VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATICKÁ DETEKCE KNIHOVNÍHO KÓDU ZE SPUSTITELNÝCH SOUBORŮ TYPU PE

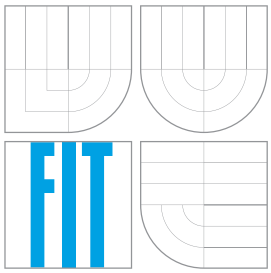
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MAREŠ

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATICKÁ DETEKCE KNIHOVNÍHO KÓDU ZE SPUSTITELNÝCH SOUBORŮ TYPU PE

AUTOMATIC LIBRARY CODE DETECTION IN PE EXECUTABLE FILES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MAREŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing.. KOLÁŘ DUŠAN,

BRNO 2008

Abstrakt

Diplomová práce zabývá rozpoznáváním importovaných funkcí ve spustitelných souborech, které jsou původem ze statických knihoven překladače. Cílem práce je automatizace procesu a zjednodušení analýzy při disassemblování. Samotná detekce je řešena pomocí vyhledávání připravených vzorků s tolerancí změn adres. Výsledná aplikace podporuje i detekci překladače a v základu obsahuje vzorky pro překladač MinGW32, Visual Studio 2005 a C++ Builder 6.

Klíčová slova

spustitelný soubor typu PE, detekce kódu statických knihoven, rozpoznání knihovních funkcí, detekce překladače

Abstract

Master's thesis describes imported functions detection in PE executables, which are from static libraries. Main reason is process automatization and analysis simplification. Detection is solved by searching prepared patterns with missmatch tolerance. Missmatch are caused by changing address during building application. Resulting application supports compiler detection and it contains patterns for MinGW32, Visual studio 2005 and C++ Builder 6.

Keywords

PE executables, static library code detection, static library function recognition, compiler detection

Citace

Petr Mareš: Automatická detekce knihovního kódu
ze spustitelných souborů typu PE, diplomová práce, Brno, FIT VUT v Brně, 2008

Automatická detekce knihovního kódu ze spustitelných souborů typu PE

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Dr. Ing. Dušana Koláře.

.....
Petr Mareš
18. května 2008

Poděkování

Chtěl bych poděkovat svému vedoucímu Doc. Dr. Ing. Dušanovi Kolářovi za velkou dávku trpělivosti, mé rodině za nikdy nekončící podporu a Simonce, která mě podržela, když mi bylo nejhůř. Děkuji všem!

Obsah

Kapitola 1

Úvod

Tato diplomová práce se zabývá automatickou detekcí knihovního kódu ze spustitelných souborů typu PE. Volně navazuje na moji bakalářskou práci [2], která otevírala téma analýzy spustitelných souborů. Jejím tématem byla detekce překladače ze spustitelných souborů. Tato práce jde dál a snaží se pomocí detekce knihovního kódu separovat část aplikace vytvořenou překladačem a část, která je dílem programátora. Část se statickými knihovnami je dále rozdělena na jednotlivé metody. Dojde k jejich pojmenování a příp. zobrazení dalších význačných prvků jako původ či její funkce. Cílem je automatizace procesu analýzy s vidinou úspory práce, času a prostoru. Diplomová práce byla vytvořena pro firmu AVG Technologies.

V praxi lze využít znalost knihovního kódu různými způsoby: Analytik nemusí prohledávat všechen binární kód, aby našel zajímavé úseky kódu. Pojmenování částí kódu nabídne větší přehlednost a nastíní základní funkci úseku bez další analýzy. Přidružená detekce překladače umožní rozdělení databáze souborů podle použitého překladače. Rozpoznaný binární kód lze nahradit kratším symbolem a zvýšit tak výkon archívu z hlediska hledání i prostoru.

Diplomová práce se skládá z několika kapitol. Úvodní kapitola, ve které se právě nacházíte, se zabývá základním popisem a smyslem této práce. Zkusíme si povědět něco o tom, kde se setkáme s detekcí knihovního kódu a zkusíme najít zcela obecné řešení problému vyhledávání. Druhá kapitola Teoretický základ nabízí základní pilíře teoretických znalostí, které jsou potřeba ke zvládnutí tématu. Povíme si stručně něco o struktuře spustitelných souborů, o formátu instrukce architektury Intel x86 a skončíme kapitolkou, která se zajímá o proces sestavení zkompilovaného kódu a statických knihoven. Třetí kapitola shrne nabyté poznatky a provede nás návrhem a implementací aplikace pro detekci kódu statických knihoven. Následující kapitola rozšíří rozpoznávací aplikaci o schopnosti detekce překladače a detekce funkcí a nabídne způsob označení, nahrazení či mazání nálezů. Kapitola obsahuje informace o rozhraní, které lze využít k zakomponování detekčních schopností do jiné aplikace. Pátá kapitola se zabývá konkrétními příklady překladačů. Jsou zde popsány umístění a formáty statických knihoven. Rovněž se zde upozorňuje na strasti při získávání vzorků. Předposlední kapitola obsahuje testy, které dokazují detekční schopnosti implementované aplikace. Následuje závěr, který shrne a zhodnotí dosažené výsledky a nastíní budoucí možnosti vývoje či rozšíření detekce knihovního kódu.

1.1 Současný stav a obecné řešení práce

Rozpoznávání kódu pocházejícího ze statických knihoven překladačů není samozřejmě nový obor. Můžeme se s ním setkat v disassemblerech např. IDA PRO nebo v jiných analytických nástrojích. Bohužel jde vždy o proprietární řešení, takže informací z otevřených zdrojů je velmi poskrovnu. V podstatě jen víme, že to jde, ale už nevíme jak.

Nejlepším řešením problému detekce kódu statických knihoven by bylo takové řešení, které je obecné a tím není závislé na překladači, prostředí či architektuře. Pokud bychom takové řešení našli, nemuseli bychom reagovat na nové verze překladačů či statických knihoven až do té doby, než by se zásadně změnil pohled na programování. Samozřejmě toto řešení jen těžko bude rozpoznávat jednotlivé funkce ze statických knihoven, jejich činnost nebo dokonce jména těchto funkcí.

Abychom mohli implementovat takový způsob vyhledávání, musíme samozřejmě umět dobře popsat, co vlastně hledáme. Zkusme se teď krátce zamyslet, jaký je vlastně kód ze statických knihoven a jak se liší od kódu, který jsme právě napsali a přeložili:

- Opakuje se v programech stejného překladače.
- Je napsán různými programátory jako všechen ostatní kód.
- Je napsán jinými programátory než kód programu.
- Zapouzdřuje systémová volání, obsahuje často volání služeb operačního systému.
- Je už přeložen, neovlivňují ho nastavení překladače, pouze linkeru. Během sestavení se pouze vybírá z různých verzí statických knihoven či exportovaných funkcí např. verze s ladícími informacemi či bez nich

Je zřejmé, že tento popis je nedostatečný a nelze podle něho programově rozpoznat knihovní kód. Nejzajímavější z hlediska vyhledávání je to, že se tento kód opakuje. Kdybychom vytvořili aplikaci, která má schopnost paměti a učení, mohli bychom poznat sekvence kódu ze statických knihoven podle opakovaných výskytů v různých aplikacích přeložených stejným překladačem. Toto obecné řešení se zdá životaschopnější, ale přináší sebou další problémy např. ve velkém množství vzorových dat.

Výhody:

- Obecné řešení pro všechny překladače.

Nevýhody:

- Opakovat se může i neknihovní kód.
- Pro učení musí mít uloženy všechny kódové sekce.
- Nikdy neobjeví všechen knihovní kód, naopak může považovat obyčejný kód za knihovní.
- Není zřejmá hranice poměru počtu nalezení sekvence a počtu vzorových souborů. Stačí, že se sekvence kódu vyskytuje v X% vzorových souborů?

- Jeden vzorek může obsahovat více funkcí knihovny.
- Různé vzorky mohou obsahovat stejné funkce knihovny.
- Analyzátor nemá žádné informace o vzorku.

Toto řešení je sice obecné, tedy že přesně stejný postup učení a detekce aplikujeme na všechny překladače, ale abychom vytvořili, tak velikou množinu učebních vzorků musíme každé prostředí či překladač instalovat a to už je lepší získat statické knihovny přímo z překladače. Navíc tím získáme jistotu, že hledáme přesně to, co máme. Jednotlivé vzorky budou odpovídat přilinkovaným funkcím a dozvíme se další informace jako jména těchto funkcí.

Výhody:

- Dokáže rozlišit přilinkované funkce.
- Jistota, že hledáme správné vzorky.(Samozřejmě i tak můžeme zaznamenat s velkou pravděpodobností falešný nález.)
- Doplňkové infomace o vzorcích např. jméno funkce a statické knihovny.

Nevýhody:

- Různé verze překladače, různé formáty uložení statických knihoven.
- Vliv nastavení linkeru.

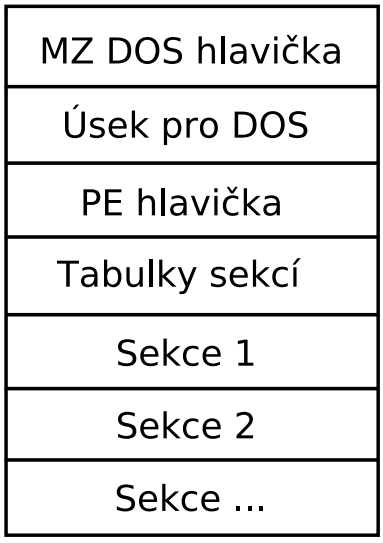
Diplomová práce se dále zabývá pouze třetím způsobem řešení detekce kódu statických knihoven, protože je nejperspektivnější a jako jediné nabízí rozpoznání přilinkovaných funkcí překladače, jak si vyžaduje zadání diplomové práce.

Kapitola 2

Teoretický základ

2.1 Formát spustitelných souborů a statických knihoven typu PE

Vzhledem k našemu problému nás hlavně zajímá, které části souboru obsahují instrukce v binární podobě, ať už jde o spustitelné soubory či statické knihovny. Musíme si tedy nejprve říci něco u strukturách, které tvoří tyto soubory. Kapitola čerpá z mé bakalářské práce [2] a ze specifikace PE a COFF souborů od Microsoftu [4].



Obrázek 2.1: Struktura spustitelného souboru

Spustitelné soubory typu PE se skládají z několika struktur, které mají za úkol popsat vzezření souboru a dále z několika sekcí, které obsahují binární kód, (ne)inicializovaná data a další zdroje potřebné k běhu programu.

2.1.1 Způsoby adresování

Než se pustíme do popisu PE souboru, řekněme si něco o způsobu adresování. Ve strukturách spustitelného souboru typu PE se můžeme setkat s dvěma typy adres. První jsou

adresy typu Raw, které ukazují přímo do spustitelného souboru např. ukazatel PointerToRawData v tabulce sekcí (struktura _IMAGE_SECTION_HEADER) ukazuje přímo na začátek sekce v souboru. RVA neboli Relative Virtual Address jsou relativní adresy, které začnou platit, až operační systém nahraje spustitelný soubor před spuštěním do paměti. Adresy jsou relativní, protože operační systém nemusí nahrát soubor na adresu udanou v ImageBase, a pak by muselo dojít k vyhledání a přepočítání všech adres. Příkladem může být proměnná VirtualAddress z tabulky sekcí, která ukazuje též na záčátek sekce, ale až po nahrání operačním systémem do paměti.

Výpočet adresy RVA na Raw

Při procházení struktur často narazíme na problém, že některé adresy na další struktury jsou RVA. Nelze je tedy použít přímo ke skoku na tuto adresu, ale musí se z nich spočítat pozice v souboru. Všimněme si, že pokud je použita RVA adresa, tak se tato struktura nachází vždy v datových oblastech některé ze sekcí. Můžeme tedy využít toho, že o každé sekci známe nejen její polohu v souboru, ale i budoucí polohu v paměti, jde též o RVA. Stačí tedy v cyklu projít celou tabulku sekcí a najít tu, kde platí:

```
ST[n].VirtualAddress <= HledanaRVA <= ST[n].VirtualAddress + ST[n].VirtualSize
```

Poté už jen stačí odečíst RVA sekce od hledané RVA a po přičtení Raw adresy sekce dostaneme Raw adresu původní RVA, tedy:

```
Raw = HledanaRVA - ST[n].VirtualAddress + ST[n].PointerToRawData
```

2.1.2 Struktury spustitelného souboru

Exe soubor začíná tzv. MZ DOS hlavičkou (struktura _IMAGE_DOS_HEADER), která je zde z důvodu kompatibility s MS DOS. Tato struktura slouží k zavedení aplikace v prostředí MS DOS. Samozřejmě nelze zavést aplikaci pro Win32, ale zavede se tzv. MS DOS Stub, který může obsahovat alternativní program. Dnes se ale tato část omezuje pouze na výpis zprávy, že aplikace potřebuje pro svůj běh operační systém na bázi Windows.

MZ DOS hlavičce nás z hlediska Windows zajímají dvě hodnoty. První dva bajty tzv. Magic bytes obsahují znaky "MZ"[1]. Tyto dva znaky slouží k ověření, že jsme opravdu otevřeli spustitelný soubor s MZ hlavičkou. Druhou hodnotou je čtyřbajtový ukazatel na další hlavičku v souboru, konkrétně na PE hlavičku.

```
typedef struct _IMAGE_DOS_HEADER
{
            WORD e_magic;
             ...
            LONG e_lfanew;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

PE hlavička začíná opět Magic bytes, tentokrát jde o hodnoty PE\0\0. Následuje povinná struktura FileHeader a volitelná OptionalHeader.

[1] Jedná se o iniciály jména Mark Zbirowsky, který se podílel na návrhu formátu spustitelných souborů.

```
typedef struct _IMAGE_NT_HEADERS
{
        DWORD Signature;
        IMAGE_FILE_HEADER FileHeader;
        IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS,*PIMAGE_NT_HEADERS;
```

Datová struktura FileHeader je zajímavá hodnotou NumberOfSections, která nás informuje o počtu sekcí. V těchto sekcích najdeme námi hledaný binární kód. Proměnné PointerToSymbolTable a NumberOfSymbols podávají informace o poloze a velikosti tabulky symbolů. Bezprostředně za tabulkou symbolů se nachází StringTable neboli tabulka řetězců. V této struktuře se pak nacházejí dlouhá jména sekcí. SizeOfOptionalHeader informuje o velikosti OptionalHeader, pokud se hodnota rovná nule, pak tato struktura neexistuje např. není v některých statických knihovnách.

```
typedef struct _IMAGE_FILE_HEADER
{
        ...
        WORD NumberOfSections;
        ...
        DWORD PointerToSymbolTable;
        DWORD NumberOfSymbols;
        WORD SizeOfOptionalHeader;
        ...
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Struktura OptionalFileHeader obsahuje jedinou zajímavou hodnotu a tou je pole DataDirectory. Každý prvek pole DataDirectory obsahuje strukturu, která obsahuje RVA adresu odkazované struktury a hodnota Size určuje počet těchto struktur. Tato tabulka popisuje různé struktury např. tabulky exportů, importů či výjimek, strukturu pro ladící informace atd.

```
typedef struct _IMAGE_OPTIONAL_HEADER
{
        ...
        IMAGE_DATA_DIRECTORY  DataDirectory[IMAGE_NUMBEROF
                                            _DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER,*PIMAGE_OPTIONAL_HEADER;

typedef struct _IMAGE_DATA_DIRECTORY
{
        DWORD VirtualAddress;
        DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Pro nás je zajímavý hned první prvek, který odkazuje na tabulku exportů. Ta popisuje všechny exportované funkce knihovny. Tabulka exportů je reprezentována strukturou _IMAGE_EXPORT_DIRECTORY.

```
0   Export table address and size
1   Import table address and size
2   Resource table address and size
3   Exception table address and size
    ...
15  Reserved1
```

Ke kódu exportovaných funkcí se dostaneme přes RVA AddressOfFunctions, která odkazuje na pole RVA ukazatelů na binární kód funkcí. Počer těchto funkcí udává proměnná NumberOfFunctions. Obdobné je to se jmény funkcí. RVA na pole RVA na jména funkcí najdeme pod proměnnou AddressOfNames. Z hlediska definice knihoven nemusí počet jmen a funkcí odpovídat, protože některé aplikace mohou přistupovat k funkcím nejen přes jména, ale i přes pořadí funkcí v knihovně.

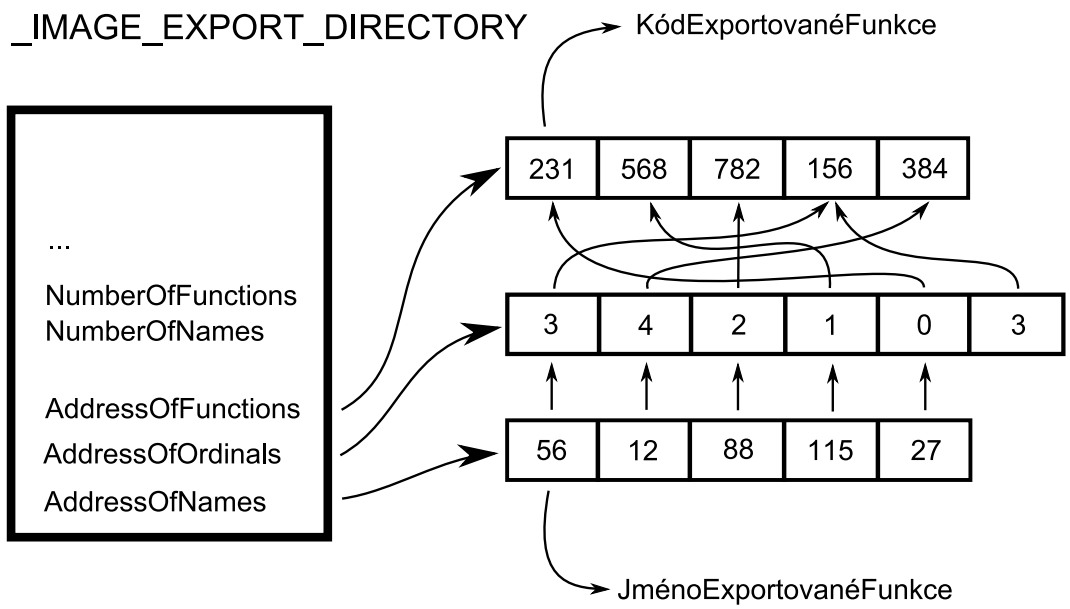
```
PocetOrdinalnichFunkci = NumberOfFunctions - NumberOfNames
```

Tento postup se ale moc nepoužívá, protože po aktualizaci knihovny může dojít k promíchání funkcí. Ordinální přístup je samozřejmě rychlejší. Poslední hodnota AddressOfNameOrdinals je RVA na pole hodnot (indexů), které přiřazují jména k funkcím. Mezi poli jmen a funkcí totiž není implicitně žádná vazba a navíc podle definice může jedna funkce vystupovat pod více jmény. Celou situaci s exportovanými funkcemi přehledně ilustruje obrázek.

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
        ...
        DWORD NumberOfFunctions;
       DWORD NumberOfNames;
       DWORD AddressOfFunctions;
       DWORD AddressOfNames;
       DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

Bezprostředně za PE hlavičkou jsou tabulky sekcí. Každá sekce v souboru má vlastní strukturu, která popisuje umístění, velikost a obsah.

```
typedef struct _IMAGE_SECTION_HEADER
{
        BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
        ...
        DWORD SizeOfRawData;
        DWORD PointerToRawData;
        ...
        DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```



Obrázek 2.2: Ilustrace přístupu k exportovaným funkcím

Hlavička sekce začíná jménem. Délka jména může být maximálně 8 znaků[2], poté je třeba využít tabulku symbolů, o které jsme mluvili u PE hlavičky. Tyto delší názvy lze využít pouze statických knihoven, protože u spustitelných souborů či DLL knihoven dojde k jejich zkrácení. Použití delšího názvu symbolizuje "\" na prvním znaku názvu. Následuje offset do tabulky řetezců.

Hodnoty SizeOfRawData a PointerToRawData nám poskytnou velikost a přesné umístění počátku sekce v souboru. Poslední relevantní hodnotou jsou bitové příznaky nastavené v Characteristics. Pro nás jsou důležité dva:

```
IMAGE_SCN_MEM_EXECUTE 0x20000000 The section can be executed as code.
IMAGE_SCN_CNT_CODE 0x00000020 The section contains executable code.
```

Pokud jsou tyto dva bity nastaveny na 1, obsahuje tato sekce spustitelný strojový kód.

Struktury jsou definovány v souboru Winnt.h. Tuto strukturu a další zde použité naleznete v příloze na konci diplomové práce.

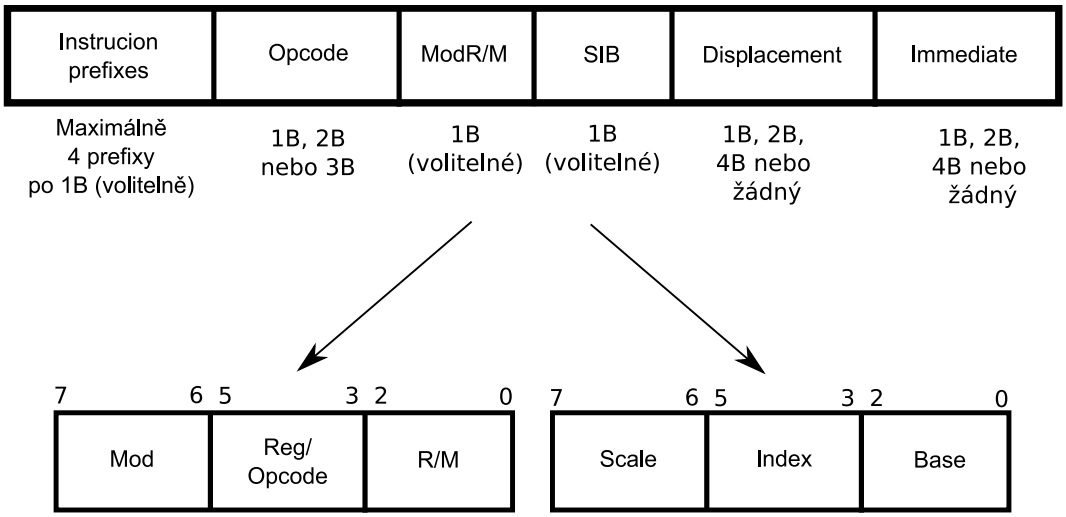
2.2 Instrukční sada procesorů Intel x86 a kompatibilních

Instrukční sada architektury x86 zasahuje detekci spustitelného kódů jen okrajově, protože řešení nepracuje s významem jednotlivých instrukcí. Na problém můžeme narazit při vyhledávání, kdy zjistíme, že se v podstatě veškeré nalezené shody liší od získaných vzorků ze statických knihoven. U některých instrukcí se během sestavení totiž mění adresy. Kapitola čerpá z druhého svazku IA-32 Intel Architecture Software Developers Manual [1].

[2]Pokud je délka jména sekce přesně 8 znaků, chybí ukončující nuly.

2.2.1 Formát instrukce

Projděme si formát instrukce. Velikost instrukce je od 1B až po teoretických 17B. Každá instrukce musí obsahovat samozřejmě operační kód, který jednoznačně popisuje její činnost.



Obrázek 2.3: Formát instrukce x86

Instruction Prefixes

Instrukce může obsahovat až 4 jednobajtové prefixy. Existují 4 skupiny a z každé z nich může být vždy použit pouze jeden prefix:

1. skupina obsahuje prefixy pro uzamykání paměti a pro opakování u instrukcí pracujících s řetězci.
2. skupina obsahuje prefixy pro předpověď větvení
3. skupina obsahuje prefixy pro změnu délky operandu
4. skupina obsahuje prefixy pro změnu délky adresy

Opcodes

Určuje samotnou instrukci. Kód instrukce je 1B až 3B dlouhý. Další tři bity MODR/M určují např. směr operace, velikost displacementu atd.

Bajty MODR/M a SIB

Obsahuje tři bitová pole. Slouží k adresování operandů. Mod pole vybere z osmi registrů nebo 24 módů adresování. Reg/opcode specifikuje registr nebo obsahuje část instrukce. Pole r/m vybere jako operand registr nebo se kombinuje s mod polem pro zpřesnění způsobu adresování.

SIB se skládá ze třech polí, které slouží k adresování:
Pole scale je měřítko bázového registru.
Pole index obsahuje číslo indexového registru.

Pole base obsahuje číslo bázového registru.

Displacement a Immediate

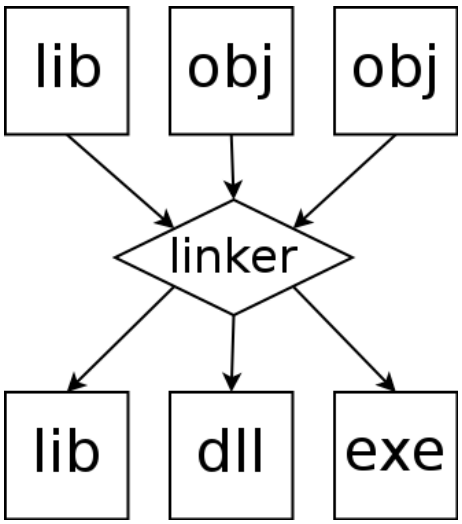
V případě, že instrukce přistupuje do paměti, může obsahovat tzv. Displacement. Jde vlastně o posunutí v paměti, které je třeba přičíst. Pole Immediate může obsahovat přímý operand.

Závěr

Během sestavení dochází ke změnám adres. V celé instrukci se může změnit zcela jistě Displacement a pole Immediate, protože zde může docházet k aritmetickým operacím s adresami. Změna SIB je nepravděpodobná. Tato zjištění ověříme později prakticky.

2.3 Připojování standardních knihoven ke spustitelným souborům v době překladu

Posledním teoretickým předpokladem pro řešení diplomové práce je způsob připojení statických knihoven během překladu. Správně by se mělo říct po překladu, protože je zdrojový kód nejprve přeložen a vznikne soubor se strojovým kódem typu object s příponou *.o. Tento soubor ještě není plnohodnotná aplikace, protože neobsahuje části kódu z jiných souvisejících souborů typu object ani ze statických knihoven překladače. K doplnění těchto částí dojde během sestavení, které má na starosti linker. Během této činnosti dochází k relokacím kódu, proto musí nutně docházet ke změnám adres. Výsledkem práce linkeru nemusí být jen spustitelný soubor, ale i statická nebo dynamická knihovna.



Obrázek 2.4: Ilustrace funkce linkeru

V objektovém souboru jsou mimo strojového kódu definovány symboly. Tyto symboly zastupují strojový kód, proměnné nebo další data např. bitmapy pro kurzory a ikony.

Máme dva druhy symbolů:

Exportované symboly jsou definované v daném modulu a zároveň jsou nabízené pro ostatní moduly.

Importované symboly jsou použity v modulu, ale nejsou v něm definovány.

Linker vlastně vyhledá v každém souboru symboly, které zde nejsou definované a spáruje je s exportovanými symboly jiného modulu. Tyto symboly může také získat ze statických knihoven, což jsou vlastně archívy objektových souborů. Linkery se liší podle toho, jestli importují knihovnu celou nebo pouze potřebné symboly.

Linker má také na starosti uspořádání aplikace v adresovém prostoru. K přesunům dochází i na základě relokací, kdy se změní bázová adresa a dojde ke změně umístění. V důsledku se musí přepočítat většina adres (pouze některé relativní adresy zůstanou nezměněny), ať se jedná o skoky, načítání či ukládání do paměti. Tato práce je velmi zjednodušena moderními operačními systémy, které vytvoří pro každou aplikaci vlastní a oddělený paměťový prostor. Zároveň OS umožňuje nahrát aplikaci do paměti na jakoukoli virtuální adresu.

Závěrem je třeba zdůraznit, že z hlediska vyhledávání přináší linkování problémy, protože je třeba reflektovat změnu adres či proměnných. Kapitola čerpala z WWW [5]

Kapitola 3

Aplikace pro detekci a rozpoznání statických knihoven spustitelných souborů typu PE

Cílem je navrhnout a implementovat aplikaci, která nalezne ve spustitelném souboru sekce se spustitelným kódem a v nich označí části, které jsou ze statických knihoven a navíc informuje o rozpoznaných knihovnách. Aplikace se musí vyrovnat s odlišnostmi mezi vzorky a nálezy, které způsobuje linker změnou adres. Další podstatnou funkcí je automatické získání vzorků z několika typů překladačů. Důraz je kladen na ověření způsobu detekce a na automatizaci detekce než na množství podporovaných překladačů.

3.1 Návrh

Návrh aplikace je plně objektový. Snaží se oddělit tři činnosti aplikace: otevírání spustitelných souborů PE, import vzorků ze statických knihoven a rozpoznávání kódu. Projděme se nyní tyto objekty podrobněji.

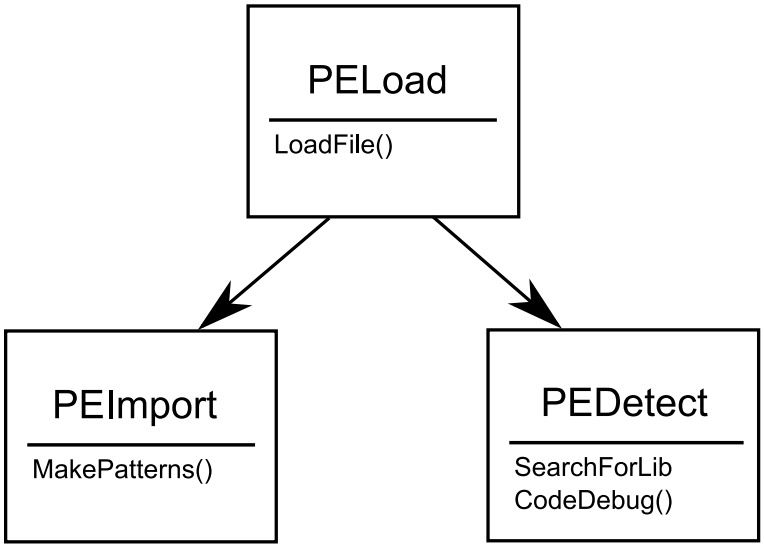
3.1.1 Třída PELoad

Třída PELoad má na starosti správné načtení spustitelného souboru či statické knihovny do paměti. Dále nalezne a zpracuje hlavičkové struktury. Třída musí být schopna rozpoznat, zda se jedná opravdu o spustitelný soubor typu PE. Třída si vystačí s jedinou veřejnou metodou LoadFile(string FileName). O chybách informuje vyvoláním výjimek.

Implementace musí vhodně reagovat na rozdíly mezi spustitelným souborem a statickými knihovnami či rozdíly mezi knihovnami samotnými. Třída provádí načítání bez zásahu uživatele a nabízí podporu pro tzv. "hloupé" načtení, kdy nejsou reflektovány struktury či formát souboru a dojde pouze ke zkopírování souboru do paměti. Takto lze vyhledávat v nepodporovaných formátech a můžeme také částečně ověřit správnost importů vzorků.

3.1.2 Třída PEImport

Třída PEImport zdědí schopnosti načítání od třídy PELoad. Sama doplní metodu MakePatterns(string PatternsOutputFile), která získá vzorky z otevřené statické knihovny a uloží je



Obrázek 3.1: Architektura aplikace

do souboru s příponou PAT. K rozpoznání formátu knihovny se použije přípona souboru. Jsou podporovány dva formáty. Typ Object - přípona .o nebo .obj, kde má každá exportovaná funkce uložený kód v samostatné sekci. Typ DLL - přípona DLL či BPL - kód exportovaných funkcí je společně uložen ve spustitelné sekci. Na jednotlivé funkce je odkazováno přes tabulku exportů.

3.1.3 Třída PEDetect

Třída PEDetect zdědí schopnosti načítání od třídy PELoad. Sama doplní metodu SearchForLibCodeDebug(), která označí nalezené vzorky a vypíše výsledky.

Způsob vyhledávání

Vyhledávání bez úprav vzorků není vůbec myslitelné. Je třeba vyřešit obecně problém přepočítávání adres. První způsob je projít vzorek instrukci po instrukci a nahrazovat části instrukcí zástupnými symboly. Tím bude mít vzorek pořád stejnou délku a vyhledávání bude fungovat.

Prakticky mám z prvotní implementace ověřeno, že tato metoda funguje, ale je třeba rozumět binárnímu kódu na úrovni instrukcí. To by znamenalo implementovat spolehlivý disassembler, který by kód přeložil a označil by místa, u kterých by mohlo dojít ke změně adresy. Ovšem implementace disassembleru by velmi zdražila odpověď na otázku, zda je detekce možná nebo ne. Použití externího disassembleru by bylo sice jednodušší, ale aplikace je pak na něm závislá a samozřejmě jsou pak také problémy s automatizací importu.

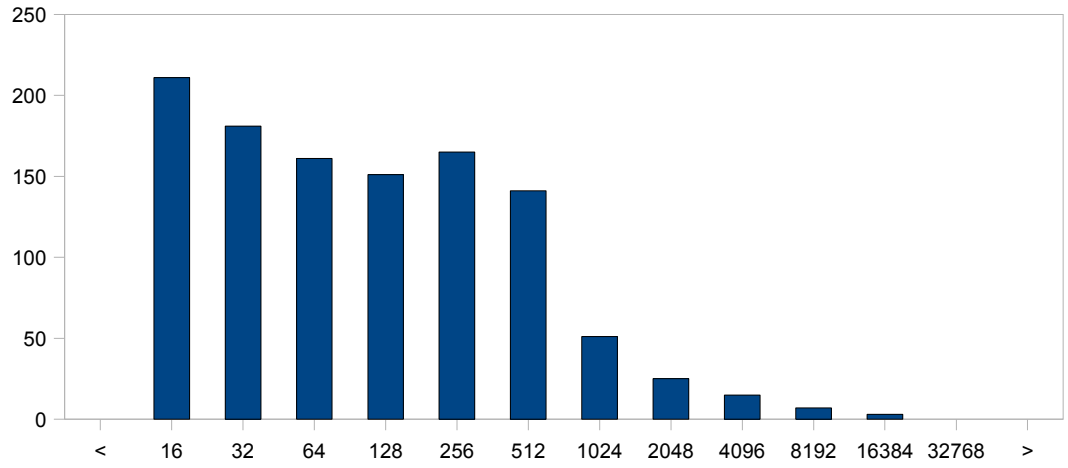
Druhou možností je nechat vzorky, tak jak jsme je získali a upravit vyhledávání. Toto upravené vyhledávání by tolerovalo vícebajtové odlišnosti a to i vícekrát ve vzorku. Tím odpadá předzpracování vzorků a dochází ke zjednodušení implementace, proto bylo rozhodnuto použít právě tento způsob vyhledávání, který toleruje chyby. Délka této chyby by měla být podle teorie formátu instrukce 4 až 8 bajtů.

Analýza vyhledávání s tolerancí neshod

Vyhledávání probíhá pro každý bajt sekce. Vzorky jsou postupně porovnávány, nakonec je vybrán nejdelší vzorek, který se shoduje. Další vyhledávání v případě úspěšného nálezu probíhá až za nálezem. Pokud rozšíříme vyhledávání o toleranci neshod, nedojde v případě první neshody k vyřazení vzorku, ale dojde pouze k navýšení počítadla bajtů neshod. K vyřazení vzorku dojde až po překročení daného počtu neshod. V případě shody je počítadlo vynulováno.

Problémem je, že nevíme, jaký vliv bude tolerance chyb na kvalitu vyhledávání. Vždy se budeme pohybovat mezi větším procentem detekce či větším množstvím falešných nálezů. Kvalitu nálezu ovlivňuje v první řadě délka vzorku, čím je vzorek delší, tím je obecně unikátnější a s klesající pravděpodobností nálezu, klesá i pravděpodobnost toho falešného. S kvalitou vyhledávání určitě souvisí poměr mezi skutečně nalezenými bajty vzorku a těmi, které prohlásíme za tolerované neshody.

Začněme délkou vzorku. Průměrná délka vzorku překladače MinGW32 je 286 bajtů. Minimální délka je 16 bajtů a maximální 16 000 bajtů. V grafu jsou vyneseny počty délek vzorků pro délky 0 - 8, 9 - 16, 17 - 32, 33 - 64, atd.



Obrázek 3.2: Délka vzorků překladače MinGW32

16 bajtů délky nejkratších vzorků stačí na bezpečné nalezení vzorku. Nesmíme ovšem zapomínat, že binární kód není náhodná sekvence bajtů a zcela jistě existuje množina hodnot, které se vyskytují v sekvenci častěji než ostatní a které nejsou zastoupeny vůbec, proto je počet kombinací diametrálně odlišný od 10^{38} všech teoretických kombinací.

$$256^{16} = 3,40 * 10^{38}$$

Délka vzorku je tedy dostatečná. Zkusme nyní najít, jaká délka tolerance neshod je pro nás přijatelná. Hledáme hodnotu takovou, aby byl úspěch detekce co nejvyšší s co nejmenší hodnotou tolerované neshody. Pokud se podíváme do tabulky, zjistíme, že výsledek měření je jednoznačný. Nejlepší poměr mají 4 bajty maximální tolerované neshody. Z teoretické

části víme, že během sestavení programu dochází ke změnám adres. Tyto adresy jsou dlouhé podle použitého prostředí, buď 4 bajty pro 32bit či 8 bajtů pro 64bit prostředí.

Tolerance neshod (B)	0	1	2	3	4	5	6	7	8
% detekce:	3,80	3,86	4,4	10,17	90,95	90,96	91,0	91,04	91,05
Počet 1B neshod	-	123	123	163	420	417	406	408	390
Počet 2B neshod	-	-	37	184	1000	1013	1012	1019	1020
Počet 3B neshod	-	-	-	402	2806	2798	2800	2795	2797
Počet 4B neshod	-	-	-	-	2531	2512	2517	2511	2490
Počet 5B neshod	-	-	-	-	-	39	38	40	29
Počet 6B neshod	-	-	-	-	-	-	42	32	66
Počet 7B neshod	-	-	-	-	-	-	-	47	52
Počet 8B neshod	-	-	-	-	-	-	-	-	75

Dále si v tabulce všimněme, že čtyřbajtové neshody tvoří pouhých 40%. Je to pravděpodobně způsobeno tím, že aplikace pracuje v malém paměťovém prostoru a adresy se liší pouze na méně významných bitech. Nyní nám zbývá zjistit, jestli použití čtyřbajtové tolerované neshody neznehodnotí nálezy nejkratších vzorků.

%neshod	počet vzorků	%neshod	počet vzorků
0%	105	26%	0
1%	28	27%	0
2%	36	28%	2
3%	26	29%	1
4%	19	30%	0
5%	30	31%	21
6%	50	32%	0
7%	27	33%	0
8%	60	34%	0
9%	34	35%	0
10%	31	36%	0
11%	21	37%	8
12%	42	38%	0
13%	15	39%	0
14%	7	40%	0
15%	24	41%	0
16%	38	42%	0
17%	37	43%	7
18%	30	44%	0
19%	9	45%	0
20%	11	46%	0
21%	49	47%	0
22%	10	48%	0
23%	6	49%	0
24%	3	50%	1
25%	47		

Provedl jsem měření viz poslední tabulka a poměr neshodných bajtů překročil 31% pouze v 16 případech z celkových 834 nálezů. To znamená, že u nejkratších vzorků zbývá

cca 10 bajtů, které se shodují. Shodu na 10 bajtech můžeme považovat ještě za dostatečnou, ale kratší vzorky už nelze doporučit, protože může docházet k falešným nálezům, které by mohly znehodnotit výsledky.

$$256^{10} = 1,2 * 10^{24}$$

Závěr tedy je, že použití vyhledávání s opakovanou tolerancí čtyřbajtových neshod lze použít bez ztráty kvalitních výsledků, pokud dodržíme minimální délku u všech vzorků.

3.2 Implementace

Implementace probíhala ve vývojovém prostředí Dev-C++ později Eclipse v jazyku C++. Obě prostředí využívájí překladač MinGW32. Obě prostředí i překladač jsou šířeny pod licencí GNU GPL.

3.2.1 Třída PELoad

Třída PELoad definuje datovou strukturu na uložení souboru.

```
struct sPEFile
{
              BYTE *pFile;
              unsigned int FileSize;

              eFileTypes FileType;

//----------- struktury ve spustitelném souboru (PE)

              PIMAGE_DOS_HEADER pMZHeader;

              BYTE *pDosStub;
              unsigned int DosStubSize;

              PIMAGE_NT_HEADERS pPEHeader;
              PIMAGE_FILE_HEADER pFileHeader;

              vector<sSectionHeader> SectionHeaders;

              PIMAGE_EXPORT_DIRECTORY IEDTable;
};
```

Před tím než dojde ke zkopírování spustitelného souboru do paměti (ukazatel pFile), musí být zjištěna velikost načítaného souboru a dojde k alokaci odpovídajícího bloku paměti. Velikost načteného souboru není programově omezena, závisí pouze na tom, jestli operační systém tak velký blok paměti přidělí. Samotné načtení struktur neboli nastavení ukazatelů do obrazu souboru v paměti, má na starosti pět privátních metod, které jsou volány postupně podle typu vstupního souboru.

Podporované formáty vstupních souborů

Třída PELoad je schopna načíst tři formáty souborů založených na formátu PE - běžný spustitelný soubor a dva formáty knihoven:

obj - formát uložení v překladači MinGW32 a Visual Studio

bpl - formát uložení v překladači Borland C++ Builder (jedná se vlastně o dynamické knihovny, protože se statickými byl problém.)

Následující tabulka ukazuje použití jednotlivých metod pro načítání daného formátu. Metody jsou volány postupně shora dolu.

Metoda	EXE	BPL	OBJ
void LoadMZHeader()	x	x	
void LoadDosStub()	x	x	
void LoadPEHeader()	x	x	x
void LoadSectionHeaders()	x	x	x
void LoadImageExportDirectory()		x	

LoadMZHeader()

Metoda načte MZ hlavičku neboli nastaví ukazatel pMZHeader na začátek souboru. Navíc dochází k ověření "Magic Bytes", zda se rovnají iniciálům MZ, pokud ne, je vyvolána výjimka.

LoadDosStub()

DosStub následuje hned za MZ hlavičkou. Obsah pro nás nemá žádnou hodnotu, ale i tak se nastaví ukazatel na DosStub a uloží se jeho délka, proměnné pDosStub a DosStubSize. Délka se spočítá tak, že se od hodnoty e_lfanew z MZ hlavičky odečte délka právě MZ hlavičky. Pokud nastane případ, kdy je délka DosStub záporná, tak DosStub vůbec neexistuje a navíc jsou částečně překryty struktury MZ a PE hlavičky. Tento úskok je překvapivě funkční, ale setkáme se s ním pouze u pochybného softwaru jako jsou viry, cracky apod. a u produktů demoscény.

LoadPEHeader()

Obdobně jako u MZ hlavičky se nastaví ukazatel na pozici PE hlavičky. Počátek je na pozici v souboru určované hodnotou e_lfanew z MZ hlavičky. Poté dojde k ověření symbolů "PE". V případě načítání statické knihovny typu object se načte z PE hlavičky pouze struktura _IMAGE_FILE_HEADER.

LoadSectionHeaders()

Tabulky sekcí neboli struktury _IMAGE_SECTION_HEADER se nachází ihned za PE hlavičkou. Jejich počet je dán proměnnou NumberOfSections z PE hlavičky. Metoda ukládá ukazatele do struktur sSectionHeader do vektoru SectionHeaders. V případě, že jméno sekce začíná lomítkem, znamená to, že jméno je delší než 8 znaků, a proto musí být dohledáno ve tabulce řetězců. Toto jméno je pro další použití uloženo do struktury sSectionHeader do proměnné LongName.

```
struct sSectionHeader
{
    PIMAGE_SECTION_HEADER pSectionHeader;
    string LongName;
};
```

Výpočet pozice jména sekce ve string table .

Tabulku řetězců nalezneme za tabulkou symbolů, na kterou odkazuje hodnota PointerToSymbolTable v PE hlavičce. Abychom získali ukazatel na tabulku řetezců, musíme k ukazateli na tabulku symbolů přičíst počet prvků tabulky vynásobený délkou jednoho prvku. Nakonec stačí přičíst offset, který je uložen v běžné proměnné pro jméno sekce za lomítkem.

```
PEFile.pFileHeader->PointerToSymbolTable +
+ PEFile.pFileHeader->NumberOfSymbols * sizeof(IMAGE_SYMBOL) + Offset
```

void LoadImageExportDirectory()

Metoda pouze nastaví ukazatel IEDTable na adresu tabulky exportů neboli _IMAGE_EXPORT_DIRECTORY. Volá se pouze u knihoven s příponou BPL.

3.2.2 Třída PEImport

Implementace třídy PEImport zahrnuje mimo jedné veřejné metody MakePatterns i čtyři privátní metody. Metody ImportObjectFilePatterns() a ImportBPLPatterns() v cyklu načítají vzorky z knihoven a pro zápis do souboru se vzorky volají metodu AddPattern(). Metoda CreateHeader() vytvoří, jak už název napovídá, hlavičku souboru se vzorky s informacemi o překladači.

```
void ImportObjectFilePatterns(fstream & OutputFile);
void ImportBPLPatterns(fstream & OutputFile);
void CreateHeader(fstream & OutputFile, string Compiler, string Version,
                  string Description);
void AddPattern(fstream & OutputFile, string Name, string Description,
                  char * Data, unsigned int Size);
```

Třída ukládá vzorky do souboru s příponou PAT do adresáře Patterns. Pokud už soubor existuje, přídá vzorky na jeho konec. Tím jsou vzorky přiřazeny k překladači, kterému původní vzorky patří. Nelze takto uložit vzorky více překladačů do jednoho souboru. Vzorky se ukládají v následujícím formátu:

```
#jméno překladače#verze překladače#jméno knihovny
*jméno funkce*popis(zde ukládám délku vzorku)*
030a5453716c4442547970650100000000
*jméno funkce*popis(zde ukládám délku vzorku)*
030a5453716c4442547970650100000000
...
```

Příklad:

```
#C++ Builder#6#cds60.bpl
*@$xp$18Dblocal@TSqlDBType*143*
030a5453716c4442547970650100000000030000000c1b8540077479706544425807747
97065424445077479706541444f07747970654942580744424c6f63616c8d4000a01b85
40000000000000000000000000881e8540000000000000000000000006c1e854008030
000145785406413854018108540441385402010854010108540001085400810854040c29
85
*@$xp$28Dblocal@TCustomCachedDataSet*2531*
071454437573746f6d436163686564446174615365740a01b8540105785404200074442 4
...
```

Omezení pro importované vzorky

Import vzorků musí mimo samotného získání zajistit i kvalitu vzorku. Nevhodné vzorky totiž mohou výrazně zvýšit množství falešných nálezů. Během testování se objevil problém se vzorky, které obsahují abnormální množství nul popř. obsahují jen nuly. Takové vzorky nedávají smysl a navíc jsou detekovány velmi často na místech, které překladač vyplnil nulami jen kvůli zarovnání. Proto během importu dochází k eliminaci těchto vzorků včetně těch, u kterých by nenulové bajty mohlo zakrýt tolerování neshod.

Výsledná omezení pro importované vzorky

- minimální délka vzorku je 16 bajtů
- vzorky nesmí obsahovat abnormální množství nulových bajtů

Více o importu, formátu a výskytu statických knihoven naleznete v kapitole Detekované překladače a formát statických knihoven.

3.2.3 Třída PEDetect

Implementace třídy PEDetect se dělí na dvě části. V první dojde k načtení vzorků a v druhé jsou tyto vzorky vyhledány. Nejprve jsou tedy načteny všechny soubory se vzorky z adresáře Patterns. Při načítání je podporováno rozdělení vzorků jediného překladače do více souborů např. podle statických knihoven. Více překladačů v jediném souboru není podporováno. Tento přístup by vytvářel nepořádek a ztěžoval záměnu vzorků některého z překladačů.

Načítání vzorků má na starosti veřejná metoda LoadCompilerPatterns, která obsahuje parser a dvě pomocné privátní metody ReadUntil a ReadPatternContent. Výsledkem jejich snažení je správně naplněný vektor objektů cCompiler. Každý objekt překladače obsahuje vektor vzorků neboli objektů cPattern.

```
//definice třídy vzorku
class cPattern
{
    public:
        int *Text;
        int Size;
        string Name;
```

```
        string Description;

        cCompiler * Compiler;

        // počítadlo na počet neshod
        int MissCount[MAX_LOST];

        // počítadlo počtu nalezu vzorku
        int Found;

        cPattern(cCompiler * Compiler)
        {
            Found = 0;
            this->Compiler = Compiler;
        }
};

typedef vector<cPattern> vPatterns;

//definice třídy překladače
class cCompiler
{
    public:
       string Name;
       string Description;
       string Version;
       vPatterns Patterns;

       cCompiler()
       {
            Name = string();
       }
};
```

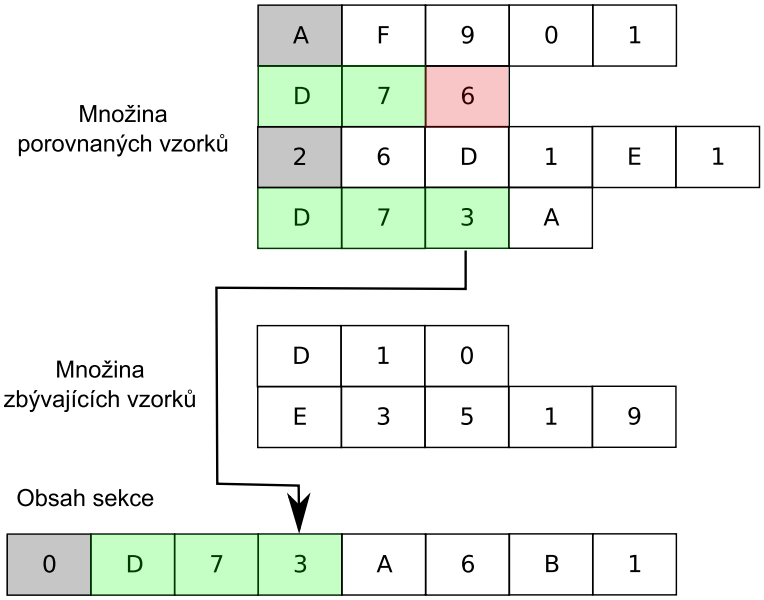
Popis implementace vyhledávání

Vyhledávání se od semestrální práce pozměnilo s cílem zjednodušení a zrychlení. Algoritmus probíhá pro každý bajt zkoumané sekce. Začíná se prvním vzorkem a porovná se první bajt, pokud se rovná, nepokračuje se dalším vzorkem, ale porovnává se další bajt. Porovnávání probíhá tak dlouho, dokud není vzorek nalezen či vyřazen. V případě nálezu se pokračuje dalším vzorkem, protože musí být jisté, že nalezený vzorek je nejdelší možný. Samozřejmě je implementována tolerance neshod. Počet tolerovaných bajtů jdoucích po sobě se nastavuje v hlavičkovém souboru PEDetect.h.

```
#define MAX_LOST 4
```

Tolerance neshodných bajtů je možná kdekoliv mimo prvního bajtu vzorku, kde to nedává smysl, protože na prvním bajtů musí být ta část instrukce, která určuje, co se

bude dělat, nikoliv adresa. Toto omezení slouží zároveň jako optimalizace prohledávacího algoritmu, protože k vyřazení nevhodných vzorků dojde po prvním porovnání.



Obrázek 3.3: Ilustrace vyhledávání

Na obrázku je zobrazen stav, kdy první a třetí vzorek byl vyřazen pro neshodu na prvních bajtech. Druhý vzorek byl i přes jednobajtovou neshodu, označen za nález. Nyní probíhá porovnávání se čtvrtým vzorkem, který bude prohlášen za nález místo druhého vzorku, protože je delší.

V PEDetect je též definováno omezení na maximální délku vzorku na 32kB:

```
#define MAX_PATTERN_SIZE 32768
```

Výsledná omezení pro toleranci neshod

- omezení délky po sobě jdoucích neshod na 4 bajty
- vzorek nesmí začínat neshodou

3.3 Výsledky

Nyní nezbývá nic jiného než prověřit správnost všech předpokladů a implementace. Testování jsem prováděl na vzorové aplikaci “Hello World!”. Výstup obsahuje informace o pozici nálezu v sekci, jméno souboru se vzorky, jméno statické knihovny a jméno sekce statické knihovny (jméno exportované funkce). Výstup je zkrácen (mimo nálezů jsou zkráceny i jména funkcí):

```
Sector size: 252416 Virtual Address:401000h
720 (4012d0) - 736 (4012e0) concept-inst.o-.text$_ZN9__gnu_cxx21_Inp
Missmatched bytes / Pattern size: 7/16 (43%)
```

```
1200 (4014b0) - 1328 (401530) globals_io.o-.text
Missmatched bytes / Pattern size: 15/128 (11%)
1328 (401530) - 4704 (402260) ios_init.o-.text
Missmatched bytes / Pattern size: 966/3376 (28%)
4704 (402260) - 7520 (402d60) eh_personality.o-.text
Missmatched bytes / Pattern size: 103/2816 (3%)
7520 (402d60) - 9616 (403590) ios.o-.text
Missmatched bytes / Pattern size: 167/2096 (7%)
9616 (403590) - 15984 (404e70) locale_init.o-.text
Missmatched bytes / Pattern size: 1556/6368 (24%)
15984 (404e70) - 22704 (4068b0) locale.o-.text
Missmatched bytes / Pattern size: 534/6720 (7%)
22704 (4068b0) - 23152 (406a70) eh_catch.o-.text
Missmatched bytes / Pattern size: 33/448 (7%)
...
230208 (439340) - 230400 (439400) locale-inst.o-.text$_ZNSt8messages
Missmatched bytes / Pattern size: 40/192 (20%)
230752 (439560) - 230944 (439620) locale-inst.o-.text$_ZNSt8numpunct
Missmatched bytes / Pattern size: 38/192 (19%)
230944 (439620) - 231136 (4396e0) locale-inst.o-.text$_ZNSt8numpunct
Missmatched bytes / Pattern size: 32/192 (16%)
231136 (4396e0) - 231328 (4397a0) locale-inst.o-.text$_ZNSt8numpunct
Missmatched bytes / Pattern size: 38/192 (19%)
...
248376 (43da38) - 248408 (43da58) fstream-inst.o-.text$_ZTv0_n12_NSt
Missmatched bytes / Pattern size: 4/32 (12%)
----------------------------------------------------------------------
Segment size: 252416 Identified segment size: 229568 Ratio: 90.9483%
----------------------------------------------------------------------
```

Program rozpoznal více než 90% binárního kódu, což se dá prohlásit za výtečné. Pokud necháme prozkoumat samotnou aplikaci PEDetect (je třeba vytvořit kopii) bude úspěch nižší, necelých 68%. Druhá aplikace zjevně obsahuje větší množství kódu, který vytvořil programátor. Jen pro připomenutí, v semestrální práci bylo procento detekce u testovací aplikace "Hello World!" o dvě procenta nižší. Toto "zlepšení" způsobila přítomnost kratších vzorků 16 až 31 bajtů, které byly v minulé verzi aplikace pod povolenou hranicí 32 bajtů. Zároveň je též vidět, že i takto krátké vzorky jsou ještě použitelné a nedochází ke znehodnocení výsledků.

```
Výsledek se vzorky , které jsou delší či rovny 32B:
----------------------------------------------------------------------
Segment size: 252416 Identified segment size: 224512 Ratio: 88.9452%
----------------------------------------------------------------------
```

V této kapitole se nám podařilo ověřit, že detekce statických knihoven na základě vzorků je možná. K vyhledávání lze využít implementovaný prohledávácí algoritmus včetně tolerance neshod s nastavením na čtyři bajty. Výsledkem je aplikace, která dokáže označit u překladače MinGW32 desítky procent knihovního kódu.

Kapitola 4

Implementace techniky pro detekci překladače a jeho knihovních funkcí

4.1 Návrh

Druhá aplikace bude rozšiřovat funkčnost první. Účelem je detekovat knihovní funkce a správně je pojmenovat. Než přistoupíme k samotnému rozpoznávání knihovních funkcí je třeba provést detekci překladače. To nám umožní vyhledávat pouze vzorky patřící ke správnému překladači, čímž dojde k výraznému zlepšení výkonu a zmenšíme počet falešných nálezů.

4.1.1 Detekce překladače

Detekci překladače lze řešit několika způsoby. Můžeme čerpat z mé bakalářské práce [2], kterou jsem dělal právě na toto téma. Zde se využívaly informace z popisných struktur spustitelného souboru, které se porovnávaly s hodnotami ze vzorových dat. Tato metoda je funkční a ověřená, ale lze ji ovlivnit editací úvodních hlaviček souboru. Druhou možností je využít vzorků ze statických knihoven, které už stejně máme pro další detekci a podle množství jejich nálezů rozhodnout, který překladač je původcem souboru. Rozhodl jsem se využít druhou metodu, protože díky existenci pouze jediných vzorků je aplikace z uživatelského hlediska jednodušší a to nám v důsledku šetří čas a zvyšuje spolehlivost.

Detekce překladače se bude podobat vyhledávání statických knihoven. Největším rozdílem bude enormní množství vzorků, které je třeba otestovat, proto je třeba pečlivě rozmyslet, jakým způsobem bude vyhledávání probíhat, aby se neobjevily s výkonnostní problémy. Samozřejmě tyto optimalizace nesmí mít žádný vliv na výsledek.

Možné optimalizace detekce překladače (řazeno podle složitosti od nejmenší)

- zkrácení prohledávané plochy
- vyřazení neúspěšných překladačů už během vyhledávání
- zmenšení množiny vyhledávaných vzorků

Hned u první optimalizace, která zkracuje prohledávanou plochu, by se mohlo stát, že vyřadíme takovou část, která obsahuje většinu vzorků, což by mohlo ovlivnit výsledky. Z praktických testů, ale víme, že rozdíly v nálezech mezi vítězným a ostatními překladači je jeden až dva řády, takže i kdybychom vyřadili např. 70% nálezů, tak budeme detekovat stejný překladač. Můžeme taky prohledávanou plochu měnit dynamicky podle rozdílu úspěšnosti mezi překladači.

Vyřazování překladačů už během vyhledávání lze aplikovat jednoduše, pouze musí být vždy jisté, že vyřazujeme beznadějný překladač a musíme si určit, kdy bude probíhat vyřazování.

Zmenšení množiny vyhledávaných vzorků je zdaleka nejefektivnější metoda. Pokud nám zbyde od každého překladače několik desítek vzorků, bude detekce blesková. Navíc můžeme mít více různě objemných množin vzorků od každého překladače, tak aby v případě nejasného výsledku mohla být použita větší množina s více vzorky. Tato metoda ale trpí zásadním problémem. Není jasné, které vzorky jsou vhodné do užší detekční množiny. Nemůžeme samozřejmě počítat s pomocí uživatele, aplikace si musí poradit sama. Tento problém se dá vyřešit jedině učením. Můžeme aplikaci spustit nad dostatečnou množinou souborů (aby každý překladač měl minimálně X výskytů), a ta by si navíc během detekce překladače ukládala statistiku, zda se vzorek vyskytoval a kolikrát. Poté by se vyřadily vzorky, které nebyly nalezeny či byl jejich výskyt sporadický. Teď to možná vypadá jednoduše, ale uvědomme si, že vyřazování vzorků musí být vůči všem překladačům spravedlivé.

Takže každý překladač dostane stejné množství vzorků? Ale přece každý vzorek je jinak dlouhý! Tak překladačům přidělíme tolik vzorků, aby jejich délka byla přibližně stejná. Ale co s opakovaným výskytem či enormními rozdíly v délce vzorků? Takže bychom přidělili tolik vzorků, aby jejich délka byla přibližně stejná, nepovolili opakovaný nález a omezili délku vzorku nějakou konstantou? Ano, to zní rozumně. Navíc jak jsem psal výše, rozdíl mezi vítězným překladačem a ostatními je jeden až dva řády, takže si můžeme dovolit menší nepřesnosti, nikoliv však hrubé chyby.

V aplikaci je nakonec implementována kombinace první a druhé optimalizace, vyřazování vzorků není implementováno.

4.1.2 Detekce knihovních funkcí

Vyhledávání knihovních funkcí těsně následuje po detekci překladače. Dále se budou vyhledávat pouze vzorky z detekovaného překladače. První aplikace, která vyhledávala statické knihovny, je nacházela pomocí vzorků, které odpovídají jedna ku jedné exportovaným funkcím, takže změny ve vyhledávání jsou minimální a orientují se pouze na sběr informací o hledaných funkcích např. jméno funkce lze získat u překladače MinGW32 z názvu sekce.

4.1.3 Označení nálezu, eliminace nálezu a zakomponování do větší aplikace

Vytvořená aplikace není samozřejmě určena pro koncového uživatele, slouží hlavně k ověření způsobu detekce. V případě splnění zadání se může stát součástí většího analytického nástroje, proto je třída PEDetect vybavena rozhraním, které umožňuje předat informace o nálezech mimo objekty aplikace.

K označení nálezu slouží struktura sMatch. Ve struktuře najdeme informace o nalezeném vzorku, ukazatel na pozici nálezu v souboru, relativní adresu nálezu v sekci a konečně adresu na místo nálezu v paměti, kam bude program nahrán operačním systémem před spuštěním.

Eliminace nálezu není implementována, protože se počítá se zakomponováním do větší aplikace, který může provést tuto činnost podle svého uvážení a s podporou disassembleru. Implementace by byla triviální, stačí nahradit nálezy např. jménem knihovny, ale protože se počítá se zakomponováním aplikace do většího celku není možné, abychom takto jednoduše zasahovali do obrazu souboru.

```
struct sMatch
{
     // relativní pozice nálezu v sekci
     int SectionPosition;

     // ukazatel do souboru
     unsigned char * FilePosition;

     // pozice v obrazu EXE souboru po nacteni OS
     int ImagePosition;

     // ukazatel na nalezený vzorek
     cPattern *Pattern;
};
```

Rozhraní třídy nabízí tři veřejné metody. Jednu pro detekci překladače a dvě pro nalezení vzorků. Metoda GetUsedCompiler vrací detekovaný překladač, v případě neúspěšné detekce vrací NULL. Dvě metody GetLibCodeMatches se liší pouze parametry, první vyhledá vzorky překladače z prvního parametru a vratí nálezy jako vektor struktur sMatch. Protože detekce chvíli trvá a kvůli možnosti okamžité reakce na nález, existuje druhá metoda GetLibCodeMatches, která využívá k informaci o nálezu callback funkci, které předá strukturu sMatch s informacemi o nálezu.

```
cCompiler * GetUsedCompiler();

vMatch * GetLibCodeMatches(cCompiler * Compiler);

void GetLibCodeMatches(cCompiler * Compiler,
                   void(*CallBackFunction)(sMatch Match));
```

Použití detekce překladače a knihovního kódu je triviální. Ukážeme si ho na příkladu malé aplikace, který bude lepší než sáhodlouhý popis.

Příklad minimální detekční aplikace:

```
#include "PE/pedetect.h"

void CallBackFunction(sMatch Match)
{
```

```
        cout << Match.Pattern->Name << endl;
}

int main(int argc, char *argv[])
{
        cPEDetect PEDetect;

        //otevře soubor
        PEDetect.LoadFile(argv[1], false);

        //detekuje překladač
        cCompiler * Compiler = PEDetect.GetUsedCompiler();

        //detekuje kód statických knihoven
        if(Compiler != null)
          PEDetect.GetLibCodeMatches(Compiler, CallBackFunction);

        // uvolníme zdroje
        PEDetect.Close();
}
```

4.2 Implementace

Původní aplikace je rozšířena v několika ohledech. Nejdůležitější je podpora pro detekci překladače, která umožní efektivněji rozpoznat statické knihovny různých překladačů. Samozřejmá je podpora pro více překladačů, ať už rozšíření třídy PEImport pro další překladače či možnost mít uloženy statické knihovny jednoho překladače ve více souborech.

Detekci překladače mají na starosti dvě privátní metody, které rozšiřují třídu PEDetect. DetectCompiler obsahuje logiku detekce a GetRecognizedLibCodeSize slouží jako pomocná metoda pro vyhledávání vzorků kódu v daném úseku souboru.

```
//* detekuje překladač vrací ukazatel na objekt cCompiler, jinak NULL
cCompiler * DetectCompiler();

//* vrací počet bajtů, které nalezne od daného překladače
//* v daném úseku souboru
unsigned int GetRecognizedLibCodeSize(unsigned char * Data,
        int VirtualAddress, int DataSize, cCompiler Compiler);
```

Popišme si výchozí detekční algoritmus: Pro každý překladač bude provedena detekce kódu statických knihoven. Ten, který rozpozná největší část sekce se spustitelným kódem, je vítěz. Samozřejmě tento způsob je nepoužitelný, protože by to trvalo neúměrně dlouho, proto jsou podle analýzy implementovány optimalizace. Detekční algoritmus využívá vyřazování nejméně úspěšných překladačů a zároveň zkracuje prohledávanou plochu podle úspěchu detekce, tedy pokud je zřejmé, že jeden překladač je výrazně lepší než ostatní, ukončí se předčasně vyhledávání. K ukončení vyhledávání dojde též v případě mizivého úspěchu detekce všech podporovaných překladačů.

Zkracování prohledávané plochy znamená, že algoritmus neprohledá celou sekci, ale prohledává ji po částech a po každé části se kontrolují parametry, pokud jsou splněny detekce končí a je znám vítěz. Počet částí definuje SECTION_DIVIDER. Protože je použito vyřazování neúspěšných překladačů, není hledán vítěz, ale ty překladače, kterým se nedaří a vítěz vlastně zbyde. Aby začlo vyřazování musí být nejprve splněna základní podmínka "minimalního rozpoznání nejlepšího překladače" neboli abychom mohli vyřazovat je třeba, abychom rozpoznali dostatečnou část sekce, protože nemůžeme vyřazovat, pokud jeden překladač rozpoznal dva vzorky, další jeden a ostatní nic. Tuto mez definuje MINIMAL_COMPILER_DETECTION_DISCARD. Pokud je tato podmínka splněna, je porovnán nejlepší překladač s ostatními. V případě, že platí

$$prekladac/nejlepsi\,prekladac < DISCARD\_COMPILER\_THRESHOLD$$

je překladač vyřazen. Vítězný překladač musí splnit poslední podmínku a tou je minimální míra rozpoznání. Hodnotu definuje MINIMAL_COMPILER_DETECTION_WINNER. Tato míra určuje minimální poměr mezi rozpoznanou a celkovou velikostí sekce. Pokud není splněna, vítěze nelze určit a metoda DetectCompiler vrací NULL

```
// ***** PARAMETRY PRO DETEKCI PREKLADACE
// * určuje počet častí, na které se rozdělí sekce během detekce
// * překladače
#define SECTION_DIVIDER 8

//* minimální úspěšnost překladače vzhledem k nejúspěšnějšímu
//* překladači, pokud se dostane překladač pod tuto hranici je
//* z další detekce vyřazen
#define DISCARD_COMPILER_THRESHOLD 0.1

//* minimální úspěšnost nejúspěsnějšího překladače, při kterém
//* proběhne vyřazování nejslabších překladačů
#define MINIMAL_COMPILER_DETECTION_DISCARD 0.01

//* minimální úspěšnost nejúspěšnějšího překladače, kdy může
//* být prohlašen za vítěze
#define MINIMAL_COMPILER_DETECTION_WINNER 0.05

//* maximální úspěšnost pro předčasné ukončení detekce překladače
//* v případě mizivého úspěchu
#define MAX_COMPILER_DETECTION_END 0.005
```

Kapitola 5

Detekované překladače a formát statických knihoven

Tato kapitola nabízí informace o umístění statických knihoven, o použitém formátu a o dalších úskalích, které je třeba překonat pro získání vzorků. Aplikace pracuje se třemi překladači, které zastupují tři největší rodiny: Opensource, Borland a Microsoft. Všechny pracují s jazykem C++.

5.1 MinGW32

MinGW32 je zde jako jediný zástupce překladače, který je šířen pod GNU GPL, proto byl tento překladač zvolen k prvotní analýze. Díky lincenci jsou dostupné i zdrojové kódy Statické knihovny nalezneme v adresáři překladače v adresáři lib. Nalezneme zde přímo soubory typu object (přípona .o) a nebo archívy (přípona .a), které obsahují velké množství dalších souborů typu object.

Formát objektových souborů využívá zjednodušený formát PE. Soubor začíná strukturou PE hlavičky _IMAGE_FILE_HEADER. Následují tabulky sekcí a samotné sekce. Každá ze sekcí označená za spustitelnou obsahuje jednu exportovanou funkci. Získání vzorků tedy znamená zkopírování obsahu sekcí. Jména funkcí odpovídají názvům sekcí. V případě delších jmen je nalezneme v tabulce řetězců, kterou jsme si popsali v teorii o formátu PE souboru.

5.1.1 Standardní knihovna

V adresáři lib nalezneme mnoho statických knihoven. Standardní knihovna má jméno libstdc++.a. Funkce z této knihovny nalezneme v aplikacích, které používají jazyk C++. Pokud použijeme jazyk C, jsou funkce volány dynamicky z DLL knihoven operačního systému.

Práce s archívy

Kvůli přehlednosti jsou objektové soubory statických knihoven zabaleny archivátorem. To pro nás představuje menší obtíž, protože buď musíme rozšířit importovací třídu o metody pro extrakci a nebo musíme využít schopnosti externího archivátoru. V této fázi došlo k

rozhodnutí, že bude použit externí archivátor. Archivátor je součástí MinGW32, konkrétně ho nalezneme v balíku BinUtils. Po instalaci bude v adresáři překladače v adresáři bin.

Samotné použití je triviální. Stačí ho spustit s parametrem -xo (půjde o rozbalení archívu) a jménem knihovny.

```
ar -xo libstdc++.a
```

Pro snadnější práci byly vytvořeny dávkové soubory, které archív nejprve rozbalí a potom vytvoří vzorky. Zde je příklad konkrétního dávkového souboru pro standardní knihovnu libstdc++.a překladače MinGW32.

```
@echo off
rem Vytvori sablony pro vyhledavani v PE souboru verze pro MinGW32

SET PEDetect="PEDetect.exe"

ar -xo libstdc++.a

for %%f in (*.o) do %PEDetect% -i %%f MinGW32 3.4.2 libstdc++.pat
```

5.2 C++ Builder 6

Druhý podporovaný překladač patří do rodiny firmy Borland. Jedna se sice už o starší překladač, ale protože jsou tato prostředí zpětně kompatibilní, měli bychom alespoň částečně rozpoznávat knihovní kód i novějších prostředí.

5.2.1 Získání vzorků

Zatímco hledání statických knihoven bylo otázkou okamžiku, jejich import se ukázal jako velmi složitý. Formát je totiž od PE zcela odlišný a mnohem méně přívětivější. Rozhodl jsem se, proto provést pokus, zda lze získat potřebné vzorky i z dynamických knihoven. Tyto knihovny jsou už samozřejmě formátu PE, takže získání vzorků by nemělo být náročné. Tyto knihovny nalezneme na instalačním CD v adresáři Install/System32/. Mají příponu BPL.

Dynamická knihovna má skoro identický formát se spustitelným souborem. Jedním z rozdílů je existence tabulky exportů. Práce s ní při získání umístění a jmen exportovaných funkcí je popsána v druhé kapitole. Jediný údaj, který nám chybí je délka každé exportované funkce, tento údaj zde implicitně není. Jeden ze způsobů, jak získat informaci o délce, je seřadit adresy počátků funkcí od nejnižší po nejvyšší a za konec funkce vzít počátek další. Tento způsob samozřejmě není úplně optimální, protože funkce mohou být zarovnány na násobky adres a ke konci může být použita výplň, kterou samořejmě ve spustitelném souboru nenajdeme. Uvidíme v testech a v uložených logovacích souborech, které budou při testech vytvořeny a budou součástí DVD.

Importovány byly všechny knihovny, proto je množina vzorků větší:
adortl60.bpl, bdecds60.bpl, bdertl60.bpl, cds60.bpl, dbexpress60.bpl, dbrtl60.bpl, dbxcds60.bpl, dsnap60.bpl, dsnapcrba60.bpl, dsnapent60.bpl, dss60.bpl, ibevnt60.bpl, ibxpress60.bpl, inet60.bpl, inetdb60.bpl, inetdbbde60.bpl, inetdbxpress60.bpl, nmfast60.bpl, qrpt60.bpl, rtl60.bpl,

soaprtl60.bpl, tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl, vcl60.bpl, vcldb60.bpl, vcldbx60.bpl, vclie60.bpl, vcljpg60.bpl, vclshlctrls60.bpl, vclsmp60.bpl, vclx60.bpl, visualclx60.bpl, visualdbclx60.bpl, webdsnap60.bpl, websnap61.bpl, xmlrtl60.bpl

5.3 Visual studio 2005

Visual studio 2005 sjednocuje všechny podporované technologie firmy Microsoft. Z množství podporovaných jazyků jsem si vybral C/C++ s překladem do nativního kódu počítače, nejde tedy o technologii .NET.

5.3.1 Standardní knihovny

Knihovny nalezneme v adresáři překladače v adresáři /VC/lib/. Formát je stejný s knihovnami MinGW32, pouze jsou jiné přípony. Místo .a je použita .lib a z .o se stala přípona .obj. Pro rozbalení se opět použije archivátor. Rozbalení knihoven pro statické linkování proběhlo v pořádku, ale knihovny pro dynamické linkování odolaly. Pokusy o rozbalení končily chybou. Rozhodl jsem se importovat vzorky alespoň z ostatních knihoven.

C Knihovna	DLL knihovna	Charakteristika
libcmt.lib	není	statické linkování
msvcrt.lib	msvcr80.dll	dynamické linkování (importní knihovna msvcr80.dll)
libcmtd.lib	není	statické linkování (ladící informace)
msvcrtd.lib	msvcr80d.dll	jako msvcrt.lib navíc obsahuje ladící informace

Vzorků v těchto knihovnách je enormní množství, proto je jejich vyhledávání pomalejší. Pro detekci byly použity knihovny: libcmt.lib, libcmtd.lib, libcpmt.lib a libcpmtd.lib.

C++ Knihovna	DLL knihovna	Charakteristika
libcpmt.lib	není	statické linkování
msvcprt.lib	msvcp80.dll	dynamické linkování (importní knihovna msvcp80.dll)
libcpmtd.lib	není	statické linkování (ladící informace)
msvcprtd.lib	msvcr80d.dll	jako msvcprt.lib navíc obsahuje ladící informace

Informace o standardních knihovnách poskytly webové stránky Microsoftu [3].

Kapitola 6

Testy

Testy spočívají ve změření detekčních schopností vzhledem k překladači a rozpoznávacích schopností pro označení nálezu funkce statické knihovny.

Před zahajením testování jsem nastavil aplikaci, aby tolerovala čtyřbajtové neshody. U detekce překladače jsem nastavil, že pro vítězný překladač stačí, aby rozpoznal jen 1% spustitelné sekce. Takto malá hodnota má sice dopad na výkon, ale lépe uvidíme, zda falešné nálezy dokáží ovlivnit výsledek detekce. Předčasné skončení detekce je možné, když překladač do poloviny sekce nerozpozná více než 0,000005% kódu.

6.1 Testované soubory a výsledky

Pro testy byla nachystána aplikace PEDetect se vzorky třech překladačů a dále bylo vytvořeno několik desítek testovacích spustitelných souborů. Nalezneme zde zástupce různých verzí Visual Studia (C++ i .net), MinGW32, 7 prostředí od Borlandu, FreePascal a několik souborů také používá kompresi UPX. Celkem je to 74 souborů, u kterých je zaručen původ překladače.

Legenda:
BL - C++ Builder 6
MN - MinGW32
VS - Visual Studio 2005

Spustitelný soubor	**Překladač**	**Verze**	**RP**	**Rozp. kód**
Builder5_w_pack_Abel.exe	C++ Builder	5	BL	16,11%
Builder6_clx_app_deb_P1.exe	C++ Builder	6	BL	5,83%
Builder6_clx_app_rel_Project1.exe	C++ Builder	6	BL	5,83%
Builder6_com_clx_wo_pack_P1.exe	C++ Builder	6	–	–%
Builder6_com_wo_pack_Project2.exe	C++ Builder	6	–	–%
Builder6_wo_pack_Project1.exe	C++ Builder	6	BL	7,49%
Builder6_wo_pack_Project5.exe	C++ Builder	6	BL	7,49%
Builder6_w_pack_Project2.exe	C++ Builder	6	BL	36,39%
Builder6_w_pack_Telefon.exe	C++ Builder	6	BL	24,22%
BuilderX_bcc32_debug.exe	BuilderX	X	–	–%
BuilderX_bcc32_release.exe	BuilderX	X	–	–%

BuilderX_MinGW32_debug.exe	BuilderX	X	–	–%
BuilderX_minGW32_rel.exe	BuilderX	X	–	–%
Delphi10_.NET_CMD.exe	Delphi	10	–	–%
Delphi10_.NET_VCL.exe	Delphi	10	–	–%
Delphi10_.NET_WFA.exe	Delphi	10	–	–%
Delphi10_C#_CMD_deb.exe	Delphi	10	–	–%
Delphi10_C#_CMD_rel.exe	Delphi	10	–	–%
Delphi10_C#_WFA_deb.exe	Delphi	10	–	–%
Delphi10_C#_WFA_rel.exe	Delphi	10	–	–%
Delphi10_Cpp_CMD_wo_pack.exe	Delphi	10	–	–%
Delphi10_Cpp_service_wo_pack.exe	Delphi	10	BL	9,05%
Delphi10_Cpp_VCL_wo_pack.exe	Delphi	10	BL	6,99%
Delphi10_Delphi_CMD_w_pack.exe	Delphi	10	BL	19,48%
Delphi10_Delphi_VCL_wo_pack.exe	Delphi	10	–	–%
Delphi10_Delphi_VCL_w_pack.exe	Delphi	10	BL	26,05%
Delphi5_CMD_wo_pack.exe	Delphi	5	BL	25,98%
Delphi5_CMD_w_pack.exe	Delphi	5	BL	16,28%
Delphi5_VCL_wo_pack.exe	Delphi	5	BL	45,31%
Delphi5_VCL_w_pack.exe	Delphi	5	BL	16,28%
Delphi6_CMD_wo_pack.exe	Delphi	6	BL	25,98%
Delphi6_CMD_w_pack.exe	Delphi	6	BL	52,20%
Delphi6_VCL_wo_pack.exe	Delphi	6	BL	38,60%
Delphi6_VCL_w_pack.exe	Delphi	6	BL	64,25%
Delphi6_w_pack_ELO.exe	Delphi	6	BL	61,74%
Delphi6_w_pack_fontexport.exe	Delphi	6	BL	40,42%
Delphi7_CLX_w_pack.exe	Delphi	7	BL	45,08%
Delphi7_CMD_w_pack.exe	Delphi	7	BL	37,98%
Delphi7_VCL_wo_pack.exe	Delphi	7	BL	40,09%
Delphi7_VCL_w_pack.exe	Delphi	7	BL	53,50%
Delphi7_w_pack_Project7.exe	Delphi	7	BL	27,18%
Delphi7_w_pack_Project72.exe	Delphi	7	BL	53,50%
Delphi9_C#_debug.exe	Delphi	9	–	–%
Delphi9_C#_VCL.exe	Delphi	9	–	–%
Delphi9_C#_WFA.exe	Delphi	9	–	–%
Delphi9_Delphi.NET_CMD.exe	Delphi	9	–	–%
Delphi9_VCL_wo_pack.exe	Delphi	9	BL	47,75%
Delphi9_VCL_w_pack.exe	Delphi	9	BL	28,79%
FreePascal_magic.exe	Free Pascal	?	–	–%
FreePascal_MATICE.exe	Free Pascal	?	–	–%
MinGW32_C++_importDLL.exe	MinGW32	3.4.2	MN	79,92%
MinGW32_C++_PEDetect.exe	MinGW32	3.4.2	MN	67,48%
MinGW32_C++_test.exe	MinGW32	3.4.2	MN	90,95%
MinGW32_C_Project15.exe	MinGW32	3.4.2	MN	1,38%
UPX_builder6_telefon.exe	UPX	?	–	–%
UPX_MinGW32_importDLL.exe	UPX	?	–	–%
VS2003_.NET_ITU03d.exe	Visual studio	2003	VS	13,73%
VS2003_.NET_ITU03r.exe	Visual studio	2003	VS	8,91%
VS2003_Cubicle.exe	Visual studio	2003	VS	1,85%

VS2003_TestPlusWin.exe	Visual studio 2003	VS	11,23%
VS2005_.NET_C#_ConsoleApp1.exe	Visual studio 2005	–	–%
VS2005_.NET_C#_Deb_Busses.exe	Visual studio 2005	–	–%
VS2005_.NET_C#_Rel_Busses.exe	Visual studio 2005	–	–%
VS2005_.NET_C++_WFA_CLR.exe	Visual studio 2005	–	–%
VS2005_.NET_C++_debug.exe	Visual studio 2005	VS	28,56%
VS2005_.NET_DB.exe	Visual studio 2005	–	–%
VS2005_.NET_Debug_Opravy.exe	Visual studio 2005	–	–%
VS2005_.NET_TestDLL.exe	Visual studio 2005	–	–%
VS2005_C++_deb_CMD.exe	Visual studio 2005	VS	22,85%
VS2005_C++_deb_Win.exe	Visual studio 2005	VS	28,56%
VS2005_C++_rel_CMD.exe	Visual studio 2005	VS	27,41%
VS2005_C++_rel_Win.exe	Visual studio 2005	VS	20,14%
VS2008_Cpp_deb_hello.exe	Visual studio 2008	VS	4,68%
VS2008_Cpp_win32_deb.exe	Visual studio 2008	VS	6,04%

Výsledky testů dopadly úspěšně. Detekční aplikace rozpoznala překladač u všech souborů, které znala popř. patřil do rodiny. Naopak u souborů, které rozpoznat neměla, docházelo k falešným nálezů jen velmi zřídka a můžeme tedy tvrdit, že nálezy budou kvalitní a vliv náhodných falešných nálezů je minimální. U prostředí Borlandu docházelo k rozpoznání funkcí i v případě souborů využívajících dynamické knihovny. Zdá se, že některé funkce jsou součástí souborů i přesto, že jsou v dynamických knihovnách. Může jít například o optimalizace běhu aplikace.

Z výsledku si také můžete všimnout, že pokud máte vzorky z jednoho prostředí, máte ho vlastně pro celou rodinu. To samozřejmě není žádné překvapení, protože aplikace ze staršího prostředí musí být přeložitelná i v novějším prostředí. Ani jedna strana nechce zahazovat práci a peníze v podobě odladěného kódu, který může dále sloužit.

Během testů se ukázalo, že implementované optimalizace nestačí na ostré nasazení. Největší vliv na výkon má množství vzorků, které se vyhledává. Následuje úspěch detekce, protože v případě nálezu se posuneme o mnohem více než jediný bajt, který nás čeká při neúspěchu. Detekci překladače lze výrazně zrychlit vyřazováním vzorků, které je popsáno již dříve. Vyhledávání funkcí statických knihoven to samozřejmě nezrychlí. Zde by se dalo využít sofistikované řešení, kdy by se všechny vzorky překladače setřídily do stromu a tím došlo k překrytí vzorků, které stejně začínají. Ale i zde by výraznému zvyšování výkonu bránilo tolerování neshod. Pro testovací učely je výkon dostatečný a je i zbytečné ztrácet čas implementací špičkového vyhledávacího algoritmu pro firmu jako je AVG Technologies, která se živí vyhledáváním škodlivého softwaru.

Kapitola 7

Závěr

Diplomová práce popisuje princip detekce binárního kódu statických knihoven ve spustitelném souboru typu PE. Vyhledávací algoritmus využívá získaných vzorků z knihoven překladačů. Vyhledávání toleruje omezený počet neshodných bajtů tak, aby byly pokryty změny, které nastanou během sestavení aplikace se statickými knihovnami. Diplomová práce dále popisuje detekci překladače a rozpoznávání knihovních funkcí. Součástí je popis sběru vzorků a funkční implementace.

Přínosem diplomové práce je navržení a ověření implementace detekce kódu statických knihoven včetně detekce překladače. Práce podrodobně popisuje kde a jakým způsobem lze získat vzorky funkcí. Popisuje problémy, které mohou nastat s kvalitou vzorků a s tolerancí neshod. Funkčnost celku je ověřena na třech zástupcích překladačů.

Testy prokázaly, že aplikace je schopna rozpoznat desítky procent spustitelného kódu překladačů, které přímo zná. Příjemným bonusem jsou nálezy v prostředích pro které aplikace přímo vzorky nemá, ale pocházejí ze stejné rodiny např. je zřejmá velká příbuznost mezi prostředími firmy Borland.

Za nedostatek diplomové práce může být považována implementace detekční aplikace, která trpí častým předěláváním částí kódů při vracení se ze slepých uliček. Navíc je v případě detekce a vyhledávání vzorků dost pomalá. Hlavním faktorem, který ovlivňuje rychlost, je počet vzorků.

Tato diplomová práce slouží jako úvod do detekce spustitelného kódu statických knihoven, proto je zde mnoho směrů, kam se může práce dále ubírat. Úpravy by si jistě zasloužil formát ukládání vzorků. Zcela jistě není třeba, aby se ukládaly celé vzorky, ale jen jejich část a některé atributy. Můžeme vzorky řadit podle abecedy, podle délky či z nich vytvořit strom. Všechny tyto úpravy zkvalitní množinu vzorků a zvýší efektivitu práce po všech stránkách. Množinu vzorků můžeme rozšířit o další překladače a můžeme také zkusit provázat vzorky různých verzí stejných překladačů. Detekci překladače můžeme zrychlit automatickýcm vybráním typických vzorků překladače.

Dodatek A

Struktury ve spustitelných souborech typu PE

A.1 MZ DOS hlavička

```
typedef struct _IMAGE_DOS_HEADER
{
       WORD e_magic;
       WORD e_cblp;
       WORD e_cp;
       WORD e_crlc;
       WORD e_cparhdr;
       WORD e_minalloc;
       WORD e_maxalloc;
       WORD e_ss;
       WORD e_sp;
       WORD e_csum;
       WORD e_ip;
       WORD e_cs;
       WORD e_lfarlc;
       WORD e_ovno;
       WORD e_res[4];
       WORD e_oemid;
       WORD e_oeminfo;
       WORD e_res2[10];
       LONG e_lfanew;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

A.2 PE hlavička

```
typedef struct _IMAGE_NT_HEADERS
{
       DWORD Signature;
       IMAGE_FILE_HEADER FileHeader;
       IMAGE_OPTIONAL_HEADER OptionalHeader;
```

```
} IMAGE_NT_HEADERS,*PIMAGE_NT_HEADERS;


typedef struct _IMAGE_FILE_HEADER
{
        WORD Machine;
        WORD NumberOfSections;
        DWORD TimeDateStamp;
        DWORD PointerToSymbolTable;
        DWORD NumberOfSymbols;
        WORD SizeOfOptionalHeader;
        WORD Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;


typedef struct _IMAGE_OPTIONAL_HEADER
{
        WORD Magic;
        BYTE MajorLinkerVersion;
        BYTE MinorLinkerVersion;
        DWORD SizeOfCode;
        DWORD SizeOfInitializedData;
        DWORD SizeOfUninitializedData;
        DWORD AddressOfEntryPoint;
        DWORD BaseOfCode;
        DWORD BaseOfData;
        DWORD ImageBase;
        DWORD SectionAlignment;
        DWORD FileAlignment;
        WORD MajorOperatingSystemVersion;
        WORD MinorOperatingSystemVersion;
        WORD MajorImageVersion;
        WORD MinorImageVersion;
        WORD MajorSubsystemVersion;
        WORD MinorSubsystemVersion;
        DWORD Reserved1;
        DWORD SizeOfImage;
        DWORD SizeOfHeaders;
        DWORD CheckSum;
        WORD Subsystem;
        WORD DllCharacteristics;
        DWORD SizeOfStackReserve;
        DWORD SizeOfStackCommit;
        DWORD SizeOfHeapReserve;
        DWORD SizeOfHeapCommit;
        DWORD LoaderFlags;
        DWORD NumberOfRvaAndSizes;
```

```
        IMAGE_DATA_DIRECTORY DataDirectory[IMAGE
                                _NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER,*PIMAGE_OPTIONAL_HEADER;
```

Struktura _IMAGE_OPTIONAL_HEADER obsahuje šestnáctiprvkové pole struktur _IMAGE_DATA_DIRETORY. Tyto struktury popisují umístění až šestnácti dalších struktur různého určení.

```
typedef struct _IMAGE_DATA_DIRECTORY
{
        DWORD VirtualAddress;
        DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

0  Export table address and size
1  Import table address and size
2  Resource table address and size
3  Exception table address and size
4  Certificate table address and size
5  Base relocation table address and size
6  Debugging information starting address and size
7  Architecture-specific data address and size
8  Global pointer register relative virtual address
9  Thread local storage (TLS) table address and size
10 Load configuration table address and size
11 Bound import table address and size
12 Import address table address and size
13 Delay import descriptor address and size
14 The CLR header address and size
15 Reserved
```

A.3 Tabulka exportů

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
       DWORD Characteristics;
       DWORD TimeDateStamp;
       WORD MajorVersion;
       WORD MinorVersion;
       DWORD Name;
       DWORD Base;
       DWORD NumberOfFunctions;
       DWORD NumberOfNames;
       DWORD AddressOfFunctions;
       DWORD AddressOfNames;
       DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

A.4 Tabulka sekce

```
typedef struct _IMAGE_SECTION_HEADER
{
        BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
                  DWORD PhysicalAddress;
                  DWORD VirtualSize;
        } Misc;
        DWORD VirtualAddress;
        DWORD SizeOfRawData;
        DWORD PointerToRawData;
        DWORD PointerToRelocations;
        DWORD PointerToLinenumbers;
        WORD NumberOfRelocations;
        WORD NumberOfLinenumbers;
        DWORD Characteristics;
} IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

Dodatek B

Popis ovládání aplikace PEDetect

Aplikace PEDetect se ovládá pomocí těchto parametrů:

```
PEDetect soubor [-debug] [-extradebug] [-dumb]
```

Provede detekci nad souborem. Přepínač -debug a -extradebug zapne ladící výpisy. Přepínač -dumb slouží pro hledání v celém souboru bez ohledu na struktury.

```
PEDetect -i vstupni_soubor vystupni_soubor jmeno_prekladace verze
```

```
PEDetect -i rtl60.bpl rtl60.bpl.pat "C++ Builder" 6
```

Získá vzorky ze vstupního souboru a uloží je nakonec výstupního souboru do adresáře Patterns.

```
PEDetect -help
```

Vypíše nápovědu.

Dodatek C

Popis struktury logovacích souborů

Na DVD v adresáři test můžete nalézt výstupy z hledání s parametrem -extradebug. Tyto soubory obsahují mnoho informací o nálezech, detekci a mnoho dalších statistik pro každý soubor. Tato příloha popisuje jejich formát.

Každý log začíná informací o načtených vzorcích jako jméno souboru a statistické informace o délce.

```
Loading patterns...
adortl60.bpl.pat
Average pattern length C++ Builder: 209
Longenst pattern: 2739
Shortest pattern: 19
```

Po načtení všech vzorků je vypsáno statistické rozložení délky vzorků všech překladačů.

```
Number of patterns of same length:  <= 2: 0
Number of patterns of same length:  <= 4: 0
Number of patterns of same length:  <= 8: 0
Number of patterns of same length:  <= 16: 213
Number of patterns of same length:  <= 32: 5728
Number of patterns of same length:  <= 64: 9668
Number of patterns of same length:  <= 128: 10263
Number of patterns of same length:  <= 256: 6448
Number of patterns of same length:  <= 512: 3233
Number of patterns of same length:  <= 1024: 1583
Number of patterns of same length:  <= 2048: 577
Number of patterns of same length:  <= 4096: 213
Number of patterns of same length:  <= 8192: 64
Number of patterns of same length:  <= 16384: 17
Number of patterns of same length:  <= 32786: 1
Number of patterns of same length:  <= 1000000000: 0
```

Následuje detekce překladače.

```
Detecting compiler:
 Scanned section: .text
  Testing C++ Builder 6 Recognized: 0
```

```
Testing Visual Studio 2005 Recognized: 0
Testing MinGW32 3.4.2 Recognized: 30560
MinGW32 detected
```

Vyhledávání vzorků začíná informacemi o sekci. Nálezy jsou popsány adresou, jménem souboru se vzorky, jménem statické knihovny a jménem funkce. Součastí jsou i informace o tolerovaných neshodách konkrétně poměr neshod vůči délce a množství neshod jednotlivých délek. V případě delšího nerozpoznaného prostoru je vypsána jeho velikost a jsou vypsány také vzorky, které se lišily nejméně. O vzorcích jsou výpsány tyto údaje: Pozice v souboru, pořadí vzorku, shodná délka a délka vzorku a jméno vzorku.

```
Searching patterns:
Sector size: 252416 Virtual Address:401000h
Not recognized space: 720B
720 (4012d0) - 736 (4012e0) _libstdc++.pat - concept-inst.o - .text
         $_ZN9__gnu_cxx21_InputIteratorConceptIPcE13__constraintsEv
Missmatched bytes / Pattern size: 7/16 (43%)
Missmatch (1B): 0
Missmatch (2B): 0
Missmatch (3B): 1
Missmatch (4B): 1
1184 (4014a0) - 1312 (401520) _libstdc++.pat - globals_io.o - .text
Missmatched bytes / Pattern size: 15/128 (11%)
Missmatch (1B): 0
Missmatch (2B): 0
Missmatch (3B): 5
Missmatch (4B): 0
Byte: 2304 LongestUnMatch: 544 MatchSize: 594 PaSize: 599 Name: MinGW32
```

Následuje celkové shrnutí výsledků.

```
----------------------------------------------------------------------
Segment size: 252416 Identified segment size: 229568 Ratio: 90.9483%
----------------------------------------------------------------------
```

Je zobrazena statistika nerozpoznaných částí sekce. V tabulce jsou počty nerozpoznaných úseků, kolik tvoří bajtů celkem a jakou tvoří část sekce v procentech. Každý řádek tabulky znamená interval 1 - 2, 3 - 4, 5 - 8, 9 - 16, atd.

```
Space Result:
MAX SIZE  COUNT     SUM     %
2           820     820     0.32486%
4             0       0     0%
8             0       0     0%
16            0       0     0%
32            2      34     0.01346%
64            2      90     0.03565%
128           1      81     0.03208%
256           2     354     0.14024%
512           6    2102     0.83275%
1024          2   16177     6.40886%
```

Další dva řádky informují o možném zlepšení detekce, kdyby se započítaly vzorky, které se lišily nejméně. Pesimistická varianta počítá délku pouze shodné části, optimistická počítá s délkou celého vzorku.

```
Possible Detection - pesimistic: 227     0.0899309%
Possible Detection - optimistic: 272     0.107759%
```

Na konci souboru nalezneme vždy statistická data o tolerovaných neshodách. Missmatch overall sčítá bajty tolerovaných neshod o stejných délkách. Poslední tabulka podává informace o rozložení poměru tolerovaných neshod a celkové délky detekovaného vzorku.

```
Missmatch overall
Missmatch overall (1B): 419
Missmatch overall (2B): 965
Missmatch overall (3B): 2841
Missmatch overall (4B): 2531

Ratio between missmatch and length of match
0%: 105
1%: 28
2%: 36
3%: 26
4%: 19
5%: 29
6%: 50
...
97%: 0
98%: 0
99%: 0
```

Dodatek D

DVD

D.1 Obsah DVD

app/PEDetect	Program PEDetect
app/PEDetect/patterns	Vzorky standardní knihovny MinGW32
app/PEDetect/src	Zdrojové kódy aplikace PEDetect(C++)
doc/	Diplomová práce ve formátu PDF.
doc/scr/	Zdrojové kódy diplomové práce (LaTEX)
patterns/All/	Aplikace PEDetect se vzorky všech překladačů
patterns/C++ Builder 6/	Aplikace PEDetect se vzorky C++ Builder 6
patterns/MinGW32 3.4.2/	Aplikace PEDetect se vzorky MinGW32
patterns/Visual Studio 2005/	Aplikace PEDetect se vzorky Visual Studio 2005
test/	Testovací soubory a výstupní logy

D.2 DVD

Literatura

[1] Intel: *IA-32 Intel Architecture Software Developers Manual*. Intel, 2004.

[2] Mareš, P.: *Analyzátor typu spustitelných souborů - bez komprese*. 2006.

[3] Microsoft: C Run-Time Libraries.
`http://msdn.microsoft.com/en-us/library/abx4dbyh(VS.80).aspx`.

[4] Microsoft: Microsoft Portable Executable and Common Object File Format Specification.
`http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[5] Wikipedia: Linker. `http://en.wikipedia.org/wiki/Linker/`.