VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZPOZNÁVÁNÍ PODOBNOSTÍ SOUBORŮ NA ZÁKLADĚ CHOVÁNÍ

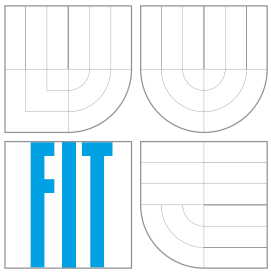
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DÁVID OTOČKA

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZPOZNÁVÁNÍ PODOBNOSTÍ SOUBORŮ NA ZÁKLADĚ CHOVÁNÍ

PROGRAM SIMILARITY RECOGNITION BASED ON BEHAVIOUR ANALYSIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. DÁVID OTOČKA
AUTHOR

VEDOUCÍ PRÁCE — Dr. Ing. PERINGER PETR
SUPERVISOR

BRNO 2009

Abstrakt

Cílem práce bylo navrhnout algoritmus, který na základě výstupu z analýzy chovaní programu, dokáže stanovit míru podobnosti s jinými programy. Pro potřeby algoritmu byla upravena Levenshteinova metoda pro výpočet rozdílu mezi dvěma řetězci a metoda NCD. U obou metod je v práci uveden spůsob jejich implementace a výsledky testů. V práci jsou popsány způsoby analýzy programů v prostředí virtuálního počítače i vysvětlení některých základních pojmů týkajících se analýzy škodlivého kódu.

Abstract

The goal of this master thesis was to design an algorithm that will be able to measure the difference between two programs based on their behavioral description. For the algorithm needs, the Levenshtein distance method between two strings and NCD method, were used. Both methods have their implementation approach and test result described. This term also discusses various methods of program analysis in virtual machine environment, as well as explanation of some basic concepts regarding malware analysis.

Klíčová slova

Malware, chovanie, DLL injection, API, analýza, podobnosť, Levenshteinova vzdialenosť, NCD

Keywords

Malware, behaviour, DLL injection, API, analysis, similarity, Levenshtein distance, NCD

Citace

Dávid Otočka: Rozpoznávání podobností souborů
na základě chování, diplomová práce, Brno, FIT VUT v Brně, 2009

Rozpoznávání podobností souborů na základě chování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Ing. Petra Peringera. Další informace mi poskytl pan Pavel Krčma. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Dávid Otočka
25. května 2009

Poděkování

Tímhle bych chtěl poděkovat vedoucímu panu Peringerovi za prvotní rady, které mi poskytl, i panu Krčmovi a AVG za jejich přístup a pomoc.

Obsah

Kapitola 1

Úvod

V posledných rokoch sa počet nových rodín/variánt škodlivých programov dramaticky zvýšil. Mnoho najviditeľnejších problémov na Internete má na svedomí obrovský ekosystém škodlivého software a nástrojov. Spam, phishing, odoprenie prístupu k službám, botnety a červy vo vysokej miere závisia na nejakej forme škodlivého kódu, obvykle sa nazývajúci ako *m*alware. Aj keď to z názvu práce nie je na prvý pohľad zrejmé, budem sa vo svojej práci zaoberať najmä analýzou vzoriek škodlivého kódu, pretože práve pri nich sa analýza ich chovania najviac používa.

Pochopenie a odhaľovanie kódu sa stáva z roka na rok ťažším problémom, ktorý je potrebné riešiť. Nanešťastie komplexnosť moderného škodlivých programov robí tento problém zložitejším. Ďalším limitujúcim faktorom je množstvo programov, ktoré sa šíri po internete, každým rokom sa objavuje čím ďalej tým viac variácií, s ktorými treba bojovať. V dôsledku tohto všetkého sa stala manuálna analýza škodlivého kódu neadekvátnou a neefektívnou. Preto musia byť používané automatické a robustné vyhodnocovanie techniky.

Cieľom tejto práce[1] je vysvetliť základné pojmy vo svete škodlivých programov, zoznámenie sa s metódami detekcie a klasifikácie takýchto programov, no najmä navrhnúť algoritmus pre odhaľovanie podobnosti škodlivých programov na základe ich textového popisu. V úvodných kapitolách popíšem problematiku skúmania malware, najmä problematiku skúmania malware vo virtuálnom počítači. Pred tým však ešte naznačím, čo je presne malware a prostredie virtuálneho počítača. Pomerne často bude v práci uvedené riešenie v rámci systému Windows, keďže programová časť práce bola vytváraná pre tento systém a najviac prípadov škodlivého kódu útočí práve na počítače so sytémom Windows. V kapitolách 2 a 3 uvediem spôsoby odhaľovania a zaznamenávania škodlivých vzoriek a v kapitole 4 spôsoby a možné algoritmy, ktorých cieľom je na základe zostavených záznamov určiť, ktoré vzorky sú si najpodobnejšie. V kapitole 5 popíšem mnou navrhnuté algoritmy pre porovnávanie škodlivých vzorkov a následne ich implementáciu. V záverečnej kapitole 7 predvediem výsledky testov implementovaného algoritmu. Kapitoly 2, 3 a časť kapitoly 4 sú inšpirované semestrálnym projektom, ktorý som spracoval na túto tému.

[1] Práca je robená pre externú firmu, v súvislosti s tým je veľa riešení prispôsobených konkrétnemu prípadu.

Kapitola 2

Škodlivé programy a ich správanie

V tejto kapitole uvediem, prečo v mojej práci budem pracovať s programami, ktoré sú považované za škodlivé. Uvediem čo znamená pojem malware, virtuálny počítač, prečo sa využíva na beh programov práve virtuálny počítač a ako sa malware v prostredí virtuálneho počítača správa. Záznam správania sa programu v počítači budem nazývať aj „behaviorálny“ popis[1].

2.1 Škodlivé programy

V ďalších častiach práce sa bude veľmi často používať pojem *m*alware, pretože práve pri jeho odhaľovaní a analýze sa virtuálny počítač a behaviorálny popis najviac používajú.

Malware je počítačový program určený k vniknutiu, alebo poškodeniu počítačového systému. Výraz malware vznikol zložením anglických slov *m*alicious(prekl. zákerný) a *s*oftware. Pod toto súhrnné označenie sa zahrnujú počítačové vírusy, trójske kone, červy a spyware [19]. Malware však nie je software, ktorý obsahuje chyby, ale bol napísaný pre legitímne účely [15]. Práve pri porovnaní a detekcii škodlivého kódu sa analýza behaviorálneho popisu najviac využíva a aj táto práca, bude inklinovať a mať na mysli toto využitie. Navyše je potrebné dodať ešte rozdelenie malware na parazitický a statický. Parazitický malware potrebuje k svojej existencii hostiteľa, ktorého by mohol napadnúť. Zpôsob napadnutia sa pri tom môže líšiť (napr. boot sektor, súbor, Word dokument ...). Ostatným typom malware sa hovorí jednotne statický malware. Pod toto označenie spadajú aj červy.

Vzhľadom na to, že je často ťažké rozlíšiť čo je trójksy kôň a čo zle napísaný program (napr. ServU server), zmenšuje sa nám rozlíšenie škodlivých vzoriek do dvoch skupín a to síce parazitického a „ostatného“ malware. Avšak parazitické vírusy sa v súčasnosti vyskytujú minimálne, tvoria asi 10% z celkového objemu malware [1]. Preto sa aj táto práca zaoberá hlavne porovnávaním druhej spomenutej skupiny.

[1] Toto pomenovanie je odvodené od anglického slova *behavior*, ktoré znamená správanie.

2.2 Analýza správania sa programov

Pojem analýza správania sa najčastejšie vyskytuje ak hovoríme o ľuďoch, alebo o zvieratách. Čo je vlastne analýza správania sa programov? Analýza správania sa programov je metóda, ktorá sa snaží identifikovať škodlivosť skúmaného programu na základe krokov, ktoré program vykonáva. Analyzuje sa najmä ako program komunikuje s komponentami systému a hlavne, ako sa správa voči samotnému operačnému systému.

Táto metóda nie je novinkou, ale je využívaná v mnohých rôznych podobách, nemusí riešiť vždy tie isté problémy a jej implementácia môže naberať rôzne podoby. Jedným z najdôležitejších aspektov pri tejto metóde je samotné získavanie dát o správaní sa programu, toto je časť keď systém, alebo nejaký iný program sleduje, čo vykonáva sledovaný program, dáta potom normalizuje, aby ich následne predal systému, ktorý analyzuje správanie sa programu. K analýze správania sa programov sa v drvivej väčšine prípadov používa prostredie virtuálneho počítača.

Virtuálny počítač

Virtuálny počítač je prostredie(obvykle program, alebo operačný systém), ktoré fyzicky neexistuje, ale je vytvorený v rámci iného prostredia. Je schopný spúšťať programy ako klasický počítač, tak isto používa vlastný operačný systém, len mu sú pridelené nami stanovené prostriedky. Hovorí sa, že „virtualizujeme“ hardware. V našom prípade využívame virtuálny počítač preto, lebo testujeme potenciálne nebezpečné súbory. Takéto prostredie sa nazýva *S*andbox. Spustením týchto súborov v prostredí, ktoré vieme kontrolovať a ľahšie zisťovať zmeny, ktoré sa v ňom udiali, zabránime škodám na našich fyzických počítačoch. Správne nastavenie virtuálneho počítača, ktoré ale nie je predmetom tejto práce možno nájsť na [13].

Všeobecne najvyužívanejším prostredím, ktoré sa využíva pri simulácii virtuálneho počítača na analýzu programov je podľa mnohých *VMware*. Výhodou použitia virtuálneho prostredia je aj to, že umožňuje simuláciu viacerých systémov v rámci jedného fyzického počítača.

Výhody využitia virtuálneho počítača pre analýzu programov

Výhod prečo je dobré na analýzu programov, ktoré sú potenciálne škodlivé, použiť virtuálny počítač je viacero. Často je veľmi prospešné mať v testovacom laboratóriu viacero systémov, aby sa dalo sledovať ako program pracuje s komponentmi simulovaného internetu. Práve s virtuálnym počítačom je možné vybudovať multikomponentné laboratórium pre sledovanie programu bez toho, aby bolo potrebných viacero fyzických počítačov.

Vziať stav systému pred samotným spustením programu a infekciou systému a následné branie stavov systému v istých periodických intervaloch po infekcii, šetrí čas aj pre samotnú analýzu. Táto funkcionalita prostredia virtuálneho počítača umožňuje ak je potrebné a žiaduce, vrátiť zmeny v systéme do predchádzajúceho stavu takmer okamžite. Najmä prostredie VMware je pre tieto prípady veľmi dobre vybavené.

Virtuálne prostredia podporujú host–only nastavenie siete pre vzájomné prepojenie viacerých virtuálnych systémov a využívajú simulovanú sieť bez ďalšieho pridaného hardware. Toto nastavenie je užitočné aj z toho dôvodu, že nie je potrebné a dokonca nežiaduce prepájať laboratórnu sieť s vonkajšou sieťou. Navyše nastavenie host–only umožňuje monitorovanie kompletnej komunikácie, ktorá prebieha simulovanou sieťou a preto je monitorovanie správania sa programu na sieti ešte jednoduchšie [21].

Získavanie dát pre analýzu programov

Na získavanie dát o správaní sa programu môže byť použitých viacero metód. Väčšina metód využíva aplikácie, alebo služby poskytnuté priamo operačným systémom(napr. Monitor procesov) na získanie niektorých, alebo všetkých dát o správaní sa programu. Veľká časť metód taktiež využíva zachytávanie funkcií API pre získanie ďalších dát, ktoré nie sú poskytované oficiálnymi rozhraniami. Ďalšími možnosťami môže byť *p*olling, čo znamená neustále vyzývanie zariadenia na zasalanie dát druhým zariadením, alebo iné prístupy. Samotný operačný systém Windows poskytuje niekoľko zabudovaných spätných volaní[2], ktoré umožňujú získavanie behaviorálneho popisu dát. Spätné volanie je vykonávateľný kód, ktorý je predávaný ako argument inému kódu. Niektoré sú implementované priamo v jadre systému, iné v užívateľskom móde. Napríklad práca s registrami, ako aj akcie vykonané na súborovom systéme sú sledovateľné z jadra systému Windows. Takisto je možné sledovať spustenie, či ukončenie programov, ako aj správanie sa na sieti.

Zachytávanie API funkcií sa často používa na doplnenie indormácií v získaných dátach o správaní sa programu, ktoré sa nepodarilo získať prostredníctvom spätných volaní. Táto metóda nie je podporovanou metódou na získavanie dát, keďže nie je explicitne podporovaná operačným systémom. Zachytávanie API funkcií je obvykle implementované zmenou ukazateľov na funkcie, modifikáciou tabuliek v DLL, alebo prepísaním strojového kódu funkcie. Aj keď je to nepodporovaná metóda je veľmi užitočná pri získavaní behaviorálneho popisu, keďže umožňuje komplexnejší pohľad na daný program [18]. Viac o zachytávaní volania API funkcií bude uvedené v kapitole 3. Príklad behaviorálneho popisu:

```
SX: 00000003
SY: 00000003
PI: 00000240
TI: 00000244
PN: c:\002b11498c2d273f035c4d848ead2fbf.ex
MN: \SDBX
FN: OPEN_SECTION
00: 00000000
01: USERENV.dll
02: 0000000E
03: 000007CC
SZ: 00000003
```

[2]angl. Callback - vykonávatelný kód, ktorý je predávaný ako argument inému kódu.

Ako je možné vidieť, takto získané dáta nie sú pre človeka príliš čitateľné. Avšak po aplikácii API hooking a pridaní popisu jednotlivých akcií môže vyzerať získaný popis programu aj takto:

```
*** c:\00033033daaf0ea1586bbefdca31ad30.ex:
Try connect to network
Write to file
C:\Program Files\Common Files\Microsoft Shared\MSINFO\paramstr.txt
Run file C:\program files\internet explorer\IEXPLORE.EXE
*** C:\program files\internet explorer\IEXPLORE.EXE:
Try connect to network
Copy file c:\00033033daaf0ea1586bbefdca31ad30.ex to
C:\Program Files\Common Files\Microsoft Shared\MSINFO\Scvhost.exe
Create service Scvhost (Scvhost) with file name
C:\Program Files\Common Files\Microsoft Shared\MSINFO\Scvhost.exe
Delete file c:\00033033daaf0ea1586bbefdca31ad30.ex
Connect to 10.2.1.100:80
Download file http://10.2.1.100/wpad.dat
DNS query my.xu-chen.cn
Connect to 10.2.1.100:80
Download file http://my.xu-chen.cn/ip.txt
Copy file C:\Program Files\Common Files\Microsoft Shared\MSINFO\Scvhost.exe to
C:\WINDOWS\System32\_Scvhost.exe
Run file C:\WINDOWS\System32\calc.exe
*** C:\WINDOWS\System32\calc.exe:
Try connect to network
```

Z takéhoto popisu už je pre človeka oveľa zrejmejšie, akú činnosť program v počítači vykonáva. Viac o analýze a spôsoboch analýzy jednotlivých programov je možné nájsť v kapitole 3.

Kapitola 3

Metódy analýzy programov vo virtuálnom počítači

V nasledujúcej kapitole uvediem, aké metódy sa používajú pri analýze programov vo virtuálnom počítači. Upozorňujem, že vymenované budú len základné z nich, pretože existuje mnoho variácií konkrétnych metód.

3.1 Metódy statickej analýzy

Ak sa začína analýza programu, ktorý môže potenciálne byť malware, najskôr sa väčšinou začína statickou analýzou. V nasledujúcich častiach tejto kapitoly zhrniem metódy statickej analýzy, ktoré nevyžadujú rozsiahle programovanie. Detailná statická analýza totiž zahrňuje najmä analýzu zdrojového kódu skúmaného programu.

Statická analýza je vo všeobecnosti bezpečnejšia ako dynamická, hlavne preto, že pri statickej analýze sa samotný kód programu nevykonáva, takže sa pri nej nezmažú, ani neukradnú žiadne dáta. Pravdepodobne jediné riziko pri statickej analýze programu je jeho náhodné, alebo chybou spôsobené spustenie. Ale aj tomuto kroku sa dá zabrániť tým, že sa statická analýza vykonáva na operačnom systéme, pre ktorý nebol daný program navrhnutý.

Identifikácia súboru

Predtým ako sa začne s čímkoľvek iným je doporučené vypočítať kryptografickú *hash* hodnotu každého súboru, ktorý je skúmaný. Je mnoho *h*ash funkcií, ale odporúča sa použiť tie, ktoré sú pravdepodobne najčastejšie používané a to MD5, SHA1, alebo SHA256. Po vypočítaní hash hodnoty môže byť táto hodnota periodicky kontrolovaná k určeniu, či bol program modifikovaný, alebo modifikoval sám seba [13].

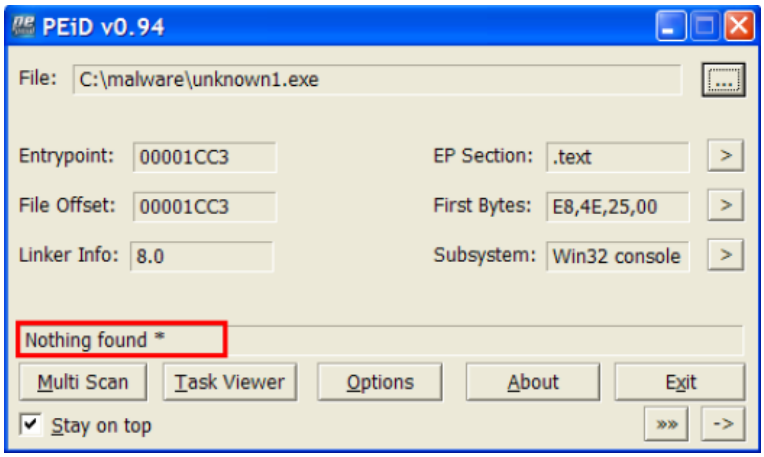
Hľadanie vírusu

Ak je podozrenie, že súbor môže obsahovať časti malware, vždy je dobré skontrolovať ho antivírovým programom. Ak antivírový program rozpozná daný malware, potom k nemu väčšinou uvedie aj stručnú analýzu, ktorá daný malware popisuje. Väčšinou sú informácie

podávané antivírovými programami minimálne, no niekedy môže byť vo výpise uvedené, aké sú znaky daného programu, jeho schopnosti a aj návod ako ho odstrániť. Ak používame virtuálny počítač je ideálne využiť služby niektorého z online vyhľadávacích klientov.

Detekcia kompresie, alebo modifikácie

Najväčšou komplikáciu pri analýze programov a malware je rozšírenie programov, ktoré modifikujú spustitelný súbor s cieľom zakryť programovú logiku pred očami analyzátora. Programy, ktoré modifikujú, alebo komprimujú obsah súborov iných potenciálne nebezpečných programov sa nazývajú „baliče“[1]. Ak takýto program zkomprimuje, či zašifruje kód, alebo spustitelný program, potom zkomprimovaný program vyzerá z pohľadu statickej analýzy úplne inak, ale vykonáva tú istú činnosť, ako keby nebol zkomprimovaný. Akonáhle bol program zkomprimovaný, je veľmi ťažké z pohľadu statickej analýzy zistiť programovú logiku, či metadáta. Z tohto dôvodu je dobré využiť program, ako napríklad PEiD, ktorý ma vo svojej databáze cez 600 rôznych kompilátorov. Je schopný odhaliť, či bol daný spustitelný súbor zkomprimovaný alebo nie, ale aj to len v obmedzenom množstve [13]. Ukážka neúspešnej detekcie kompilátora je na obrázku 3.1. Statická analýza nad takýmto súborom stráca zmysel.



Obrázok. 3.1: Ukážka programu PEiD.

Reťazce nachádzajúce sa v programe

K pochopeniu, čo daný program robí by bolo ideálne mať prístup k manuálu, ktorý krok po kroku mapuje funkcie a možnosti analyzovaného programu. Samozrejme škodlivé programy obyčajne pri sebe taký manuál nemávajú, ale veľa sa dá zistiť z analýzy vstavaných reťazcov vo vnútri programu. Napríklad programy často vypisujú užívateľovi svoj stav, alebo či sa náhodou niekde nevyskytla chyba. Takéto reťazce znakov obvykle končia vstavané do spustiteľného programu a ich analýza môže byť veľmi užitočná. Existuje mnoho

[1] angl. packers

programov napr. Sysinternals, Bintext a pod., ktoré vedia zo spustitelného súboru vytiahnuť reťazce. Pri ich analýze sa väčšinou zameriavame na tie dôležitejšie, ale treba si dať pozor na ich význam, pretože môžu byť v programe použité práve preto, aby analyzátora zmiatli [13].

Rozklad programu na strojový kód

Ďalším a väčšinou posledným krokom statickej analýzy je rozklad spustitelného súboru a analýza strojových inštrukcií, ktoré program tvoria. Opäť existuje mnoho programov schopných rozložiť spustitelný súbor na jeho strojové inštrukcie, najčastejšie používaným je však IDA Pro. Táto technika už však požaduje mať potrebné znalosti strojového kódu a je ďaleko zložitejšia, ako všetky vopred menované [13]. Možný výstup programu je zobrazený na obrázku 3.2. Väčšinou rozkladom súboru a analýzou jeho zdrojových kódov vieme určiť, či je daný program škodlivý, alebo nie, nevýhodou je, že takýto proces je ťažké automatizovať.

```
0100762C                push    eax             ; dwLanguageId
0100762D                push    eax             ; dwMessageId
0100762E                lea     eax, [ebp+Buffer]
01007634                push    eax             ; lpSource
01007635                push    FORMAT_MESSAGE_ALLOCATE_BUFFER or FORMAT_MESSAGE_FR
0100763A                call    ds:FormatMessageW
01007640                mov     eax, [ebp+var_4]
01007643                leave
01007644                retn
01007644 sub_10075F7    endp
01007644
01007645
01007645 ; ---------------  S U B R O U T I N E  -------------------------------------
01007645
01007645 ; Attributes: bp-based frame
01007645
01007645 sub_1007645    proc near               ; CODE XREF: sub_10064BB+15C↑p
01007645                                        ; sub_1007645+AA↓p
01007645
01007645 Type           = dword ptr -14h
01007645 cbValueName    = dword ptr -10h
01007645 phkResult      = dword ptr -0Ch
01007645 var_8          = dword ptr -8
01007645 cbData         = dword ptr -4
```

Obrázok. 3.2: Možný výstup programu IDA Pro.

3.2 Metódy dynamickej analýzy

Niektoré metódy dynamickej analýzy boli spomenuté už vyššie. V tejto kapitole ich bližšie rozvediem a nastienim, čo znamená dynamicky analyzovať program v prostredí virtuálneho počítača a predstavím aj niektoré ďalšie nástroje, ktoré je možné využiť pri dynamickej analýze. Dynamická analýza pozoruje a zaznamenáva správanie sa programu pri jeho vykonávaní, v našom prípade v prostredí, ktoré sa nazýva *sandbox*, väčšinou pri dynamickej

analýze vieme, že ide o malware a preto ho púšťame v kontrolovanom prostredí. Pri dynamickej analýze rozlišujeme dva globálne prístupy:

- Zaznamenanie stavu virtuálneho počítača pred vykonaním programu a následné porovnanie s konečným stavom systému po dokončení vykonávania programu.
- Monitorovanie akcií vykonávaných programom počas celého jeho behu za pomoci špecializovaného nástroja napr. debuggeru.

Prvá možnosť je ľahšia na implementáciu avšak dáva len veľmi hrubé výsledky, ktoré však v niektorých prípadoch stačia na to, aby sme vedeli určiť, čo binárny súbor vykonáva. Tento prístup však analyzuje len celkový efekt programu na systém bez toho, aby bral do úvahy dynamické zmeny. Napríklad vytvorenie súboru a jeho zmazanie tesne pred koncom vykonávania. Druhý prístup je zložitejší, ale práve ním sa bude táto práca zaoberať, keďže dáva lepšie a podrobnejšie výsledky.

Dynamická analýza má na rozdiel od statickej nevýhodu v tom, že v danom časovom momente je možné analyzovať len jednu vzorku, ktorá je práve spustená, kdežto statická analýza skúma zdrojový kód a tým dáva možnosť analyzátorom pozorovať všetky možné vykonávania [9].

Monitor procesov

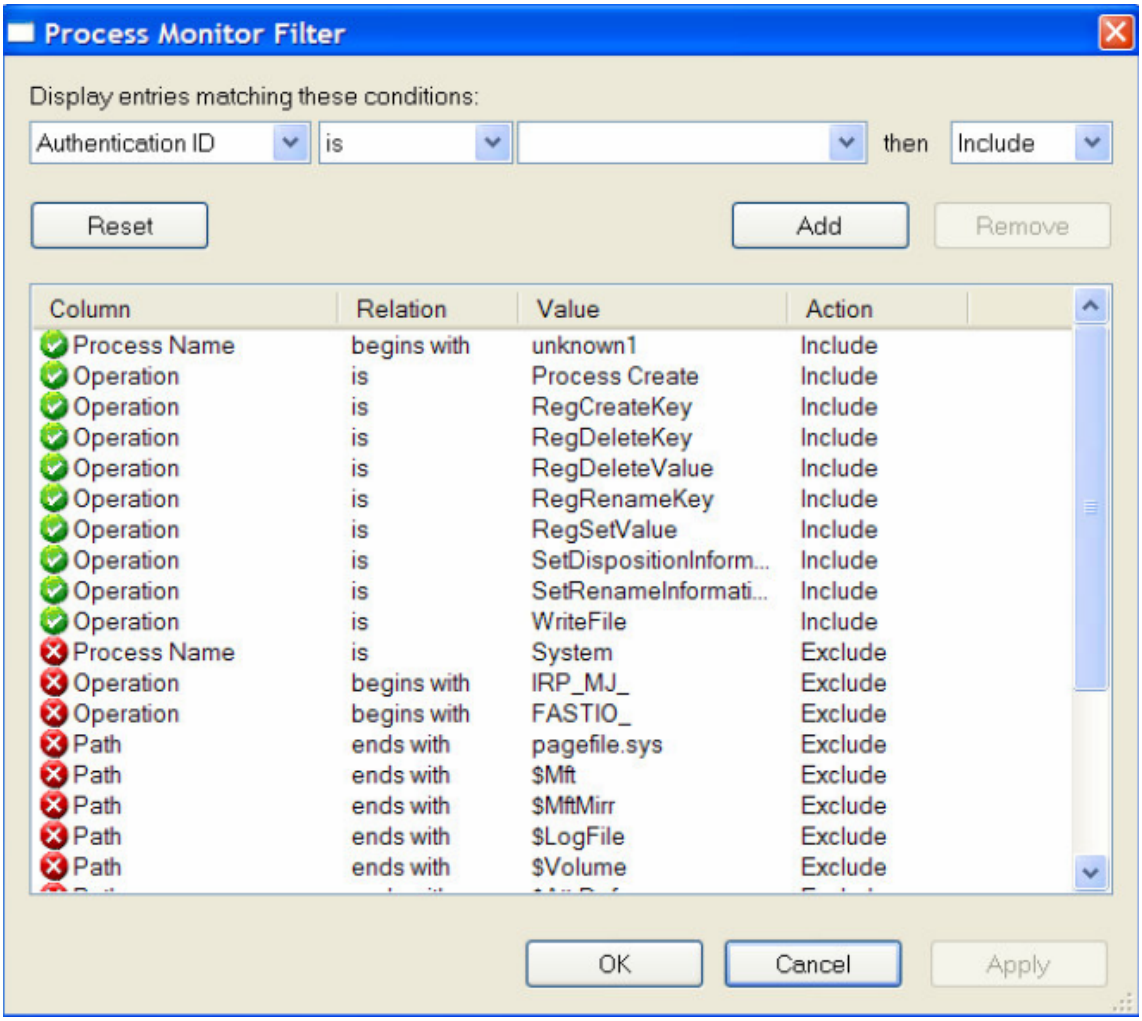
Monitor procesov je nástroj *SysInternals*, ktorý umožnuje užívateľom monitorovanie všetkých súborov, registrov a aktivít jednotlivých procesov v systéme Windows. Pri takomto nástroji je väčšinou najdôležitejšie, jeho počiatočné nastavenie, aby nás nezahlcoval zbytočnými informáciami. V tomto nástroji možno nastaviť ľubovolné filtrovanie správ a tým pádom je hneď možné odhaliť potencionálne škodlivé vzorky. Príklad efektívneho nastavenia filtra je možné vidieť na obrázku 3.3.

Ďalšie informácie týkajúce sa monitora procesov je možné nájsť v [13]. Podobne použiteľným nástrojom, ale na kontrolu sieťových aktivít je program *Wireshark*.

DLL injection

Ak v súčasnosti hovoríme o technike DLL injection hovoríme o zdrojovom kóde, alebo DLL[2], ktorá je vložená priamo do pamäte bežiaceho procesu, v rámci ktorého chceme náš kód vykonať. Výhodou je, že systém Windows bol navrhnutý pre podporu takejto techniky a preto môže byť technika DLL injection využívaná akoukoľvek aplikáciou [17]. Nevýhodou je použitie tejto techniky v aplikáciách, ktoré ju využívajú na ohrozenie bezpečnosti užívateľa. Je totiž do značnej miery používaná v *s*pyware a *r*ootkitoch. Využitie tejto techniky je úzko spojené so zachytávaním volaní API funkcií, vlastne je to jedna z techník ako realizovať zachytávanie volaní API funkcií. V nasledujúcich odstavcoch ukážem viacero spôsobov, ktorými sa dá realizovať vloženie kódu, ktorý sa vykoná v rámci inej aplikácie a prostredníctvom neho zachytávať volania API funkcií, alebo vykonávať inú činnosť.

[2]angl. Dynamic Linked Library – dynamická knižnica umožňujúca programom zdieľanie kódu k vykonávaniu osobitných funkcií. Systém Windows obsahuje Dll, ktoré v sebe zahŕňajú funkcie a zdroje umožnujúce programom bežať v prostredí Windows.



Obrázok. 3.3: Možný filter monitoru procesov.

Vloženie realizované pomocou registrov

Systém Windows, prehliadač systému Windows, ako aj mnoho ďalších aplikácií využíva systémovú knižnicu *u*ser32.dll. K tomu, aby sa v tomto prípade naša knižnica načítala do cieľovej aplikácie stačí len vložiť meno knižnice ako hodnotu do nasledujúceho registru:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion
\Windows\AppInit_DLLs
```

Hodnota registru obsahuje jednu, alebo viacero dynamických knižníc oddelených medzerou, alebo čiarkou. Všetky knižnice obsiahnuté v registre sú načítané vo všetkých aplikáciách založených na Windows, ktoré bežia v rámci aktuálneho sedenia[3] [10]. Tento jednoduchý prístup má ale niekoľko nevýhod:

- K aktivácii, alebo deaktivácii procesu vloženia je potrebný reštart systému.

[3]angl. session

- Knižnica bude vložená len do aplikácií, ktoré používajú *u*ser32.dll, konzolové aplikácie vo väčšine prípadov funkcie z tejto knižnice nepoužívajú.
- Nemáme žiadnu kontrolu nad procesom vloženia. Knižnica je vložená do každej jednej aplikácie vyžívajúcej *u*ser32.dll, bez ohľadu na to, či chceme, alebo nie.

Vloženie pomocou filtrovania správ Windows

Systém zasielania správ vo Windows umožňuje inštaláciu filtrov pre zasielané správy. K inštalácii filtrov Windows poskytuje rozhranie, ktoré je schopné vložit zvolenú knižnicu do adresového priestoru každého procesu. Funkcia *S*etWindowsHookEx môže byť využitá k zachyteniu jednej, alebo viacerých systémových udalostí. Zachytenie môže byť vytvorené pre ktorúkoľvek metódu vstupu, alebo správy Windows vytvorenej pre jednu konkrétnu aplikáciu. Najčastejším cieľom zachytávania sú aplikácie bežiacej na rovnakej pracovnej ploche, ako vlákno volajúce spomenutú funkciu. Výsledkom je, že výsledky zachytených volaní, môžu byť potom pozmenené [11].

Vloženie pomocou ladenia podsystému

Podsystém ladenia, ktorý poskytuje systém Windows umožňuje jednej aplikácii ladenie a ovplyvňovanie zdrojového kódu inej aplikácie. Ak má užívateľ využívajúci nástroj na ladenie dostatočné práva v rámci systému, je schopný modifikáciou zdrojového kódu spustiť v rámci inej aplikácie nové vlákno, alebo priamo čítať a zapisovať do adresového priestoru aplikácie. Funkcie, ktoré je možné v takomto prípade použiť:

- *C*reateRemoteThread – Funkcia umožňuje spustenie kódu na ďialku v adresovom priestore akéhokoľvek bežiaceho procesu, ku ktorému máme prístupové práva. Jedným z typických využití je volanie *C*reateRemoteThread počas špecifikácie adresy vo funkcii *L*oadLibrary a mena našej knižnice. Tento proces umožní nahranie našej knižnice do adresového priestoru cieľovej aplikácie. Akonáhle sme v adresovom priestore cieľovej aplikácie, sme schopní meniť výsledky API volaní [11].
- *W*riteProcessMemory – Umožnuje zapísať kód do ktorejkoľvek časti pamäte procesu, ku ktorému máme prístupové práva. Potom pomocou funkcie *S*etThreadContext môžeme modifikovať rozšírený ukazateľ inštrukcií[4] vlákna, aby presmeroval vykonávanie vlákna na novo zapísané bajty do pamäte. Táto metóda funguje podone ako keď použijeme funkciu *C*reateRemoteThread, s tým rozdielom, že sa nenačíta žiadna nová knižnica a vložený kód existuje len v pamäti konkrétneho procesu [11].

Všetky spomenuté techniky možno využiť v rámci odhaľovania malware, alebo v rámci akejkolvek inej aplikácie, kde potrebujeme zdielať kód. Problémom je ich časté využitie malwarom, pretože sa odhaľujú len veľmi ťazko.

[4]angl. Extended Instruction Pointer(EIP) – obsahuje adresu ďalšej inštrukcie, ktorá sa má vykonať [22].

Zachytávanie volaní API funkcií

Programátori používajú API, čo je vlastne aplikačné rozhranie systému ako prístup k vzdialeným zdrojom. Aplikácie využívajú API radšej ako priame systémové volania a tým ponúkajú možnosť dynamickej analýzy. Sme schopní monitorovať relevantné volania API a ich parametre. Po vložení našej knižnice, alebo kódu do adresového priestoru jednou z techník popísaných vyššie sme schopní zachytiť a modifikovať volanie a výsledok volania API funkcie [11]. Na zachytenie sa obvykle používajú dve techniky, ktoré predstavím v nasledujúcich odstavcoch.

Zachytenie pomocou tabulky importovaných funkcií

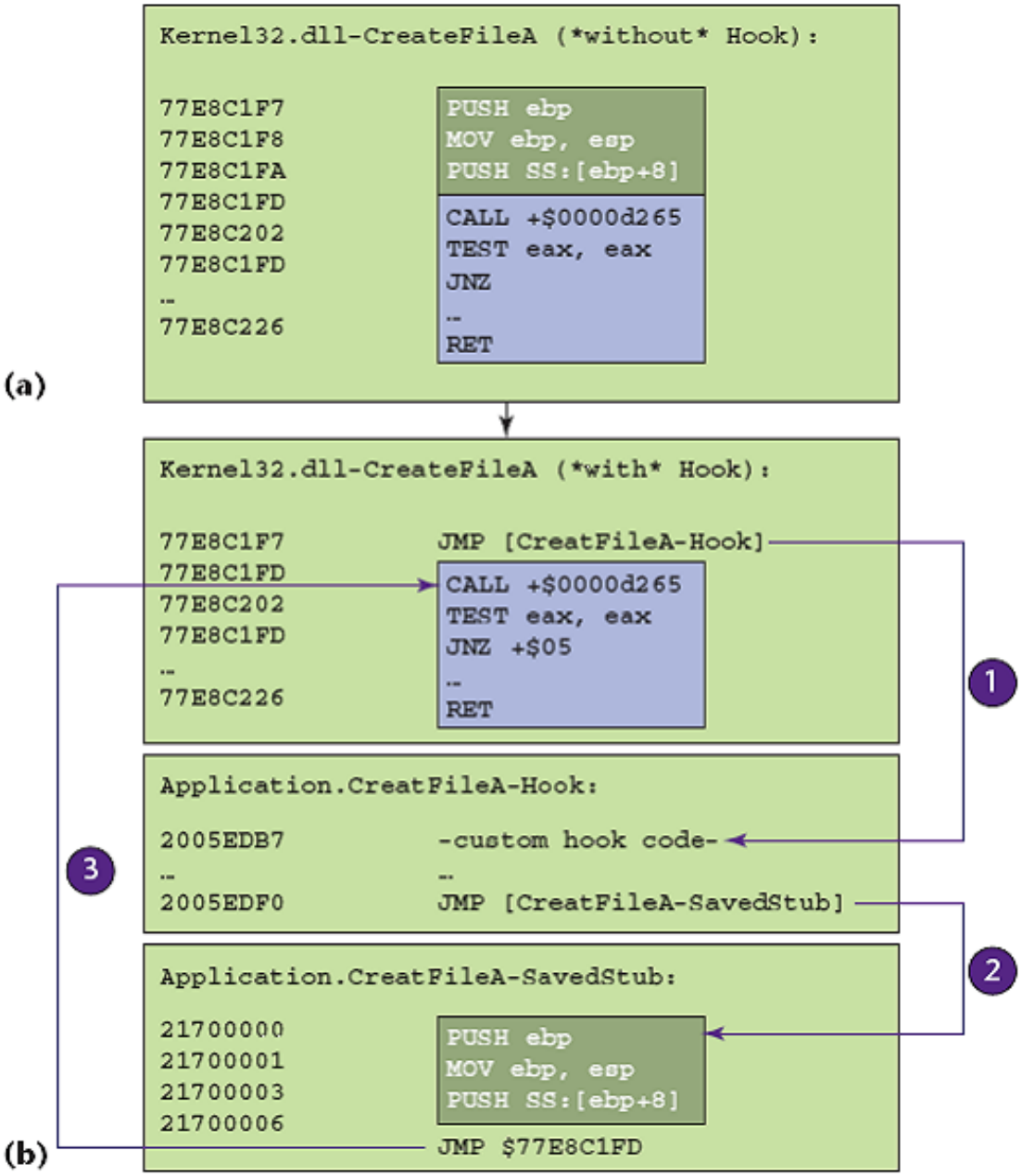
Táto metóda využíva fakt, že pri kompilácii programu, nie sú všetky volania API funkcií spojené s knižnicami, v ktorých sa nachádzajú. Volania týchto API funkcií sú presmerované cez tabulku importovaných funkcií[5] využitím štandardných inštrukcií strojového jazyka. Pri načítaní pamäte procesom sa vyhodnotia adresy v tabulke importovaných funkcií a inštrukcie začnú vykonávať kód na získanej adrese. Toto umožnuje binárnym súborom spustitelnosť a funkčnosť na viacerých operačných systémoch bez potreby znovu kompilovať program. Akonáhle je naša knižnica v adresovom priestore cieľového procesu, funkcie v nej môžu spracovať formát prenositelného spustitelného súboru a nájsť umiestnenie API funkcie, ktorú chceme ovplyvniť v tabulke importovaných funkcií. Potom už len treba nahradiť API funkciu, našou funkciou. výsledkom bude vykonanie našej funkcie vždy keď je volaná nami nahradená API funkcia. Viac informácií o tomto spôsobe zachytenia volania API, ako aj o tom ako vyzerá formát prenositelného spustitelného súboru sa možno dočítať v [11].

Zachytenie pomocou inline funkcií

Na rozdiel od metódy zachytávania pomocou tabulky importovaných funkcií táto metóda modifikuje funkcie priamo v základných systémových knižniciach(napr. *k*ernel32.dll). Obrázok 3.4 ukazuje in–line prepisovanie kódu API funkcie v DLL. V časti *a* je pôvodný kód funkcie, v časti *b* prepisuje inštrukcia JMP prvý blok funkcie a predáva kontrolu našej funkcii kedykoľvek, keď analyzovaná aplikácia zavolá danú API funkciu (1). Naša funkcia vykoná požadované úkony a potom zavolá blok API funkcie, ktorý má uložený (2). Uložený blok vykoná prepísané inštrukcie a presunie sa k nemodifikovanej časti API funkcie (3)[9].

V našom prípade využívame zachytávanie volaní API funkcií na zistenie krokov, ktoré skúmaný program vykonáva, čiže na zaznamenávanie jeho činnosti. Podrobnejšie o zachytávaní volaní API funkcií sa možno dočítať v [9], [11], [10]. Viac praktických skúseností, tipov možno o analýze malware nájsť v [13], alebo [6].

[5] angl. Import Address Table – tabulka obsahujúca ukazatele na API funkcie, ktoré program využíva, ich knižnicu a ich adresu v pamäti [11].



Obrázok. 3.4: Prepisovanie vstupného bodu API funkcie.

Kapitola 4

Reprezentácia, algoritmy porovnania a klasifikácie programov

Cieľom tejto kapitoly je ukázať, akým spôsobom môže byť analýza programu reprezentovaná v počítači a akým spôsobom možno porovnať výstupy analýzy programu a určiť ich vzájomnú podobnosť. Taktiež bude nastienená otázka klasifikácie, pre zaradenie jednotlivých druhov malware do tried kvôli ľahšiemu rozpoznávaniu škodlivých programov a vzájomnému spájaniu podobných programov do tried. Na to, aby sme výstup z analýzy mohli porovnávať musíme dáta ukladať a vhodne reprezentovať, aby sa s nimi dalo pracovať.

4.1 Reprezentácia správania programu

Výsledky analýzy sa väčšinou ukladajú do databázy a uložený popis môže mať rôzne formy, záleží na tom, aká metóda bola zvolená k analýze. Môžeme mať napríklad rozložený zdrojový kód programu, či popis postupne vykonávaných strojových inštrukcií, alebo len uchované stavy systému pred, počas a po spustení programu. Táto práca sa bude zaoberať špecifickým textovým popisom správania súboru, ktorý bude vysvetlený nižšie a bol získaný metódou zachytávania API funkcií.

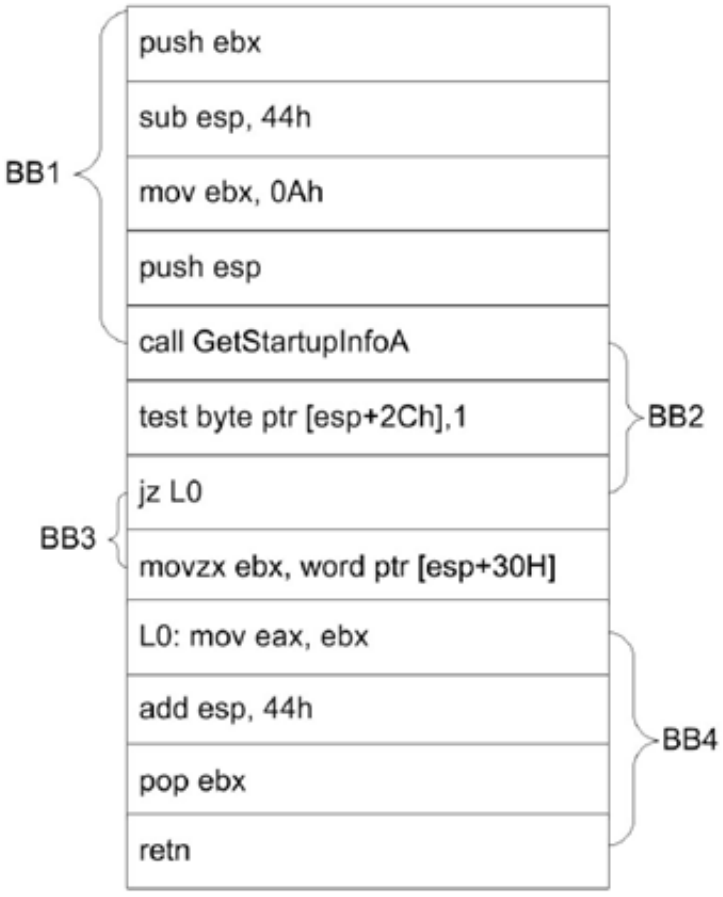
Jedným zo spôsobov reprezentácie je ukladanie identifikácie súborov, ktoré sú hashované napríklad algoritmom MD5 priamo do databáze, v rôznych fázach spusteného systému, pričom k nim môžu byť doložené počty, koľkokrát daný program pracoval s registrami, súbormi, či so sieťou. Takúto reprezentáciu znázorňuje tabulka 4.1 [3].

Ďalším spôsobom môže byť ukladanie blokov inštrukcií strojového kódu vykonávaného programu a k nim zostavenie príslušného diagramu riadiaceho toku [1] Za blok sa považuje postupnosť inštrukcií v strojovom kóde, ktoré neobsahujú inštrukciu skoku, alebo cieľ inštrukcie skoku. Príklad takéhoto spôsobu je možno vidieť na obrázku 4.1 a obrázku 4.2, viac o ňom je možné sa dozvedieť v [7].

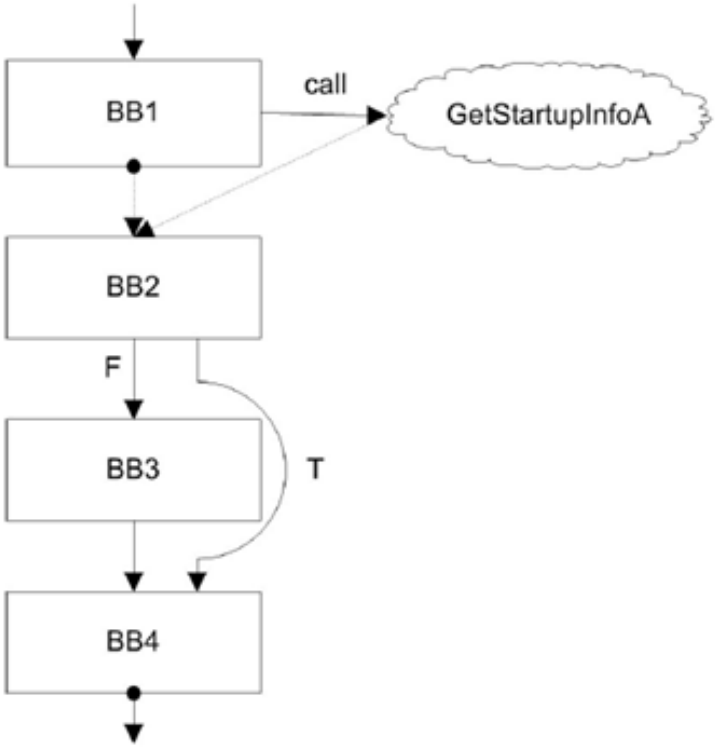
[1] angl. Control Flow – diagram znázorňujúci postupnosť operácií programu prostredníctvom grafu [2].

MD5	P/F/R/N
71b99714cddd66181e54194c44ba59df	8/13/27/0
be5f889d12fe608e48be11e883379b7a	8/13/27/0
df1cda05aab2d366e626eb25b9cba229	1/1/6/1
5bf169aba400f20cbe1b237741eff090	1/1/6/2
eef804714ab4f89ac847357f3174aa1d	1/2/8/3
80f64d342fddcc980ae81d7f8456641e	2/11/28/1
12586ef09abc1520c1ba3e998baec457	1/4/3/1
ff0f3c170ea69ed266b8690e13daf1a6	1/2/8/1
36f6008760bd8dc057ddb1cf99c0b4d7	3/22/29/3
c13f3448119220d006e93608c5ba3e58	5/32/28/1

Tabulka 4.1: 10 unikátnych vzoriek malware. Pre každú vzorku je v druhom stĺpci uvedený počet zaznamenaných operácií s procesmi, súbormi, registrami, alebo operácie spojené so sieťou.



Obrázok. 4.1: Inštrukcie strojového kódu so základným označením blokov.



Obrázok. 4.2: Diagram riadiaceho toku pre príklad z obrázka 4.1.

V našom prípade máme uložený textový popis akcií vykonaných skúmaným programom počas jeho behu vo virtuálnom počítači. Textový súbor bol vytvorený s cieľom zrozumiteľne znázorniť činnosť programu bez toho, aby sa musel skúmať zdrojový kód. Celý tento popis, ktorý bol spracovaný programom napísanom v jazyku C, je uložený ako reťazec v databázi. Príklad takéhoto reťazca je možné vidieť na obrázku 4.3.

```
"*** c:\0060d5241ea2fefb00fec6c2c6481d21.ex  :
Try connect to network
Copy file ""c:\0060d5241ea2fefb00fec6c2c6481d21.ex"" to ""C:\Program Files\Common
Files\Microsoft Shared\MSINFO\rejoice91.exe""
Write to file ""C:\AutoRun.inf""
Copy file ""c:\0060d5241ea2fefb00fec6c2c6481d21.ex"" to ""C:\rejoice91.exe""
Create service Windows_rejoice2007_91 (Windows_rejoice2007_91) with file name
""C:\Program Files\Common Files\Microsoft Shared\MSINFO\rejoice91.exe""
Run file ""C:\Program Files\Common Files\Microsoft Shared\MSINFO\rejoice91.exe""

Heuristic score=9, guess=NEW VIRUS

  EXE file
```

Obrázok. 4.3: Ukážka jednej z možných textových reprezentácií správania sa malware.

4.2 Metódy porovnávania reprezentácií malware

V riadkoch tejto podkapitoly uvediem niektoré základné techniky, ktoré sa využívajú pri vzájomnom porovnaní a klasifikácii jednotlivých reprezentácií malware. Nebudem všetky algoritmy podrobne rozpisovať keďže to nie je predmetom tejto práce, ale najmä preto, že spomenuté algoritmy sa využívajú najmä pri porovnaní nebehaviorálnych reprezentácií.

N–gramy a n–permy

Využitie n–gramov je bežná technika v zpracovaní textu. N–gram je reťazec n znakov, ktoré sa objavujú v určitej sekvencii. Pri využívaní n–gramov ako nástroja na analýzu malware je daný program rozložený na sekvenciu n znakov, čo môžu byť buď len čisté bajty, výrazy v assembleri, zdrojové riadky a iné. Čím viac sa n posúva k číslu 1, tým menší význam má zvolená sekvencia. Snáď prvé využitie n–gramov pre analýzu malware bolo výskumnou skupinou IBM, ktorý skúmali metódu na automatické rozpoznávanie vírusov napadajúcich $b$oot sektory [12].

N–permy sú variáciou n–gramov. Pre každú sekvenciu znakov, ktorá može byť n–gramom, n–perm predstavuje každú možnú permutáciu danej sekvencie. N–permy sú s n–gramami identické až na to, že pri n–permoch je poradie znakov vo vybranej sekvencii vzhľadom na provnávanie irelevantné. Napríklad $a$bcab je reťazec obsahujúci tri 3–gramy, $a$bc, $b$ca a $c$ab, každý práve s jedným výskytom. Avšak len jeden 3–perm $a$bc s-tromi výskytmi. Je preto jasné, že n–permov je nepomerne menej ako n–gramov, ale zároveň sú n–permy toleratnejšie vzhľadom na poprehadzovanie znakov [12].

Oba prvky, ako n–gramy, tak aj n–permy môžu byť využité a upravené meraniu podobnosti, ktorá bude založená na vektoroch výskytu n–permov, či n–gramov. Takéto vektory môžu byť reprezentáciou programov a to tak, že jeden vektor reprezentuje jeden program. Podobnosť a klasifikácia môže byť merateľná výpočtom kosínusu dvoch vektorov:

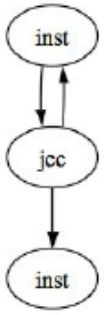
$$\frac{V_1 V_2}{|V_1||V_2|} \tag{4.1}$$

Daná metóda sa používa najmä pri porovnaní rozloženého kódu malware. Jej výhoda je rýchlosť a relatívna jednoduchosť, taktiež by mala byť odolná voči preusporiadaniu kódu, alebo voči posunom kódu. Nevýhodou je, že odolnosť voči posunu kódu sa naruší akonáhle je n väčšie ako blok, ktorý sa posúva. Tento fakt velkosť n dosť limituje, pretože čím je n kratšie tým nesie o skúmanom programe menšiu informáciu a ako je už spomenuté vyššie aj menší význam z pohľadu porovnávania [5]. O rôznych spôsoboch využitia n–permov a n–gramov sa možno dočítať aj v [23] a [16]. O tom, ako ľahko je možné danú metódu obísť sa dá dočítať v [5].

Grafy volania a grafy riadiaceho toku

Základným konceptom pri využití grafov volania[2] a grafov riadiaceho toku je spustitelný súbor rozbaliť, potom ho rozložiť na postupnosť inštrukcií v assembleri. Následne z tejto postupnosti vytiahnuť grafy toku a reštruktualizovať ich, aby sa, ak je to možné, odstránili nepodmienené skoky. Potom z aplikácie vytiahnuť grafy volania a následne tieto grafy začať porovnávať. Pri porovnávaní sa treba zamerať hlavne na štruktúru programu namiesto jednotlivých inštrukcií [5]. Graf toku programu môže vyzerať napríklad ako je znázornené na obrázku 4.4, o zostavení grafu toku sa možno dočítať v [4].

```
0x1288 add edi, 0x4
0x128b mul dword [0x4056c9]
0x1291 mov [edi], eax
0x1293 inc dword [0x4056c5]
0x1299 cmp dword [0x4056c5], 0x270
0x12a3 jnz 0x1288
0x12a5 pop edi
```



Obrázok. 4.4: Inštrukcie v strojovom kóde a prislúchajúci graf riadiaceho toku. Ins znamená množinu inštrukcií, $j$cc značí podmienený skok.

Výhody využitia grafov volania sú v tom, že berú do úvahy celý program, avšak problémom je potreba mať program kompletne rozložený do inštrukcií strojového kódu a porovnávanie grafov trvá pomerne dlho. Pri porovnávaní berieme do úvahy úzku špecializáciu grafov, ktoré obsahujú veľa informácií navyše. Našou snahou pri porovnaní je vybudovať medzi dvomi porovnávanými grafmi mapovanie, ktoré rešpektuje ich štruktúru. To znamená, že susedia uzlu K z grafu 1, by mali byť susedmi uzlu K' z grafu 2. Presný popis algoritmu porovnania grafov možno nájsť v [5], je to pomerne komplikovaná metóda a preto v tejto práci nebude rozoberaná.

Levenshteinova vzdialenosť

Pri behaviorálnej klasifikácii reprezentovanej ako postupnosť udalostí, činností vykonávaných pozorovaným programom a následne zaznamenaných do textovej reprezentácie je možné využiť algoritmy, ktoré boli určené na prácu s reťazcami. Jedným z takých algoritmov je Levenshteinova vzdialenosť. Levenshteinova vzdialenosť[3] sa používa ako metrika k meraniu množstva rozdielov medzi dvomi sekvenciami. Táto metrika udáva minimálne množstvo operácií, ktoré treba vykonať, aby sme previedli jeden z porovnávaných reťaz-

[2]angl. Callgraph – reprezentácia funkcií a procedúr, ktoré sú pospájané hranami na základe toho, ako sa navzájom volajú pri behu programu.

[3]Pomenovaná po Vladimirovi Leveshteinovi, ktorý ju objavil v roku 1965. Nazývaná aj editačná vzdialenosť[8].

cov na druhý. Operácie môžu byť vloženie, zmazanie, alebo nahradenie jediného znaku. Algoritmus využíva maticu o rozmeroch (n+1, m+1), kde m a n sú dĺžky porovnávaných reťazcov a cena akejkoľvek operácie je 1. Klasický algoritmus Levenshteinovej vzdialenosti je reprezentovaný naseledujúcim pseudokódom:

```
int LevenshteinDistance(char s[1..m], char t[1..n])

  // d je tabulka s m+1 riadkami a n+1 stlpcami
  declare int d[0..m, 0..n]

  for i from 0 to m
      d[i, 0] := i
  for j from 0 to n
      d[0, j] := j
  for i from 1 to m
      for j from 1 to n
      {
          if s[i] = t[j] then cost := 0
                  else cost := 1

          d[i, j] := minimum(d[i-1, j] + 1,
                             d[i, j-1] + 1,
                             d[i-1, j-1] + cost)
      }
return d[m, n]
```

Ako je vidno jedná sa o maticový algoritmus, ktorý je jednoduchý na implementáciu. Nevýhodou tohoto algoritmu je, že na získanie popisu musíme samozrejme škodlivý kód nechať bežať a zaznamenávať udalosti. Algoritmus sa v rôznych formách využíva pri analýze DNA, rozpoznávaní reči, či odhaľovaní plagiátorstva. Viac je možné sa dočítať v [8]. Klasifikácia a porovnanie vzoriek je potom založené na cene transformácie jedného popisu na druhý.

Hodnotenie časovej a priestorovej náročnosti algoritmu

Časová zložitosť algoritmu je *O(m*n)*, pričom m a n sú dĺžky porovnávaných reťazcov, ak je dĺžka oboch reťazcov n potom časová zložitosť bude $O(n^2)$. Priestorová zložitosť algoritmu je taktiež *O(m*n)*, keďže pre jeho realizáciu musíme vytvoriť maticu o rozmeroch *(m, n)*, resp. *(m+1, n+1)*, matice možno vidieť v [8]. Priestorovú zložitosť sme však schopní znížiť na *O(m)*, keďže potrebujeme ukladať len predchádzajúci a súčasný riadok v každom časovom okamihu. Nevýhodou algoritmu je, že sa ťažko paralelizuje z pohľadu porovnania operácií, keďže je v ňom veľa dátových závislostí.

NCD

Ďalším možným algoritmom využitelným na textovú reprezentáciu správania sa súborov je NCD[4]. NCD sa dá použiť aj priamo na spustitelné súbory, čo však nie je tento prípad. Táto metóda bola úspešne použitá aj v iných oblastiach [20]. Výhodou tejto metódy je, že na rozdiel od Levenshteinovej vzdialenosti je odolnejšia voči polymorfizmu(napr. náhodné mená súborov) a početnosti záznamov. NCD je definovaná vzťahom:

$$NCD(x,y) = \frac{C(x+y) - min(C(x), C(y))}{max(C(x), C(y))} \tag{4.2}$$

$x + y$ je konkatenáciou x a y a $C(x)$ je zkomprimovaná dĺžka x. Kompresia môže byť napríklad pomocou knižnice Zlib. NCD vlastne reprezentuje, aká časť informácie sa medzi dvomi porovnávanými vzorkami prekrýva. Výsledkom je, že správanie, ktoré je podobné, nie však identické je posudzované ako navzájom blízke. Viac o použití NCD sa možno dozvedieť v [3].

[4]angl. Normalized Compression Distance – normalizovaná kompresná vzdialenosť.

Kapitola 5

Návrh algoritmu pre porovnanie behaviorálnych reprezentácií

V nasledujúcej kapitole popíšem mnou navrhnutý algoritmus na porovnávanie špecifických textových reprezentácií správania sa programov a jeho implementáciu. Poukážem na ktoré veci som sa zameral a v čom sú výhody, nevýhody algoritmu. Predstavím štruktúru programu a to, aké prvky boli použité pri jeho stavbe a prečo. Zároveň musím uviesť, že návrh môjho algoritmu, ako aj ďalších vecí spojených pri porovnávaní vzorkov je úzko spojený s dátami a metódou výskumu malware v konrétnom prípade. Je možné algoritmus použiť aj v iných oblastiach, či na iný štýl behaviorálneho popisu, avšak potom je nutné za týmto účelom popisy, alebo algoritmus a jeho pravidlá upraviť.

5.1 Upravený algoritmus Levenshteinovej vzdialenosti

O využití Levenshteinovej vzdialenosti ako miery podobnosti dvoch textových popisov súborov bolo čo to už napísané v kapitole 4. V tejto podkapitole rozoberiem mnou navrhnutú úpravu algoritmu tak, aby sa dali popisky súborov jednoduchšie porovnávať, ale zároveň, ak ich niekto bude čítať mali aj nejakú výpovednú hodnotu. Nebolo by vhodné, aby popis predstavovali len nejaké čísla, či špeciálne znaky. Inšpiráciu pre úpravu algoritmu som hľadal v [14].

Spracovanie textového popisu

Na realizáciu algoritmu, ktorý bude vykonávať porovnanie dvoch popisov programov, je potrebné spracovať, respektíve znormalizovať textovú reprezentáciu, ktorej príklad je uvedený na obrázku 4.3. Často sa stáva, že pri vytváraní súborov malware využíva pseudonáhodné generátory a tým informácia o tom, aký presne súbor bol vytvorený len zbytočne mätie porovnávacie algoritmy a najmä Levenshteinova metóda na to dopláca. Dôležité však je, aká je cesta k vytvorenému, skopírovanému, či mazanému súboru, pretože niektoré zložky na disku sú dôležitejšie z pohľadu analýzy malware napr. *Windows\System32*. To isté platí aj pre registre, IP adresy, čísla portu a posielaných paketov. Parsovanie textových popisov

Vstup	**Výstup**
Run file [path]	Run file {Special}
Copy file [path] to [path]	Copy file {Special}
Move file [path] to [path]	Move file {Special}
Write to file [path]	Write file {Special}
Delete file [path]	Delete file {Special}
Download file [address]	Download file
Write [path] to section [section name] in file [path]	Write to section file {Special}
Load kernel service driver [register]	Load KSD {Special}
Unload kernel service driver [register]	Unload KSD {Special}
Set registry key [register] [path]	Set registry key {Special}
Try connect to network	Try connect to network
Hide process	Hide process {Special}
Open another process for writing	Open for write process {Special}
Undocumented suspicious image load	Image load {Special}
Create service [name] with file name [path]	Create service {Special}
Delete service	Delete service
Connect to [IP address]	Connect to
Listen on port [port number]	Listen on port
Send e-mail via smtp server [IP address]	Email send
Hook API function	Hook API function
DNS query [IP address]	DNS query
Sended packets [packets]	Packets send
Other changes in registry [registers]	Registry other change

Tabulka 5.1: Pravidlá pre reťazce v spracovávanom textovom popise.

ešte podrobnejšie rozoberiem pri popise implementácie parsera, keďže môj algoritmus, rovnako ako parser a celý program je úzko nadviazaný na vzorky, ktoré mi boli poskytnuté z oblasti výskumu malware.

Textová reprezentácia sa spracuje pomocou pravidiel uvedených v tabulke 5.1. Pre vytvorenie potrebných pravidiel mi bol poskytnutý zdrojový kód analyzátora, ktorý vzorky analyzuje na výskyt vírusu a následne upravuje vzorky do podoby, v akej sú v databáze. Na základe obsahu zdrojového kódu som stanovil pravidlá, pre spracovanie a určil prípady, ktoré sa môžu vyskytnúť.

Cieľom je orezať výstup len na základné operácie, ktoré malware robí, aby sa potom popis ľahšie porovnával. Základné operácie a ich ceny sa zjednotia do tabulky, ktorú bude využívať porovnávací algoritmus a bude uložená v externom súbore slúžiacom ako konfigurácia. Viac o tabulke bude v podkapitole 5.1. Význam výrazov v hranatých zátvorkách je viac menej jasný, vysvetlím však význam parametra *Special* v spracovanom texte. Cieľom pridania parametra je väčšia diverzifikácia spracovaných popisov a zvýraznenie operácií, ktoré manipulujú z pohľadu skúmania malware s dôležitejšími adresármi, alebo registrami.

Pretože je rozdiel, ak nám súbor zapisuje, alebo spúšťa niečo v adresári *My Documents*, alebo *Windows\System32*. Za parametrom *Special* môže byť schovaných viacero reťazcov, ktoré môžu byť napísané za sebou, obvykle oddelené od seba a základu pomlčkou a to síce:

- Root - koreňový adresár(napr. C:\)
- System32 - adresár Windows\System32
- Service - ide o jeden z dôležitejších registrov
- Run - ide o jeden z dôležitejších registrov
- Self - súbor pracuje sám so sebou, nastáva hlavne pri kopírovaní, či spúšťaní

Po použití pravidiel a znormalizovaní reprezentácie z obrázku 4.3 pomocou našej tabulky dostaneme nasledovný popis:

```
Program try connect to network
File Copy-Self
File Write-Root
File Copy-Self-Root
Service Create
File Run
```

Otázne je, ako vyriešiť početnosť výskytu niektorých činností. Veľmi často tvorcovia malware do svojho kódu umiestňujú príkazy, ktorými sa snažia algoritmy podobnosti rozhodiť. Napríklad tým, že sa do cyklu dá otvárať ten istý súbor, alebo náhodne generovaný súbor. Je príliš problematické v týchto prípadoch vedieť, či to je úmysel, alebo to má z pohľadu porovnania správania nejaký význam. Dokonca by som povedal, že je to nemožné. Preto som sa po konzultácii rozhodol brať všetky riadky obsahujúce tú istú informáciu ako jeden a po spracovaní ich aj ako jeden riadok reprezentovať. To znamená, že postupnosť akcií:

```
Run file C:\program files\internet explorer\IEXPLORE.EXE
Run file C:\WINDOWS\System32\cmd.exe
Connect to 10.2.1.100:80
Download file http://10.2.1.100/wpad.dat
DNS query zyp110.vicp.net
Run file C:\program files\internet explorer\IEXPLORE.EXE
Connect to 10.2.1.100:80
```

bude spracovaná na postupnosť:

```
File Run
File Run-System32
Connect to
Download file
DNS query
```

Takto sa budem snažiť aspoň obmedziť možné techniky pre „zmätenie“ Levenshteinovej metódy, avšak samozrejme ak budú mená súborov, či IP adresy, čísla portov generované náhodne, tak sa s tým bohužial nedá nič robiť a budem to brať ako viacero vykonávaných akcií.

Ďalšou vecou, ktorá stojí za zmienku je, že skúmané vzorky často spúšťajú iné procesy. Z hľadiska analýzy sa ukázalo byť výhodnejšie, aby sa akcie, ktoré vykonávajú procesy spustené sledovaným vzorkom, zaznamenávali v rámci skúmaného vzorku. Preto aj pri spracovaní textového popisu bude táto vlastnosť zachovaná a akcie procesov spustených v rámci testovania vzorky budú ponechané v popise vzorky. V priebehu testovania sa zistí, aké má poradie krokov význam na mieru podoby daného malware. Pri zachovaní poradia krokov, tj. bez ich preusporiadania napr. podľa abecedy očakávam väčšie rozdiely medzi jednotlivými vzorkami, kedže Levenshteinova metóda je na poradie citlivá.

Meranie podobnosti textových popisov malware

Klasický Levenshteinov algoritmus pracuje s jednotkovými cenami operácií vkladania, mazania, alebo nahradzovania. Pre potreby porovnania textovej reprezentácie behaviorálneho popisu programov sa musí algoritmus upraviť a to nasledovne. Definujeme špecifické operácie vloženia, mazania a nahradenia. Cena prevodu jedného behaviorálneho popisu na druhý bude daná sumou operácií, ktoré treba vykonať na dokončenie prevodu [14].

$$Cena(Prevod) = \sum_{i} Cena(Opercia_i) \tag{5.1}$$

Metrikou rozdielu dvoch popisov potom bude práve cena prevodu jedného vzorku na druhý. Aby sa meranie podobnosti ešte viac „zjemnilo“, stanovíme cenu za jednotlivé operácie, ktoré môžu byť použité pri prevode jedného popisu na druhý. Algoritmus sa pritom vždy bude snažiť nájsť prevod s čo najmenšou cenou. Stanovením ceny za jednotlivé operácie docielime toho, aby sme vyberali čo najpodobnejšie vzorky, keďže je rozdiel, ak sledovaný program zapisuje do registrov, alebo niekam inam. Vytvorená tabulka môže byť predmetom ďalšej diskusie v závislosti na presnosti porovnávania vzoriek. Bude vychádzať z tabulky vytvorenej pri normalizácii textového výstupu 5.1, kde budú uložené všetky operácie, ktoré by sa mohli pri analýze vyskytnúť a boli zachytené. Základnú podobu tabulky operácií s cenami je možné vidieť v tabulke 5.2 a tabulke cien pre dôležitejšie adresáre a registre 5.3.

Ceny jednotlivých operácií sa môžu po implementácii algoritmu meniť v závislosti na požadovanej presnosti, či dôležitosti jednotlivých úkonov, ktoré malware vykonáva. Pôvodná veľkosť matice bude odpovedať nie dĺžkam jednotlivých reťazcov, ale počtu operácií, ktoré obsahujú jednotlivé popisy. Porovnávať sa nebudú jednotlivé znaky reťazcov, ale celé reťazce ako je uvedené v 5.2. Podľa celkovej ceny sa potom rozhodne, ako veľmi sú si vzorky podobné. Tabulky v plnom rozsahu som uviedol aj preto, aby bolo na prvý pohľad vidieť, ktoré skupiny operácií medzi sebou súvisia a aká je ich dôležitosť. V návrhu počítam s tým, že čím je daná operácia z pohľadu heuristickej analýzy dôležitejšia, tým väčšiu cenu má aj jej prípadné zmazanie, alebo pridanie. Pravidlá pre nahradzovanie sú potom tvorené len

Operácia	**Objekt 1**	**Objekt 2**	**Cena**
Vloženie—Mazanie	Service Create		7
	Service Delete		7
	KSD Load		7
	KSD Unload		7
	Process Hide		9
	Process Open for write		9
	Hook API function		9
	File Copy		5
	File Move		5
	File Delete		5
	File Write		10
	File Download		10
	File Write to section		10
	Registry Set key		10
	Listen on port		8
	Email Send		8
	File Run		6
	Connect to		4
	Program try connect to network		1
	Registry Other change		2
	Packets Send		2
Nahradenie	Service Create	Service Delete	3
	Service Create	KSD Load	3
	Service Create	KSD Unload	3
	Service Create	Image Load	3
	Service Delete	KSD Load	3
	Service Delete	KSD Unload	3
	Service Delete	Image Load	3
	KSD Load	KSD Unload	3
	KSD Load	Image Load	3
	KSD Unload	Image Load	3
	Process Hide	Process Open for write	4
	Process Hide	Hook API function	4
	Process Open for write	Hook API function	4
	File Copy	File Delete	2
	File Copy	File Move	2
	File Delete	File Move	2
	File Write	File Download	3
	File Write to section	Registry Set key	3
	System32	Root	2
	Services	Run	1

Tabulka 5.2: Možné ceny operácií pri meraní podobnosti.

Špeciálny adresár	**Cena**
System32	4
Root	4
Services	3
Run	3
Self	1

Tabulka 5.3: Možné ceny pri práci s dôležitejšími adresármi.

navzájom súvisiacimi operáciami. Nemožno napríklad nahradiť mazanie súboru so zápisom do registrov. Napríklad operácie *Listen on port* a *Email Send* súvisia spolu tak, že obe sa dajú považovať za taktiku trójskeho koňa, kedy ide hlavne o získanie informácií od užívateľa. Spolu spojené sú aj operácie pracujúce s dôležitejšími adresármi, alebo registrami. Preto je možné medzi sebou nahradiť len *System32* a *Root*, keďže tieto dva súvisia skôr s adresárovou štruktúrou a *Services* a *Run* súvisia skôr s registrami.

Pôvodný zámer bol mať rôzne operácie pre vkladanie a mazanie tej istej operácie, avšak od toho som upustil, keďže potom by nebola splnená základná podmienka porovnávania a to síce, že pri porovnaní x a y, kde každé písmeno predstavuje inú vzorku musí platiť:

$$Rozdiel(x, y) = Rozdiel(y, x) \tag{5.2}$$

Samozrejme čím vyššie číslo nám z porovnania vzoriek vzíde, tým väčší rozdiel je medzi dvomi skúmanými vzorkami. Stále je však nevýhodou algoritmu, že výsledná rozdielnosť je len jednorozmerná hodnota, čo nemá príliš velkú výpovednú hodnotu. Ďalšou nevýhodou je kladenie vyššieho dôrazu na rozdielnosť, ako na podobnosť pri porovnávaní vzoriek s menším počtom operácií so vzorkami s väčším počtom operácií.

Porovnávanie, či vyhladávanie reťazca v rámci iného reťazca rýchlosti nepridáva, ale vzorky sú aj po spracovaní čitatelné pre človeka a to bolo tiež jedným z cieľov. Preto som sa snažil spracované informácie čo najviac skrátiť, pretože čím dlhšie reťazce, tým potenciálne väčšia časová zložitosť. Otvorená je aj otázka kódovania spracovaných popisov na menšie, je ale otázne či to bude mať požadovaný efekt na rýchlosť porovnávania.

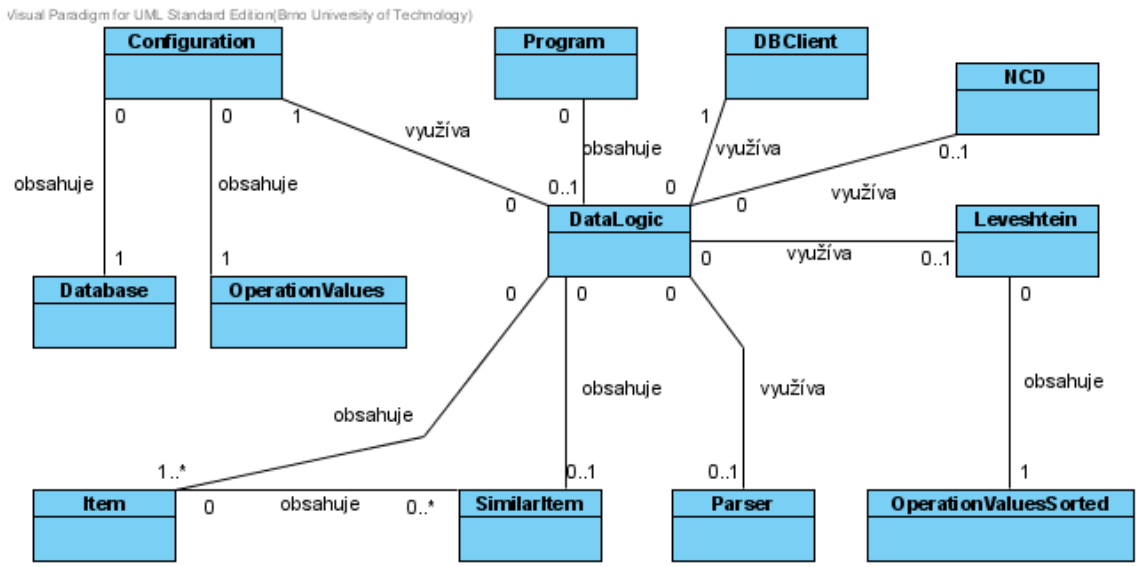
5.2 Algoritmus NCD

Pri algoritme NCD využijem vzťah z podkapitoly 4.2. Výhodou použitia NCD je, že popis vzorkov nebude potrebné zvlášť upravovať. Textový popis v tomto prípade zmodifikujem minimálne a to odstránením opakujúcich sa reťazcov v nespracovanom popise. Pri porovnávaní nesracované popisy spojím do jedného reťazca, keďže vzťah pre NCD pracuje s popisom vzorku ako celkom a nie s jednotlivými časťami. Výsledok vzťahu bude ležať v intervale $< 0, 1 >$ pričom čím bude výsledné číslo bližšie k nule, tým budú vzorky podobnejšie. Uvažovať je možné ešte o použití NCD na popisy spracované pre metódu Leveshtein a zistiť vplyv na výsledné hodnoty.

Kapitola 6

Implementácia navrhnutého algoritmu

Program je písaný ako klasická konzolová aplikácia v jazyku C# s využitím .NET Framework 2.0. Zvolil som starší framework kôli možnému použitiu na počítači so starším frameworkom, aj keď aktualizácia by nemala byť problém. Zároveň program nepoužíva žiadne funkcie, na ktoré by vyšší framework potreboval. Implementácia prebiehala v prostredí Visual Studio 2008 a bola pri nej využitá jedna externá knižnica a to síce knižnica pre komunikáciu s databázou MySQL, v ktorej sú uložené behaviorálne popisy vzorkov. Verzia databáze MySQL, v ktorej boli pri testovaní uložené popisy bola 5.1.3, ale nemal by byť problém pracovať aj so staršou verziu databáze.



Obrázok. 6.1: Diagram tried obsiahnutých v programe, ich vzájomná náväznosť.

Celkovo sa naprogramovaná aplikácia skladá z 12–tich tried, ktoré možno vidieť v diagrame tried znázornenom na obrázku 6.1. Kompletný diagram tried aj s metódami v jed-

notlivých triedach je možné vidieť v prílohe na CD. Základná trieda programu je *P*rogram, spracováva príkaz zadaný do príkazového riadku a na základe parametrov nastavuje riadiace premenné a vytvára inštanciu triedy obsahujúcej logiku, prípadne vypisuje nápovedu, alebo upozorňuje na chybné parametre.

6.1 Konfigurácia databáze obsahujúcej textové popisy a pravidiel pre porovnávanie textových popisov

Konfiguráciu spravuje trieda *C*onfiguration, ktorá v sebe obsahuje ďalšie dve triedy pre uchovanie informácií potrebných k behu a funkčnosti programu. Pri konfigurácii som využil možnosť XML serializácie v .NET, preto je trieda *C*onfiguration implementovaná ako serializovatelná. Informácie sa do konfiguračnej triedy dostávajú po spustení programu so správnymi parametrami zo súboru *c*onfig.xml, ktorý sa musí nachádzať v rovnakom adresári ako spustitelný súbor. XML formát som použil pre jeho ľahkú čitatelnosť a tým pádom aj modifikovateľnosť, ako aj pre dobrú podporu práce s týmto formátom v .NET. Konfiguračný súbor predstavuje zoznam pravidiel, ich cien a informácie pre program, aby sa mohol pripojiť do databáze.

Súčasťou súboru, v ktorom sa nachádza trieda *C*onfiguration sú aj spomenuté dve vedľajšie triedy a to *D*atabase a *O*perationValues. Dôvodom pre vytvorenie týchto tried bola hlavne prehľadnosť konfiguračného súboru a umožnenie jeho ľahšej editácie. Trieda *D*atabase obsahuje premenné uchovávajúce informácie o databáze ako meno servera, meno databáze, login, heslo a názov tabulky v ktorej sú vzorky v databázi uložené. Všetky tieto premenné sú v XML súbore reprezentované ako atribúty elementu *D*atabase. V triede *O*perationValues sú uložené v zoznamoch názvy operácií a ich ceny tak, ako je uvedené v tabulke 5.2. Ďalšou významnou informáciou obsiahnutou v konfigurácii je počet najpodobnejších položiek, ktoré sa majú vypísať ako výstup programu. Pre ilustráciu, ako konfiguračný súbor môže vyzerať slúži obrázok 6.2. Vysvetlenie elementov použitých v konfiguračnom súbore:

- <Configuration> – základný element predstavujúci triedu *C*onfiguration, v ktorom musí byť uzavreté všetko ostatné.
- <SimilarCountNumber> – číslo uzavreté v tomto elemente predstavuje počet najpodobnejších vzoriek, ktoré program vypíše.
- <Database> – element týkajúci sa databázi, význam základných atribútov je jasný. Atribút *T*ableNameItems obsahuje názov tabulky, kde sa očakávajú nespracované popisy vzoriek. Atribúty *T*ableNameLevenstein a *T*ableNameNCD obsahujú názvy tabuliek v databázi, v ktorých sa očakávajú vzorky s popismi pre porovnanie algoritmom Levenshtein, respektíve NCD. Atribút *C*ount znamená počet položiek vytiahnutých z tabulky v databázi na porovnanie, ak je 0 bude sa operovať so všetkými položkami. Atribút *S*tartPosition vyjadruje štartovaciu pozíciu v rámci tabulky v databázi, od ktorej sa prvky začnú vyberať.

- <OperationCost> – element predstavujúci triedu *O*perationValues. Elementy <InsertOrDelete>, <Replace> a <Special> sa musia nachádzať v rámci počiatočného a koncového elementu <OperationCost>.
- <InsertOrDelete> – element predstavujúci cenu za vymazanie, alebo pridanie vybranej operácie z textového popisu vzorky. Atribút *N*azov udáva názov operácie, ktorej sa mazanie, pridanie týka. Atribút *H*odnota udáva cenu operácie.
- <Replace> – podobný význam ako element <InsertOrDelete>, predstavuje cenu za nahradenie jednej operácie druhou.
- <Special> – element udávajúci cenu dôležitejších informácií

```
<?xml version="1.0"?>
<Configuration>
  <SimilarCountNumber>50</SimilarCountNumber>
  <Database Server="localhost" DBName="DIP" DBLogin="dip"
            DBPass="dip" Count="100" StartPosition="0" TableNameItems="dip_table"/>
  <OperationCost>
    <!--Insert-->
    <InsertOrDelete Nazov="Service Create" Hodnota="7" />
    <InsertOrDelete Nazov="Service Delete" Hodnota="7" />
    <InsertOrDelete Nazov="Process Hide" Hodnota="9" />
    <InsertOrDelete Nazov="Process Open for write" Hodnota="9" />

    <!--Replace-->
    <Replace NazovPrvy="Service Create" NazovDruhy="Service Delete" Hodnota="3" />
    <Replace NazovPrvy="Process Hide" NazovDruhy="Process Open for write" Hodnota="4" />

    <!--Special-->
    <Special Nazov="System32" Hodnota="4" />
    <Special Nazov="Root" Hodnota="4" />
    <Special Nazov="Services" Hodnota="3" />
    <Special Nazov="Run" Hodnota="3" />
    <Special Nazov="Self" Hodnota="1"/>
  </OperationCost>
</Configuration>
```

Obrázok. 6.2: Príklad súboru config.xml.

6.2 Štruktúra databáze obsahujúcej textové popisy

Ako bolo už spomenuté vyššie, pre uloženie behaviorálneho popisu vzoriek po výstupe z analyzátora sa využíva databáza MySQL[1]. Preto som musel aj svoj program prispôsobiť tomuto faktu a pre prístup a prácu s databázou využívam externú knižnicu dostupnú na stránkach MySQL. Na pripojenie a prácu s databázou slúži trieda *D*BClient. Trieda obsahuje funkciu pre pripojenie sa k databázi, funkciu pre vyčítanie jednej konkrétnej položky, funkciu pre vloženie jednej konkrétnej položky a funkciu pre vyčítanie zvoleného počtu položiek od konkrétnej pozície v tabulke. Ak počet ani pozícia nie sú definované, vyčítava

[1]Databáza nebola navrhovaná, pracuje sa s už existujúcou databázou.

sa celý obsah tabulky od začiatocnej pozície. Operácia SELECT je pomerne drahá a preto pri požiadavke na vyčítanie všetkých hodnôt v tabulke zavolám SELECT na všetky položky, neukladám ich však všetky do pamäte, ale postupne vyčítavam výsledok výrazu SELECT po riadku z databázového serveru. Týmto spôsobom nevolám zbytočne SELECT na každú položku, ale ani nemusím ukladať celú databázu do operačnej pamäte. Tabulka v databázi obsahujúca vzorky je tvorená štyrmi stĺpcami.

- MD5 – Typ *v*archar(32). Unikátne ID vzorky, pomocou ktorého sa dá dohladať aj v ostatných tabulkách.
- SANDBOX – Typ *m*ediumtext. Reťazec obsahujúci behaviorálny popis vzorky, ktorý je potreba spracovať.
- RESULT – TyP *t*inyint. Udáva, či bola vzorka hodnotená ako možný vírus pozitívne, alebo negatívne.
- DATE_UPD – Typ *t*imestamp. Dátum a čas poslednej aktualizácie vzorky

Aby správne fungovalo spracovanie popisu vzoriek musí byť v databázi vytvorená tabulka pre ich uloženie a to so stĺpcami MD5, SANDBOX a RESULT. Ak táto tabulka nebude v databázi počas spracovania vytvorená, vkladanie spracovaných vzoriek neprebehne úspešne. Pri vkladaní položky, ktorá už v databázi existuje sa aktualizuje informácia v stĺpci SANDBOX u položky v databázi, podmienkou je mať nastavený stĺpec MD5 ako primárny kľúč tabulky so vzorkami.

6.3 Spracovanie dát, logika a porovnanie

Trieda *D*ataLogic sa stará o logiku celého programu. Pri jej vytvorení sa načíta konfigurácia pre databázu a zoznamy pravidiel, pomocou ktorých sa operácie porovnávajú. Z dôvodu serializácie sú hodnoty v konfigurácii uložené v klasických zoznamoch, avšak po načítaní dát zo súboru do triedy *C*onfiguration sa obsah zoznamov s menami a cenami operácií presunie do slovníkov(typ Dictionary) v štruktúre *O*perationValuesSorted. Výhoda týchto slovníkov oproti klasickému zoznamu je, že slovník je pri vyhľadávaní založený na hashovacej tabulke a je pri hľadaní a prístupe ku konkrétnemu prvku na základe mena stĺpca rýchlejší. Klúčom pre vyhľadanie v slovníku je meno operácie, s ktorou sa pracuje a vrátenou hodnotou je jej cena. Pôvodný zámer bol použiť klasický usporiadaný zoznam (operácia vkladania do neho je kratšia), ale vkladať sa do slovníku bude len pri štarte aplikácie a počas celého behu sa bude hlavne vyhľadávať, preto som použil v tomto o niečo rýchlejší slovník. Po spustení funkcie *S*tart() sa ako prvé vytvorí spojenie s databázou, potom sa na základe nastavených riadiacich premenných zavolá funkcia, ktorá využíva inštancie vytvorených tried pri spracovaní a porovnaní behaviorálnych popisov.

Reprezentácia vzorkov v rámci programu

Trieda *I*tem predstavuje vzorku načítanú z databáze. Obsahuje tri základné zoznamy a to zoznam nespracovaných popisov činností, zoznam spracovaných popisov činností a zoznam

najpodobnejších vzoriek k danej vzorke. Zoznam nespracovaných popisov je vytvorený po načítaní zvorky s databáze, kedy sa reťazec zo stĺpca SANDBOX rozdelí na menšie reťazce, reprezentujúce jednotlivé činnosti. Zoznam najpodonejších vzoriek k danej je tvorený objektami triedy *S*imilarItem, ktorá uchováva informáciu o názve vzorky a vzájomnej vzdialenosti vypočítanej metódou Leveshtein, alebo NCD.

Spracovanie textového popisu vzorku

O spracovanie reťazca s popisom činností vzorku načítaného z databáze sa stará trieda *P*arser a v nej funkcia *P*arse a *P*arseNCD. Funkcia *P*arse na základe pevne daných pravdiel uvedených v tabulke 5.2, spracuje rozdelené popisy zo zoznamu nespracovaných popisov do zoznamu spracovaných popisov. Pri aplikácii pravidiel, prechádza vstupné pravidlá a zisťuje, či sa zvolené pravidlo nenachádza v rámci reťazca zo zoznamu nespracovaných popisov. Pravidlá sú pritom tvorené tak, aby sa na každý reťazec v rámci zoznamu činností dalo aplikovať maximálne jedno pravidlo. Spracovaný zoznam popisov obsahuje operácie uvedené v tabulke 5.1, pričom operácia je tvorená základom a prípadne menom dôležitého adresára, registru oddeleného od základu pomlčkou. Dôležitejších adresárov, registrov môže byť aj viacero a sú od seba taktiež oddelené pomlčkami. Príklad možno vidieť v podkapitole 5.1. Funckia *P*arseNCD slúži na prípravu popisu pre metódu NCD. Pri spracovaní pre metódu NCD nie sú využívané žiadne špeciálne pravidlá, len sa znormalizuje popis vzorky tak, aby sa v ňom nevyskytovali rovnaké operácie a odstránia sa informácie s najnižšou výpovednou hodnotou. Za informácie s najnižšou výpovednou hodnotou sú považované zmeny v registroch nachádzajúce sa za reťazcom *O*ther registry changes vo výstupe z analyzátora a informácie o odoslaných paketoch.

Levenshteinovo porovnanie

Na porovnanie spracovaných vzoriek Leveshteinovým algoritmom slúži trieda *L*evenshtein obsahujúca funkciu na výpočet rozdielu medzi dvomi vzorkami. Vstupom do funckie sú dva zoznamy, jeden obsahujúci spracované popisy vzorky,ktorú porovnávam, druhý obsahujúci spracované popisy vzorky, sk torou porovnávam. Funkcia v sebe implementuje navrhnutý upravený Levenshteinov algoritmus, kde cenu vloženia, alebo zmazania operácie určuje z už spomenutého slovníku operácií. Preto je dôležité, aby sa v konfiguračnom súbore nezabudlo na žiadne pravidlo, ktoré môže vzniknúť spracovaním behaviorálneho popisu, pretože potom sa dané pravidlo v slovníku nenájde a program vypíše správu o chybe a upozorní na fakt, že dané pravidlo v konfiguračnom súbore nie je. Ak operácia obsahuje okrem základu aj dodatočné údaje, ich hondota sa vyčíta podobne, ale zo zoznamu dôležitých informácií a pričíta k cene za pridanie, zmazanie základu. Pri určovaní ceny za nahradenie jednej operácie za druhú je výpočet ceny o niečo zložitejší. V prvom rade sa vezmú základy a algoritmus sa pozrie, či je vôbec možné základy medzi sebou nahradiť. Ak nie, tak sa cena nahradenia nastaví na veľmi vysokú, aby si ju Levenshteinov algoritmus určite nezvolil. Ak sa základy dajú nahradiť, potom dochádza k porovnaniu dodatočných dôležitých adresárov, registrov, ak ich operácia obsahuje. Ak sa dôležité registre nedajú nahradiť ich ceny sa pričítajú k cene za nahradenie základu, ak sa nahradiť dajú, k cene základu sa pričíta cena za ich

nahradenie.

NCD

Pri porovnávaní metódou NCD je volaná funkcia z triedy *N*CD, ktorá implementuje algoritmus NCD z podkapitoly 4.2. Návratová hodnota je na rozdiel od Leveshteinovej vdialenosti typu *f*loat. Vstupné parametre funkcie tvoria popis vzorky, ktorú porovnávam a popis vzorky, s ktorou porovnávam, ale na rozdiel od funkcie v triede *L*eveshtein je tu popis vzorky reprezentovaný ako jeden súvislý reťazec a nie ako zoznam reťazcov. Je to z toho dôvodu, že v algoritme NCD sa nepracuje s jednotlivými operáciami, ktoré reprezentujú činnosti vzorky v sledovanom systéme, ale so všetkými operáciami ako celkom, súvislým reťazcom, rovnako ako je to uložené v databázi. Pri komprimovaní textového popisu je použitá metóda GZip, ktorá je podporovaná vo frameworku .NET.

6.4 Ovládanie programu

Keďže sa jedná o konzolovú aplikáciu, ovládanie programu je reprezentované parametrami príkazovej riadky, s ktorými je možné program úspešne spustiť. Parametre a ich význam je možné vidieť v tabulke 6.1. Príklady spustenia programu a ich význam možno vidieť v tabulke 6.2.

Parameter	**Význam parametra**
[MD5]	ID porovnávanej/spracovávanej vzorky.
–h	Vypíše nápovedu k programu.
–p	Spracuje všetky vzorky a uloží ich do tabulky zadanej v konfiguračnom súbore.
–c	Znamená porovnávanie vzoriek.
–ncd	Porovnanie/spracovanie vzoriek metódou NCD.

Tabulka 6.1: Parametre programu a ich význam

Program možno použiť aj na porovnanie nespracovaných popisov, avšak značne sa tým spomalí čakanie na výsledky, keďže program musí v takom prípade pri každom načítaní vzorky z databáze túto vzorku spracovať, preto sa doporučuje radšej pri väčších množstvách dát najskôr vzorky spracovať do samostatnej tabulky a až potom porovnávať. Pri zadaní nesprávneho parametra, ale ak nastane chyba pri behu programu, program vypíše do konzole oznámenie, aká chyba nastala a program sa ukončí. Mená podobných vzoriek a hodnoty podobnosti sú vypisované na štandartný výstup od najmenšieho rozdielu, ich počet závisí na čísle maximálneho počtu vypísaných vzorkov, ktoré sa udáva v konfigurácii.

6.5 Využitie implementovaných algoritmov

Implementované algoritmy nemusia byť využité len na porovnanie vzoriek. Výsledky porovnania sa dajú využiť na vytváranie zhlukov podobných vzoriek, ktoré môžu predstavovať

Obsah príkazového riadku	**Činnosť**
program –h	Zobrazí nápovedu k programu.
program –c [MD5]	Porovnanie vzorky s ID = [MD5] s ostatnými v databázi metódou Leveshtein.
program –p	Spracovanie vzoriek v databázi pre metódu Leveshtein do tabulky zadanej v konfiguračnom súbore.
program –p [MD5]	Spracovanie vzorky s ID = [MD5] pre metódu Leveshtein do tabulky zadanej v konf. súbore.
program –p –c [MD5]	Porovnanie vzorky s ID = [MD5] s ostatnými v databázi metódou Leveshtein, pričom sa vzorky pred porovnaním spracujú.
program –ncd –c [MD5]	Porovnanie vzorky s ID = [MD5] s ostatnými v databázi metódou NCD.
program –ncd –p	Spracovanie vzoriek v databázi pre metódu NCD do tabulky zadanej v konf. súbore.
program –ncd –p [MD5]	Spracovanie vzorky s ID = [MD5] pre metódu NCD do tabulky zadanej v konf. súbore.
program –ncd –p –c [MD5]	Porovnanie vzorky s ID = [MD5] s ostatnými v databázi metódou NCD, pričom sa vzorky pred porovnaním spracujú.

Tabulka 6.2: Príklady spustenia programu s rôznymi parametrami.

istú rodinu škodlivých vzoriek. Nové vzorky by sa potom porovnávali len s predstavitelmi zhlukov a podľa výsledku by boli priradené už k existujúcemu zhluku, alebo by bol vytvorený zhluk nový. V prípade vytvorenia nového zhluku stojí za zváženie opätovné preusporiadanie vzoriek v rámci zhlukov. Proces, ktorý som naznačil v predchádzajúcich riadkoch sa nazýva klasifikácia. V minulosti klasifikáciu vykonávali ľudia, avšak v súčasnosti je vzhľadom na počet vzoriek škodlivého kódu kolujúceho vo virtuálnom svete takéto porovnávanie časovo príliš náročné.

Pri vytváraní zhlukov sa všeobecne využívajú dva spôsoby a to hierarchické zhlukovanie a zhlukovanie pomocou segmentácie. Hierarchické zhlukovanie zoskupuje objekty do hierarchií zhlukov(obyčajne stromových). Výpočetná zložitosť takýchto algoritmov je $O(n^2)$ a ich nevýhodou je horšia prispôsobiteľnosť k počtu porovnávaných objektov. Zhlukovanie pomocou segmentácie rozdeluje objekty do zhlukov na základe premennej k, ktorá udáva počet požadovaných zhlukov, k musí byť menšie ako počet objektov, s ktorými zhlukovanie pracuje. Zhluky sú potom vytvorené podľa kritéria, ktorým je v našom prípade vypočítaná hodnota vzdialenosti medzi dvomi vzorkami. Objekt je potom priradený k najviac sa zhodujúcemu zhluku, porovnávanie so zhlukom môže byť realizované porovnaním s reprezentatívnym vzorkom pre daný zhluk, alebo porovnaním s priemerom vzorkov obsiahnutých v zhluku. Viacej praktických informácií o zhlukovaní možno nájsť v [14], alebo [3].

Kapitola 7

Testy implementovaných algoritmov

V nasledujúcej kapitole otestujem mnou navrhnuté algoritmy na porovnávanie textových popisov súborov. Algoritmy otestujem na množine reálnych, ale aj umelo vytvorených vzoriek. Zameriam sa na ich úspešnosť a rýchlosť a zároveň ich porovnám medzi sebou a vyzdvihnem ich klady a zápory. V databázi, ktorú mám k dispozícii je 191616 vzoriek, z toho 12381 bolo testovaných ako škodlivých. Program bol testovaný na počítači s konfiguráciou: *I*ntel Core2 Duo 2GHz, 2GB RAM, Windows Vista 32-bit. Použitá databáza *M*ySQL 5.1 je súčasťou balíka *E*asyPHP.

7.1 Test spracovania textových popisov vzoriek

V testoch tejto podkapitoly sa zameriam na rýchlosť spracovania textových popisov vzoriek a ich vloženia do databáze. Budem pracovať s náhodne vybranými vzorkami z databáze, ktorá mi bola poskytnutá, ale aj s celým obsahom databáze. Výsledky sú zhrnuté v nasledujúcej tabulke.

Spracovanie pre metódu	**Počet vzoriek**	**Priemerná dĺžka trvania** (s)
Levenshtein	191616	730
Levenshtein(len škodlivé)	12381	172
Leveshtein(náhodne, len škodlivé)	50	5,6
NCD	191616	460
NCD(len škodlivé)	12381	65
NCD(náhodne, len škodlivé)	50	3,7

Tabulka 7.1: Rýchlosť spracovania celej databáze a náhodne vybraných položiek.

Z výsledku testu je zrejmé, že spracovanie textového popisu pre metódu NCD je asi o polovicu rýchlejšie. Je to očakávaný pomer, keďže na rozdiel od spracovania textu pre porovnanie Levenshteinovou metódou sa textové popisy len zbavujú rovnakých riadkov a

málo dôležitých informácií. Pri spracovaní textu pre Leveshteinovu metódu je potrebné časté vyhladávanie podreťazca v reťazci, preto je časová zložitosť podstatne vyššia. Pri spracovaní 50–tich vzoriek náhodným výberom je údaj z časti skreslený. Položky s vyšším indexom v databázi obsahujú v popise viac činností ako položky s nižším indexom a preto ich spracovanie trvá dlhšie. Je teda možné pri spracovaní 50–tich vzoriek dosiahnúť čas 0,5 sekundy, ale aj 9 sekúnd, opäť však platí, že spracovanie pre metódu NCD je rýchlejšie. Testoval som aj paralelné spracovanie v rámci jedného systému, jeden program vzorky spracovával pre metódu NCD a druhý pre metódu Levenshtein. Výsledné časi boli o 10–15% pomalšie, ale záleží aj na tom kde sa server, v ktorom je databáza s tabulkami, nachádza. Správnosť spracovania textového popisu bola overená vybratím náhodných vzoriek z tabulky spracovaných vzoriek a ich porovnaním s prislúchajúcou vzorkou z tabulky nespracovaných vzoriek.

7.2 Test porovnania textových popisov vzoriek

V nasledujúcich testoch otestujem rýchlosť porovnávania pri náhodne vybraných vzorkách a rýchlosť porovnávania, ak vzorku porovnávam s celou databázou. Ďalej sa zamieriam na úspešnosť v určovaní podobnosti vzoriek pri oboch algoritmoch, na náhodne vybranej vzorke z databáze, ako aj na mnou umelo vytvorených vzorkách. Keďže ide o náhodne vyberané vzorky, údaje z testov nemôžu pokrývať všetky možné prípady a to treba pri ich zvažovaní brať do úvahy. Výsledky jednotlivých testov budú uvedené v tabulkách. Význam stĺpcov v tabulkách:

- MD5 – ID vzorky v tabulke nachádzajúcej sa v databázi.
- Vzdialenosť – Hodnota vzdialenosti medzi dvomi vzorkami provnávaných príslušnou metódou, ktorej sa tabuľka týka.
- Vírus – Hodnota udáva, či bola vzorka v databázi vyhodnotená ako škodlivá.
- Úspešnosť – Percentuálne vyjadrenie pomeru vzoriek vyhodnotených ako škodlivých a vzoriek vyhodnotených negatívne zo vzoriek, ktoré algoritmus označil ako podobné k testovanej vzorke.

Testy náhodne vybranej vzorky s celou databázou

V tabulke 7.2 sú výsledky prvého testu, keď som náhodne zvolenú vzorku testoval v rámci celej databáze metódou Levenshtein, v tabulke 7.3 je tá istá vzorka testovaná metódou NCD. Obidve metódy pracovali s normalizovanými popismi vzoriek, to znamená, že boli z textového popisu vzoriek vylúčené opakujúce sa akcie. Čas trvania algoritmu bol v prvom teste 132 sekúnd, pri druhom teste 604 sekúnd. Ak by sme testovali len vzorky, ktoré boli označené ako škodlivé, boli by algoritmy ešte rýchlejšie.

Ako je možné vidieť z oboch tabuliek, ak zoberieme prvých 10 vzoriek z oboch algoritmov, výsledky sú porovnateľné. Levenshteinova metóda určila ako podobné vzorky 5 vzoriek, ktoré neboli vyhodnotené ako škodlivé. Po analýze týchto vzoriek sa ukázalo,

P.č.	MD5	Vzdialenosť	Vírus
1.	ea76509e0f1e83c4f3fa5936cfda83bd	62	Áno
2.	15ef92ee72e719b56e700840f1a91519	93	Áno
3.	e1733085b608245700f232558e85ce84	94	Áno
4.	54efa7dedd9d95fca4722d8264513790	102	Áno
5.	c0486732450e9129544d5065a26c547d	103	Áno
6.	e9716d186ba2997676d20c32b2cf65a8	137	Áno
7.	b4ccfc2a182caefaf9c91a0bc7f96ad0	138	Áno
8.	bcfffad6b3c2a04e690913937af835f0	158	Áno
9.	27ce59dd8942af976a1857194d385954	164	Áno
10.	05ec385ac0132de627230788308126cb	175	Nie
11.	c9c0e69e4e312b86267ef1a549a3a14f	182	Áno
12.	a838b392ac4008fe5ff87cb6b33bdca9	185	Nie
13.	0228e3e9d5f66d7542d3fbb599c44a37	187	Nie
14.	18c9c4d5ff97cc6daebb38f806ea71d5	189	Nie
15.	5b0f462f9ce1536cbb65d130ac0541f5	189	Nie

Tabulka 7.2: 15 najpodobnejších vzoriek so vzorkou 2631d390b6f6b16b3f2a39976cec419e vyhodnotených v rámci celej databáze metódou Levenshtein.

P.č.	MD5	Vzdialenosť	Vírus
1.	c0486732450e9129544d5065a26c547d	0,274642	Áno
2.	e9716d186ba2997676d20c32b2cf65a8	0,2805392	Áno
3.	e1733085b608245700f232558e85ce84	0,2813816	Áno
4.	b4ccfc2a182caefaf9c91a0bc7f96ad0	0,2923336	Áno
5.	15ef92ee72e719b56e700840f1a91519	0,3083403	Áno
6.	ea76509e0f1e83c4f3fa5936cfda83bd	0,3142376	Áno
7.	54efa7dedd9d95fca4722d8264513790	0,3336141	Áno
8.	bcfffad6b3c2a04e690913937af835f0	0,3676901	Áno
9.	c9c0e69e4e312b86267ef1a549a3a14f	0,3909349	Áno
10.	be6f1abbeb09e86e2fc340a5508ab776	0,4303112	Áno
11.	27ce59dd8942af976a1857194d385954	0,5048991	Áno
12.	d71927b4454f5032a1b58076568dbee6	0,5425442	Nie
13.	badcb3c66542f45a22bfe99c8613a749	0,5796125	Nie
14.	ba4190c850e0fddfef6e5212f359fc99	0,6352148	Nie
15.	253b7f0c9853115e46d8fe92829a95d3	0,6537489	Áno

Tabulka 7.3: 15 najpodobnejších vzoriek so vzorkou 2631d390b6f6b16b3f2a39976cec419e vyhodnotených v rámci celej databáze metódou NCD.

že obsahujú síce veľké množstvo operácií zápisu do súboru avšak, len do zložky *T*emp, ktorú využívajú všetky aplikácie na uchovávanie informácií. To, že boli vyhodnotené medzi

15–timi najpodobnejšími môže byť spôsobené normalizáciou výstupu zo sandboxu, teda odstránením opakujúcich sa operácií. Pri metóde NCD boli vyhodnotené ako podobné 3 vzorky, ktoré neboli označené ako škodlivé. Po ich analýze sa ukázalo, že obsahujú porovnatelné operácie s infikovanou vzorkou, avšak všetky ich operácie pracujú s málo významnými adresármi(napr. Dokumenty, Temp). Metóda NCD teda správne vyhodnotila tieto vzorky ako podobné, keďže miera prekrývajúcej informácie je dostatočne vysoká, rozdielne sú len mená adresárov. Zaujímavá informácia je, že ani jedna zo vzoriek, ktoré neboli označené ako škodlivé, vyhodnotená metódou Leveshtein sa nezhoduje so vzorkami, ktoré boli vyhodnotené metódou NCD ako podobné, ale neboli označené ako škodlivé. Je to spôsobené tým, ako tieto dve metódy vzorky navzájom porovnávajú. Pri metóde NCD obsahujú vzorky, ktoré boli vyhodnotené ako podobné, ale neboli označené ako vírus podobnú sekvenciu operácií a približne rovnako dlhý popis ako vzorka, s ktorou boli porovnávané, pracovali iba s menej dôležítými adresárami z pohľadu heuristiky. Pri Levenshteinovej metóde majú vzorky neoznačené za škodlivé spoločné sekvencie operácií, ale kratší popis ako vzorka, s ktorou sú porovnávané. Ak by mali totiž rovnako dlhý popis je veľmi pravdepodobné, že aj analyzátor by ich označil ako škodlivé. Túto rozdielnosť v porovnávaní Levenshteinovou a NCD metódou skúsim dokázať v testoch na umelo vytvorených vzorkách. Test v tabulke 7.4 ukazuje výsledok, pri porovnávaní rovnakej vzorky Leveshteinovou metódou bez normalizácie a v tabulke 7.5 sú znázornené výsledky pri porovnávaní textových popisov, ktoré sú zoradené podľa abecedy. Prvý test trval 157 sekúnd, druhý 114 sekúnd. Ako je vidieť

P.č.	**MD5**	**Vzdialenosť**	**Vírus**
1.	ea76509e0f1e83c4f3fa5936cfda83bd	58	Áno
2.	15ef92ee72e719b56e700840f1a91519	85	Áno
3.	e1733085b608245700f232558e85ce84	90	Áno
4.	c0486732450e9129544d5065a26c547d	95	Áno
5.	54efa7dedd9d95fca4722d8264513790	98	Áno
6.	e9716d186ba2997676d20c32b2cf65a8	129	Áno
7.	b4ccfc2a182caefaf9c91a0bc7f96ad0	134	Áno
8.	bcfffad6b3c2a04e690913937af835f0	153	Áno
9.	27ce59dd8942af976a1857194d385954	156	Áno
10.	c9c0e69e4e312b86267ef1a549a3a14f	177	Áno
11.	a838b392ac4008fe5ff87cb6b33bdca9	195	Nie
12.	0228e3e9d5f66d7542d3fbb599c44a37	201	Nie
13.	18c9c4d5ff97cc6daebb38f806ea71d5	203	Nie
14.	5b0f462f9ce1536cbb65d130ac0541f5	203	Nie
15.	80da7b99e10dbbf81229b9ceebfabc6e	203	Nie

Tabulka 7.4: 15 najpodobnejších vzoriek so vzorkou 2631d390b6f6b16b3f2a39976cec419e vyhodnotených v rámci celej databáze metódou Levenshtein, vzorky v tomto prípade neboli normalizované.

z oboch tabuliek, výsledky podobných vzoriek, ktoré boli označené ako škodlivé sú porovna-

P.č.	MD5	Vzdialenosť	Vírus
1.	e9716d186ba2997676d20c32b2cf65a8	57	Áno
2.	ea76509e0f1e83c4f3fa5936cfda83bd	58	Áno
3.	c0486732450e9129544d5065a26c547d	63	Áno
4.	54efa7dedd9d95fca4722d8264513790	78	Áno
5.	15ef92ee72e719b56e700840f1a91519	89	Áno
6.	bcfffad6b3c2a04e690913937af835f0	103	Áno
7.	e1733085b608245700f232558e85ce84	104	Áno
8.	b4ccfc2a182caefaf9c91a0bc7f96ad0	114	Áno
9.	265f6c4f14609e303eb475c2e0f681c6	127	Nie
10.	dd401e9e9c89463a62d3708cb8286fce	141	Nie
11.	c9c0e69e4e312b86267ef1a549a3a14f	147	Áno
12.	4af4c075bd88b65eb15603f249d5818a	148	Áno
13.	56a0bb80f698f94b882f875f74d3eacc	148	Nie
14.	f01e2fbccf92af023ddac323bd3a94c2	151	Nie
15.	f560ae56d87e6d78328edbfff89caff0	157	Nie

Tabulka 7.5: 15 najpodobnejších vzoriek so vzorkou 2631d390b6f6b16b3f2a39976cec419e vyhodnotených v rámci celej databáze metódou Leveshtein, činnosti u jednotlivých vzoriek boli abecedne zoradené.

teľné, až na mierne odchylky v hodnote vzdialenosti spôsobenou opakujúcimi sa činnosťami pri neznormalizovaných vzorkách a rovnakým poradím pri vzorkách, ktorých činnosti boli zoradené podľa abecedy. Podstatne rozdielny je však výsledok u vzoriek neoznačených ako škodlivé, najmä v tabulke 7.5 kde sa porovnávali vzorky so zoradenými činnosťami podľa abecedy. Vyvsvetlujem to tým, že operácie škodlivých vzorkov sú na rozdiel od operácií normálnych aplikácií viac chaotické, čo sa prejavilo po usporiadaní vzoriek. Vzdialenosti medzi vzorkami sú síce menšie, ale vzorky nepovažované za škodlivé sa nám dostávajú viac do popredia, keďže aj činnosti vykonávané škodlivými vzorkami sú zoradené a znormalizované. Pri teste s neznormalizovanými a zoradenými vzorkami bol výsledok zhodný s tabulkou 7.5 len hodnoty vzdialeností boli pri posledných vzorkách o niečo vyššie. Prekvapením bol výsledok metódy NCD použitej na úplne nespracované vzorky. Výsledky ukázali veľké odchylky v podobnosti nespracovaných vzoriek, dve najpodobnejšie vzorky boli dokonca vzorky, ktoré neboli označené ako škodlivé. Môže to však súvisieť aj s voľbou vzorky, ktorá bola na porovnanie použitá. V tabulkách 7.6 a 7.7 sú výsledky porovnania niektorých vzoriek vyhodnotených ako podobné k vzorke testovanej v tabulke 7.2.

Z výsledkov testov je vidieť najmä pomerne velkú presnosť metódy NCD, pokým Levenshteinova metóda vyhodnotila viac vzoriek zle. Tento trend je spôsobený tým, že príliš veľa informácií sa v rámci spracovania vzorky pre metódu Levenshtein musí vymazať. Pri metóde NCD zostavajú mená súborov, registrov aj IP adries zachované. Levenshteinovu metódu by som preto použil najmä na porovnávanie už vyhodnotených vzoriek, alebo v prípade potreby rýchleho aj keď čiastočne nepresného získania výsledkov, keďže trvanie testu v ta-

P.č.	MD5	Vzdialenosť	Vírus
1.	e9716d186ba2997676d20c32b2cf65a8	34	Áno
2.	b4ccfc2a182caefaf9c91a0bc7f96ad0	75	Áno
3.	bcfffad6b3c2a04e690913937af835f0	84	Áno
4.	2631d390b6f6b16b3f2a39976cec419e	103	Áno
5.	c9c0e69e4e312b86267ef1a549a3a14f	108	Áno
6.	27ce59dd8942af976a1857194d385954	111	Áno
7.	18c9c4d5ff97cc6daebb38f806ea71d5	129	Nie
8.	5b0f462f9ce1536cbb65d130ac0541f5	129	Nie
9.	80da7b99e10dbbf81229b9ceebfabc6e	129	Nie
10.	dad5e87e8e3af9be7c1b71710f2b126e	129	Nie
11.	33498ac8fa0cbee445188644cc1312a5	130	Nie
12.	05ec385ac0132de627230788308126cb	132	Nie
13.	8ceee4da489036d4e4b0fbf1aed15679	135	Nie
14.	e982cc9b8528fccc9ec242f1e1ca827a	136	Nie
15.	6deebec72008cad6eece265db4aa8f1b	139	Nie

Tabulka 7.6: 15 najpodobnejších vzoriek so vzorkou c0486732450e9129544d5065a26c547d vyhodnotených v rámci celej databáze metódou Leveshtein.

P.č.	MD5	Vzdialenosť	Vírus
1.	e9716d186ba2997676d20c32b2cf65a8	0,1709937	Áno
2.	b4ccfc2a182caefaf9c91a0bc7f96ad0	0,2181329	Áno
3.	2631d390b6f6b16b3f2a39976cec419e	0,2788543	Áno
4.	bcfffad6b3c2a04e690913937af835f0	0,3274854	Áno
5.	c9c0e69e4e312b86267ef1a549a3a14f	0,3392351	Áno
6.	ea76509e0f1e83c4f3fa5936cfda83bd	0,3447038	Áno
7.	54efa7dedd9d95fca4722d8264513790	0,3464991	Áno
8.	15ef92ee72e719b56e700840f1a91519	0,3484589	Áno
9.	e1733085b608245700f232558e85ce84	0,3857515	Áno
10.	be6f1abbeb09e86e2fc340a5508ab776	0,432341	Áno
11.	27ce59dd8942af976a1857194d385954	0,4904899	Áno
12.	d71927b4454f5032a1b58076568dbee6	0,5143626	Nie
13.	253b7f0c9853115e46d8fe92829a95d3	0,615799	Áno
14.	299f667f012b72dadf9e6c44e96e17eb	0,6265709	Áno
15.	3e7f05a7233af3221a6fd23346a1db9c	0,6310592	Áno

Tabulka 7.7: 15 najpodobnejších vzoriek so vzorkou c0486732450e9129544d5065a26c547d vyhodnotených v rámci celej databáze metódou NCD

bulke 7.6 bolo 95 sekúnd, pokým trvanie testu v tabulke 7.7 bolo 533 sekúnd. Napokon vzorky na prvých miestach sú takmer zhodné pri oboch algoritmoch.

Pre zaujímavosť sú v tabulke 7.8 uvedené percentuálne úspešnosti vyhodonotenia podobných vzoriek, so zameraním sa na fakt, či vzorky označené ako podobné, boli označené ako škodlivé, alebo nie. Pri tomto teste bolo určovaných 50 najpodobnejších vzoriek k zadanej vzorke. Týmto testom chcem dokázať, že ak by mal byť program použitý na testovanie prítomnosti vírusu vo vzorke, jeho úspešnosť by nebola príliš vysoká. Program je primárne určený na zisťovanie podobnosti medzi vzorkami, ktoré boli označené ako vírusy programom, ktorý bol na heuristiku určený.

P.č.	**Metóda**	**MD5**	**Úspešnosť** %	Trvanie (s)
1.	Leveshtein	2631d390b6f6b16b3f2a39976cec419e	22	97
2.	NCD	2631d390b6f6b16b3f2a39976cec419e	99	552
3.	Levenshtein	0008e7de19ddabc0b12bfe95a97c2737	56	43
4.	NCD	0008e7de19ddabc0b12bfe95a97c2737	100	477
5.	Levenshtein	18929d23c4e49f1e85f8b84de6dbc1ab	20	42
6.	NCD	18929d23c4e49f1e85f8b84de6dbc1ab	44	485

Tabulka 7.8: Porovnanie úspešnosti algoritmov u náhodne zvolených vzoriek. Pojem úspešnosť je vysvetlený na začiatku kapitoly 7.

Z testu je vidieť, ako veľmi Leveshteinova metóda v úspešnosti zaostáva za metódou NCD, je však nepomerne rýchlejšia. Všetko je závislé od popisu porovnávanej vzorky. Príčina velkého rozdielu je najmä v zachovaní pomenovaní súborov v spracovaných popisoch vzoriek pri metóde NCD na rozdiel od metódy Levenshtein, pri metóde Levenshtein sa veľa informácií o činnosti vzorky pri spracovaní popisu zahadzuje. Ďalšie príčiny rozdielnosti ozrejmím pri testovaní umelo vytvorených vzoriek.

Testy vytvorených vzoriek

V nasledujúcich testoch budem pracovať s tabulkou vzoriek, ktorú možno nájsť na priloženom CD[1]. Tabulka obsahuje niekoľko vzoriek označených ako škodlivé a niekoľko vzoriek, ktoré naopak za škodlivé považované nie sú. Základom sú 4 vzorky označené A,B,C,D, názvy vzoriek, ktoré sú od nich odvodené začínajú vždy písmenom vzorky od ktorej sú odvodené(tj. A1, A2, C1 ...). V odvodených vzorkách sú niektoré operácie navyše, prípadne ich je menej, alebo sú navzájom vymenené. Vzorky, ktoré nie sú škodlivé majú počiatočné písmeno X(tj. X1, X2 ...).

Výsledky testov potvrdili väčšiu presnosť metódy NCD, ale vyššiu rýchlosť metódy Leveshtein. Pri vzorke B, ktorá má pomerne málo informácií o činnosti vzorky síce obe metódy správne určili najpodobnejšie vzorky $B1$, B2, B3 avšak zároveň hneď za vzorkami $B1$, B2, B3 sa nachádzajú pri oboch metódach ako najpodobnejšie vzorky $X3$ a $X1$. V testoch sa potvrdila aj väčšia citlivosť metódy Levenshtein, najmä ak sa porovnávané vzorky líšia

[1]Keďže je práca robená pre externú firmu nemohol som poskytnúť celý obsah databáze, preto som vybral niekoľko vzoriek z databáze, ktorá mi bola poskytnutá a rôzne som ich obmenil. Pre ilustráciu funkčnosti algoritmov to je postačujúce.

o operáciu, ktorá má v rámci pravidiel metódy vyššiu cenu. Ak totiž do vzorky, ktorá je na počiatku identická ako vzorka, s ktorou ju porovnávam, pridám riadky obsahujúce napríklad zmenu dôležitých registrov, tak aj vzorky ktoré sa líšia len v dvoch–troch riadkoch sú pri porovnaní touto metódou značne odlišné. Naopak za výhodu považujem rýchlosť metódy a možnosť nastaviť cenu jednotlivých pravidiel reprezentujúcich činnosti. Vo vzorke $D3$ som presunul niekoľko operácií z konca na začiatok textového popisu a pri metóde Levenshtein sa rozdiel medzi vzorkami znatelne zvýšil, na rozdiel od metódy NCD. Za úvahu stojí opäť rola poradia operácií a jej význam. Z testov na reálnych vzorkách sa prikláňam k názoru, že poradie operácií ma istý význam ak ide o porovnávanie činností vzoriek označených za škodlivé a ostatných vzoriek. Ak by bolo porovnanie len medzi vzorkami označenými ako škodlivé považujem za lepšie poradie operácií zjednotiť.

Zhodnotenie vykonaných testov

Použitie Levehsteinovej metódy je najvhodnejšie najmä na zisťovanie podobnosti v rámci vzoriek označených ako škodlivé. Pri porovnaní Levenshteinovou metódou zároveň platí, že čím je popis činností vzorky kratší, tým horšie sú výsledky algoritmu. Neodporúčam preto využitie Levenshteinovej metódy na vzorky, ktorých popis je pomerne krátky, v takom prípade je lepšie použiť metódu NCD. Pri dlhších popisoch je naopak výhodnejšie uprednostniť metódu Levenshtein pretože výsledky oboch metód sú porovnateľné a metóda Levenshtein je rýchlejšia. Výhodou metódy Levenshtein nad metódou NCD je aj komutatívnosť v porovnaní vzoriek. Porovnanie dvoch vzoriek metódou NCD nie je podľa výsledkov testov komutatívne. Presnosť Levenshteinovej metódy sa dá zvýšiť správnym nastavením cien jednotlivých operácií. V konfiguračnom súbore, ktorý sa nachádza v prílohe sú ceny operácií stanovené na základe ich dôležitosti z pohľadu heuristiky.

Kapitola 8

Záver

V rámci mojej práce som sa zameral na problematiku odhaľovania podobných typov škodlivých súborov. V práci som uviedol, aké techniky zisťovania škodlivého kódu sa dajú použiť a ako je možné uchovávať informácie o činnosti skúmaných programov. Cieľom bolo navrhnúť algoritmus, ktorý relatívne rýchle porovná informácie o činnosti programov uložené ako text v databázi. Keďže činnosti boli uložené ako text, snažil som sa využiť algoritmy, ktoré zisťujú rozdielnosť medzi textom. Tieto kritériá spĺňal algoritmus pre výpočet Levenshteinovej vzdialenosti a algoritmus normalizovanej kompresnej vzdialenosti(NCD). Oba algoritmy som upravil tak, aby sa dali použiť na textové popisy činnosti skúmaných programov, ktoré mi boli poskytnuté. Výsledky testov naimplementovaných algoritmov boli uspokojivé, ale zároveň ukázali, že takýto spôsob porovnávania nie vždy stačí. Nevýhodou je výsledok porovnania, ktorý je reprezentovaný číslom, teda jednorozmernou hodnotou, ktorá má veľmi malú výpovednú hodnotu. Ideálnym prípadom pri porovnávaní programom by bolo ich porovnanie na úrovni strojového kódu, takáto metóda by bola podľa mňa presnejšia, aj keď si nedovolím tvrdiť, že bezchybná.

S rozmachom internetu a so zvyšovaním počítačovej gramotnosti užívateľov vzniká ďaleko viac škodlivého kódu ako v minulosti. Často sú to len mierne upravené verzie starších škodlivých programov. Myslím si, že práve v odhaľovaní takýchto škodlivých programov by mohol výsledok mojej práce pomôcť. V práci som ukázal akým spôsobom by sa dali porovnávať textové popisy činnosti programov, ako aj to, ako ďalej výsledky z porovnania využiť. V testoch som zhrnul výhody aj nevýhody mnou implementovaných algoritmov, prípadne ich ďalšie rozšírenie. V budúcnosti by som sa skôr zameral na využitie získaných dát z porovnania programov pre ich klasifikáciu do tried reprezentujúcich skupiny podobných programov.

Literatura

[1] Predictions for Top Security Threats in 2007 as Hacking Comes of Age. Technická zpráva, [cit. 2009-4-26].
URL http://www.govtech.com/gt/print_article.php?id=102603

[2] Analyzer, R. A.: Program control flow diagram. [online], [cit. 2009-4-30].
URL http://publib.boulder.ibm.com/infocenter/rassan/v5r5/index.jsp?topic=/com.ibm.raa.doc/MVShelp/udiacf.htm

[3] Bailey, M.; aj.: Automated classification and analysis of internet Malware. [online], [cit. 2008-12-30].
URL http://www.eecs.umich.edu/fjgroup/pubs/malware-raid07.pdf

[4] Bonfante, G.; Kaczmarek, M.; Marion, J.: Control Flow Graphs as Malware Signatures. [online], [cit. 2009-4-29].
URL http://hal.archives-ouvertes.fr/docs/00/17/62/35/PDF/tcv07b.pdf

[5] Carrera, E.; Flake, H.: Automated Structural Classification of Malware. [online], [cit. 2008-04-26].
URL http://www.sourceconference.com/2008/sessions/pdf/carrera-AutomatedStructuralMalwareClassification.pdf

[6] Distler, D.: Malware Analysis: An Introduction. [online], [cit. 2008-12-26].
URL http://www.sans.org/reading_room/whitepapers/malicious/malware_analysis_an_introduction_2103?show=2103.php&cat=malicious

[7] Gheorghescu, M.: An automated virus classification system. [online], [cit. 2008-12-30].
URL http://download.microsoft.com/download/8/8/3/88375a9e-bd27-4ced-83ce-453272a74b86/Automated_Virus_Classification.pdf

[8] Gilleland, M.: Levenshtein Distance, in Three Flavors. [online], [cit. 2008-12-30].
URL http://www.merriampark.com/ld.htm

[9] Holz, T.; Willems, C.; Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. [online], [cit. 2008-12-28].
URL
http://pi1.informatik.uni-mannheim.de/filepool/publications/j2holz.pdf

[10] Ivanov, I.: API hooking revealed. [online], [cit. 2009-5-10].
URL http://www.codeproject.com/KB/system/hooksys.aspx

[11] Kapoor, A.; Sallam, A.: Rootkits Part 2: A Technical Primer. [online], [cit. 2009-5-12].
URL http:
//www.mcafee.com/us/local_content/white_papers/wp_rootkits_0407.pdf

[12] Karim, E.; Walenstein, A.; A., L.: Malware Phylogeny Generation using Permutations of Code. [online], [cit. 2009-4-26].
URL http://www.cacs.louisiana.edu/~walenste/pubs/
2005-jicv-karim-walenstein-lakhotia-parida.pdf

[13] Kendall, K.: Practical malware analysis. [online], [cit. 2008-12-26].
URL http://www.blackhat.com/presentations/bh-dc-07/Kendall_McMillan/
Paper/bh-dc-07-Kendall_McMillan-WP.pdf

[14] Lee, T.; Mody, J.: Behavioral classification. [online], [cit. 2008-12-30].
URL
http://blogs.technet.com/antimalware/archive/2006/05/16/428749.aspx

[15] Moir, R.: Defining Malware: FAQ. [online], [cit. 2008-12-26].
URL http://technet.microsoft.com/cs-cz/library/dd632948(en-us).aspx

[16] Santos, I.; Penya, Y.; Devesa, J.; aj.: N-Grams-based file signatures for malware detection. [online], [cit. 2009-4-27].
URL http://paginaspersonales.deusto.es/ypenya/publi/penya_ICEIS09_
N-grams-based%20File%20Signatures%20for%20Malware%20Detection.pdf

[17] Shewmaker, J.: Analyzing DLL Injection. [online], [cit. 2008-12-29].
URL http://bluenotch.com/files/Shewmaker-DLL-Injection.pdf

[18] Swiderski, F.: What is behaviour analysis? [online], [cit. 2008-12-27].
URL http://blog.hautesecure.com/forums/t/29.aspx

[19] TechTerms: Malware. [online], [cit. 2009-4-26].
URL http://www.techterms.com/definition/malware

[20] Wehner, S.: Analyzing worms and network traffic using compression. Technická zpráva, 2005.

[21] Zeltser, L.: Using VMware for malware analysis. [online], [cit. 2008-12-26].
URL
http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1254046,00

[22] Zezula, L.: Výuka assembleru. [online], [cit. 2009-5-12].
URL http://www.zezula.net/cz/teach/assembler_01.html

[23] Zubair, M. S.; Khayam, S. A.; Farooq, M.: Embedded Malware Detection using Markov n-grams. [online], [cit. 2009-4-27].
URL http://wisnet.niit.edu.pk/publications/2008/Zubair_DIMVA08.pdf

Dodatek A

Obsah CD

Bin - Zložka obsahujúca spustitelné súbory programu.
Configuration - Zložka obsahujúca konfiguračný súbor k programu.
Doc - Zložka obsahujúca programovú dokumentáciu.
Latex - Zložka obsahujúca zdrojové súbory textu diplomovej práce.
Source - Zložka obsahujúca zdrojové súbory programu.
ClassDiagram.png - Diagram tried programu.
README.txt - Obsahuje popis zložiek a súborov prítomných na CD, ako aj návod na otestovanie priloženého programu.