Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Bakalářská práce

Virtuální GPU cluster

Jakub Samek

Vedoucí práce: Ing. Tomáš Zahradnický, EUR ING, Ph.D.

13. února 2015

Poděkování

Děkuji Ing. Tomáši Kadlecovi, hlavnímu správci ICT fakulty, za pomoc s přípravou učeben a vedoucímu práce Ing. Tomáši Zahradnickému, EUR ING, Ph.D. za podnětné konzultace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 13. února 2015 .....

Odkaz na tuto práci

Samek, Jakub. *Virtuální GPU cluster*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato bakalářská práce se zabývá možnostmi clusteringu počítačů vybavených grafickou kartou pro obecné výpočty s ohledem na běh bezpečnostních nástrojů k prolamování hesel. Praktickou část práce představuje implementace GPU clusteru v prostředí FIT ČVUT s možností distribuovaného běhu vybraných programů a měření výkonnosti clusteru.

Klíčová slova prolamování hesel, hash, cluster, grafická karta, GPGPU

Abstract

This thesis deals with the possibilities of clustering computers equipped with a general purpose computing capable graphics card, in relation to running various security tools for cracking passwords. The practical core of the thesis inheres in the implementation of a GPU cluster in the facilities of the Faculty of Information Technology, CTU with the possibility of a distributed run of selected programs and measuring the cluster performance.

Keywords password cracking, hash, cluster, GPU, GPGPU

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Od počátku 90.let se vyrábí grafické karty s urychlováním 3D grafiky pro oblast počítačových her. GPU je vlastně procesor na grafické kartě, který zpracováváním grafických dat snižuje v počítači zátěž na hlavní procesor (CPU). Je zajímavé, že od té doby se výkon GPU zdvojnásobuje asi každých 6 měsíců, zatímco u CPU je to každých 18 měsíců. Architektura GPU je navržena pro masivně paralelní úlohy, což, jak se ukazuje, není vhodné jen pro 3D rendering náročných počítačových her, ale také například pro různé fyzikální simulace nebo kryptografické výpočty.

Pro taková ostatní využití grafických karet se používá označení GPGPU (General-Purpose computation on Graphics Processing Units). Výkon GPU nelze zvyšovat nad fyzikální limity, ale je možné zvyšovat výpočetní výkon systému spojováním více GPU v jednom počítači (i to má své hranice) nebo dokonce spojováním více počítačů do GPU clusteru.

Jednou z vhodných GPGPU aplikací je prolamování hesel, u kterého se využije jak masivní paralelizace na GPU, tak distribuovaného výpočtu rozdělením úlohy na nody clusteru. Motivací této práce je zodpovědět následující otázky:

- Existuje software pro řešení úlohy prolamování hesel na GPU clusteru?
- Lze takový cluster zprovoznit v prostředí fakulty?
- Jak by byl takový cluster výkonný?
- Jak je systém škálovatelný, zvýší se výkon přidáním dalších výpočetních prostředků?
- Obstojí dnešní systémy ukládání hesel útoku takového clusteru?

Kapitola 1

Prolamování hesel a možnosti distribuovaného řešení

1.1 Autentifikace a ukládání hesel

K ověřování totožnosti (nebo jinak autentizaci) se v počítačových systémech často používá heslo. Uživatel - žadatel o autentizaci - je považován za oprávněného, pokud prokáže znalost hesla. Takové heslo musí být někde uloženo, aby systém byl schopný vyhodnotit, zda se uživatel prokázal platným heslem. Bezpečnost autentifikace tedy závisí nejen na vhodně zvoleném hesle, metodě zadání hesla a na způsobu přenosu zadaného heslo k autentifikačnímu systému, ale také na způsobu uložení hesla v systému. V případě, že jsou hesla uložená v otevřené formě (tedy ve stejné podobě jako je uživatel zadává do systému) jsou v případě neoprávněného přístupu do systému všechna hesla kompromitována. Proto se hesla uchovávají v zašifrované či jinak řečeno „zahashované“ podobě.

1.1.1 Hashování

Hashovací funkce zpracovávají neomezeně dlouhá vstupní data na výstupní kód s předem omezenou délkou. Výstupu hashovací funkce říkáme hash. Po hashovacích funkcích pro ukládání hesel požadujeme, aby byly dostatečně odolné vůči kolizím.

Bezkoliznost 1. řádu
Požadujeme, aby bylo výpočetně nezvládnutelné nalezení libovolných dvou různých vstupů M_1 a M_2 takových, že výstupy po zahashování těchto vstupů jsou stejné, tedy $h(M_1) = h(M_2)$.

Bezkoliznost 2. řádu
Hashovací funkce h je odolná proti kolizi 2. řádu, pokud je výpočetně

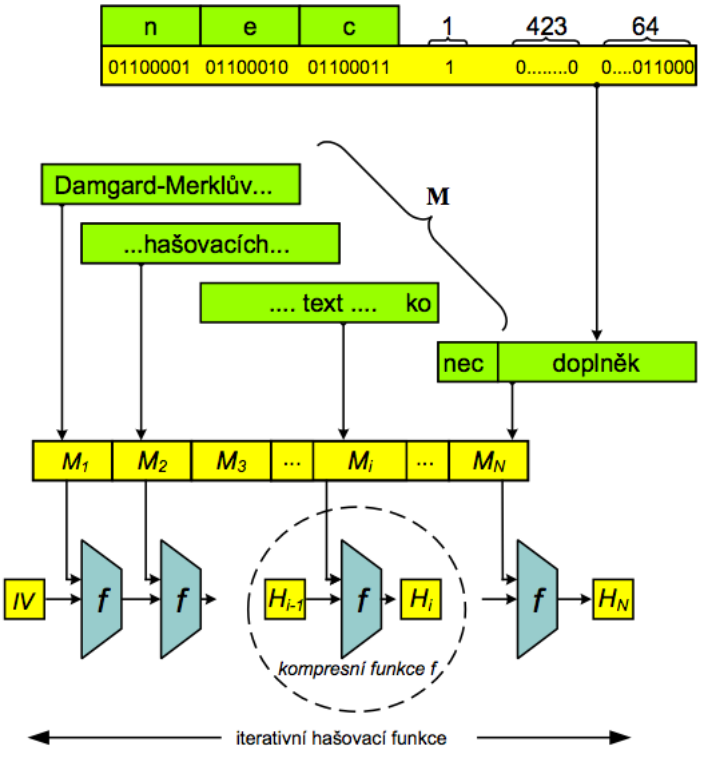
nezvládnutelné pro daný vstup M_1 najít druhý vstup M_2, $M_1 \neq M_2$, takový, že $h(M_1) = h(M_2)$.

Například pokud by hashovací funkce nebyla bezkolizní 2. řádu, pak bychom k danému hashi snadno nalezli heslo s tímto hashem.

Současné prakticky používané hashovací funkce používají Damgard-Merklův princip iterativní hashovací funkce s využitím kompresní funkce f:

$$H_i = f\left(H_{i-1}, M_i\right), H_0 = IV$$

kde M_i jsou bloky vstupní zprávy a H_i jsou výstupy funkce f označované jako kontext, který je použit jako vstup pro další iteraci zpracovávající následující blok. Kontextem pro první iteraci je inicializační vektor IV, což je vhodně zvolená konstanta. Tímto způsobem se zpracuje celý vstup a výsledným hashem je poslední kontext (případně jeho část). Kompresní funkce musí být bezkolizní a velmi robustní, aby zajistila dokonalé promíchání bitů a jednocestnost.



Obrázek 1.1: Iterativní hashovací funkce [1]

Využití hashovacích funkcí je několik, my se budeme zabývat hashováním hesel pro jejich ukládání v bezpečné formě. Známě používané funkce jsou například MD5, SHA1, SHA256, SHA512, NTLM, Bcrypt, PBDKF.2.

S narůstajícím výkonem výpočetních prostředků a možností clusterizace se u některých hashovacích algoritmů doba potřebná k prolomení hesel hrubou silou nebezpečně zmenšuje. Z toho důvodu se zvyšuje bezpečnost uložených hesel použitím složitějších a tedy pomalejších hashovacích funkcí (Bcrypt, PBDKF.2) nebo zvýšením počtu iterací (hashovací funkce se nepoužije pouze jednou, ale několikrát - vstupem každé další iterace je výstup předchozí). Pěkné srovnání rychlostí hashovacích funkcí je dostupné na [2].

1.1.2 Entropie a síla hesla

Při volbě hesla jsme často vybízeni k volbě silného hesla. Jak se taková síla hesla měří? Pomocí entropie. Entropie charakterizuje počet stavů, ve kterém se daný systém může nacházet. Příklad: jeden bajt může nabývat $2^8 = 256$ stavů. Informační entropie bajtu je tedy 8 bitů. Pro popsání stavu, ve kterém se systém nachází, je potřeba přesně tolik bitů informace, jaká je jeho entropie. Pokud v dané množině symbolů (například malá písmena abecedy) je pravděpodobnost výskytu všech symbolů stejná, pak se entropie H zprávy o délce n symbolů spočítá předpisem

$$H = n \cdot \log_2 s,$$

kde s je počet symbolů v dané množině. Nyní si můžeme udělat přehled běžně používaných znakových sad a entropie jejích znaků.

Tabulka 1.1: Entropie hesla

rozsah symbolů	počet symbolů s	entropie prvku zprávy H
0-9	10	3.3
a-z	26	4.7
a-z, 0-9	36	5.2
a-z, A-Z, 0-9	62	6.0
a-z, A-Z, 0-9, mezera, speciální znaky (ASCII vytisknutelné)	95	6.7

Z pomocí této tabulky snadno spočítáme, že heslo *?.Vz4.5* má entropii $7 \cdot 6.7 = 46.9$, což je přibližně stejně jako u hesla *hezkeheslo* ($10 \cdot 4.7 = 47$). Z entropie lze jednoduše spočítat počet kolik pokusů je třeba při jeho testování hrubou silou, u těchto dvou hesel je to $2^{47} \approx 1.4 \cdot 10^{14}$, což, jak se ukáže, není u některých algoritmů mnoho.

1.2 Prolamování hesel

Prolamováním hesel rozumíme proces, ve kterém se snažíme získat heslo v otevřené podobě z dat obsahujících jeho tajnou podobu. V této práci budeme

tajnou podobou uvažovat tu, ve které se heslo ukládá zahashováním.

1.2.1 Možnosti útoku

Způsobů jak se pokusit prolomit heslo je několik, uvedu ty hlavní. Pro úspěšné prolamování hesel je zásadní najít rovnováhu mezi časem potřebným k útoku, potřebným výkonem a pravděpodobností úspěchu. Při prolamování zahashovaných hesel se k ověření správnosti hesla postupuje stejně jako při autentifikaci - kandidát na heslo je zahashován a porovnán s uloženými daty, pokud se shodují, tak kandidát je hledané heslo.

1.2.1.1 Útok hrubou silou

Při útoku hrubou silou se jednoduše zkouší všechny možnosti z nadefinovaného prostoru možných hesel. Hesla jsou omezená délkou a je tedy jasné, že dostatečně dlouhým během takového útoku bude pokus o prolomení vždy úspěšný. Problémem této metody je, že doba běhu u složitých hesel či u pomalých hashovacích funkcí může snadno přesáhnout hranice doby dokončení v reálném čase.

1.2.1.2 Slovníkový útok

U slovníkového útoku se snažíme heslo uhodnout zkoušením slov ze seznamu. Hesla si často uživatelé volí sami a málokdy je volí náhodně. Na internetu pravidelně vychází statistiky nejpoužívanějších hesel a jsou to hesla často triviální. Zdrojem dat pro takové slovníky jsou velké databáze uniklých hesel [3] nebo jazykové slovníky. Bývají to maximálně miliony slov, což je objem dat, který u většiny hashovacích funkcí lze i se slabým hardwarem otestovat.

1.2.1.3 Útok předpočítaním tabulek

V předchozích metodách se hesla ověřují tak, že je kandidát zahashován a výsledek porovnán s uloženým heslem v tajné podobě. S předpočítáním hashovacích tabulek lze hashování (nejpomalejší část ověřovacího procesu) vynechat. Do tabulky si předem předpočítáme hash pro zvolená hesla a pak se v případě útoku na zahashované heslo pouze hledá v tabulce, což je velmi rychlé. Takové tabulky velmi rychle nabývají na objemu, přesto to pro nějaké hashovací funkce může mít smysl. Například pro MD5 se všechny hesla kratší než 10 znaků vejdou na 1 TB disk [4]. Používání hashovacích tabulek (často také Rainbow tabulek) se nehodí u algoritmů používajících tzv. solení [5], bylo by nutné pro každou sůl předpočítat zvláštní tabulku.

1.2.1.4 Další útoky

Programy na prolamování hesel implementují i další možnosti útoků. Mezi nejzajímavější považuji útoky rozšiřující slovníkový útok. V některých nástrojích

je možné slovníková hesla před otestováním modifikovat - například změnit velikosti písmen, přidávat na začátek či konec slova náhodný řetězec nebo třeba slova řetězit za sebe s volitelným oddělovačem. Potom i zdánlivě složitá a dlouhá hesla jako například *VenkuJeKrasne1.* mohou být snadno prolomitelná [6].

1.2.2 Výpočetní prostředky

Již víme jaké jsou možnosti prolamování a nyní se zaměříme na rychlost, s jakou lze hashe hesel prolamovat a jaký hardware je na to vhodný.

Rychlost prolamování měříme jako počet vyzkoušených hesel oproti zadání za vteřinu a tento údaj nazýváme hashrate. Pokud tedy například prolamujeme MD5 hash rychlostí 10 MH/s, znamená to, že program zahashuje a otestuje 10 milionů různých hesel za vteřinu.

1.2.2.1 CPU

Z logiky věci na procesorech jsou všechny používané algoritmy implementovatelné, a tak je lze použít na prolamování hesel. Dobrý program by měl být schopen paralelizace výpočtu, a tedy využívat pro výpočet všechna jádra a procesory systému.

1.2.2.2 GPU

Velmi stručně je síla GPU v tom, že na rozdíl od procesorů, které nabízejí jednotky až desítky výpočetních jader, grafické karty mají takových jader stovky. Některé úlohy tak lze díky dobře zvládnuté paralelizaci vykonávat o několik řádu rychleji než na procesorech. Hashování, potřebné k prolamování hesel, k takovým úlohám patří a programy s podporou běhu na GPU dosahují řádově lepších výsledků. Popsat podrobnější fungování programů na grafických kartách a konkrétní rozdíly oproti klasickým procesorům, jako je správa vláken, přístup k pamětím a podobně je nad rozsah této práce.

Výrobci grafických karet ke svým kartám vhodným pro GPGPU zveřejňují API pro programátory, hlavními jsou CUDA pro karty Nvidia, Stream pro karty AMD a OpenCL, jež se snaží být obecným rozhraním pro výpočetní akcelerátory. Tato rozhraní dávají programátorům možnost upravit stávající implementace pro akceleraci na grafických kartách a většinou lze použít některý ze známých vyšších jazyků jako je C či C++.

1.2.2.3 FPGA

Field programmable gate array, tedy programovatelná hradlová pole, lze také využít na prolamování hesel. Musí být samozřejmě dostatečně velká, aby tam šlo hashovací funkce implementovat. Našel jsem jen několik dostupných hotových produktů, za nejzajímavější považuji Pico Computing SC5/M505-48 [7],

kde výrobce uvádí rychlost prolamování 164 milionů NTLM hesel za vteřinu při odběru $1.5kW$. To vše v jednom poli sestaveném do 4U skříně. Cena takových řešení bývá vysoká, na dotaz, kolik stojí zmiňované zařízení, jsem dostal odpověď „hodně přes 100 000 €“.

1.2.3 Rychlost prolamování

Se znalostí měření síly hesla a rychlosti prolamování si nyní můžeme pro lepší představu udělat tabulku přibližné doby potřebné k prolomení hesla hrubou silou.

Hodnoty rychlostí prolamování jsem zvolil tak, aby přibližně pokrývaly výkon používaných výpočetních prostředků při aplikaci rychlých hashovacích algoritmů jako je MD5 nebo NTLM. Dnešní procesory by měly zvládat okolo 10MH/s na jedno jádro. Běžné (i neherní) grafické karty by měly dosahovat rychlostí nad 1000MH/s, od GPU clusteru očekávám výkon alespoň 100 GH/s. Poslední hodnotu, tedy 10 TH/s, považuji za „postavitelné maximum“.

Tabulka 1.2: Rychlost prolamování

Heslo			Hashrate			
znaky	délka	entropie	10 MH/s	1000 MH/s	100 GH/s	10 TH/s
a-z, A-Z, 0-9	4	23.8	1s	<1s	<1s	<1s
a-z, A-Z, 0-9	5	29.8	1m33s	<1s	<1s	<1s
a-z, A-Z, 0-9	6	35.7	1h36m	57s	<1s	<1s
a-z, A-Z, 0-9	7	41.7	4d3h	1h	35s	<1s
a-z, A-Z, 0-9	8	47.6	256d	2d13h	37m	22s
a-z, A-Z, 0-9	9	53.6	43r	159d	1d14h	23m
a-z, A-Z, 0-9	10	59.5	2703r	27r	98d	23h40m
a-z, A-Z, 0-9	11	65.5		1676r	16r	61d
a-z, A-Z, 0-9	12	71.5			1039r	10r

Z hodnot v tabulce se dá říct, že dvanácti znakové heslo (nebo přesněji heslo s entropií nad 70) bezpečně odolá útoku hrubou silou. Ovšem uvážíme-li pokročilé slovníkové útoky, pak například heslo *dnesjekrasne1* nelze považovat za bezpečné, i když má velkou entropii.

1.2.4 Distribuované prolamování

Pro zrychlení prolamování by kromě paralelizace v rámci jednoho stroje bylo vhodné mít možnost spustit prolamování distribuovaně. Tedy spustit úlohu na několika počítačích zároveň a to tak, aby výkonnost celého systému rostla lineárně s počtem zapojených strojů. Například pokud máme 5 stejných počítačů, tak distribuované prolamování hesla by na nich mělo trvat pětinu času, než když spustíme výpočet pouze na jednom.

Podpora pro distribuovaný běh programu na prolamování hesel by měla být v takovém programu přímo zabudována (například využitím některé z implementací MPI s podporou GPU jako MVAPICH2 [8]). Alternativně by měl mít program rozhraní pro rozdělení výpočtu na části, což by dávalo možnost naprogramovat nadstavbu pro distribuovaný běh.

Problém prolamování hesel je velmi dobře rozdělitelný na menší části. Například pokud máme 5 počítačů a chceme prolomit heslo za pomocí slovníku s 100000 slovy, lze jeho rozdělením určit, aby první počítač zkoušel prvních 20000 slov, druhý dalších 20000 a tak podobně. Stejné by to bylo i v případě útoku hrubou silou. Máme 10 počítačů a pokud uvažujeme pouze číselná hesla, můžeme výpočet rozdělit třeba tak, že první bude počítat pouze hesla začínající nulou, druhý jedničkou a tak dále. Nároky na sdílenou paměť nejsou téměř žádné, pokud některý z počítačů najde řešení, měl by mít možnost informovat o tom ostatní.

1.3 Nástroje k prolamování hesel

Napsat program na prolamování hesel je jednoduché, velmi zjednodušeně jde o implementaci hashovací funkce a poté stačí v cyklu porovnávat zahashované zadání s uloženým heslem. Programů na prolamování existuje mnoho, liší se hlavně svou výkonností, množstvím implementovaných algoritmů a možnostmi útoků. Vybral jsem menší skupinu těch, které považuji za aktuálně nejlepší. Jejich hlavní vlastností jsem zjednodušeně zhodnotil v tabulce.

* Hashcat ani oclHashcat / cudaHashcat samy o sobě distribuovaný běh neumí, ale existují nadstavby (wrappery), které to umožňují.

1.3.1 Hashcat

Hashcat (a jeho verze pro GPU oclHashcat / cudaHashcat) se chlubí tím, že jde o nejrychlejší program na prolamování hesel na světě a také tím, že jako jediný implementuje modifikační pravidla [16] pro použití na GPU. Můj průzkum to potvrzuje a navíc Hashcat vede i v množství implementovaných hashovacích algoritmů. Považuji ho za nejlepší program k zprovoznění ve školním prostředí. Hashcat sám o sobě nemá vestavěnou podporu distribuovaného běhu, ale protože pomocí přepínačů `-s` a `-l` lze omezit prostor hodnot k propočítávání (keyspace), lze téměř jakoukoliv úlohu rozdělit mezi více výpočetních stanic (nodů) [17].

1.3.1.1 VirtualCL

VirtualCL [18] je obecné řešení pro distribuovaný běh jakékoliv aplikace využívající OpenCL. Po instalaci potřebných nástrojů je pak na zvoleném počítači z clusteru možné přistupovat ke všem grafickým kartám dostupným v clusteru tak, jakoby byly nainstalované v počítači fyzicky. Není tak třeba

Tabulka 1.3: Nástroje k prolamování hesel

	John the Ripper[9]	Hashcat[10]	oclHashcat[11]	Multiforcer[12]	Elcomsoft[13]	Passware[14]	Pyrit[15]
počet algoritmů	+	++	++	–	+	+	
možnosti útoků	++	++	++	–	+	+	–
podpora GPU	–	– –	++	+	++	++	+
distribuovaně CPU	+	+*	–	+	+	+	+
distribuovaně GPU	– –	– –	+*	+	+	+	++
multiplatformní	++	++	++	++	– –	–	+
otevřený kód	+	–	–	–	–	–	+
cena	+	+	+	+	–	–	+

v programech, které podporují více grafických karet, dělat jakékoliv změny. Technologie je velmi náročná na odezvu sítě a doporučuje se používat síťové rozhraní Infiniband.

S využitím VirtuaCL v roce 2012 demonstroval J.M. Gosney (na internetu známý jako epixoip) svůj oclHashcat GPU cluster [19] [20], sestavený z pěti výpočetních nodů s celkem 25 GPU. Tento cluster se stal velmi známým a jako první široké veřejnosti předvedl sílu distribuovaného prolamování hesel pomocí grafických karet.

VirtualCL už není doporučenou technologií pro distribuovaný běh Hashcatu [21], navíc OpenCL verze Hashcatu (oclHashcat) je připraven výhradně pro karty AMD/ATI, přestože karty Nvidia OpenCl podporují (pro karty Nvidia se používá cudaHashcat).

1.3.1.2 Disthc

Projekt Disthc [22] umožňuje distribuovaný běh Hashcatu i jeho GPU verze. Architektura řešení je centralizovaná klient-server a k distribuci výpočtu využívá možnosti omezit Hashcat na propočítávání jen části keyspace. K rozběhnutí je nejdříve nutné nakonfigurovat serverovou část včetně vstupních dat k prolamování a dalších podrobností útoku. Na výpočetních klientech je třeba také provést konfiguraci, a to především cestu k Hashcatu a velikost keyspace odpovídající výkonu klienta. Pak je útok možné z centrální konzole spustit a ukončit.

Disthc v současné podobě nepovažuji za dobrý wrapper, pro každou úlohu je nutný zásah do konfigurace na serveru i na klientech, není zde automatické vyvažování zátěže (je třeba to řešit ručním testováním výkonnosti klientů a konfigurací) a centrální ovládání nefunguje spolehlivě.

1.3.1.3 Hashtopus

Relativně novým wrapperem je na začátku roku 2014 představený Hashtopus. Jde opět o architekturu klient-server, kde server je realizován webovou aplikací. Nabídka funkcí je ohromující, uvedu ty hlavní:

- správa přes webové rozhraní
- široké možnosti importu/exportu dat
- podrobné statistiky o průběhu úloh
- vizuální reprezentace rozdělení úloh mezi klienty
- zpracování více úloh najednou
- fronty úloh
- jednotlivé úkolování klientů

- multiplatformní klientská aplikace
- automatické rozdělování zátěže podle výkonnosti klienta

Hashtopus přistupuje k distribuovanému prolamování hesel tak, jak si představuji. Při větším počtu nodů a častém přepínání úloh (což je při prolamování běžné, zkoušíme různé triky, slovníky, pravidla, vylepšujeme zadání úlohy) je možné spravovat celý cluster pohodlně.

1.3.2 Pyrit

Pyrit je specializovaný nástroj na prolamování sdíleného hesla do bezdrátových sítí zabezpečených pomocí WPA/WPA2-PSK. Hlavní předností Pyritu je možnost předpočítat si databázi klíčů dopředu a samotný útok je pak velice rychlý.

1.3.2.1 WPA/WPA2-PSK

Po tom, co bylo zabezpečení WEP k přístupu do bezdrátových sítí několikrát prolomeno, vznikl nový mechanismus WPA. Jednou z možností jak zabezpečit WPA síť, je použitím předem domluveného sdíleného hesla - PSK. Pokud se chce klient připojit k přístupovému bodu, musí se prokázat znalostí sdíleného hesla. K bezpečnému ověření se používá takzvaná čtyřcestná výměna (4way handshake). Během ní se k ověření znalosti hesla a domluvení zabezpečeného komunikačního kanálu používá takzvaný PMK, který se z domluveného hesla odvodí takto:

```
PMK = PBKDF2(HMAC-SHA1, sdílené heslo, název sítě, 4096, 256)
```

(hashovací funkce PBDKF2 se spustí s parametry: operační mód HMAC-SHA1, sdílené heslo, název sítě, 4096 iterací a délkou výstupu 256 bitů). Prolamování sdílených WPA hesel se provádí právě přes PMK, oproti zachyceným síťovým rámcům. Netestuje se přímo PMK, ale z něj odvozená MIC zpráva obsahující mimo jiné i MAC síťových rámců. Celý proces je složitější, pro jeho celého pochopení i s jeho slabinami doporučuji tento článek [23].

Šifrování PBDKF2 je v nastavení použitém k výpočtu PMK velmi silné a přímý útok hrubou silou je i na GPU pomalý. Pyrit umožňuje si PMK předpočítat do databáze. Při samotném prolamování pak dochází k výraznému zrychlení (je vynecháno pomalé hashování) a porovnávají se MIC kódy vypočtené z předpočítaných hodnot oproti zachyceným paketům. Tuto techniku lze přirovnat k předpočítávání Rainbow tabulek a jde o klasický případ Time-Memory Trade-Off [24] (výměna výpočetní složitosti za prostor na disku). Pro každý název sítě (SSID) je nutné předpočítat zvláštní databázi. Název sítě totiž je součástí šifrování, jak je vidět z předpisu.

Hashcat WPA-PSK prolamování také umožňuje, ale nenabízí možnost předpočítat si databázi klíčů.

1.4 Závěr

K implementaci jsem vybral program Hashcat s wrapperem Hashtopus na prolamování hesel a program Pyrit pro prolamování bezdrátových sítí zabezpečených pomocí WPA/WPA2-PSK.

Kapitola 2

Návrh GPU clusteru na FIT ČVUT

V následující kapitole zmapuji výpočetní prostředky fakulty a navrhnu postup jak na nich zprovoznit distribuovaný běh programů Hashcat a Pyrit.

2.1 Dostupný hardware

Zajímají nás především počítače vybavené dedikovanou grafickou kartou AMD či Nvidia, takové jsou v učebnách 350, 351 a ve čtveřici výpočetních nodů na clusteru **star**. Rozeberu konkrétní vybavení počítačů a na závěr přidám celkový přehled grafických karet na fakultě.

2.1.1 Učebna 350 - GPU laboratoř

V učebně je 25 shodných desktop počítačů s následující konfigurací:

Tabulka 2.1: Hardware počítačů učebny 350

Procesor	Intel Core i5-2400
Základní deska	Gigabyte HA65M-UD3H-B3K
Operační paměť	8 GB DDR3
GPU	Gigabyte GTX 480 N560OC-1GI
Pevný disk	500GB SATA III

Počítače jsou zapojené do gigabitové ethernetové sítě. Na síti je dostupný DHCP server, který přiděluje adresy IPv4 sítě 10.3.50.0/24, konkrétní použitý rozsah pro počítače je 10.3.50.100-10.3.50.125. Všechny počítače mají přímý přístup na internet skrze gateway 10.3.50.1. Standardně má po startu uživatel možnost zvolit si mezi operačními systémy na pevném disku (Gentoo Linux a

Microsoft Widnows 7) a síťovým bootovaním. Počítače jsou sestavené v obyčejných desktop skříních, v klidovém stavu (pár minut po startu operačního systému) má grafická karta teplotu přibližně 50 °C, odběr měřený wattmeterem v klidovém stavu je 74 W.

2.1.2 Učebna 351 - Multimediální učebna

Druhá učebna je velmi podobná, konfigurace je následující:

Tabulka 2.2: Hardware počítačů učebny 351

Procesor	Intel Core i5-3470
Základní deska	Intel DQ77MK
Operační paměť	8 GB DDR3
GPU	Gigabyte GTX 560 Ti N560OC-1GI
Pevný disk	500GB SATA III

Stejně jako v učebně 350 se používá gigabitový ethernet, DHCP přiděluje adresy z rozsahu 10.3.51.100 až 10.3.51.124. Počítače také mají přístup na internet, ale přímá komunikace mezi sítěmi učeben 350 a 351 není možná. PC jsou opět sestavené v desktop skříních, klidová teplota GPU je 31 °C a odběr nezatížené sestavy je 52 W.

2.1.3 Cluster star

Ve škole je pro testovaní paralelních úloh připraven HPC cluster **star**, který je tvořen především systémem IBM BladeCenter H, plně obsazeným počítači IBM BladeCenter LS21 a IBM BladeCenter LS22. Ty ovšem nemají výkonnou grafickou kartu a pro náš účel se hodí čtveřice počítačů s Nvidia GPU připojených do clusteru mimo BladeCenter.

Počítače clusteru **star** používají gigabitovou síť 192.168.1.0/24. Na nodech je nainstalován Gentoo Linux a dá se k nim přistupovat přes hlavní uzel (front-end) star.fit.cvut.cz, který je veřejně dostupný z internetu. Nody nemají přístup na internet a ani do jiných sítí v rámci školy. Podobně jako v učebnách jsou nody s GPU obyčejné počítače v desktop casech, počítače jsou fyzicky umístěné ve 13. patře budovy A v menší neklimatizované místnosti.

Konfigurace GPU nodů:

Tabulka 2.3: Počítače clusteru **star** vybavené GPU

Node	GPU	CPU	Paměť
051	Nvidia GeForce GTX 470	Intel Celeron E3400	8GB DDR3
052	Nvidia GeForce GTX 480	Intel Core i5 760	8GB DDR3
053	Nvidia GeForce GTX 590	Intel Core i5 760	8GB DDR3
054	Nvidia Tesla K40	Intel Core i7 950	24GB DDR3

2.1.4 Přehled grafických karet

Na závěr této sekce uvádím souhrnnou tabulku grafických karet 2.4, které máme na fakultě, včetně jejich vlastností. V terminologii Nvidia CUDA jádrem rozumíme jinak běžnější pojem stream procesor. V tabulce neuvádím takt stream procesorů. U karet Nvidia to je v základním nastavení vždy dvojnásobek základního taktu, na kterém karta běží (Core clock).

Z vlastností jednotlivých karet předpokládám nejvyšší výkon u karty Tesla K40, má nejvíce stream procesorů i nejvyšší takt. Porovnání karet v učebnách bude zajímavé, GTX 560 Ti má vyšší takt, ale GTX 480 má zase více CUDA jader.

2.2 Návrh implementace

Jak jsem zmínil v úvodní kapitole, především se na školních počítačích pokusím zprovoznit distribuovaný běh programu Hashcat za pomocí wrapperu Hashtopus a dále pak program Pyrit. Hardwarovou výbavu a síťové zázemí už máme zmapované, teď je čas udělat si představu, jak postupovat při zprovoznění clusteru pro běh vybraných programů v prostředí fakulty.

2.2.1 Hashcat s wrapperem Hashtopus

Architektura tohoto distribuovaného řešení Hashtopus pro Hashcat vyžaduje centrální webový server s přístupem do MySQL databáze, a pak instalaci klientského programu hashtopus-agent do všech počítačů zapojených do clusteru. Ty logicky musí mít možnost se serverem spojit.

Nepředpokládám velké nároky na síťové připojení klientů k serveru, a tedy pokud se mi nepodaří zajistit centrální server přímo ve škole, připravím nějaký server dostupný z internetu. Učebny 350 a 351 by s připojením neměly problém, ale GPU nody clusteru **star** by bylo potřeba nějak protunelovat skrz hlavní uzel.

Celý proces přípravy prostředí klientského prostředí a instalace programu hashtopus-agent by bylo dobré co nejvíce zautomatizovat, stejně tak následný běh agenta. Pro pár počítačů není problém stáhnout, zaregistrovat a spouštět agenta ručně, ale pro těch několik desítek, co máme k dispozici už by to bylo náročné. Autor wrapperu Hashtopus toto neřeší. Šlo by to zajistit prostřednictvím balíků do linuxových distribucí (kromě samotného programu by obsahovaly i obslužnou službu, kterou by administrátor zařadil do systému). V prostředí Windows můžeme situaci řešit podobně, instalačním balíkem a službou. Další možností je vytvořit modul do automatizačních nástrojů jako je například Ansible. Nebo napsat jednoduše nějaký vlastní programem pracující například s SSH.

V implementační části tedy budu postupovat zhruba takto:

Tabulka 2.4: Přehled GPU na fakultě

název	počet	počet CUDA jader	Core clock (Mhz)	Memory clock (Mhz)	Velikost paměti	max TDP
GeForce GTX 470	1	448	607	3348	1280 MB	215 W
GeForce GTX 480	26	480	700	3696	1536 MB	250 W
GeForce GTX 560 Ti	25	382	900	4000	1024 MB	170 W
GeForce GTX 590	1	1024	607	3414	3072 MB	365 W
Tesla K40	1	2880	745	3004	12188 MB	245 W

- zprovoznění Hashcatu na jednotlivých typech počítačů (stačí jeden z 350, jeden z 351 a GPU nody clusteru **star** jednotlivě - GPU jsou různé a SW výbava nemusí být všude stejná)
- příprava webového a databázového serveru
- zprovoznění programu hashtopus-agent na klientských počítačích
- automatizace
- otestování funkčnosti celého clusteru

2.2.2 Pyrit

Jak jsem už zmiňoval, síla programu Pyrit spočívá v možnosti předpočítat si databázi PMK k jednotlivým SSID dopředu, a to distribuovaně s použitím GPU. Taková databáze ale velmi rychle nabírá na objemu. Například pro jednu síť zabere databáze všech 8-místných číselných kombinací přibližně 4GB na disku, ale databáze pro všechna 8 znaková hesla (bez speciálních znaků) by už měla přibližně 2 TB. To už je docela velký objem dat, přesto má smysl Pyrit vyzkoušet. Srovnání s Hashcatem bude zajímavé, pro otestování bude stačit malá databáze a z měření by mělo být jasné, zda se vůbec vyplatí si předpočítávat databázi PMK pro jednotlivá SSID.

Řídící i výpočetní část Pyritu je obsažená v jednom programu. Na výpočetních strojích se pouští Pyrit v serve modu a pak plní příkazy řídícího počítače. Těmi může být přímo hledání hesla pro zachycený WPA handshake a nebo předpočítávání klíčů do databáze. Stačí, když k databázi bude mít přístup pouze řídící počítač, přičemž možností realizace databáze je více. Může to být soubor nebo jakákoliv SQL databáze podporovaná knihovnou SQLALchemy, kterou Pyrit využívá (například SQLite, Postgresql, MySQL).

Na rozdíl od Hashcatu s wrapperem Hashtopus zde lze při předpočítávání databáze předpokládat větší požadavky na připojení, podle výkonnosti grafické karty na výpočetním stroji očekávám tok směrem k řídícímu počítači asi 500-1000 kb/s.

Stejně jako u programu hashtopus-agent by bylo dobré ovládat Pyrit hromadně na všech počítačích, možnosti jsou podobné jako u Hashtopusu. U Pyritu je situace trochu jednodušší v tom, že není potřeba žádná registrace na centrální řídící počítač a na výpočetních strojích ani není třeba žádná konfigurace. Pyrit totiž v serve modu poslouchá na TCP portu a čeká na zadání úlohy.

Postup implementace:

- zprovoznění Pyritu na jednotlivých typech počítačů (opět stačí jeden z 350, jeden z 351 a GPU nody clusteru **star** jednotlivě), tak aby šel spustit benchmark

- výběr centrálního řídícího počítače a druhu databáze
- automatizace
- otestování funkčnosti celého clusteru před měřením

Kapitola 3

Implementace

V této kapitole podrobně rozeberu kroky, které vedly ke zprovoznění distribuovaného běhu programů Hashcat a Pyrit ve školním prostředí.

Prvním rozhodnutím, které jsem udělal, byla volba operačního systému. Oba programy je možné provozovat jak na OS Windows tak v Linuxu. Myslím, že v našem případě je Linux vhodnější. Možnosti skriptování, ovládání procesů, vzdálené správy, logování, přidělování práv a podobně jsou podle mého názoru na Linuxu lepší a ovládám je lépe, ale nepochybuji o tom, že by vybrané programy šly zprovoznit distribuovaně i na Windows.

Toto rozhodnutí se týkalo jen počítačů v učebnách, nody clusteru **star** pracují na Linuxu a není možnost zavádět jiný OS. Hlavní správce ICT infrastruktury FIT ČVUT mi připravil testovací počítač se vzdáleným přístupem (přes SSH) shodný s těmi, jaké jsou v učebně 350, a to i se shodně nainstalovaným operačním systémem Gentoo Linux. Po změnách v operačním systému testovacího počítače potřebných ke zprovoznění clusteru se z této instalace vytvoří síťově bootovatelný image. Ten bude přidán do bootovacího menu na počítačích v učebnách. Počítače učeben 350 a 351 jsou si velmi podobné a nebylo tedy třeba připravovat systém pro učebnu 351 zvlášť.

Snažil jsem programy zprovoznit tak, aby k distribuovanému běhu uživatel nepotřeboval administrátorská práva, v systému tedy budu používat uživatele `nvidia`. Ten je kromě skupiny `users` i ve skupině `video`, a díky tomu má potřebná právo pro přístup k video kartě. Tato práva lze získat i jinak, na Gentoo je toto doporučený postup. V průběhu implementace se ukáže, že přeci jen na nějaké úkony jsou rootovská práva potřeba, a tak pro testování má můj uživatel `nvidia` povoleno vykonávání příkazů jako superuživatel bez hesla pomocí `sudo`. To zajišťuje řádek

```
nvidia ALL=(ALL) NOPASSWD: ALL
```

v souboru **/etc/sudoers**.

3.1 Hashcat s wrapperem Hashtopus

3.1.1 Hashcat

Program Hashcat pro použití s grafickými kartami je šířen ve dvou verzích, oclHashcat pro systémy s GPU od firmy AMD a cudaHashcat pro ty s GPU Nvidia. Hashcat je volně ke stažení v podobě 7zip archivu na stránce projektu [11]. Archiv obsahuje samotný zkompilovaný program pro operační systémy Linux a Windows a jeho pomocné soubory, především zkompilované programy pro grafické karty ve složce kernels. Rozbalený cudaHashcat zabírá na disku přibližně 3 GB a oclHashcat 4 GB.

Jedinou podmínkou pro běh programu cudaHashcat na Linuxu je instalace dostatečně aktuální verze ovladače grafické karty a jeho správné zavedení do Xorg grafického serveru. Toto jsem v učebnových počítačích řešit nemusel, od správců byl v operačním systému správně zaveden ovladač od Nvidia ve verzi 343.33, který je postačující na cudaHashcat verze 1.31 (aktuální v době psaní práce). Na GPU nodech clusteru **star** je ovladač starší (334.21) a nebyla možnost jej aktualizovat. Bylo nutné proto použít starší verzi programu cudaHashcat - 1.30. Pro naše účely rozdíly mezi těmito verzemi nejsou důležité (jde o lepší podporu nových karet GTX 970/980 a přidání několika dalších hashovacích algoritmů).

Pro snížení velikosti síťového image, jsem program nainstaloval na sdílené NFS uložiště do `/mnt/data/nvidia/cudaHashcat-1.31`. Podobně tak na clusteru **star** jsem použil doporučené NFS uložiště `/mnt/data/`.

Hashcat tedy funguje, testovací benchmark pro MD5 na počítači z učebny 350:

```
$ ./cudaHashcat64.bin -m 0 -b
cudaHashcat v1.31 starting in benchmark-mode...

Device #1: GeForce GTX 480, 1535MB, 1401Mhz, 15MCU

Hashtype: MD5
Workload: 1024 loops, 256 accel

Speed.GPU.#1.:  2185.5 MH/s

Started: Tue Feb  3 17:52:00 2015
Stopped: Tue Feb  3 17:52:16 2015
$
```

3.1.2 Ventilátory

Při testování programu Hashcat se ukázalo, že ve výchozím automatickém nastavení se ventilátory grafických karet neroztočí naplno ani při plné zátěži a karty se velmi brzy přehřály (do pár minut karty dosahovaly 90 °C, kdy se Hashcat ve výchozím nastavení z bezpečnostních důvodů ukončí; toto nastavení je určitě rozumné, výrobce u většiny karet uvádí maximální provozní teplotu 95 °C).

U grafických karet Nvidia lze upravovat parametry karty pomocí programu `nvidia-settings`. Aby by bylo možné ovládat ventilátor, je třeba v konfiguraci Xorg serveru takové změny povolit přidáním volby `Coolbits 4` (nebo 5, to navíc povolí i změny taktu) do sekce `Device`. Na našem Gentoo systému tedy `/etc/X11/xorg.conf.d/10-nvidia.conf` upravíme, aby vypadal takto:

```
Section "Device"
Identifier  "Card0"
Driver      "nvidia"
BusID       "PCI:1:0:0"
Option      "Coolbits" "4"
EndSection
```

Pak už lze ventilátor ovládat, u našich jednokartových systémů se nastaví ventilátor na plný výkon takto:

```
$ sudo nvidia-settings -a [gpu:0]/GPUFanControlState=1 \
-a [fan:0]/GPUCurrentFanSpeed=100 -c :0

Attribute 'GPUFanControlState' (pc350-199:0[gpu:0]) assigned value 1.

Attribute 'GPUCurrentFanSpeed' (pc350-199:0[fan:0]) assigned value 100.

$
```

a vrácení zpět na automatiku:

```
$ sudo nvidia-settings -a [gpu:0]/GPUFanControlState=0 -c :0

Attribute 'GPUFanControlState' (pc350-199:0[gpu:0]) assigned value 0.

$
```

Při plné zátěži karty s ventilátorem nastaveným na 100% výkonu se teplota po několika minutách ustálí okolo 83 °C, což je přijatelné. Je třeba si uvědomit, že karty jsou nainstalované v obyčejných uzavřených desktop skříních. Teplotu by bylo možné ještě snížit o několik stupňů zajištěním lepšího proudění vzduchu ve skříni.

V učebnách 350 i 351 je několik počítačů (celkem 4 z 50), které i přes plný výkon ventilátorů po několika desítkách minut dosáhnou 90 °C. Příčinou je pravděpodobně odcházející ventilátor nebo příliš prachu v počítačových skříních.

V GPU nodech clusteru **star**, kde nemám možnost spouštět programy s vyššími právy, se mi nepodařilo ventilátory ovládat. Počítač s kartu Tesla problém s přehříváním nemá, zbylé tři počítače s GPU z clusteru **staru** ano.

3.1.3 Hashtopus server

Distribuované řešení Hashtopus pro Hashcat je šířeno jako 7z archiv na stránkách autora [25] a jde o opensource s licencí GNU GPL a všechny části jsou zveřejněné i na autorově GitHub stránce [26].

Serverová část je napsána v jazyce PHP a jako datové uložiště používá MySQL databázi. Přesné požadavky jsou:

- web server (doporučený je Apache 2) s nakonfigurovaným PHP (verze 5.3 a vyšší)

- právo zápisu pro systémového uživatele spouštějícího PHP skripty do webového adresáře `hashtopus` s instalací
- MySQL server (verze 5 a vyšší) s právem zápisu do adresáře webového adresáře `hashtopus` s instalací
- MySQL uživatel vybraný pro práci s databází musí mít k databázi všechna práva a navíc i FILE právo zápisu do webového adresáře `hashtopus` s instalací

Poslední požadavek implikuje, že MySQL server by měl být na stejném stroji jako webový server. Při prostudování exportní funkce (řádky 2020 až 2052 v `admin.php`) se to opravdu potvrzuje. Tento požadavek lze vynechat, ale nebude fungovat export hashlistů z databáze do souboru. Řešení exportu přímo z databáze pomocí INSERT INTO FILE nepovažuji za vhodné. Exportní funkci by šlo celkem jednoduše opravit, ale to nechám na autorovi, se kterým jsem tento problém diskutoval.

Ve škole je pro studenty i zaměstnance připraven webový server **webdev.fit.cvut.cz** s možností PHP skriptování, kde je pro studenty spuštěn i MySQL server. Webovou část jsem na tomto serveru úspěšně otestoval, verze všech komponent byly připravené v dostatečných verzích. Databázi se mi v době psaní práce ve škole nepodařilo zprovoznit, aplikace pro vytváření databázových přístupů nebyla dostupná a navíc by velikost databáze byla omezená na 20 MB, což by pro delší testovací soubory nestačilo. Rozhodl jsem se tedy pro vlastní server, ale je jisté, že na školním webovém serveru s databází by serverová část fungovala korektně.

Můj server je obyčejné novější PC, připojení na internet rychlostí 20 Mb download i upload, ping ze školních počítačů na server pod 3ms. Webový server byl v době psaní práce dostupný z internetu na **http://redash.cz:11711**. Jako operační systém na serveru používám aktuální linuxovou distribuci Debian 7 Wheezy. Na server jsem nainstaloval balíčky `apache2`, `mysql-server`, `php5`, `php5-mysql` a `php5-gd`. Kromě administrátorského hesla k databázi není třeba nic dalšího konfigurovat, výchozí nastavení je vyhovující.

Pokračoval jsem instalací programu Hashtopus podle návodu [27]:

- vybalil jsem stažený archivu do webového adresáře `hashtopus122`
- v MySQL serveru jsem vytvořil uživatele hashtopus a databázi `hashtopus122`
- přidělil jsem databázovému uživateli hashtopus veškerá práva k databázi

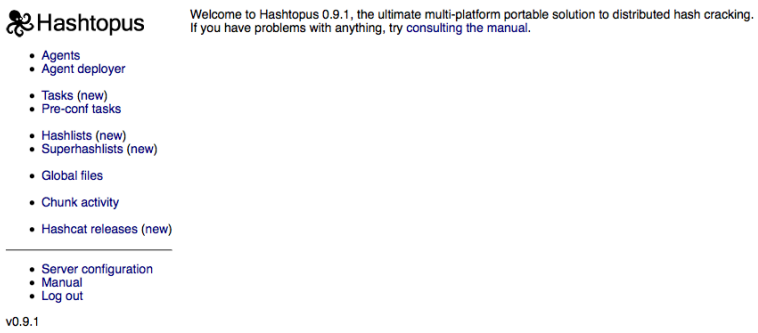
```
GRANT ALL PRIVILEGES ON hashtopus122.* TO 'hashtopus'@'localhost';
```

- do databáze jsem naimportoval obsah souboru `hashtopus.sql`

```
$ mysql -u hashtopus -p -h localhost hashtopus122 < hashtopus122.sql
```

- do souboru v `dbconfig.php` jsem vyplnil přístupové údaje k databázi
- na **http://redash.cz:11711/hashtopus122/admin.php** jsem změnil administátorské heslo a krátce otestoval, že web funguje v pořádku
- doinstaloval jsem javascriptový program **JSColor** [28] pro pohodlné vybírání barev na označování úloh, tj. vybalil jsem archiv do webového adresáře s instalací `hashtopus`

Serverová část je v tuto chvíli připravená a administrační rozhraní je dostupné na **http://redash.cz:11711/hashtopus122/admin.php**. Popis webového rozhraní a návod k použití je dostupný ve webovém rozhraní nebo na stránkách autora [27].



Obrázek 3.1: Hashtopus

3.1.4 Hashtopus-agent

Nyní se zaměříme na klientskou část - program hashtopus-agent. Instaluje se do počítačů s grafickou kartou, má nasledující požadavky:

- **.NET 2.0** kompatibilní prostředí (v našem případě na linuxu tedy **Mono 2.0**)
- TCP připojení na server
- prostředí připravené pro běh programu Hashcat

Do operačního systému pro učebny jsem tedy doinstaloval `mono` balíčkovým systémem (`emerge dev-lang/mono`) a o totéž jsem požádal administrátory clusteru **star**. Počítače v učebnách mají přímý přístup na internet, s připojením na server Hashtopus (**http://redash.cz:11711/hashtopus122**) tedy nebude problém. U nodů clusteru **star**, jsem na hlavním uzlu spustil SSH tunel:

```
ssh -g -N -f -L 11711:redash.cz:11711 samekja2@localhost
```

který pro GPU nody zajistí přesměrování. Pokud se tedy některý z GPU nodů připojí na **http://star:11711/hashtopus122**, dostane se na Hashtopus server.

Samotného agenta máme podle návodu stáhnout přímo na klientu z administračního webového rozhraní - v menu **Agent deployer** položka **Download agent**. Důvodem je to, že během požadavku na stažení se do `hashtopus.exe` na konec souboru zapíše URL serveru, na který se potom agent připojuje. Nepovažuji tento postup „konfigurace“ serverového hostitele za vhodný, později v práci ukážu alternativní postup.

Nově nainstalovaný agent se při prvním běhu potřebuje zaregistrovat na server. Pro registraci se používají vouchery, které lze generované z administračního portálu v menu **Agent deployer**. Po registraci agent dostane zpět autorizační token pro příští přihlášení (soubor `hashtopus.token`). Po zadání voucheru se agent pokusí stáhnout vhodnou verzi Hashcatu a pak už čeká na přidělení úlohy. Ukázkový první běh agenta:

```
nvidia@pc350-100 ~/hashtopus $ mono hashtopus.exe
Hashtopus 0.9.1
Registering to server...Enter registration voucher: iNXRV
OK.
Logging in to server...OK.
Creating tasks directory...OK
Loading task...failed: No active tasks.
Waiting for next assignment...
Waiting for next assignment...
Waiting for next assignment...
```

a ve webovém rozhraní si můžeme zobrazit detail připojeného klienta agenta:

Agent details:

Property	Value
ID:	111
Activity:	☑ Agent is active
Machine name:	lab350-100
Operating system:	
Access token:	l6jqoVpu3G
Machine ID:	BADID_969995
CPU platform:	64-bit
GPU platform:	NVidia Set
GPU driver:	34322
Graphic cards:	NVIDIA Corporation GF100 [GeForce GTX 480] (rev a3)
Hashcat version:	1.31
Idle wait:	0 seconds Set
Extra parameters:	Set
Hashcat errors:	☐ Ignore *(Be careful, this might lead to endless loops!)*
Trust:	☑ Trust agent with secret data
Assignment:	(unassigned) Assign
Last activity:	Action: task Time: 03.02.2015, 18:39:24 IP: 147.32.232.134
Time spent cracking:	00:16:03

Obrázek 3.2: Detail připojeného agenta

Poznámka: snímek obrazovky jsem dělal až po některých z měření, nezobrazuje čistý stav po registraci agenta.

3.1.5 Automatizace

V tuto chvíli máme celý systém připravený. Když se na každém počítači stáhne, spustí a zaregistruje agent, je možné všechny najednou ovládat z webového rozhraní. Vzhledem k počtu strojů by bylo vhodné tento proces automatizovat.

Rekapitulace nutných úkonů ke spuštění programu hashtopus-agent na klientských počítačích:

- stažení agenta - doporučené je ho stáhnout z webového administračního rozhraní serveru, tím se do spustitelného souboru dostane jeho URL, kam se bude agent připojovat. Alternativně lze stáhnout `hashtopus.exe` přímo z **http://server/hashtopus/hashtopus.exe** a URL server přidat manuálně, v Bashi:

  ```
  echo -n "http://server/hashtopus/server.php" >> hashtopus.exe
  ```

- při prvním spuštění odbavit registraci agenta na server
- ovládat jeho běh, tedy možnost ho spustit i ukončit (včetně procesů, které spustil)

Při rozhodování, jak automatizaci realizovat, jsem uvažoval následující možnosti:

- instalační balíček do linuxových distribucí, včetně obslužné služby a konfiguračního souboru
- Ansible Playbook [29]
- vlastní program - Bash skript

Instalační balíček se službou

Podobně jako se instalují jiné služby do linuxových systémů (třeba MySQL server), balíček by:

- zajistil závislosti
- provedl uživatele konfigurací - zadáním serveru (podle toho by se upravil spustitelný soubor), zadáním voucheru pro registraci na server a případně nastavením cest.
- stáhnul a nainstaloval agenta podle konfigurace

Pak už by stačilo jen spouštět službu. Bylo by možné dát ji mezi služby po spuštění počítače a nebo ji spouštět hromadně vzdáleně - třeba pomocí jednoduchého skriptu přes `ssh` nebo použitím `parallel-ssh`.

Nevýhody:

- bez dalšího skriptování nelze proces instalace a registrace udělat bezobslužně (musí se zadávat voucher)
- spouštět služby a instalovat balíky může většinou pouze administrátor
- bylo by dobré připravit balíky pro hlavní distribuce

Výhody:

- je to systémové řešení

Ansible Playbook

Pro pohodlnou správu služeb, instalaci aplikací, nasazování informačních systémů, sestavování celých operačních systémů a celou řadu dalších věcí se dnes velmi často používají automatizační nástroje. Známé a rozšířené jsou například Puppet [30], Chef [31] a Ansible [32]. Z nich považuji Ansible za nejvíce lightweight - funguje přes SSH, což je také jeho jedinou závislostí k použití.

Ansible je možné použít nejen ke spouštění jednoduchých příkazů na vzdálených hostitelích, ale je možné pomocí něj automatizovat daleko složitější úkony. Pěkně to ilustruje tento návod [33], kde autor postupně představuje jednotlivé částí Ansible na instalaci webového serveru. Výsledkem je tzv. Playbook, který webový server nainstaluje na daných hostitelích včetně požadované konfigurace jako umístění webových adresářů, nastavení práv a uživatelských rolí k jeho ovládání. Ansible Playbook si lze zjednodušeně představit jako scénář příkazů ke spuštění na vzdáleném stroji.

Při testování a psaní vlastních jednoduchých úloh, modulů a Playbooků jsem pochopil, že možnosti takové automatizace jsou opravdu velké a zcela jistě by bylo možné napsat Playbook pro instalaci a ovládání programu hashtopus-agent. Na druhou stranu jsem také nabyl dojmu, že v našem případě by Playbook byl buď složitý na konfiguraci (hlavně bez předchozích znalosti fungování Ansible), nebo by se část výhod automatizačního nástroje ztratila (nutnost práce s vouchery pro registraci, externí skripty pro ovládání ventilátorů a podobně).

Nevýhody:

- je třeba se s Ansible naučit pracovat i pro ovládání
- složitost konfigurace a vlastního programu

Výhody:

- je to systémové řešení
- jediný požadavek na klienty je `ssh` přístup
- konfigurace všech parametrů pro klienty z jednoho místa

Vlastní program - bash skript

Všechny části automatizace je poměrně snadné řešit skriptováním.

Nevýhody:

- není to systémové řešení
- pro korektní běh je třeba upravit agenta (dokud autor projektu změny nezařadí do hlavní verze)

Výhody:

- nejsou potřeba administrátorská práva
- jediný požadavek na klienty je nainstalované `mono` a `ssh` přístup
- jednoduchá konfigurace všech parametrů pro klienty z jednoho místa

Všechny z probraných možností jsou dobré a lze jimi dosáhnout řešení. Rozhodl jsem se pro vlastní program.

3.1.6 Hashtopusbyssh

Jak již bylo uvedeno, rozhodl jsem se pro napsání vlastního programu. Kromě jeho hlavní funkce, tedy bezobslužné instalace o ovládání běhu programu hashtopus-agent, jsem chtěl, aby pro jednotlivé klienty bylo možné:

- určit SSH uživatele a port pro připojení
- určit cestu, kam se hashtopus-agent stáhne a odkud se bude spouštět
- určit zda, se má hashtopus-agent pokusit o stažení Hashcatu
- použít už existující instalaci Hashcatu v systému
- zvolit příkazy ke spuštění před startem a po ukončení programu hashtopus-agent
- odinstalovat hashtopus-agent

Program je napsaný pro interpret příkazů BASH. Jediný externí program, který nemusí být dostupný na všech linuxových systémech a je použit ve skriptu, je `wget`. Ve všech běžných linuxových distribucích je přítomen v základní instalaci nebo ho lze jednoduše doinstalovat. Skript je dostupný veřejně na [34] a je uvolněn pod licencí GPL v2, jeho kopie je i na přiloženém CD.

V této kapitole proberu konfiguraci, popis použití a detaily pro běh ve školním prostředí.

3.1.6.1 SSH

Jak již název programu napovídá, pro připojení k výpočetním uzlům budu používat SSH. Aby nebylo nutné pro jednotlivé stroje učeben zadávat heslo při každém úkonu, využil jsem možností autentifikace pomocí klíčů. Protože všechny počítače učeben budou používat stejný image operačního systému, stačí přidat vygenerovaný veřejný klíč do souboru s autorizovanými klíči `/home/nvidia/.authorized_keys`.

Na počítačích clusteru **star** je bezheslové přihlašovaní pomocí SSH už připravené.

3.1.6.2 Konfigurace

Program se konfiguruje dvěma soubory: hlavním konfiguračním souborem, kde se především nastaví cesta k serveru a výchozí hodnoty pro klienty a pak souborem definujícím jednotlivé klienty, kde poběží hashtopus-agent. Ukázkové konfigurační soubory i s popisem direktiv jsou dostupné v příloze na CD i na GitHub stránce programu.

Hlavní konfigurační soubor

Hlavní konfigurační soubor je textový soubor, na každém řádku může být maximálně jedna volba, syntaxe je `název volby` mezera `hodnota`. Seznam možných voleb je v tabulce 3.1. Na pořadí nezáleží, neplatné volby jsou ignorovány a je povolené komentování znakem # kdekoliv na řádku.

Direktivy `sshusername`, `sshport`, `hashtopuspath`, `oclhashcatpath`, `cudahashcatpath` a `hashcatdownload` mohou být uvedeny i v konfiguračním souboru jednotlivých klientů, kde mají vyšší prioritu.

V učebnách jsem tedy například používal tento konfigurační soubor:

```
$ cat hashtopusbyssh.conf
hashtopusserver http://redash.cz:11711/hashtopus122
cudahashcatpath /mnt/data/nvidia/cudaHashcat-1.31
hashcatdownload no
voucherfile /mnt/data/nvidia/configs/hashtopus.vouchers
$
```

Konfigurace klientů

Tabulka 3.1: Hlavní konfigurační soubor

Direktiva	typ	popis	výchozí hodnota
`sshusername`	řetězec	uživatelské jméno pro ssh spojení	uživatel, který spouští skript
`sshport`	číslo	port pro ssh spojení	22
`hashtopusserver`	URL	URL hashtopus serveru	`http://localhost/hashtopus`
`hashtopuspath`	cesta	cesta k instalaci hashtopus agenta	`hashtopus`
`oclhashcatpath`	cesta	cesta k OpenCL Hashcat na hostiteli	`oclHashcat-1.31`
`cudahashcatpath`	cesta	cesta k CUDA Hashcat na hostiteli	`cudaHashcat-1.31`
`hashcatdownload`	yes/no	má se agent pokoušet o stažení programu Hashcat?	yes
`hashtopuslog`	cesta	soubor, kam bude přesměrován výstup agenta	`hashtopus.log`
`hashtopuspdig`	cesta	soubor s číslem procesové skupiny spuštěného agenta	`hashtopus.pgid`
`voucherfile`	cesta	soubor s registračními vouchery	`hashtopus.vouchers`

Konfigurační soubor pro jednotlivé klienty je opět textový soubor. Používá podobnou syntaxi jako soubor **/etc/passwd**, na každém řádku jeden hostitel a uvádí se rovnou hodnoty parametrů oddělené dvojtečkou. Stejně jako v hlavním konfiguračním souboru je povoleno komentování.

Význam sloupců:

1. hostitel (IP adresa nebo hostname)
2. SSH uživatel
3. SSH port
4. cesta k adresáři programu hashtopus-agent (případně kam se má stáhnout)
5. cesta k adresáři programu Hashcatu na hostiteli
6. výrobce GPU karty (ati či nvidia)
7. verze driveru (důležité jen u karet AMD)
8. volba zda stahovat Hashcat (yes nebo no)
9. soubor s příkazy, které se mají spustit před startem agenta
10. soubor s příkazy, které se mají spustit po ukončení agenta
11. poznámky

Povinné parametry jsou jen hostitel a výrobce GPU karty, ostatní se v případě absence nastaví podle hlavního konfiguračního souboru nebo na výchozí hodnotu. Oddělovače hodnot (dvojtečky) musí být vždy všechny.

3.1.6.3 Spouštění a příkazy

Program se spouští z příkazové řádky interpretu BASH a jeho chování se ovládá přepínači:

Tabulka 3.2: Přepínače hashtopusbyssh

Přepínač	argument	popis
-h		zobrazí krátkou nápovědu
-c	příkaz	příkaz, co se má provést (`start`, `stop`, `cleanup`)
-C	soubor	hlavní konfigurační soubor programu
-H	soubor	konfigurační soubor jednotlivých klientů

Spuštění programu hashtopus-agent na počítačích z konfigurace `hosts.conf` a hlavním konfiguračním souborem `hashtopusbyssh.conf` by tedy vypadalo takto:

```
$ hashtopusbyssh -c start -C hashtopusbyssh.conf -H hosts.conf
```

Následuje popis průběhu jednotlivých příkazů.

start

Program se po zpracování konfiguračních souborů v cyklu připojuje ke vzdáleným klientům přes `ssh` a postupně provede:

- pokud se mají na klientu spustit vlastní příkazy ze souboru, tak se spustí
- kontrolu přítomnosti a případně vytvoření pracovního adresáře pro program hashtopus-agent
- kontrolu přítomnosti Hashcatu na vzdáleném stroji
- pokud v pracovním adresáři programu hashtopus-agent není adresář `hashcat`, tak vytvoří symbolický link na již ověřenou cestu Hashcatu
- stažení `hashtopus.exe`
- v případě, že počítač má GPU značky AMD, tak skript vytvoří pomocný soubor `catalyst_ver.txt` s verzí ovladače
- pokud neexistuje přístupový token na server (soubor `hashtopus.token`), tak se skript pokusí o registraci. Skriptu je možné nadefinovat seznam voucherů, které může používat pro registraci agentů na server.
- po úspěšné registraci smazání použitého voucheru
- spuštění agenta, jeho výstup je přesměrován do logovacího souboru a do pomocného souboru se zapíše číslo procesové skupiny agenta (důležité pro zastavení).

stop

Podobně jako při startu se program připojuje ke klientům a postupně provádí:

- pokud existuje soubor s číslem procesové skupiny, tak celou skupinu procesů ukončí a soubor smaže
- pokud se mají na klientu provést příkazy po ukončení agenta, tak se provedou

cleanup

Z hostitelů podle jejich konfigurace odebere celý adresář s instalací programu hashtopus-agent.

3.1.6.4 Běh v učebnách

Pro jednotlivé skupiny počítačů na fakultě jsem si připravil hlavní konfigurační soubory i konfigurace hostitelů, všechny jsou dostupné na přiloženém CD a jejich názvy napovídají obsah. Například soubor `lab350-5.hosts` obsahuje konfiguraci pro 5 počítačů učebny 350. Pro učebny 351 a 350 používám shodný hlavní konfigurační soubor `hashtopusbyssh-lab.conf`. Pro počítače clusteru **star** používám pozměněný soubor `hashtopusbyssh-star.conf` z důvodu potřeby tunelovat spojení na server. Protože jsou všechny tři sítě oddělené, je třeba skript pouštět pro každou síť zvlášť z počítače v té síti. Pro počítače clusteru **star** jsem skript spouštěl z hlavního uzlu, pro počítače z učebny 350 z přípravného `pc350-199`, kam mám vzdálený přístup. Učebna 351 je jediná kam bez vzdáleného přístupu, a proto po nastartování počítačů (což je stejně nutné dělat fyzicky) jsem si z jednoho udělal reverzní `ssh` tunel:

```
$ ssh -fN -R 22222:localhost:22 samekja2@star.fit.cvut.cz
```

čímž jsem již mohl skript v učebně 351 také spouštět vzdáleně.

Před během jsem na serveru přidal dostatečné množství registračních voucherů, které se pak použijí pro registrace. Také využívám volby spustit vlastní příkazy před během programu hashtopus-agent a po jeho ukončení na ovládání ventilátorů - soubory `fanTo100` a `fanToAuto` obsahující příkazy z 3.1.2 Ventilátory.

Po příkazu `start` v obou učebnách jsou tedy všechny klientské počítače připravené k plnění úloh, což zkontrolujeme ve webovém rozhraní 3.3.

V počítačích pracovní adresář programu hashtopus-agent vypadá takto:

```
nvidia@lab350-107 ~/hashtopus $ ls -al
total 48
drwxr-xr-x  4 nvidia users   200 Feb  3 18:33 .
drwxr-xr-x 15 nvidia users   180 Feb  3 18:28 ..
drwxr-xr-x  1 nvidia users    80 Feb  3 18:33 files
lrwxrwxrwx  1 nvidia users    33 Feb  3 18:28 hashcat -> /mnt/data/nvidia/cudaHashcat-1.31
drwxr-xr-x  3 nvidia users    80 Feb  3 18:33 hashlists
-rw-r--r--  1 nvidia users 35374 Feb  3 18:28 hashtopus.exe
-rw-r--r--  1 nvidia users   443 Feb  3 18:33 hashtopus.log
-rw-r--r--  1 nvidia users     5 Feb  3 18:33 hashtopus.pgid
-rw-r--r--  1 nvidia users    10 Feb  3 18:28 hashtopus.token
drwxr-xr-x  2 nvidia users    60 Feb  3 18:29 tasks
nvidia@lab350-107 ~/hashtopus $
```

3.1.6.5 Změny v hashtopus-agent

Pro účely mého skriptu jsem upravil program hashtopus-agent (je uvolněný též pod licencí GPL). Protože český autor projekt zveřejňuje projekt na serveru GitHub, jsou možnosti forkování a úprav velice pohodlné. Je možné, že se změny časem objeví v hlavní větvi zdrojového kódu.

Hashtopus

- Agents
- Agent deployer
- Tasks (new)
- Pre-conf tasks
- Hashlists (new)
- Superhashlists (new)
- Global files
- Chunk activity
- Hashcat releases (new)

- Server configuration
- Manual
- Log out

v0.9.1

List of agents (52):

id	Act	Name	OS	CPU	GPU brand	Driver	GPUs	Hashcat	Last activity	Assignment
111	☑	lab350-100		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	task at 03.02.2015, 18:39:24 IP: 147.32.232.134	(unassigned)
112	☑	lab350-101		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	solve at 03.02.2015, 18:39:24 IP: 147.32.232.134	(unassigned)
113	☑	lab350-102		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	down at 03.02.2015, 18:39:28 IP: 147.32.232.134	(unassigned)
114	☑	lab350-103		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	task at 03.02.2015, 18:39:29 IP: 147.32.232.134	(unassigned)
115	☑	lab350-104		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	down at 03.02.2015, 18:39:29 IP: 147.32.232.134	(unassigned)
116	☑	lab350-105		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	chunk at 03.02.2015, 18:39:30 IP: 147.32.232.134	(unassigned)
117	☑	lab350-106		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	task at 03.02.2015, 18:39:30 IP: 147.32.232.134	(unassigned)
118	☑	lab350-107		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	chunk at 03.02.2015, 18:39:31 IP: 147.32.232.134	(unassigned)
119	☑	lab350-108		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	chunk at 03.02.2015, 18:39:31 IP: 147.32.232.134	(unassigned)
120	☑	lab350-109		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	down at 03.02.2015, 18:39:32 IP: 147.32.232.134	(unassigned)
121	☑	lab350-110		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	chunk at 03.02.2015, 18:39:32 IP: 147.32.232.134	(unassigned)
122	☑	lab350-111		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	task at 03.02.2015, 18:39:33 IP: 147.32.232.134	(unassigned)
123	☑	lab350-112		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	task at 03.02.2015, 18:39:33 IP: 147.32.232.134	(unassigned)
124	☑	lab350-113		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	chunk at 03.02.2015, 18:39:34 IP: 147.32.232.134	(unassigned)
125	☑	lab350-114		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	task at 03.02.2015, 18:39:34 IP: 147.32.232.134	(unassigned)
126	☑	lab350-115		64-bit	NVidia	34322	NVIDIA Corporation GF100 [GeF...	1.31	chunk at 03.02.2015, 18:39:35 IP: 147.32.232.134	(unassigned)

Obrázek 3.3: Hashtopus - agenti

Mezi spouštěcí argumenty programu hashtopus-agent jsem přidal argument `nodownload`, který vyřadí z programu funkci stahování programu Hashcat. Funkce samo-stahování Hashcatu je nespolehlivá, při testování jsem několikrát narazil na různé chyby, proč se stažení či nějaký následný proces přípravy nezdařil. Navíc v mém testovacím školním prostředí je vhodnější, aby Hashcat byl na sdíleném uložišti a do adresáře hashtopus-agenta se pouze linkoval. Systémový image je díky tomu menší a navíc například změna verze stačí udělat na jednom místě.

Druhou změnou je rozšíření způsobu registrace na server. Pokud v adresáři, odkud se spouští hashtopus-agent, existuje soubor `hashtopus.voucher`, je jeho obsah použit jako registrační voucher. Motivací pro toto rozšíření byla jednak nemožnost předávat voucher nějakým parametrem či souborem a také jednodušší možnost kontroly, zda se registrace podařila. Po úspěšné registraci hashtopus-agent soubor s voucherem smaže, pokud se však registrace nezdařila, soubor zůstává na svém místě. Díky této úpravě je tedy možná bezobslužná registrace na server.

Samozřejmě, nutnost manuálního zadání voucheru lze řešit i jinak, například:

```
$ echo -n "voucher" | mono hashtopus.exe
```

a úspěšnost registrace by se potom dala vyčíst analyzováním výstupu agenta nebo vhodným dotazem na server.

Poslední změnou je vyřazení části funkce `diagnoseSystem()`, kde se hashtopus-agent snaží čtením `/proc/mounts` zjistit jednoznačný identifikátor systému. Testoval jsem agenta na několika systémech a na většině z nich tato funkce vedla k padání programu (Segmentation fault). Přímo v hashtopus-agent je zabudované chování, že když se nepodaří identifikátor zjistit, tak se nějaký vygeneruje. Funkci se jsem tedy jen poupravil tak, že se identifikátor bude generovat vždy.

Přesné změny lze zobrazit na GitHubu [35]

Agenta jsem pak zkompiloval v Linuxu příkazem

```
$ xbuild /p:Configuration=Release hashtopus.csproj
```

a výsledný `hashtopus.exe` jsem nakopíroval do webové složky.

3.1.7 Další pomocné skripty

V průběhu implementace jsem napsal další dva jednoduché skripty, jsou to spíše jednoúčelové programy pro školní prostředí, přesto se hodí je zmínit. Oba jsou dostupné na CD a mém GitHubu mezi Gists [36].

3.1.7.1 sethostname

Při přípravě obrazu operačního systému pro učebny jsem zapomněl na správné nastavení názvu jednotlivých počítačů. Funkčně to ničemu nevadí, konfigurační soubor pro `hashtopusbyssh` jsem psal s IP adresami, ale Hashtopus

používá název počítače jako název agenta ve svém webovém rozhraní a v seznamu agentů jsem tedy po první registraci agentů učebny 350 měl 25x agenta s názvem FIT-PC.

Program `sethostname` má dva parametry:

- základ názvu počítače
- soubor se seznamem počítačů (se stejnou syntaxí jako mají host listy pro `hashtopusbyssh`)

a pracuje tak, že se připojí k hostiteli, nastaví mu hostname ve tvaru `základ z parametru-poslední část jeho IP adresy`. Například

```
$ ./sethostname lab351 lab351-25.hosts
```

(`lab351-1.hosts` je seznam počítačů) nastaví jména počítačů pro učebnu 351 na jejich obvyklá jména `lab351-xxx`. Po změně jména se ještě restartuje Xorg server (jinak se ztratí práva přístupu k Xorg serveru).

3.1.7.2 fetchtemp

Při testovacích měřeních se i přes nastavení ventilátorů na plný výkon některé stroje přehřívaly. Chtěl jsem monitorovat, za jak dlouho vychladnou pro další test, případně jak rychle stoupá teplota. Skript `fetchtemp` s jediným parametrem - souborem s hostiteli se stejnou syntaxí jako je seznam klientů pro `hashtopusbyssh`. Program se postupně ke všem počítačům připojí a vypíše teplotu.

Ukázka běhu:

```
$ ./fetchtemp lab350-5.hosts
10.3.50.100: 77
10.3.50.101: 83
10.3.50.102: 82
10.3.50.103: 79
10.3.50.104: 79
$
```

3.1.8 Návrhy na zlepšení

Hashtopus se z uživatelského hlediska používá velmi pohodlně, přehledná správa agentů clusteru přes webové rozhraní funguje dobře, stejně tak zadávání úloh i kontrola jejich průběhu. Přesto má Hashtopus několik menších chyb a nedostatků, které uvedu. Jsem přesvědčen, že v dalších verzích bude většina z nich vyřešena.

- Při importu souborů se vždy do databáze uloží jeden prázdný řádek navíc. Funkčně s tím není žádný problém, ale zobrazování počtu hashů je pak na všech místech o 1 zvětšené.

- Problém s importem dlouhých hashů. Při importování dlouhých hashů, například hashe klíčenky 1Password [37], dojde k chybě. Pravděpodobně jde o chybně nastavený databázový sloupec nebo proměnnou v aplikaci.

- Není možné ovládat celé skupiny agentů. Zadání úlohy pro 50 nodů tedy znamená 50 úkonů ve webové aplikaci. Při testování jsem připravil patch přidávající možnost hromadného zadávání úloh.

- Možnosti tvoření front pomocí priority hashlistů nefunguje správně.

- Automatické stahování Hashcatu nefunguje na linuxových systémech správně.

Celkově je na Hashtopusu vidět, že ještě není dokončen a s každou další verzí dochází k opravám a vylepšením. To hlavní, tedy rovnoměrné rozložení výkonu mezi agenty při prolamování hesel funguje velmi dobře.

3.2 Pyrit

Pyrit je opensource program psaný v jazyce Python a jeho zdrojové kódy jsou dostupné na stránce projektu [15] . Kromě samotného Pyritu je pro akceleraci výpočtů pomocí grafických karet Nvidia nutné ještě nainstalovat python modul cpyrit-cuda, dále pro analýzu zachycených paketů je třeba knihovna Scapy a pro přístup k SQL uložištím knihovna SQLAlchemy.

Pyrit jsem nejdříve pro testování nainstaloval na clusteru **star**. Tam nemám práva roota, nemohl jsem tedy využít balíčkového systému (pyrit i cpyrit-cuda jsou v Gentoo dostupné v poslední verzi). Pyrit a potřebné knihovny jsem tedy nainstaloval ručně zkompilováním ze zdrojových kódů na sdílené uložiště `/mnt/data/samekja2/local`.

Testovací běh pyritu na `node-051`:

```
$ /mnt/data/samekja2/local/bin/pyrit benchmark
Pyrit 0.4.0 (C) 2008-2011 Lukas Lueg http://pyrit.googlecode.com
This code is distributed under the GNU General Public License v3+

Running benchmark (20514.4 PMKs/s)... \

Computed 20514.36 PMKs/s total.
#1: 'CUDA-Device #1 'GeForce GTX 470'': 22023.8 PMKs/s (RTT 2.8)
#2: 'CPU-Core (SSE2)': 291.5 PMKs/s (RTT 3.0)
#3: 'Network-Clients': 0.0 PMKs/s (RTT 0.0)
$
```

V učebnách jsem Pyrit neinstaloval z balíčkovacího systému přímo do systému, ale program jsem si kompiloval sám na sdílené uložiště. Kvůli problémům s distribuovaným během jsem zkoušel poslední verzi Pyritu ze SVN, různé patche od uživatelů a podobně, takže to bylo i praktičtější.

3.2.1 Distribuovaný běh

Architekturou jde opět o centralizovaný systém, ale používat označení server pro centrální bod a klient pro výpočetní nody je v případě Pyritu nevhodné. Na výpočetních strojích, které budou poskytovat svou grafickou kartu, je třeba v konfiguračním souboru `~/.pyrit/config` nastavit volbu `rpc_server true` a pak pustit Pyrit v serve módu:

```
$ /mnt/data/samekja2/local/bin/pyrit serve
Pyrit 0.4.0 (C) 2008-2011 Lukas Lueg http://pyrit.googlecode.com
This code is distributed under the GNU General Public License v3+

Serving 0 active clients; 0 PMKs/s; 0.0 TTS
```

Počítač čeká na zadání úlohy na TCP portu 17935. Na výpisu je vidět, že v terminologii Pyritu jsou servery výpočetní uzly a klientem je počítač, který se k serverům připojuje a zadává jim práci.

Na počítači, který bude řídit výpočet, je třeba v konfiguračním souboru `~/.pyrit/config` do řádku `rpc_knownklients` vypsat mezerou oddělené adresy vzdálených počítačů, jejichž GPU se bude používat. To jsem dělal pro testování a měření ručně. K Pyritu jsem nedělal propracovanější ovládací skript, protože jak se ukázalo, Pyrit jsem nebyl schopen stabilně zprovoznit pro větší počet počítačů najednou. Podrobněji to rozeberu v následující kapitole.

Pro jednoduchou automatizaci jsem napsal dva malé skripty `pyritserve` a `pyritstop`. První spustí Pyrit v režimu `serve` na počítačích ze seznamu (stejný formát souboru jako u `hashtopusbyssh`, používají se pouze první tři sloupce) a druhý ho na zadaných počítačích ukončí jednoduše příkazem `killall pyrit`.

Kapitola 4

Testování a měření

Náš cluster (počítače obou učeben a počítače clusteru **star**) je připravený a krátce otestovaný pro distribuovaný běh obou vybraných nástrojů.

Všechna ostrá měření jsem dělal pouze v učebnách, kde jsem použil všechny počítače - dohromady tedy 50 strojů. Počítače clusteru **star** jsem vynechal, protože po velmi krátké době (1-3 minuty) měření se grafické karty přehřály (s výjimkou `node-054`, který se chladí dobře). Jak jsem již uvedl dříve, v systému clusteru **star** se mi nepodařilo ovládat ventilátory. Orientační výkon pro srovnání jednotlivých karet na fakultě jsem přesto odměřil.

4.1 Hashcat

U všech měření jsem definoval úlohu (Task) v Hashtopusu stejně:

- typ útoku bruteforce -a 3
- maska 8 jakýkoliv znaků ?a?a?a?a?a?a?a?a
- profil zatížení maximální -w 3
- velikost úseků (chunk size) 10 minut

Pro měření jsem vybral několik hashovacích funkcí, pro které jsem měl připravené různě dlouhé hashlisty. Hlavní údaj, který jsem měřil, je rychlost zkoušení hashů oproti zadaní, tedy hashrate.

Zkoumal jsem, jestli výkon roste lineárně s počtem zapojených počítačů, tedy například jestli 10 stejných počítačů zvládne úlohu 10krát rychleji než jeden. Dále jsem se u některých funkcí snažil zjistit, zda velikost zadání (počet hashů v úloze) má vliv na rychlost počítání. Na závěr stručně uvádím energetickou náročnost clusteru a porovnám s jinými existujícími řešeními.

Task details:

Property	Value
ID:	73
Name:	eharmony-1-a-8ch Change
Attack command:	#HL# -w 3 -a 3 ?a?a?a?a?a?a?a?a
Chunk size:	600 seconds Set
Benchmark:	Autoadjust by default
Color:	# Set
Status timer:	5 seconds
Priority:	0 Set
Keyspace size:	7737809375
Keyspace dispatched:	73732675 (0.95%) Purge
Keyspace searched:	13100800 (0.17%)
Time spent:	00:06:27
Estimated time:	00:00:00
Speed:	0.00 H/s
Hashlist:	eharmony-1

Obrázek 4.1: Příklad zadané úlohy

4.1.1 Start clusteru

Počítače z učeben je nutné manuálně startovacím tlačítkem spustit a z bootovacího menu vybrat možnost **CUDA benchmark**. Na vzdálené spouštění počítačů a změny pořadí bootovacích voleb jsem nedostal dostatečná oprávnění (technicky to možné je). Počítače z clusteru **star** by měly být v provozu nepřetržitě. Před měřením jsem na vybraných počítačích skriptem `sethostname` nastavil správné názvy počítačů.

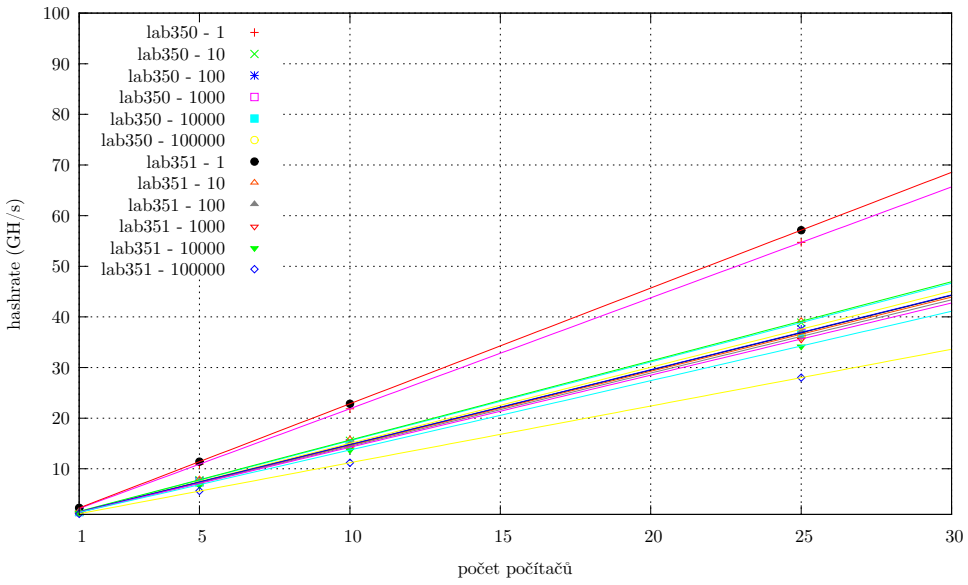
4.1.2 MD5

Jako první testovaný algoritmus pro test clusteru jsem zvolil MD5. V minulosti se používal pro uchovávání hesel v linuxových systémech a často se používal ve webových aplikacích. Dnes už není doporučeno uchovávat hesla zahashováním pomocí MD5. Důvodem je dostupnost diskové kapacity pro Rainbow tabulky [4] (Rainbow tabulka 1 až 9 znaků dlouhých hesel bez speciálních znaků se vejde do 1 TB) a také to, že prolamování s využitím grafických karet je poměrně rychlé, což se pokusím měřením ukázat. Jako zdroj dat jsem použil uniklou databázi internetové seznamky eharmony [3]. Testoval jsem různě dlouhé seznamy (1, 10, 100, 1000, 10000 a 100000 hesel) pro demonstraci toho, jak si Hashcat poradí s větším objemem hashů. Do prolamování těchto hashlistů jsem zapojoval postupně různý počet členů clusteru, aby se ukázalo, jak dobře lze clusteru škálovat výkon.

Tabulka 4.1: Měření výkonu Hashcat - MD5

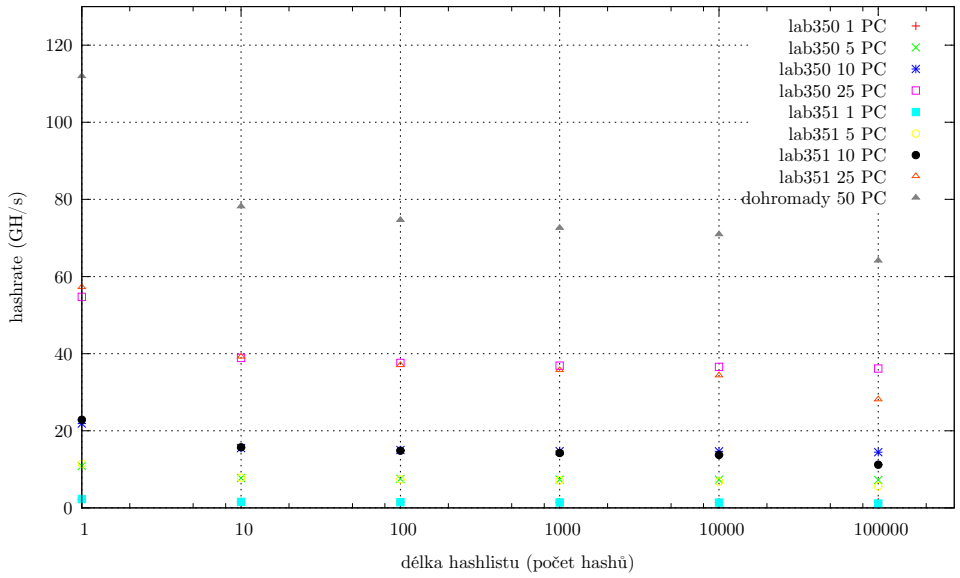
Počítače		Hashrate pro hashlisty (MH/s)					
učebna	počet	1	10	100	1 000	10 000	100 000
351	1	2285	1570	1480	1420	1370	1113
351	5	11400	7840	7390	7120	6850	5620
351	10	22850	15710	14830	14249	13750	11184
351	25	57131	39120	36933	35622	34241	28005
350	1	2184	1546	1501	1471	1465	1443
350	5	10895	7740	7510	7355	7329	7225
350	10	21860	15523	15022	14730	14661	14449
350	25	54722	38877	37578	36875	36577	36144
obě	50	111850	78085	74548	72500	70817	64000

Naměřené hodnoty ukazují, že cluster se při prolamování škáluje perfektně lineárně. Pro lepší znázornění vynesu hodnoty do grafu a proložím regresní přímku pro jednotlivé délky hashlistů.



Obrázek 4.2: Hashcat MD5 - škálování

Graf nám škálování potvrzuje a je vidět, že cluster škáluje lineárně nezávisle na velikosti zadání - ale výkon klesá, jak je vidět v následujícím grafu.



Obrázek 4.3: Hashcat MD5 - délka zadání

(osa x je logaritmická) Je vidět, že grafické karty GTX 480 počítačů z učebny 350 zvládají velké zadání lépe. Výkon u nich klesá pomaleji s velikostí zadání než u GPU GTX 560 Ti z učebny 351, přestože v hrubé síle na jeden hash jsou pomalejší.

4.1.3 NTLM

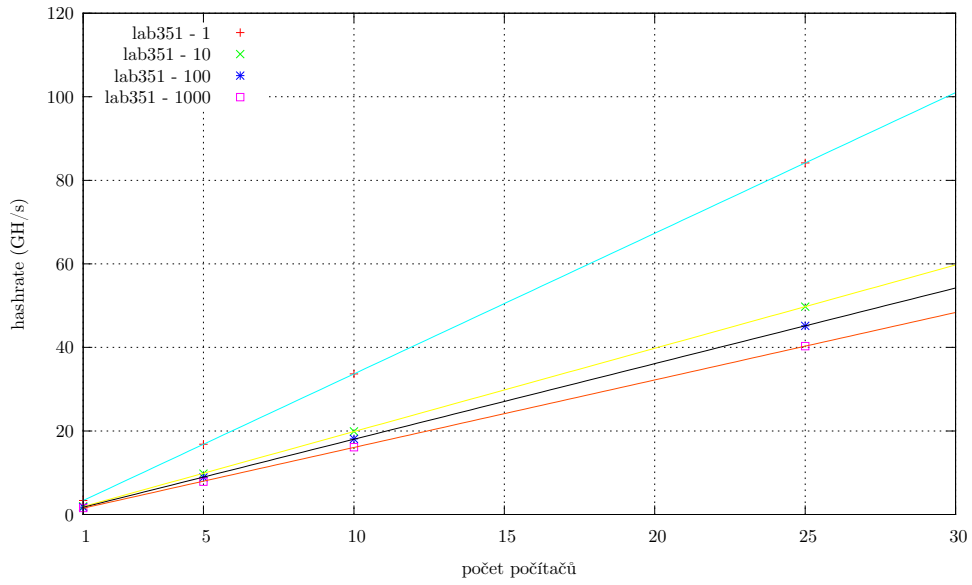
Druhým z rychlých algoritmů, který jsem vybral je NTLM. Používá se k ukládání hesel v operačních systémech Microsoft Windows.

Jako zdroj dat jsem vzal 1000 slov ze slovníku rockyou [38] a na webové stránce [39] jsem si nechal vygenerovat hashe.

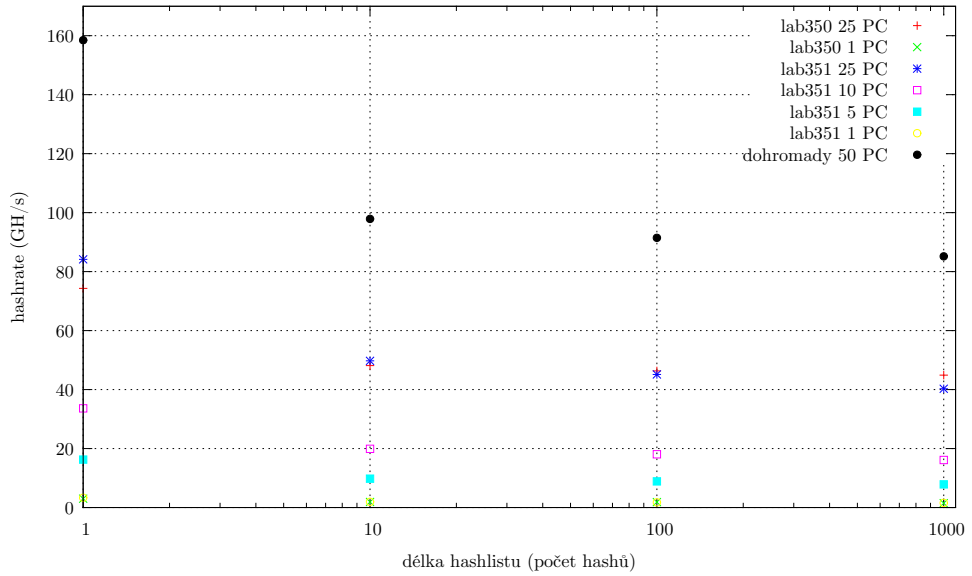
Tabulka 4.2: Měření výkonu Hashcat - NTLM

Počítače		Hashrate pro hashlisty (MH/s)			
učebna	počet	1	10	100	1 000
351	1	3347	1955	1765	1570
351	5	16820	9785	8855	7870
351	10	33653	19900	18055	16100
351	25	84161	49757	45156	40292
350	1	2984	1899	1847	1471
350	25	74347	48147	46317	44919
obě	50	158490	97901	91482	85183

Jak je vidět z tabulky, výkon opět roste lineárně. Rychlost prolamování NTLM na našem clusteru je při jednom hashi o 40% vyšší než u MD5.



Obrázek 4.4: Hashcat NTLM - učebna 351, škálování



Obrázek 4.5: Hashcat NTLM - délka zadání

Opět se ukazuje, že ačkoliv má GTX 560 Ti větší výkon na jeden hash, tak při delším zadání je na tom GTX 480 lépe.

Jednoduchým výpočtem zjistíme, že jakékoliv 8 znakové NTLM heslo (včetně speciálních znaků, entropie 52.55) cluster prolomí za méně než 12 hodin:

$$2^{52.55}/(158 \cdot 10^9) \approx 40000s \approx 11.2h$$

4.1.4 SHA512

Jako prvního zástupce „pomalého“ hashovacího algoritmu jsem zvolil 5000 krát iterovaný SHA-512 se solí. Tento algoritmus se shodným nastavením se v současné době používá jako výchozí pro ukládání hesel na linuxových systémech.

Hashlisty pro toto měření jsem si generoval pomocí nástroje `mkpasswd`. Jako zdroj dat jsem vzal náhodně 1000 slov z výše uvedeného slovníku rockyou:

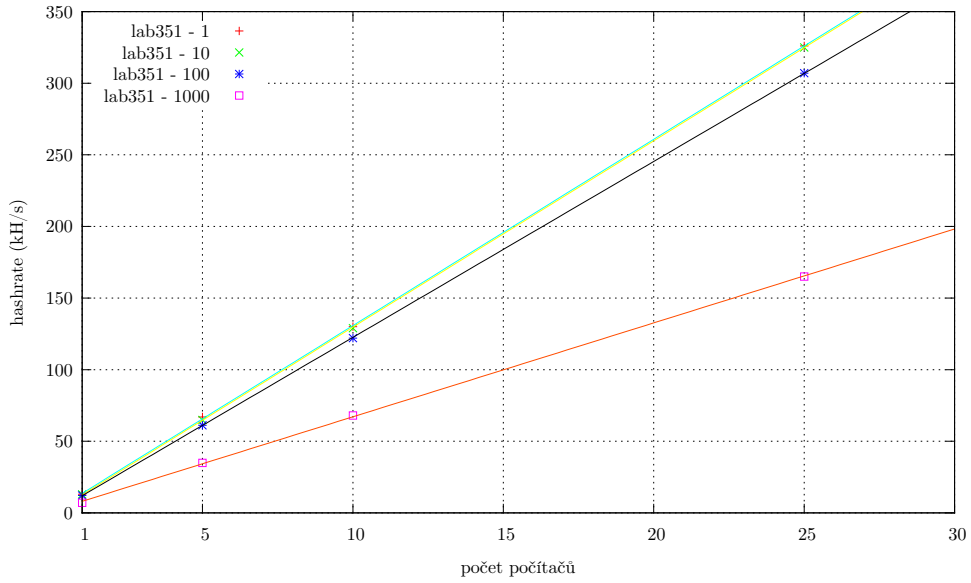
```
$ while read line ; do \
echo $line | mkpasswd -m sha-512 -s ; \
done < rockyouRandom1000 > sha512-1000
```

Tento soubor jsem pak už jen zkracoval pro konkrétní měření.

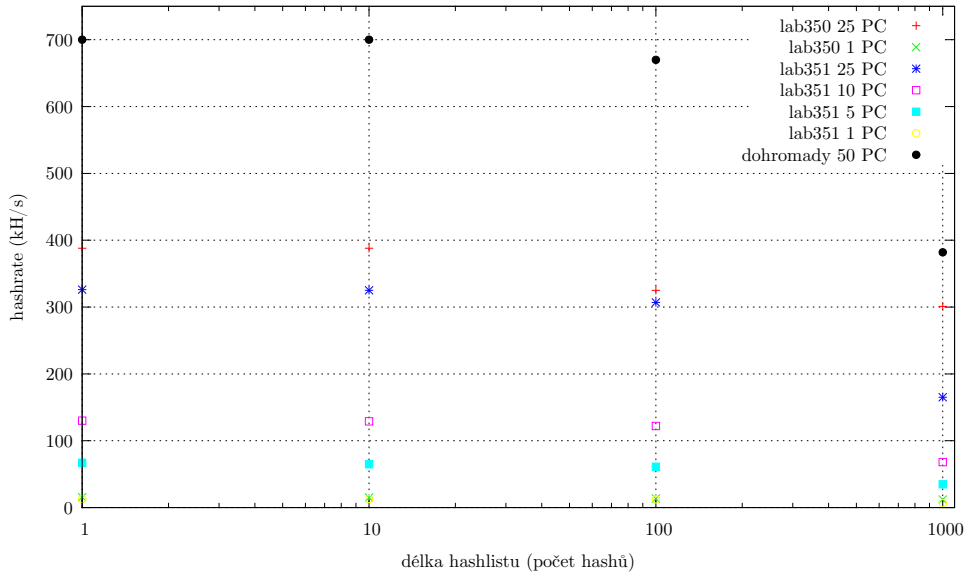
Tabulka 4.3: Měření výkonu Hashcat - SHA512

Počítače		Hashrate pro hashlisty (kH/s)			
učebna	počet	1	10	100	1 000
351	1	13	12.9	12.2	7
351	5	67	64.9	61.2	34.9
351	10	130	129	122	68
351	25	326	325	307	165
350	1	15.4	15.1	13.4	12
350	25	388	388	325	301
obě	50	700	700	670	382

U SHA512 docházelo k výpadkům zobrazování hashratu u jednotlivých agentů a tím se snižoval celkový hashrate. Delším během s menší maskou jsem si ověřil, že jde jen o zobrazovací chybu Hashcatu (změřil jsem čas do vyčerpání celého keyspace a vydělil počtem vteřin k získání reálného hashrate) a hodnoty tedy uvádím maximální, když zobrazování fungovalo správně.



Obrázek 4.6: Hashcat SHA512 5000 rund - učebna 351, škálování



Obrázek 4.7: Hashcat SHA512 5000 iterací - délka zadání

U tohoto algoritmu je propad výkonu s přibývajícím počtem hashů u GTX 560 Ti z učebny 351 vidět nejlépe.

4.1.5 WPA-PSK

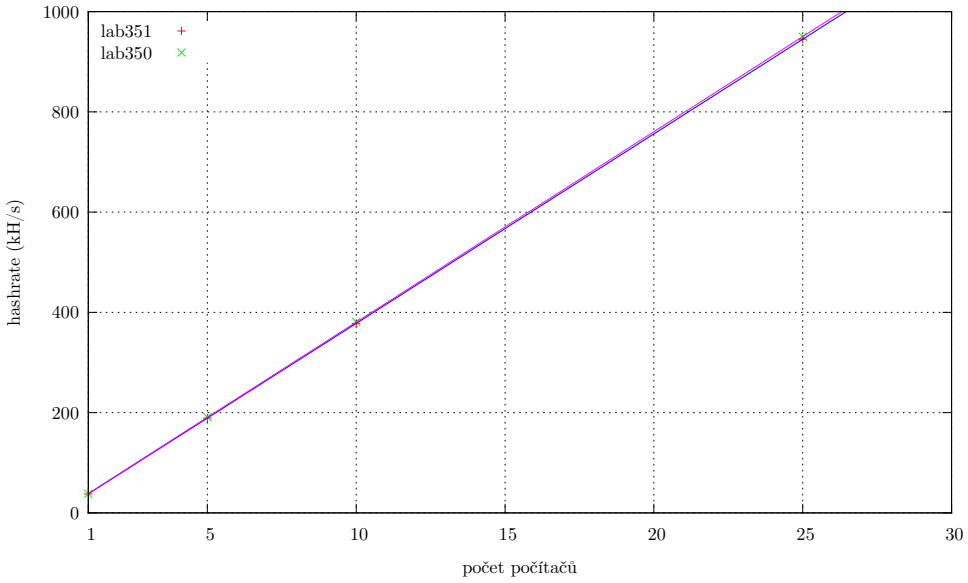
Protože Pyrit se používá pouze pro prolamování WPA-PSK, naměříme hashrate i u Hashcatu, abychom mohli oba programy porovnat.

Hashcat jako zdroj dat pro prolamování WPA-PSK nepřijímá výstup programu `airodump-ng`, kterým se běžně 4cestný handshake odchytává. Má svůj speciální formát pro WPA-PSK a je třeba zachycený handshake z `airodump-ng` do něj nejdříve překonvertovat. Detailně je to popsáno přímo na webu Hashcatu [40] a k dispozici je i webový formulář [41] pro konverzi. Jako zdroj dat pro měření jsem použil testovací soubor [42] ze seznamu testovacích hashů [43] od vývojářů Hashcatu. Později jsem pro ověření hodnot použil i odchycený handshake dodaný s Pyritem, výsledný hashrate byl stejný.

Tabulka 4.4: Měření výkonu Hashcat - WPA-PSK

Počítače		Hashrate (kH/s)
učebna	počet	
351	1	37.8
351	5	189
351	10	378
351	25	945
350	1	38.1
350	5	191
350	10	381
350	25	950
obě	50	1895

Rozdíl výkonu karet z učeben je při prolamování WPA-PSK zanedbatelný. Pokud se podíváme do tabulky 1.2, je vidět, že s takovým clusterem by prolomení osmimístného číselného hesla (entropie 26.5) trvalo méně než minutu, ale na prolomení osmiznakového hesla, kde mohou být čísla, malá i velká písmena (entropie 47.6), by výpočet musel běžet déle než tři a půl roku.



Obrázek 4.8: Hashcat WPA-PSK - škálování

4.1.6 Zhodnocení

Cluster jsem otestoval pro distribuované prolamování hesel hrubou silou. Na všech měřených algoritmech se cluster choval spolehlivě a výkon rostl lineárně se zapojenou výpočetní silou. U algoritmů kde jsem zkoumal vliv délky zadání na celkový výkon se jasně ukázalo, že s delším zadáním výkon klesá a u karet GTX 560 Ti je znatelnější propad.

4.2 Pyrit

Měření výkonnosti Pyritu je třeba rozdělit na dvě části:

- předpočítávání klíčů v databázi pro naimportovaná hesla
- útok na handshake pomocí předpočítané databáze

4.2.1 Předpočítání databáze

Pro testování jsem používal dříve připravené skripty `pyritserve` a `pyritstop`. Do databáze jsem před měřením naimportoval seznam všech 8 místných číselných kombinací. Použil jsem výchozí volbu ukládání databáze do souboru.

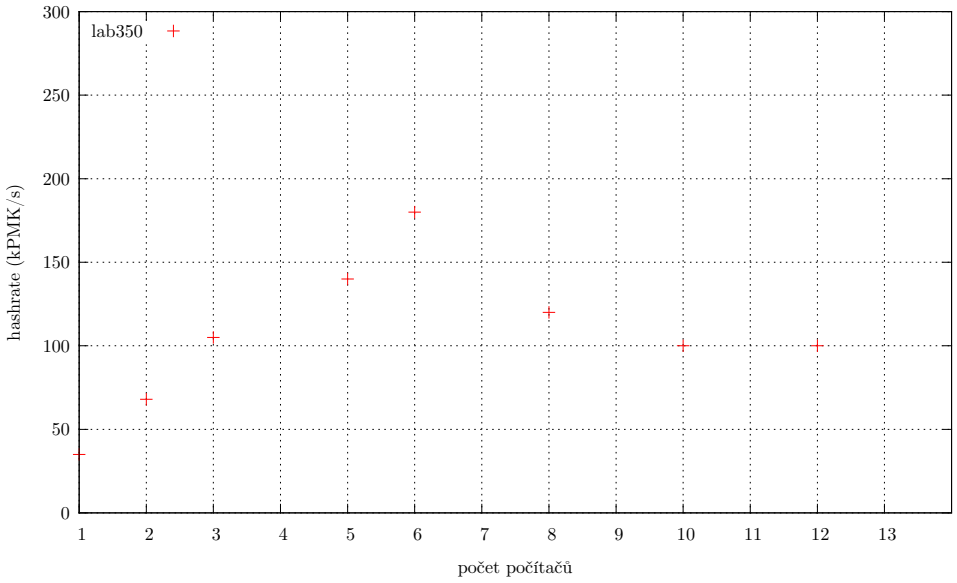
```
$ pyrit -i /mnt/data/samekja2/8digit import_passwords
```

Tabulka 4.5: Měření výkonu Pyrit - předpočítávání klíčů

Počítače		Hashrate (kPMK/s)
učebna	počet	
351	1	34
350	1	35
350	2	68
350	3	105
350	6	180
350	8	120
350	10	100
350	12	100

Jak je vidět z naměřených hodnot, Pyrit na školních počítačích funguje dobře distribuovaně pouze do šesti zapojených počítačů, pak výkon klesá. Na hlavním počítači, který zadává práci strojům v `serve` módu, dochází k velkému vytížení procesoru (všechna 4 jádra na 100% výkonu). Nejprve mi dobře fungovaly jen tři zapojené počítače, po úpravě konfiguračního souboru zadávajícího počítače, kde jsem parametrem `limit_ncpus=1` omezil, aby zadávají stroj používal jen jádro pro vlastní předpočítávání a zbytek výkonu procesoru měl volný na rozdávání úloh, jsem se dostal na stabilní běh 6 počítačů. Pro 10 a 12 počítačů sice v tabulce udávám hodnoty, ale výpočet se v obou případech po několika minutách zastavil. Z hodnot pro jeden počítač je vidět, že Hashcat je při prolamování WPA-PSK rychlejší.

Hotová databáze zabírala podle očekávání 4GB na disku.



Obrázek 4.9: Pyrit - učebna 350, škálování

Datový tok sítě dosahoval směrem z řídícího počítače do výpočetních strojů maximálně 60Mb/s, rezerva tedy byla dostatečná (to, že lze na síti v učebnách dosáhnout opravdu rychlosti 1 Gbit/s, jsem testoval). Diskové I/O také mělo rezervu, sledoval jsem vytížení pomocí programu `iotop`. Nestabilitu a nemožnost škálovat větší počet počítačů tedy dávám za vinu nedostatečnému výkonu řídícího počítače, ale domnívám se, že i se silnějším strojem by brzy nastaly další problémy. Pyrit je už delší dobu neudržovaný (poslední commit do svn je v roce 2011), na webu projektu je v záložce issues několik stran diskuzí k všemožným problémům. Ty se občas někomu podaří patchem zdrojového kódu opravit (několik jsem jich také zkusil), ale na stabilní distribuovaný běh ve větším počtu nodů Pyrit v aktuální podobě připraven není.

Je to škoda, určitě má smysl pro testování bezpečnosti Wifi sítí si pro několik nejčastějších názvů sítí vygenerovat tabulky pro osmimístná číselná hesla, slovníky s pravidly apod. tak, aby databáze ještě měla rozumnou velikost. Osobně bych se asi do takového vlastního testování nepouštěl. Pokud není možnost útoku přes WPS PIN (popis útoku [44] a nástroj [45]) a nebo zachycený handshake nerozluští služby jako je **gpuhash.me** [46] (které disponují opravdu velkými databázemi, ale nepodařilo se mi dohledat jaký software používají na počítání databází a vyhledávání v nich), považoval bych síť za bezpečnou. Ale to je námět na další bakalářskou práci.

4.2.2 Útok na handshake pomocí předpočítané databáze

Samotný útok na handshake nevyužívá grafickou kartu, ani možnosti distribuovaného výpočtu, využívají se pouze všechna CPU jádra procesoru na počítači, ze kterého je útok spuštěn. Na počítačích obou učeben jsem dosahoval podobných výsledků okolo 4000 kPMK/s. Na počítačích clusteru **star** jsem dosahoval horších výsledků. Abych se ujistil, že úzké hrdlo není jen ve zvoleném druhu úložiště. zkoušel jsem prolamovat handshake i z předpočítané databáze z SQLite databáze a z MySQL databáze uložené na diskovém poli. Výsledky ukázaly, že záleží opravdu jen na procesoru.

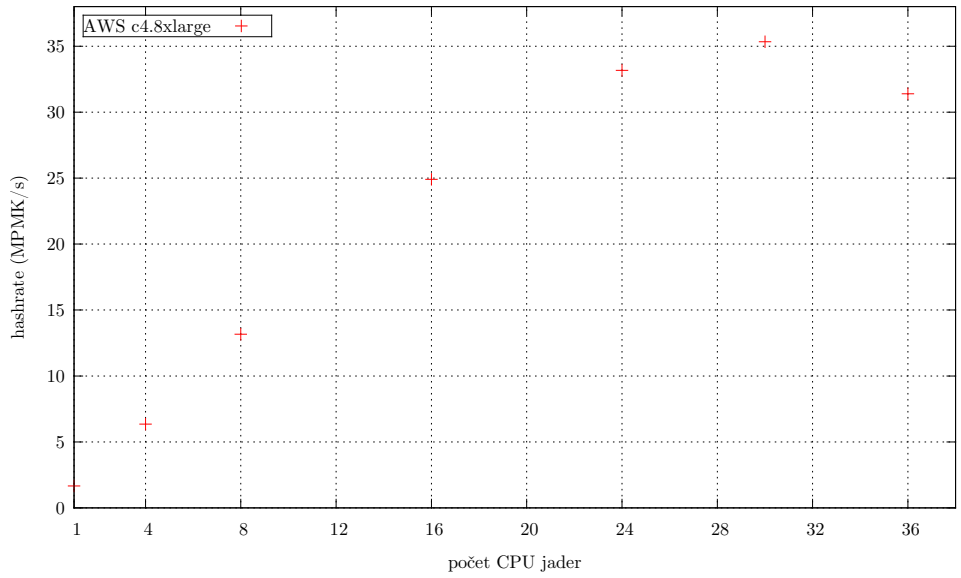
Příklad prolamování s využitím předpočítané databáze na `node-053`:

```
$ pyrit -r wpapsk-linksys.dump.gz attack_db
Pyrit 0.4.0 (C) 2008-2011 Lukas Lueg http://pyrit.googlecode.com
This code is distributed under the GNU General Public License v3+

Connecting to storage at 'file:///mnt/data/samekja2/pyritdb'... connected.
Parsing file 'wpapsk-linksys.dump.gz' (1/1)...
Parsed 6 packets (6 802.11-packets), got 1 AP(s)

Picked AccessPoint 00:0b:86:c2:a4:85 ('linksys') automatically.
Attacking handshake with Station 00:13:ce:55:98:ef...
Tried 100000000 PMKs so far (100.0%); 2713541 PMKs per second.

Password was not found.

$
```

Dále jsem chtěl ověřit, že na výkonnějším systému bude prolamování rychlejší, a tak jsem na testování použil instanci virtuálního stroje c4.8xlarge z nabídky Amazon AWS EC2. Tento virtuální stroj disponuje 60 GB paměti a celkem 36 CPU jádry procesorů Intel Xeon E5-2666 v3. Vyhledávání jsem spustil na stejné databázi jako na školních strojích, naměřené hodnoty:

Tabulka 4.6: Pyrit - prolamování Amazon AWS c4.8xlarge

počet CPU jader	kPMK/s
1	1665
4	6350
8	13164
16	24903
24	33167
30	35338
36	31392



Obrázek 4.10: Pyrit - prolamování Amazon AWS c4.8xlarge

Tabulka i graf jasně ukazují, že s výkonnějším procesorem výkon prolamování skoro lineárně roste až do počtu 30 jader. Prolamování hesla oproti připravené databázi je tedy velmi rychlé, navíc jak jsem ověřil, čím rychlejší procesor, tím lineárně rychlejší vyhledávání, ale i to má své limity. Na těchto naměřených hodnotách je dobře vidět, v čem je síla programu Pyrit - bylo by třeba přibližně 900 grafických karet (podobných jako jsou v učebnách) v Hashcat clusteru, aby bylo možné prolamovat WPA-PSK stejně rychle jako to dělá Pyrit oproti už připravené databázi.

4.3 Porovnání GPU karet

Na clusteru **star** jsem odměřil pouze několik měření MD5, abych mohl porovnat výkon s kartami v učebnách. Delší spolehlivé měření nebylo vzhledem k přehřívání možné, navíc v průběhu psaní práce byl cluster **star** několikrát nedostupný z důvodu instalace nových nodů a údržby.

Tabulka 4.7: Porovnání GPU na fakultě

Karta		Hashrate MD5 (MH/s)		
název	počet na fakultě	1	1000	100000
GeForce GTX 470	1	1764	1190	1104
GeForce GTX 480	26	2184	1471	1443
GeForce GTX 560 Ti	25	2285	1420	1113
GeForce GTX 590	1	4029	2717	1795
Tesla K40	1	4870	3210	2446

Podle očekávání je Tesla K40 nejvýkonnější z testovaných školních grafických karet, ale čekal jsem, že výkon této karty bude oproti GTX 590 ještě vyšší vzhledem k počtu CUDA jader - viz 2.1.4 Přehled grafických karet.

4.4 Energetická náročnost

V průběhu měření jsem wattmeterem měřil odběr počítačů v učebnách. Maximální odběr v učebně 350 byl 280 W na počítač při prolamování 100 000 hashů MD5, při ostatních měřeních to bylo okolo 250 W. V druhé učebně byl odběr ve všech případech přibližně od 35 W nižší, průměrně 215 W. Celkový odběr cluster z učeben je tedy celkem okolo 11.5 kW.

4.5 Srovnání s existujícími řešeními

Na závěr porovnám výkon funkčního školního clusteru s jinými existujícími systémy na prolamování hesel. Pro představu stačí k porovnání jeden hashovací algoritmus, dále zohledňuji odběr energie, prostorovou náročnost a přibližnou cenu.

Pro srovnání jsem vybral následující systémy:

SC5/M505-48 FPGA
: FPGA pole od Pico Computing jsem zmiňoval na začátku práce. Informace čerpám z produktového datasheetu na stránkách výrobce [7].

J.M.Gosney 25 GPU
: J.M.Gosney ve svém clusteru použil výhradně karty AMD a to 10x 7970, 4x 5970, 3x HD6990 a jedno 5870. Kromě 5870 jde vždy o nejvýkonnější modely dané řady. Odběr energie odhaduji. Výkon byl změřen v roce 2012 programem oclHashcat-lite 0.11.

J.Samek 5xHD7970
: Ke srovnání přidávám i můj vlastní stroj, který jsem dříve používal na těžení kryptoměn. Je vybaven pěti GPU AMD HD7970. Sestaven je „amatérsky“, v obyčejné otevřené počítačové skříni. Grafické karty jsou

připojené přes prodlužovací PCIe riser kabely a rozmístěné ve výšce pro lepší ventilaci, proto výška 7U. Hashrate jsem měřil v programu oclHashcat 1.31.

Brutalis 8xR9 290X

Firma sagitta.systems [47] (jejíž spoluzakladatel je již zmiňovaný J.M.Gosney) se specializuje na hardwarová a softwarová řešení pro, jak uvádějí, „HPC - High Performance Cracking“. Šasi Brutalis je možné vybavit až osmi GPU, obsahuje dostatečně silné zdroje pro všechny karty, a to vše v jednom 4U počítači. AMD R9 290x patří k nejvýkonnějším současným grafickým kartám. Úplný benchmark lze nalézt na Gosneyho GitHubu [48].

Brutalis 8xGTX 980

Totožné jako předchozí, jen s jinou GPU. Též jde o jednu z nejvýkonnějších aktuálně dostupných karet. Benchmark [49].

FIT cluster 50 GPU

Nakonec uvádím zprovozněný GPU cluster v prostředí FIT ČVUT, cenu jedné školní sestavy jsem odhadl na 10 000 Kč.

Tabulka 4.8: Porovnání GPU clusterů

	NTLM (GH/s)	odběr (kW)	velikost	cena (tis. Kč)
SC5/M505-48 FPGA	164	1.5	4U	2700
J.M.Gosney 25 GPU	348	7	20U	250
J.Samek 5xHD7970	83	1.5	7U	55
Brutalis 8xR9 290X	141	3	4U	465
Brutalis 8xGTX 980	135	3	4U	465
FIT cluster 50 GPU	158	11.5	dvě místnosti	500

Cena systémů Brutalis je vysoká, ale je třeba si uvědomit, že jde o profesionální odladěné řešení postavené na aktuálně nejlepších kartách pro prolamování hesel. Navíc součástí dodaného systému je řídící software Hashstack [50]. Z uživatelského hlediska jde o podobný systém jako je Hashtopus, ale jeho možnosti jsou širší. Mezi hlavní výhody patří podpora běhu různých bezpečnostních nástrojů (s možností přidání vlastního), podpora více uživatelů, lepší monitoring nodů a lepší možnosti plánování útoků. Systémy Brutalis považuji za to nejlepší, co lze na prolamování hesel koupit.

Jak ukazují hodnoty mého počítače, amatérsky sestavená řešení s větším počtem GPU v jednom nodu poskytují nejlepší poměr cena / výkon. Ze zkušenosti s těžením kryptoměn vím, že odladit takový systém pro stabilní (několikaměsíční) běh může být náročné. Jsem si jistý, že stejně výkonný systém se stejnými kartami jako mají sestavy Brutalis, bych byl schopen sestavit

za méně než poloviční cenu. Pravděpodobně by šlo o dva počítače, každý se čtyřmi GPU. Pokud by škola plánovala zřídit GPU cluster na prolamování hesel, volil bych právě tuto cestu, tedy několik málo počítačů s více špičkovými grafickými kartami.

Uvážíme-li fakt, že grafické karty ve školních počítačích jsou 3 až 4 roky staré (a výkon GPU roste rychleji než u procesorů) podařilo se zprovoznit cluster se slušným výkonem. A to bez jakýchkoliv dalších investic do hardwaru. Nevýhody školního clusteru jsou především odběr energie a fakt, že výpočetní nody jsou počítače v učebnách a nelze tedy cluster používat nepřetržitě.

Závěr

V úvodní kapitole jsem přiblížil problematiku prolamování hesel a zmapoval jsem aktuální možnosti pro efektivní distribuované prolamování. Z těchto poznatků jsem se rozhodl pro implementaci nástrojů Hashcat s wrapperem Hashtopus a Pyrit.

Následně jsem prozkoumal jaké výpočetní prostředky fakulty by byly vhodné k implementaci vybraných programů a stručně jsem nastínil jak budu při implementaci postupovat.

Ve fakultních učebnách 350, 351 a clusteru **star** jsem postupně zprovoznil jednotlivé části distribuovaného řešení Hashtopus pro program Hashcat. Součástí je i ovládací skript pro automatizované spouštění klientské části Hashtopusu na jednotlivých výpočetních nodech. Následovalo zprovoznění programu Pyrit, který má možnost distribovaného běhu vestavěnou.

Krátce otestovaný GPU cluster jsem podrobil měření. U Hashcatu jsem testoval výkonnost při prolamování hesel jednotlivých hashovacích algoritmů a zkoumal, zda výkon clusteru roste lineárně s počtem zapojených počítačů a jaký vliv na výkon má velikost zadání. U rychlých i pomalých hashovacích algoritmů se ukázalo, že cluster škáluje perfektně lineárně. Při testování větších zadání výkon klesal. Zajímavostí je, že v učebnách byla karta GTX 480 u delších hashlistů výkonnější než GTX 560 Ti, přestože při jednom hashi měla výkon nižší. Kromě přehřívání malé skupiny nodů nebyl s během clusteru v učebnách žádný problém.

S programem Pyrit jsem měl potíže s distribuovaným během na více než šesti počítačích. Příčinou byl nedostatečný výkon procesoru stroje rozdělujícího práci a nedoladěnost samotného programu. Prolamování z předpočítané databáze fungovalo korektně a jeho rychlost byla omezená pouze procesorem daného počítače. To jsem potvrdil testem na několikanásobně výkonnějším virtuálním stroji.

Na závěr testovací části jsem porovnal grafické karty používané ve škole, uvedl energetickou náročnost celého clusteru během prolamování a srovnal zprovozněný cluster s ostatními existujícími řešeními.

Pro praktické využívání GPU clusteru na prolamování hesel na fakultě by bylo zajímavé umožnit automatické spouštění celého clusteru (například v noci) se zadáváním úloh podle plánu a následným reportováním výsledků.

Literatura

[1] prof. Ing. Róbert Lórencz, C.: *Bezpečnost - 5. Hašovací funkce, MD5, SHA-x, HMAC.* [cit. 2015-02-09]. Dostupné z: https://edux.fit.cvut.cz/archive/B141/BI-BEZ/_media/bez_n5.pdf

[2] *Hashcat Benchmarks | Musings of an Information Security Student.* [cit. 2015-02-09]. Dostupné z: http://benjaminellett.com/hashcat-benchmarks/

[3] *The Password Project.* [cit. 2015-02-09]. Dostupné z: http://thepasswordproject.com/leaked_password_lists_and_dictionaries

[4] *RainbowCrack Project.* [cit. 2015-02-09]. Dostupné z: http://project-rainbowcrack.com/index.htm

[5] *Salted Password Hashing - Doing it Right.* [cit. 2015-02-09]. Dostupné z: https://crackstation.net/hashing-security.htm

[6] *Anatomy of a hack: How crackers ransack passwords like qeadzcwrsfxv1331.* [cit. 2015-02-09]. Dostupné z: http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/

[7] *Pico Computing SC5-4U.* [cit. 2015-02-09]. Dostupné z: http://picocomputing.com/brochures/SC5-4U.pdf

[8] *MVAPICH.* [cit. 2015-02-09]. Dostupné z: https://developer.nvidia.com/mvapich

[9] *John the Ripper password cracker.* [cit. 2015-02-09]. Dostupné z: http://www.openwall.com/john/

[10] *hashcat - advanced password recovery.* [cit. 2015-02-09]. Dostupné z: http://hashcat.net/hashcat/

[11] *oclHashcat - advanced password recovery*. [cit. 2015-02-09]. Dostupné z: http://hashcat.net/oclhashcat/

[12] *Cryptohaze Multiforcer*. [cit. 2015-02-09]. Dostupné z: http://www.cryptohaze.com/multiforcer.php

[13] *High-performance distributed password recovery with NVIDIA GPU acceleration*. [cit. 2015-02-09]. Dostupné z: https://www.elcomsoft.com/edpr.html

[14] *Passware Kit Forensic*. [cit. 2015-02-09]. Dostupné z: http://www.lostpassword.com/index.htm

[15] *pyrit - WPA/WPA2-PSK and a world of affordable many-core platforms*. [cit. 2015-02-09]. Dostupné z: https://code.google.com/p/pyrit/

[16] *rule_based_attack [hashcat wiki]*. [cit. 2015-02-09]. Dostupné z: https://hashcat.net/wiki/doku.php?id=rule_based_attack

[17] *oclHashcat v1.20*. [cit. 2015-02-09]. Dostupné z: http://hashcat.net/forum/thread-3323-post-19187.html

[18] *VCL Cluster Platform*. [cit. 2015-02-09]. Dostupné z: http://www.mosix.org/txt_vcl.html/

[19] *Password Cracking HPC*. [cit. 2015-02-09]. Dostupné z: http://heim.ifi.uio.no/hennikl/passwords12/www_docs/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf

[20] *New 25 GPU Monster Devours Passwords In Seconds | The Security Ledger*. [cit. 2015-02-09]. Dostupné z: https://securityledger.com/2012/12/new-25-gpu-monster-devours-passwords-in-seconds/

[21] *vcl_cluster_howto [hashcat wiki]*. [cit. 2015-02-09]. Dostupné z: https://hashcat.net/wiki/doku.php?id=vcl_cluster_howto

[22] *unix-ninja/disthc*. [cit. 2015-02-09]. Dostupné z: https://github.com/unix-ninja/disthc/

[23] *Wi-Fi security – WEP, WPA and WPA2*. [cit. 2015-02-09]. Dostupné z: http://www.hsc.fr/ressources/articles/hakin9_wifi/hakin9_wifi_EN.pdf

[24] *Making a Faster Cryptanalytic Time-Memory Trade-Off*. [cit. 2015-02-09]. Dostupné z: http://lasec.epfl.ch/pub/lasec/doc/Oech03.pdf

[25] *Hashtopus - Index of /beta*. [cit. 2015-02-09]. Dostupné z: http://hashtopus.nech.me/beta/

[26] *curlyboi (cURLy bOI).* [cit. 2015-02-09]. Dostupné z: https://github.com/curlyboi/

[27] *Hashtopus - manual.* [cit. 2015-02-09]. Dostupné z: http://hashtopus.nech.me/manual.html

[28] *JSColor JavaScript / HTML Color Picker, Selector, Chooser.* [cit. 2015-02-09]. Dostupné z: http://jscolor.com/

[29] *Playbooks - Ansible Documentation.* [cit. 2015-02-09]. Dostupné z: http://docs.ansible.com/playbooks.html

[30] *Puppet Labs: IT Automation Software for System Administrators.* [cit. 2015-02-09]. Dostupné z: http://puppetlabs.com/

[31] *Chef | IT automation for speed and awesomeness | Chef.* [cit. 2015-02-09]. Dostupné z: https://www.chef.io/chef/

[32] *Ansible is Simple IT Automation.* [cit. 2015-02-09]. Dostupné z: http://www.ansible.com/home

[33] *An Ansible Tutorial | Servers for Hackers.* [cit. 2015-02-09]. Dostupné z: https://serversforhackers.com/getting-started-with-ansible/

[34] *childintime/hashtopusbyssh.* [cit. 2015-02-09]. Dostupné z: https://github.com/childintime/hashtopusbyssh

[35] *Comparing curlyboi:master...childintime:master · childintime/hashtopus-agent.* [cit. 2015-02-09]. Dostupné z: https://github.com/childintime/hashtopus-agent/compare/curlyboi:master...master

[36] *childintime's Gists.* [cit. 2015-02-09]. Dostupné z: https://gist.github.com/childintime

[37] *1Password.* [cit. 2015-02-09]. Dostupné z: https://agilebits.com/onepassword

[38] *Rockyou.* [cit. 2015-02-09]. Dostupné z: http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2

[39] *LM/NTLM Hash Generator - TobTu.* [cit. 2015-02-09]. Dostupné z: http://www.tobtu.com/lmntlm.php

[40] *cracking_wpawpa2 [hashcat wiki].* [cit. 2015-02-09]. Dostupné z: http://hashcat.net/wiki/doku.php?id=cracking_wpawpa2

[41] *hashcat cap2hccap - advanced password recovery.* [cit. 2015-02-09]. Dostupné z: https://hashcat.net/cap2hccap/

[42] *hashcat example hccap*. [cit. 2015-02-09]. Dostupné z: http://hashcat.net/misc/example_hashes/hashcat.hccap

[43] *example_hashes [hashcat wiki]*. [cit. 2015-02-09]. Dostupné z: http://hashcat.net/wiki/doku.php?id=example_hashes

[44] *Brute forcing Wi-Fi Protected Setup*. [cit. 2015-02-09]. Dostupné z: https://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf

[45] *reaver-wps - Brute force attack against Wifi Protected Setup*. [cit. 2015-02-09]. Dostupné z: https://code.google.com/p/reaver-wps/

[46] *GPUHASH.me - online WPA cracker and hash bruteforcer for WiFi wpa handshake*. [cit. 2015-02-09]. Dostupné z: https://gpuhash.me/

[47] *Sagitta HPC - High Performance Password Cracking*. [cit. 2015-02-09]. Dostupné z: https://sagitta.systems/

[48] *World's First 8x R9 290X oclHashcat Benchmark*. [cit. 2015-02-09]. Dostupné z: https://gist.github.com/epixoip/8171031

[49] *World's First 8x GTX 980 cudaHashcat Benchmark*. [cit. 2015-02-09]. Dostupné z: https://gist.github.com/epixoip/c0b92196a33b902ec5f3

[50] *Hashstack - Distributed Password Cracking - Sagitta HPC*. [cit. 2015-02-09]. Dostupné z: https://sagitta.systems/software/

Příloha **A**

Seznam použitých zkratek

CPU Application Programming Interface

CPU Central processing unit

GPU Graphic processing unit

GPGPU General-Purpose computation on Graphics Processing Units

MPI Message Passing Interface

PMK Pairwise master key

PSK Pre-shared key mode

SSID Service Set Identifier

WPA Wi-Fi Protected Access

WEP Wired Equivalent Privacy

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `impl` ..... adresář s implementací
  - `hashtopusbyssh` ..... adresář hlavního programu
    - `hashtopusbyssh` ..... hlavní program
    - `hashtopusbyssh-lab.conf` ..... konfigurace pro učebny
    - `hashtopusbyssh-star.conf` ..... konfigurace pro cluster **star**
  - `hostlists` ..... konfigurace počítačů
  - `utils` ..... pomocné skripty
- `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `thesis.pdf` ..... text práce ve formátu PDF