**Semestrální projekt MI-PRC ZS2019/2020:**

**GPU Cryptohash Recovery**

**Pavlína Kopecká, Libor Kuchař**

**magisterské studium, FIT CVUT, Thákurova 9, 160 00 Praha 6**

**Prosinec 3, 2019**

# Definice problému a popis sekvenčního algoritmu:

## Popis problému:

Mnoho aplikací (zejména těch webových) si musí nějakým způsobem ukládat hesla svých uživatelů. V současné době stále existují weby, které ukládají hesla jako plaintext, toto řešení je ale nebezpečné, protože pokud se útočník nějakým způsobem dostane k databázi hesel, tak mu nic nebrání se k původnímu heslu dostat. Proto je doporučeno hesla zahashovat pomocí silné kryptografické hashovací funkce a následně tuto hodnotu uložit.

Bezpečná hashovací funkce by měla splňovat následující tři body: (<https://cs.wikipedia.org/wiki/Kryptografická_hašovací_funkce>)

1. **Odolnost vůči získání předlohy** – hashovací funkce je jednosměrná.
2. **Odolnost vůči získání jiné předlohy** – hashovací funkce má omezený obor hodnot, existuje tedy více předloh, který mají stejný výsledný hash.
3. **Odolnost vůči nalezení kolize:** Je obtížné systematicky najít dvojici vstupů (x,y), pro které h(x)=h(y).

Jak je patrné z 1. bodu, nelze z hashe systematicky získat původní hodnotu. Jediný způsob, jak získat původní hodnotu je nějakým způsobem „odhadnout“ tuto hodnotu a zahashovat jí pomocí použité kryptografické funkce a tyto hodnoty následně porovnat. Jakým způsobem ale původní heslo odhadnout?

1. **Útok hrubou silou** – jediné 100% spolehlivé řešení, vyzkouší se všechny možné kombinace hesel v dané délce. Počet kombinací je ale aL kde a je délka abecedy (malá + velká písmena a čísla je 62) a L je délka hesla. Od určité délky není časově možné heslo prolomit.
2. **Slovníkový útok** – používají se hesla ze slovníku. Tento typ útoku má ten problém, že pokrývá pouze malé množství všech možných hesel, na druhou stranu velká část uživatelů používá velmi slabá hesla, na která tento typ útoku můžeš postačovat. V dnešní době existují už komplexní slovníky obsahující nejčastější hesla.
3. **Rozšířený slovníkový útok** – tato metoda využívá jako základ slovníkový útok, ale slova z tohoto slovníku nějakým způsobem upravuje (např. zaměňuje velikost písmen, přehazuje podobná písmena, přidává před/za/do slova řetězce různé délky atd...)
4. **Duhové tabulky**

Všechny zmíněné útoky lze provádět pomocí CPU (např. slovníkový útok nemá cenu vůbec provádět na GPU z důvodu, že bottleneck je stejně čtení z disku), v případě delších komplexnějších hesel ale CPU již selhává. Tato práce se tedy zabývá nástrojem na získávání hesel z MD5 hashů za použití CUDA, konkrétně se bude jednat o **útok hrubou silou**, a **rozšířený slovníkový útok.**

## Popis sekvenčního řešení:

Hashovací funkce MD5 byla převzata a mírně upravena z https://gist.github.com/creationix/4710780.

### Útok hrubou silou:

Největším problémem tohoto útoku bylo samotné generování všech řetězců dané délky. Původně tento typ útoku byl řešen rekurzí, kvůli přenositelnosti na GPU bylo nutné tento algoritmus upravit na iterativní.

Iterativní algoritmus funguje následovně:

1. K řetězci délky L, který obsahuje znaky z abecedy délky A se budeme chovat jako k číslu o počtu číslicích L a základu A. K tomuto číslu je možné navrhnout softwarovou sčítačkou a pomocí této sčítačky můžeme postupně proiterovat všechny řetězce o délce L. V případě že dojde k přenosu z nejvyššího řádu (poslední znak se vrátí opět na první) tak víme, že jsme proiterovali všechny možnosti.
2. Nastavíme hodnoty sčítačky na počáteční hodnotu (v sekvenčním algoritmu to jsou samé 0).
3. Tento řetězec zahashujeme pomocí funkce MD5.
4. Porovnáme výsledný hash se zadaným hashem. Pokud se shoduje vrátíme daný řetězec = jedná se o původní hodnotu a pokračujeme ke kroku 7), jinak pokračujeme na krok 5)
5. Inkrementujeme řetězec o 1 (přičteme k nejnižší číslici číslo 1).
6. Pokud nastane přenos z nejvyššího řádu tak byly vyzkoušeny všechny kombinace – pokračujeme krokem 7, jinak pokračujeme na krok 3)
7. V případě nalezení dané hodnoty jí vypíšeme, jinak vypíšeme „No matches“, uvolní se prostředky, program se ukončí.

### Rozšířený slovníkový útok:

*Finální verze programu obsahuje pouze omezený rozšířený slovníkový útok, podporuje pouze přidávání řetězců různé délky za dané slovo. Tento typ útoku slouží spíše pro ilustraci zrychlení na GPU.*

Sekvenční řešení vždy přečte jeden řetězec ze slovníku, poté následně tento řetězec zahashuje pomocí funkce MD5, které následně porovná se zadaným hashem. Pokud se hash shoduje tak tento řetězec je původní heslo. Pokud hash neshoduje, tak se za daný řetězec pomocí bruteforce funkce přidávají všechny řetězce dané délky a abecedy. Tyto rozšířené řetězce se taktéž hashují pomocí funkce MD5 a následně porovnávají se zadaným hashem.

# CUDA:

## Popis řešení pro GPU:

### Útok hrubou silou:

Nejprve host nakopíruje používanou abecedu a zadaný hash do paměti konstant GPU a připraví paměť pro uložení nalezeného řetězce. Poté následně spustí kernel s N bloky a M vlákny (obě konstanty jsou nastavitelné během spuštění programu). Kernel se spouští zvlášť pro řetězec každé délky (takže např, pokud zkoušíme hesla v rozsahu 1-3 znaků, tak se nejprve pustí kernel pro řetězce délky 1, pak 2 atd..). Po spuštění kernelu se čeká až je dokončen a host následně zkontroluje, jestli se podařilo nalézt původní řetězec. Pokud ano, tak se vypíše a program končí, pokud ne pokračuje se s dalším rozsahem, či pokud vyčerpal všechny možnosti tak program končí s hláškou „No matches“.

Poté se na device pro každé vlákno zjistí, od jaké počáteční konfigurace sčítačky má začít počítat. Pro ošetření, aby se projely všechny kombinace se používá zaokrouhlování přídělu práce směrem nahoru, je tedy možné, že některá vlákna vykonávají stejnou práci. Začátek práce je určeno 64-bitovým číslem (bylo by možné rozšířit i na 128, či 256-bitové číslo), počáteční permutace se zjistí tak, že se toto číslo postupně modulí a dělí (stejný princip jako bychom převáděli třeba desítkové číslo na dvojkové).

Poté se zavolá funkce podobná sekvenční funkci, ovšem s jiným počátečním nastavením sčítačky než samé 0. Tato funkce také vždy po určitém počtu iterací (podle nastavení THRESHOLD) kontroluje, jestli nějaké z vláken již původní hodnotu nenašlo (pokud ano skončí práci).

V případě nalezení správného hesla (hashe sedí) se toto heslo zapíše do paměti, kterou připravil host. Vzhledem k nízké pravděpodobnosti, že dvě vlákna naleznou řetězec se stejným otiskem se vůbec neřeší atomické operace.

### Rozšířený slovníkový útok:

Vzhledem k libovolné velikosti slovníku (teoreticky může být velký i několik TB) a předem neznámé velikosti RAM a VRAM program tento slovník „porcuje“ po blocích určité délky. Načítání slov na hostovi probíhá ve dvou fázích. V první fázi se zjišťuje, jak dlouhé je nejdelší slovo v dané skupině slov a kolik slov bude vlastně načteno. Všechna slova budou muset být zarovnána na délku nejdelšího slova (aby bylo možné se stringy smysluplně pracovat). Počet slov se určí v závislosti na nastavení GRANULARITY (určuje na kolik slov bude zarovnávat, např. pokud je GRANULARITY 100 a při 565. slovu nebude stačit paměť načte se pouze 500 slov) a MEMORY\_RATIO, který určuje, jak velkou část paměti VRAM využít na ukládání slov. Maximální délka slova je omezena konstantou MAX\_WORD\_LENGTH.

Po získání počtu slov v dané iteraci se následně tato slova načtou do paměti hostitele, vzhledem k nemožnosti použití strlen na device se tato informace ukládá na poslední bajt daného slova (vždy je alokováno max(strlen) + 2, jeden bajt pro null byte a druhý právě pro uložení délky aktuálního slova). Vzhledem k datovému rozsahu byte (resp. unsigned char) pro uložení maximální délky slova tato implementace nepočítá se slovy delších, než je 255 znaků.

Před samotným spuštěním kernelu je ještě do paměti symbolů nakopírován hash, slovníky a pravidla pro rozšířený slovníkový útok. Taktéž se připraví paměť na uložení řetězce, který má stejný hash jako hledaný hash. Poté se nakopírují slova ze slovníku do paměti VRAM, slova jsou uložena v 1D poli a každé slovo je zarovnané na délku nejdelšího slova + 2. Poté se konečně zavolá kernel, během vykonávání kernelu se připraví do paměti hosta další slova ze slovníku.

Na straně device se podle bloku a vlákna získá oblast paměti nad kterou má dané vlákno pracovat, poté je podobný postup jako v sekvenční variantě s tím rozdílem, že vždy po konstantním počtu vyzkoušených slov (určený v DICTIONARY\_THRESHOLD) se kontroluje, jestli již náhodou nebyl řetězec s odpovídajícím hashem nalezen.

## Měření:

|  |  |
| --- | --- |
| **Sestava 1 – Windows** | |
| **Procesor** | AMD Ryzen 5 1600 **@3.4GHz** |
| **Grafická karta** | MSI GeForce GTX 1080 SEA HAWK X (GDDR5X 10108MHz) GTX1080 (1708MHz) |
| **Paměť** | RAM: G.SKILL 16GB KIT DDR4 3200MHz CL14 Flare X for AMD |
| **Pevný disk** | Seagate BarraCuda 2TB 7200RPM |
| **Operačný systém** | Windows 10 Professional |
| **Poznámka:** | Úplná optimalizace (/Ox) |

### Parametry blocks a threads

Tyto dva parametry mají na výpočetní čas obrovský vliv, správné nastavení těchto parametrů dokáže chod programu značně urychlit, naopak špatně zvolené parametry délku dobu zvyšují. Bohužel měření ukazují, že neexistuje univerzální nastavení těchto parametrů pro všechna data, ke všemu se tyto parametry liší i grafická karta od grafické karty. I při špatném nastavení parametrů je výpočet na GPU rychlejší, než výpočet na CPU.

Následující data byla měřena na sestavě 1.

Nejkratší čas pro zebra1 vychází na **24 sekund** pro konfiguraci 96 vláken a 16 bloků, nejvyšší čas je **362 sekund** pro konfiguraci 128 vláken a 112 bloků. Průměrný čas napříč všemi konfiguracemi je **247 sekund.** Pro 64 vláken je průměrný čas **217 sekund,** pro 96 vláken **221 sekund** a pro 128 vláken **301 sekund**.