenerována – hesla, která se nedají vytvořit žádnou kombinací fragmentů.

7.1 Měření pravděpodobnosti nalezení hesla

K měření pravděpodobnosti nalezení hesla nám postačila samostatná verze generátoru. U integrované verze generátoru by bylo velmi obtížné provádět statistické testy. Pro každé vygenerované heslo by bylo nutné provést zašifrování testovacího souboru, extrakci metadat a následné ověření hesla. Vezmeme-li v potaz, že poslední tři zmíněné fáze pro statistické měření nepotřebujeme, bylo lepší využít pro měření pravděpodobnosti nalezení hesla speciálně navrženého experimentálního nástroje.

7.1.1 Postup experimentu

Pro experimentální účely jsme vybraly tisíc hesel ze slovníku *RockYou*, *Myspace* a *phpBB*. K testovaným datům jsme připojily ještě tisíc náhodně vygenerovaných hesel (v tabulce označení *pwgen1000*). Tento slovník byl uměle vygenerován pomocí programu *pwgen* dostupného prostřednictvím Ubuntu balíčků. Hesla ve slovníku *pwgen1000* mají délku tři až dvanáct znaků, přičemž každé heslo obsahuje alespoň jednu číslici.

Experiment proběhl tak, že se postupně vygenerovala hesla ze slovníků vytvořených ve fázi učení. Po vytvoření hesla se prohledala množina hesel zmíněných v předchozím odstavci (celkem 5000 hesel). Při nalezení hesla v této množině se zaznamenal výskyt a aktuální číslo iterace. Experiment byl ukončen po 4 miliardách vygenerovaných hesel. Do souboru byla vždy po pěti milionech iterací zapsána informace o průměrném procentuálním výskytu hesla.

Do experimentů byl pro úplnost zařazen i slovník s názvem *brute force*. Jelikož metoda brute force najde dříve či později heslo, můžeme stanovit pravděpodobnost nalezení na 100 %, aniž bychom zkoušeli všechna hesla. Situace je komplikovanější v případě, kdy

chceme zjistit počet iterací potřebných k nalezení hesla. Když známe znakovou sadu použitou pro generování metodou brute force, není obtížné spočítat výslednou iteraci, kdy bude heslo nalezeno. K těmto účelům byl využit pomocný skript, jehož popis je nad rámec bakalářské práce. Z důvodu automatizovaného opakovaného spouštění skriptu byla z kódu vyjmuta část interagující s uživatelem. Pro jednoduchost byly slovníky, ještě před předložením pomocnému skriptu, značně zjednodušeny – velká písmena byla převedena na malá a speciální znaky byly vypuštěny.

7.1.2 Zhodnocení experimentu

V tabulce 7.1 můžeme vidět průměrnou procentuálního hodnotu výskytu vygenerovaných hesel v experimentálních slovnících. Na obrázku 7.2 můžeme vidět počet iterací převedený na průměrný čas potřebný k prolomení hesla. Jako referenční jsme vzali hodnotu 22 938 300 000 hesel za sekundu, což je průměrná rychlost útoku na formát PDF 1.7 (Adobe Acrobat 9) na počítači Brutalis[1]. Hodnoty ve vyšrafovaných polích v tabulce 7.1 by v experimentu se všemi možnými kombinacemi fragmentů nabývaly 100 %, jelikož hledaná hesla patří do množiny trénovacích hesel – trénovací množina hesel je podmnožinou všech vygenerovaných hesel. Například náhodně vybraná hesla z množiny hesel *RockYou* musí patřit do podmnožiny hesel všech možných kombinací získaných učením na množině *RockYou*. Zelené podbarvení výsledku značí výborný výsledek (větší než 10 %) a červené nedostačující (menší než 10 %).

		Náhodná hesla ze slovníků			
		Myspace1000	phpBB1000	RockYou1000	Pwgen1000
Slovníky použitý při generování	Brute force	100 %	100 %	100 %	100 %
	Myspace	87.7 %	10.2 %	8.5 %	0.5 %
	phpBB	35.4 %	80.6 %	12.5 %	0.5 %
	RockYou	17.8 %	31.8 %	46.4 %	0 %

Obrázek 7.1: Průměrná hodnota výskytu tisíce hesel v generátorem vygenerovaných heslech (4 miliardy) vyjádřená v procentech. Slovník *pwgen1000* byl vytvořen uměle pomocí programu *pwgen*.Zelené podbarvení výsledku značí výborný výsledek (větší než 10 %) a červené nedostačující (menší než 10 %). Hodnoty ve šrafovaných polích by při experimentu při vygenerování všech přípustných hesel nabývaly 100 %.

Na základě výsledků z obrázku 7.1 můžeme říct, že slovníky vytvořené na základě učení s reálnými hesly, podávaly dobré výkony při použití proti heslům stejného typu. Avšak při prolamování uměle vytvořených hesel (hesla programu *pwgen*) vykazovaly tyto slovníky jen malou míru úspěšnosti. Nižší procentuální hodnota úspěchu nalezení hesla u některých slovníků je oproti brute force útoku, vyvážena počtem iterací potřebných k vygenerování hesla.

Mohlo by se zdát, že použití slovníku s více trénovací hesly ve fázi tréninku, povede k lepším výsledkům. Avšak slovníky vytvořené na základě největší trénovací množiny *RockYou* podávaly spíše průměrné výkony. Jednou z mnoha možných příčin může být velké množství kombinací pro jednu šablonu (pre-terminální řetězec). Jelikož slovníky *RockYou* obsahují největší počet pre-terminálních řetězců a fragmentů, musí zákonitě vygenerovat největší množství hesel. Pro jednu šablonu se dostáváme až na několik desítek tisíc kombinací, což

[1] https://sagitta.pw/hardware/gpu-compute-nodes/brutalis/

může vést k tomu, že generátor skoční v našem experimentu z důvodu omezení počtu iterací (4 miliardy), již na prvních několika pre-terminálních řetězcích. Tudíž nestihne během experimentu přejít na další o něco málo pravděpodobnější pre-terminální řetězec. Tento faktor může být eliminován při reálném prolamování jednoho hesla, kdy nejsou generování kladena žádná omezení.

		Náhodná hesla ze slovníků			
		Myspace1000	phpBB1000	RockYou1000	Pwgen1000
Slovníky použitý při generování	Brute force	35744741 roků	1366711 roků	96721065 roků	21 dní
	Myspace	13 ms	240 ms	19 ms	240 ms
	phpBB	36 ms	15 ms	40 ms	28 ms
	RockYou	24 ms	14 ms	25 ms	∞

Obrázek 7.2: Průměrná hodnota počtu iterací k získání hesla převedená na časovou jednotku – počítáno s hodnotou 22 938 300 000 hesel za sekundu. Slovník *pwgen1000* byl vytvořen uměle pomocí programu *pwgen*. Zelené podbarvení značí výborný výsledek (méně než 10^{10} iterací, zlomky sekund) a červené nedostačující (více než 10^{10} iterací, tisíce let).

7.2 Měření počtu iterací k nalezení hesla

Experiment jsme zaměřili měření poměru mezi počtem vygenerovaných hesel a počtem prolomených hesel. Jinými slovy nás zajímalo, jak s rostoucím časem (počtem vygenerovaných hesel) poroste počet prolomených hesel.

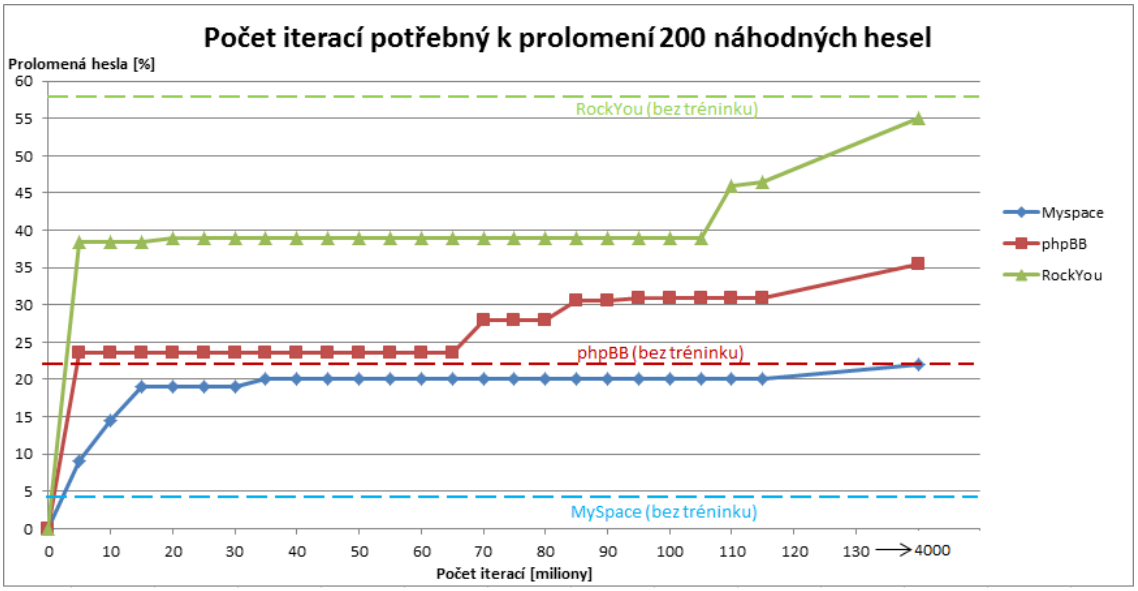
7.2.1 Postup experimentu

Před experimentem jsme náhodně vybrali dvě stě hesel ze slovníku, který nebyl součástí experimentu. Nejlépe nám pro tyto účely posloužil uniklý slovník *Faithwriters* z webu *www.wiki.skullsecurity.org/Passwords*. Další postup byl velmi obdobný předchozímu experimentu. Nechali jsme počítač generovat hesla za pomocí slovníků vzniklých ve fázi učení a zaznamenávali jsme si, zdali se vygenerované heslo vyskytovalo ve dvou stech náhodně vybraných heslech – v tomto případě jsme si zaznamenávali i iteraci, kdy k nalezení (shodě) došlo. Experiment byl ukončen po vygenerování 4 miliard hesel.

7.2.2 Zhodnocení experimentu

Výsledky měření jsou vyneseny do grafu, který je možné vidět na obrázku 7.3. Na tomto obrázku si můžeme všimnout, že největší část hesel byla prolomena již v prvních deseti milionech iterací, pak již počet uhodnutých hesel příliš nerostl. Pro přehlednost není do grafu vynesen celý průběh experimentu. Poslední hodnota označuje koncový stav po čtyřech miliardách vygenerovaných hesel. Přerušovaná čára v grafu značí, kolik hesel by bylo prolomeno za použití daného slovníku při klasickém slovníkovém útoku (bez tréninku). Za povšimnutí stojí fakt, že použití slovníků získaných z *RockYou* vykazovalo nižší úspěšnost než slovníkový útok *RockYou*. Z návrhu generátoru vyplývá, že slovník použitý v trénovací fázi je podmnožinou všech vygenerovaných hesel. Tudíž se nemůže stát, že slovníkový útok vykáže větší míru úspěšnosti v prolomených heslech než útok založený na pravděpodobnostních

pravidlech[2]. V našem případě tento problém nastal z důvodu omezení počtu iterací – hesla by byla vygenerována, avšak experiment byl ukončen ještě než se tak stalo.



Obrázek 7.3: Počet iterací potřebný k prolomení 200 náhodných hesel z množiny *Faithwriters*. Přerušovaná čára v grafu značí, kolik hesel by bylo prolomeny za použití daného slovníku při klasickém slovníkovém útoku (bez tréninku). Poslední naměřená hodnota je zároveň koncovým stavem experimentu.

Na obrázku 7.4 je možné vidět porovnání počtu prolomených hesel při použití trénovací fáze (rule base attack) a bez tréninku (slovníkový útok). Na problém, který ovlivnil výsledek experimentu *RockYou*, je poukázáno v sekci 7.1.2.

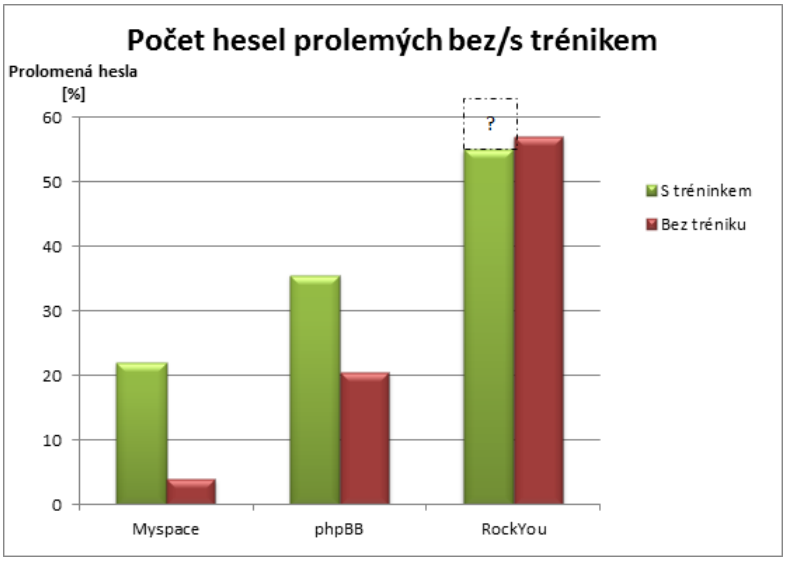
7.3 Výkonnostní experiment

Jedním z dalších aspektů, který se dá u generátoru měřit je počet vygenerovaných hesel za sekundu. Předpokládali jsme, že nejlepších výsledků bude dosahovat brute force generátor, který je velmi jednoduchý a k vytvoření hesla potřebuje jen pár operací. V experimentech jsme porovnávali jednak rychlost jednotlivých generátorů, ale také zrychlení GPU varianty prolamování oproti CPU variantě.

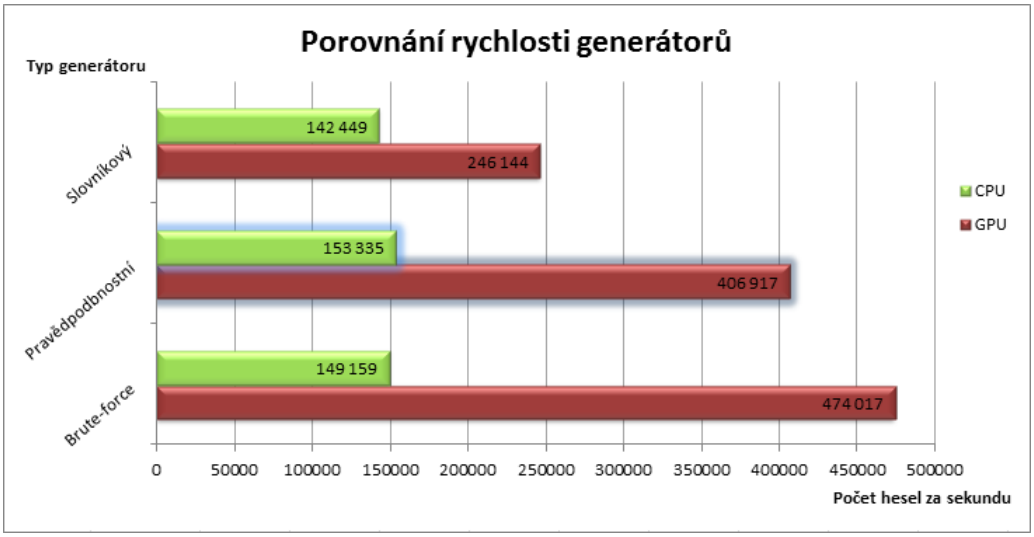
7.3.1 Experimentální prostředí

Do experimentů jsme zařadili tři typy generátorů – brute force, slovníkový generátor a generátor popisovaný v této práci. Slovníkový generátor můžeme spíše označit jako pseudo generátor, jelikož hesla nejsou generována v pravém slova smyslu. V případě slovníkového generátoru se hesla pouze vybírají ze slovníků a popřípadě jsou dále zasílána na grafickou kartu, kde si hesla dále přebírají crackry k ověření. Postupně jsme pouštěli nástroj Fitcrack s parametrem *-b* (benchmark) a zaznamenávali jsme si výslednou průměrnou hodnotu značící počet vygenerovaných hesel za sekundu. Pro experimentální účely jsme si vybrali PDF formát (revize 4).

[2]Předpokládáme stejný slovník použitý při slovníkovém útoku, jako ve fázi učení.



Obrázek 7.4: Porovnání počtu prolomených hesel s/bez tréninku. Zelenou barvou (levé sloupce) je označen útok založený na pravděpodobnostních pravidlech (rule base attack), červenou barvou (pravé sloupce) slovníkový útok. Problém v neúplnosti výsledku experimentu *RockYou* je vysvětlen v sekcích 7.1.2 a 7.2.2



Obrázek 7.5: Porovnání počtu vygenerovaných hesel za sekundu různými generátory hesel.

Experimentální sestava:

- CPU: Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz
- GPU: AMD R9 Fury X
- RAM: 16 GB

Za zapůjčení sestavy bych rád poděkoval inženýru Radku Hranickému.

7.3.2 Zhodnocení experimentu

Na obrázku 7.5 je možné vidět výsledky testování rychlosti generátorů. Dle očekávání byl v testech nejlepší brute force generátor, slovníkový generátor se umístil až na třetím místě. Náš generátor založený za pravidlech (na obrázku označen jako *Pravděpodobnostní*) v rychlosti předčil slovníkový generátor, avšak zůstal v závěsu za generátorem brute force. Při provádění testů na CPU jsou výsledky všech generátorů srovnatelné.

7.4 Zhodnocení experimentů

Z experimentu 7.1 nám vyšlo, že námi navržený generátor dokáže předčit brute force generátor především v počtu iterací potřebných k nalezení správného hesla a tím razantně zkrátit dobu potřebnou k prolomení hesla. Z experimentu 7.2 vyplývá, že největší část hesel je prolomena již při prvních deseti milionech iterací, dále roste pravděpodobnost prolomení hesla jen pomalu. V počtu vygenerovaných hesel se generátor popisovaný v této práci umístil za generátorem brute force, avšak rapidně předčil slovníkový generátor.

Kapitola 8

Závěr

Prolamování hesel pomocí techniky brute force je velmi výpočetně náročné. Akcelerace výpočtů na grafických kartách pomocí nástroje OpenCL dobu prolomení hesla značně snižuje, avšak problém s kardinalitou množiny hesel přetrvává.

Nezanedbatelný vliv v procesu tvoření hesla hraje ovlivnění okolím, pohlaví a sociální původ člověka. Lidem je příjemnější volit hesla spjatá s objekty reálného světa, jako jsou například jména osob a věcí, než potenciálně bezpečnější náhodné sekvence znaků. Jak jsme se mohli přesvědčit v kapitole 5, velkou roli hrají i restriktivní politiky, uplatňované při tvorbě hesla. Při analýze hesel z *RockYou*, *Myspace* a *phpBB* jsme se přesvědčili, že když nejsou uživatelé svázáni pravidly, tvoří ve většině případů hesla kratší a slabší a vyhýbají se používání speciálních znaků. Oproti tomu se v testovacích sadách hesla sestavená z písmen následovaných čísly vyskytovala poměrně často. Tento jev je zajímavým tématem pro další zkoumání a práce zaměřené na lidské uvažování, nejen v kontextu generování hesel.

Pro navýšení pravděpodobnosti uhádnutí hesel byl navržen generátor hesel, vyžívající předchozí zkušenosti s hesly. Před generováním je nutné provést trénink[1]. Základní myšlenkou trénovací fáze je rozdělení hesla na navzájem logicky spjaté struktury – oddělení písmenných, číselných částí a celků obsahujících speciální znaky. Při trénování se mimo jiné ukládají šablony (pre-terminální řetězce) potřebné ke generování hesel. Struktury s větším počtem výskytů, větší pravděpodobností, jsou umístěny fyzicky výše v generovaných slovnících.

Kapitola 3 popisuje strukturu nástroje Fitcrack. Prolamování se dá rozdělit na dvě hlavní části, generování hesla a ověření hesla. Fáze ověření je podstatně časové náročnější než fáze vytvoření hesla, proto je nutné se primárně zaměřit na redukci počtu hesel, iterací k dosažení správného hesla. V současné době nástroj podporuje prolamování hned několika nejznámějších formátů, například PDF, Word, Zip, Rar, Open Office a další. Mimo brute force metodu generování hesla nástroj integruje modul založený na Markovových řetězcích.

Generátor po spuštění postupně vytváří všechny přípustné kombinace fragmentů dosazováním do pre-terminálních řetězců (šablon). Dosazovaní vždy začíná od fyzicky v souboru výše umístěných, pravděpodobnějších fragmentů. Množinu hesel označujeme jako vyčerpanou, až ve chvíli, kdy je vyzkoušeno poslední možné heslo – poslední přípustná kombinace fragmentů.

Kapitola 6 popisuje implementaci trénovacího skriptu a generátoru. Trénovací skript je napsán v jazyce Python, kód generátoru je napsán v jazyce C++/OpenCL. Dále jsou v této kapitole představeny použité datové struktury využité při generování. Sekce 6.3 ukazuje

[1] Fázi trénování lze přeskočit dodáním již existujících slovníků.

rozdílnost návrhu pro grafickou kartu a CPU. V této sekci je také popsána problematika paralelizace výpočtu na GPU.

Z experimentů popsaných v kapitole 7 vyplývá, že generátor založená na pravděpodobnostních gramatikách prolomí průměrně více hesel než klasický slovníkový útok[2] a zároveň zmenšuje počet iterací k obnovení hesla. Generátor jako takový neovlivňuje pravděpodobnost vytvoření hesla, musí se spoléhat na předložené slovníky. Celkovou úspěšnost generování zásadně ovlivňuje použitý slovník ve fázi učení. Experiment měření pravděpodobnosti nalezení hesla 7.1 ukázal, že větší slovník ve fázi učení nemusí znamenat zákonitě větší úspěšnost prolomení hesla. Z výsledků experimentů 7.2 je možné usuzovat, že největší pravděpodobnost prolomení hesla je již při prvních deseti milionech iterací.

Při použití generátoru popsaného v této práci se podařilo redukovat průměrný počet iterací k nalezení hesla z řádově 10^{20} iterací (tisíce let) na 10^{8} iterací (jednotky sekund). Rychlost generátoru je o něco menší v porovnání s brute force generátorem, ale oproti slovníkovému útoku si generátor založený na pravidlech vede podstatně lépe viz obrázek 7.5. Je nutné zmínit, že útok na základě pravidel (pravděpodobnostních gramatikách) má ve většině případů menší pokrytí množiny všech přípustných hesel. Metoda brute force heslo vždy nalezneme, avšak je velmi pravděpodobné, že se tak stane až za několik let. Generátor popsaný v této práci poráží metodu hrubé síly především počtu iterací k nalezení hesla. Hlavní podíl na výsledné síle generátoru hraje slovník použití ve fázi učení.

Rozšířením a námětem pro další práce je ukládat fragmenty i s jejich pravděpodobnostní hodnotou a dopočítat celkovou pravděpodobnost hesla přímo za běhu generátoru. Tímto rozšířením by se odstranil problém se špatně seřazenými hesly popsaný v sekci 6.1.2. Zajímavé by bylo změřit a porovnat pokles výkonu generátoru s nárůstem úspěšnosti. Dále by stálo za zvážení provést na základě rozsáhlých experimentů se stávajícím učícím skriptem a pravděpodobnostním generátorem vytvoření super slovníku/ů, které by teoreticky dokázaly podávat konstantně vysoký výkon při útoku na jakékoliv heslo jakéhokoliv typu.

[2] Předpokládáme stejný slovník použitý při slovníkovém útoku, jako ve fázi učení.

Literatura

[1] *Cplusplus* . [online] [cit. 2017-05-13]. Dostupné z: http://www.cplusplus.com/reference/unordered_map/unordered_map/, 2000-2017.

[2] Florencio, D.; Herley, C.: *A large-scale study of web password habits*. Published in Proc. of the 16th Int. Confer-ence on WWW, pages 657–666, 2007.

[3] Houshmand, S.; Aggarwal, S.; Flood, R.: Next Gen PCFG Password Cracking. *IEEE Transactions on Information Forensics and Security*, ročník 10, č. 8, Aug 2015: s. 1776–1791, ISSN 1556-6013, doi:10.1109/TIFS.2015.2428671.

[4] Manning, C. D.; Schiitze, H.: *Foundations of statistical natural language processing*. Cambridge: Mass.: MIT Press, 1999, ISBN 0-262-13360-1, s.

[5] Mazurek, M. L.; Komanduri, S.; Vidas, T.; aj.: *Measuring Password Guessability for an Entire University*. Technická zpráva, Carnegie Mellon University, Pittsburgh, PA 15213, 2013.

[6] Meduna, A.: *Automata and Languages: Theory and Applications*. London, GB: Springer Verlag, 2000, ISBN 1-85233-074-0, 892 s.

[7] Mlodinow, L.: *Vědomí podvědomí*. Praha: Argo, 2013, ISBN 978-80-257-0909-2.

[8] Munshi, A.: *OpenCL Programming Guide*. Upper Saddle River, NJ: Addison-Wesley, 2012, ISBN 978-0-321-74964-2.

[9] Rabiner, L.; Juang, B.: An introduction to hidden Markov models. *IEEE ASSP Magazine*, ročník 3, č. 1, Jan 1986: s. 4–16, ISSN 0740-7467, doi:10.1109/MASSP.1986.1165342.

[10] Schmied, J.: *GPU akcelerované prolamování šifer*. Diplomová práce, Vysoké učení technické v Brně. Fakulta informačních technologií, 2014.

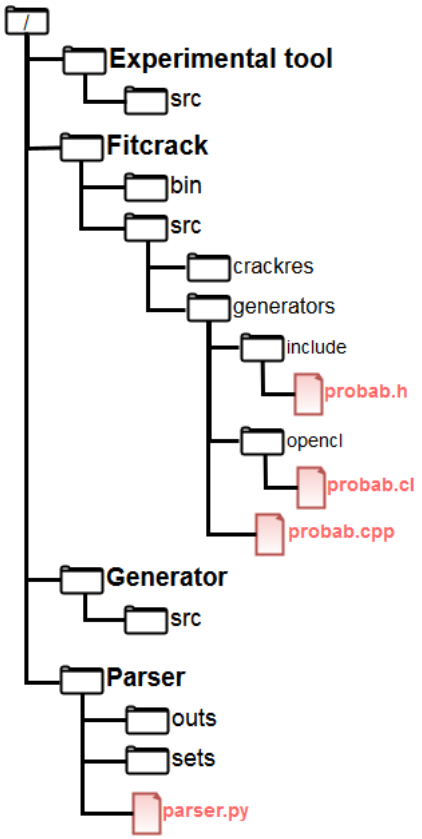
[11] Weir, M.; Aggarwall, S.; de Medeiros, B.; aj.: Password Cracking Using Probabilistic Context-Free Grammars. In *2009 30th IEEE Symposium on Security and Privacy*, May 2009, ISSN 1081-6011, s. 391–405, doi:10.1109/SP.2009.8.

Přílohy

Příloha A

Obsah přiloženého paměťového média

Přiložené médiu obsahuje zdrojové kódy nástroje Fitcrack (Fitcrack/), experimentálního nástroje využitého při experimentech (Experimental Tool/), kódy samostatného generátoru (Generator/) a trénovacího skriptu (Parser/). Na obrázku A.1 je možné vidět celou adresářovou strukturu. Pro jednoduchost jsou ve složce *Fitcrack* naznačeny jen nejdůležitější podsložky.



Obrázek A.1: Adresářová hierarchie

Postup instalace a pouštění nástroje Fitcrack je popsáno v souboru README ve složce Fitcrack. Samostatná verze generátoru ve složce *Generator* je spustitelná pomocí příkazu `./generator <PATH> <[COUNT]>`, kde `PATH` je cesta k se složce obsahující slovníky pocházející z fáze učení, nepovinný parametr `COUNT` zastupuje počet hesel ve výsledném slovníku. Příkaz `./generator out/ 100` vygeneruje sto hesel ze slovníků uložených ve složce *out*. Nápověda k trénovacímu skriptu *parser.py* je dostupná po spuštění s parametrem *–h*. Experimentální nástroj ve složce *Experimental Tool* je dostupný pouze ve verzi beta a očekává přesné pořadí parametrů a přesný počet hesel v experimentální sadě. Podrobnosti o experimentálním nástroji lze zjistit nahlédnutím do zdrojového kódu uloženého v *genertor.cpp*