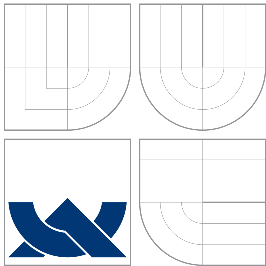
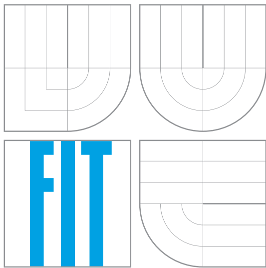


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZPĚTNÝ PŘEKLAD Z VYBRANÝCH FORMÁTŮ SPUSTITELNÝCH SOUBORŮ

DECOMPILATION FROM SELECTED OBJECT FILE FORMATS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
MICHAL BANDZI

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. PETER MATULA

BRNO 2016

Zadání bakalářské práce/17988/2015/xbandz00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Bandzi Michal**
Obor: Informační technologie
Téma: **Zpětný překlad z vybraných formátů spustitelných souborů**
Decompilation from Selected Object File Formats
Kategorie: Překladače

Pokyny:

1. Studujte problematiku zpětného inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace. Dále se seznamte se zpětným překladačem společnosti AVG Technologies.
2. Seznamte se s vybranými formáty spustitelných souborů (např. Mach-O, Intel HEX, SREC a další).
3. Navrhněte metody, které umožní zpětný překlad kódů uložených v těchto formátech do vyšší formy reprezentace.
4. Po konzultacích s vedoucím metody navržené v předchozím bodě implementujte.
5. Vytvořené řešení důkladně otestujte sadou minimálně dvaceti testů, zohledněte překladače a jejich nastavení. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- Křoustek, J.: Rekonfigurovatelná analýza strojového kódu, disertační práce. Brno, VUT FIT, 2015.
- Matula, P.: Nástroje pro konverzi formátů spustitelných souborů, bakalářská práce. Brno, FIT VUT, 2011.
- Popis platformy LLVM [online]. 2015 [cit. 2015-09-23]. Dostupný z WWW: <www.llvm.org>.
- Levine, J. R.: Linkers and Loaders, Morgan Kaufmann Publishers, Inc., 1999.
- Další dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- První tři body zadání, částečně bod čtvrtý.

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

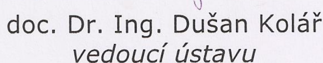
Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Matula Peter, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

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

L.S.



doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Objektové súbory obsahujú strojový kód, ktorý môže byť vykonaný procesorom. Každý objektový súbor má formát, ktorý popisuje jeho štruktúru. Pre vykonanie spätného prekladu je nutné súbor spracovať a previesť dáta do vnútornej reprezentácie spätného prekladača. Táto práca pojednáva o návrhu a implementácii nových modulov pre podporu spracovania formátov, ktoré budú súčasťou Rekonfigurovateľného spätného prekladača. Cieľom práce je pridanie podpory pre formáty Intel HEX a Mach-O a nová implementácia už podporovaného formátu Portable Executable. Implementácia modulov pre Intel HEX a Mach-O bola úspešná a je možné použiť ich pre spätný preklad. Spracovanie formátu PE nedosahuje dostatočnej kvality kvôli chybám knižnice LLVM, na ktorej je implementácia založená.

Abstract

Object files contain machine code that can be executed by processor unit. Structure of an object file is defined by its file format. In order to decompile an object file, it is necessary to process and convert file data to internal representation of decompiler. This thesis discusses design and implementation of new modules for file format processing that will be part of the Retargetable Decompiler project. The goal of this work is to add support for Intel HEX and Mach-O file formats and new implementation of already supported Portable Executable file format. Implementation of modules for file formats Intel HEX and Mach-O was successful and it is possible to use them for reverse compilation. Processing of PE file format is not possible in sufficient quality due to errors in used LLVM library.

Kľúčové slová

objektové súbory, binárne súbory, univerzálne binárne súbory, reverzné inžinierstvo, spätný preklad, spätný prekladač, Intel HEX, Portable Executable, PE, Mach-O, Mach-O Universal Binary

Keywords

object files, binary files, universal binaries, reverse engineering, decompilation, decompiler, Intel HEX, Portable Executable, PE, Mach-O, Mach-O Universal Binary

Citácia

BANDZI, Michal. *Zpětný překlad z vybraných formátů spustitelných souborů.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Matula Peter.

Zpětný překlad z vybraných formátů spustitelných souborů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Petra Matulu. Uviedol som všetky literárne zdroje a publikácie, z ktorých som čerpal.

. . . .
Michal Bandzi
17. mája 2016

Poďakovanie

Ďakujem svojmu vedúcemu Ing. Petrovi Matulovi za odborné vedenie, za poskytnuté rady a za čas, ktorý mi pri tvorbe práce venoval.

© Michal Bandzi, 2016.
Táto práca vznikla ako školské dielo na FIT VUT v Brně. Práca je chránená autorským zákonom a jej využitie bez poskytnutia oprávnenia autorom je nezákonné, s výnimkou zákonne definovaných prípadov.

Obsah

1 Úvod **4**

2 Reverzné inžinierstvo **6**
2.1 Nástroje reverzného inžinierstva 6
2.2 Spätný prekladač 7
2.2.1 Porovnanie vybraných spätných prekladačov 7

3 Rekonfigurovateľný spätný prekladač spoločnosti AVG **10**
3.1 Štruktúra rekonfigurovateľného spätného prekladača 10
3.1.1 Projekt LLVM 10
3.1.2 Predspracovanie 11
3.1.3 Front-end 12
3.1.4 Middle-end 12
3.1.5 Back-end 13

4 Knižnica fileformatl **14**
4.1 Identifikácia formátu 14
4.2 Vnútorná reprezentácia 14
4.2.1 Trieda Section 14
4.2.2 Doplňujúce triedy 14
4.2.3 Interné dátové typy 15
4.3 Detekčné metódy 15
4.4 Pridanie podpory nového formátu 16
4.5 Nástroj fileinfo 16

5 Formáty objektových súborov **17**
5.1 Intel HEX 18
5.1.1 Vytvorenie súboru s formátom Intel HEX 18
5.1.2 Štruktúra formátu Intel HEX 18
5.2 Portable Executable 20
5.2.1 Štruktúra formátu Portable Executable 20
5.3 Mach-O 22
5.3.1 Štruktúra formátu Mach-O 23
5.3.2 Mach-O Universal Binary 25

6 Spätný preklad súborov formátu Intel HEX **26**
6.1 Požiadavky 26
6.2 Návrh 26

6.2.1 Lexikálna analýza . . . 26
6.2.2 Interpretácia dát . . . 27
6.2.3 Rozdelenie vstupných dát na logické sekcie . . . 27
6.2.4 Chýbajúce informácie . . . 27
6.2.5 Serializácia dát . . . 28
6.3 Implementácia . . . 28
6.4 Testovanie . . . 29
6.4.1 Jednotkové testy . . . 29
6.4.2 Regresné testy . . . 29
6.5 Dosiahnuté výsledky . . . 29
6.5.1 Spätný preklad . . . 29
6.5.2 Nástroj fileinfo . . . 30

7 Spätný preklad súborov formátu Portable Executable 35
7.1 Požiadavky . . . 35
7.2 Návrh . . . 35
7.2.1 Načítanie sekcií . . . 35
7.2.2 Načítanie symbolov, exportov a importov . . . 36
7.2.3 Detekčné metódy . . . 36
7.2.4 Rich Header . . . 37
7.3 Implementácia . . . 37
7.4 Testovanie . . . 37
7.5 Dosiahnuté výsledky . . . 37
7.5.1 Chybné načítanie importov . . . 38
7.5.2 Chybné načítanie oneskorených importov . . . 39
7.5.3 Kontrola vstupných súborov . . . 39
7.5.4 Zhrnutie . . . 40

8 Spätný preklad súborov formátu Mach-O 41
8.1 Požiadavky . . . 41
8.2 Návrh . . . 41
8.2.1 Zistenie bitovej šírky a usporiadania bajtov . . . 42
8.2.2 Podpora univerzálnych binárnych súborov . . . 42
8.2.3 Spracovanie príkazov pre načítanie vstupného súboru . . . 42
8.2.4 Návrh detekčných metód . . . 44
8.2.5 Detekcia zašifrovaných binárnych súborov . . . 44
8.3 Implementácia . . . 44
8.4 Testovanie . . . 45
8.4.1 Overenie správnosti načítania importov . . . 45
8.5 Dosiahnuté výsledky . . . 45
8.5.1 Identifikácia zašifrovaných súborov . . . 45
8.5.2 Výsledky spätného prekladu . . . 45
8.5.3 Univerzálne súbory . . . 46

9 Záver 49
9.1 Budúci vývoj . . . 49

Literatúra 51

Prílohy **53**
Zoznam príloh 54

A Obsah CD **55**

B Pôvodné zdrojové kódy **56**

Kapitola 1

Úvod

Spustiteľný súbor je súbor obsahujúci inštrukcie, ktoré budú po jeho spustení vykonané počítačom. Môže sa jednať buď o inštrukcie cieľovej architektúry uložené v binárnej forme, doplnené dodatočnými informáciami, alebo inštrukcie interpretovaného jazyka. Každý takýto súbor má určitý formát, ktorý popisuje jeho vnútornú štruktúru.

Binárne spustiteľné súbory vznikajú prekladom (anglicky compilation) zdrojového súboru, vytvoreného užívateľom-programátorom, pomocou prekladača (anglicky compiler) zvoleného programovacieho jazyka. Prekladač analyzuje vstupný zdrojový súbor a prevedie ho na inštrukcie cieľovej architektúry, pribalí k nim statické dáta a ďalšie potrebné informácie a uloží ich do výstupného súboru, ktorý je následne možné spustiť. Opačný proces, prevedenie spustiteľného súboru do vyššieho programovacieho jazyka, sa nazýva spätný preklad (anglicky decompilation). Nástroj, ktorý túto činnosť vykonáva, sa nazýva spätný alebo reverzný prekladač (anglicky decompiler). Jedná sa o významný nástroj softvérového reverzného inžinierstva.

Reverzné inžinierstvo je proces získavania nových vedomostí alebo dokumentácie o určitom objekte, ktorý bol vytvorený ľudskou činnosťou [10]. V prípade softvérového reverzného inžinierstva je takýmto objektom softvér alebo jeho časť. Historicky odbor vznikol v podstate spolu so softvérom samotným. Prvé spätné prekladače sa začali objavovať približne desať rokov po klasických prekladačoch [7]. Motívy pre spätný preklad a všeobecne reverzné inžinierstvo môžu byť rôzne, od zvyšovania bezpečnosti (hľadanie chýb alebo škodlivého softvéru) až po urýchlenie vývoja softvéru.

Počas historického vývoja informačných technológií vzniklo niekoľko desiatok rôznych formátov objektových súborov pre rôzne architektúry, operačné systémy a účely. Medzi najznámejšie a dnes najpoužívanejšie patrí formát ELF, typický pre operačné systémy rodiny Unix, formáty COFF a PE využívané operačnými systémami Windows, formát Mach-O, ktorý nájdeme v operačných systémoch rodiny Mac OS a mnohé ďalšie. Ak chceme spustiteľný súbor preložiť späť do vyššieho programovacieho jazyka, potrebujeme získať zdrojový kód a ďalšie potrebné informácie. Aby sa nám to podarilo, musíme poznať vnútornú štruktúru týchto súborov a byť schopný tieto informácie dodať vo forme, ktorej rozumie spätný prekladač.

Táto práca si dáva za cieľ rozšíriť *Rekonfigurovateľný spätný prekladač* (anglicky *Retargetable Decompiler*) vyvíjaný spoločnosťou *AVG Technologies CZ, s.r.o.* o podporu nových formátov nielen spustiteľných objektových súborov. Rekonfigurovateľný spätný prekladač je možné jednoducho rozšíriť naprogramovaním dodatočných modulov, ktoré majú za úkol analyzovať vstupný súbor a poskytnúť ďalším častiam spätného prekladača všetky potrebné informácie pre vykonanie prekladu.

Práca je členená do niekoľkých kapitol. V kapitole 2 sa nachádzajú všeobecné informácie o reverznom inžinierstve a spätnom preklade. V kapitole 3 je detailnejšie popísaný Rekonfigurovateľný spätný prekladač spoločnosti AVG. Kapitola 4 sa podrobne zaoberá knižnicou *fileformatl*, ktorá bude rozšírená v rámci tejto práce. Kapitola 5 poskytuje pohľad na formáty jednotlivých objektových súborov. Kapitoly 6, 7 a 8 postupne riešia návrh, implementáciu, testovanie a zhodnotenie nových modulov pre podporu formátov Intel HEX, Portable Executable a Mach-O. Kapitola 9 obsahuje zhrnutie tejto práce.

Kapitola 2

Reverzné inžinierstvo

Podľa [10], z ktorej nasledujúca kapitola vychádza, je reverzné inžinierstvo proces získavania nových vedomostí alebo dokumentácie o určitom objekte, ktorý bol vytvorený človekom. V prípade softvérového reverzného inžinierstva je takýmto objektom softvér alebo jeho časť. Dôvody reverzného inžinierstva môžu byť rôzne, podľa [10] ich môžeme rozdeliť na dve hlavné skupiny: bezpečnosť a ďalší vývoj softvéru.

Z hľadiska bezpečnosti sa jedná predovšetkým o vyhľadávanie škodlivého alebo nebezpečného softvéru, predovšetkým počítačových vírusov a červov. Tento prístup sa využíva napr. v antivírusových programoch. Reverzné inžinierstvo je však možné zneužiť aj na opačnú činnosť – prehľadávanie softvéru a hľadanie bezpečnostných dier, na ktoré bude možné škodlivým softvérom zaútočiť. Ďalej môže byť reverzné inžinierstvo použité na odhalenie fungovania niektorých kryptografických algoritmov a algoritmov pre zabezpečenie digitálneho obsahu, predovšetkým tých, kde je kľúčové utajenie použitého algoritmu. Tretie využitie reverzného inžinierstva je kontrola proprietárneho softvéru, ktorého zdrojové kódy nie sú normálne dostupné užívateľom.

Z hľadiska vývoja softvéru môže byť reverzné inžinierstvo použité pre zaistenie interoperability nedostatočne dokumentovaného softwéru, nahliadnutie do konkurenčného projektu, prípadne pre kontrolu kvality a robustnosti softvéru, ktorého zdrojové kódy nie sú verejné.

2.1 Nástroje reverzného inžinierstva

Podľa [10] môžeme pri reverznom softvérovom inžinierstve použiť predovšetkým štyri hlavné druhy nástrojov:

- **Nástroje pre monitorovanie systému** – tieto nástroje zaznamenávajú informácie súvisiace s používaním operačného systému spusteným programom. Vytvárajú tak obraz o využití siete, prístupe k súborom, registrom a službám operačného systému.

- **Disassembler** – relatívne jednoduchý nástroj, ktorý prevedie binárne inštrukcie programu do zápisu formou textových operačných inštrukcií danej architektúry. Veľká väčšina týchto programov je schopná pracovať len s jednou architektúrou, existuje však aj málo disassemblerov, ktoré sú schopné zamerať sa na viacero architektúr.

- **Ladiace nástroje** – aj keď ich primárnym účelom nie je reverzné inžinierstvo, je možné ich na tieto účely použiť. Užívatelia môžu program krokovať a sledovať tak priebeh jeho činnosti inštrukciu po inštrukcii.

- **Spätný prekladač** – nástroj, ktorý je schopný previesť vstupné binárne súbory do programovacieho jazyka vyššej úrovne.

2.2 Spätný prekladač

Podľa [7], z ktorej táto podkapitola vychádza, je spätný prekladač program, ktorý číta program napísaný v strojovom kóde – zdrojovom jazyku – a prekladá ho na ekvivalentný program napísaný v jazyku vyššej úrovne. Vnútorná štruktúra spätného prekladača je veľmi podobná štruktúre klasických prekladačov. [7] hovorí o nasledovných fázach spätného prekladu:

- **Syntaktická analýza** – analyzátor prevedie bajty zdrojového programu na gramatické vety daného strojového jazyka. Najväčší problém tejto fázy je rozhodovanie, čo sú dáta a čo inštrukcie, táto fáza sa tak stáva závislá na cieľovej architektúre.
- **Sémantická analýza** – táto fáza vyhľadáva sémantický význam v skupinách inštrukcií. Fáza prebieha pomocou vyhľadávania typických idiómov a je rovnako závislá na architektúre.
- **Generovanie prechodného kódu** – je nutné pre ďalšiu analýzu. Tento kód by nemal byť náročný na jeho generáciu a zároveň by mal vhodne reprezentovať cieľovú architektúru.
- **Generovanie grafov toku riadenia programu** – je potrebné pre vytvorenie obrazu o vyšších riadiacich prvkoch programu a prípadnú elimináciu medziskokov.
- **Analýza toku dát** – v tejto fáze dochádza k eliminácii použitia dočasných registrov a príznakov, ktoré v jazykoch vyššej úrovne nie sú dostupné.
- **Analýza toku programu** – vytvára na základe grafov toku riadiace štruktúry vyššej úrovne (if-else vetvenie, cykly).
- **Generovanie kódu** – finálna fáza spätného prekladu, vygenerovanie kódu cieľového jazyka.

Podľa [7] môžeme tieto fázy rozdeliť do troch skupín:

- **Front-end** – tieto fázy sú závislé na vstupnej architektúre. Patrí sem syntaktická a sémantická analýza, generovanie prechodného kódu a generovanie grafu toku riadenia programu.
- **Univerzálny spätný prekladač** – fázy tejto skupinu nie sú závislé ani na architektúre vstupného strojového kódu ani na cieľovom jazyku. Patrí sem analýza toku dát a analýza toku riadenia programu.
- **Back-end** – táto fáza je závislá na cieľovom jazyku. Jedná sa o generovanie výsledného kódu daného jazka.

2.2.1 Porovnanie vybraných spätných prekladačov

Táto podkapitola sa zameriava na porovnanie najznámejších a najpoužívanejších spätných prekladačov predovšetkým z hľadiska podpory rôznych objektových formátov, architektúr a cieľových jazykov vyššej úrovne.

Názov	Podporované architektúry	Podporované formáty
Hex-Rays	x86, x64, ARM32, ARM64	PE, ELF a iné v rámci IDA
REC Studio 4	x86, x64, MIPS, PPC, mc68k	COFF, ELF, PE, Mach-O
SmartDec	x86, x64	PE, ELF a iné v rámci IDA
C4Decompiler	Intel 64, x86	PE
Hopper	x86, x64, ARM32, ARM64	ELF, PE, Mach-O
RetDec	x86, ARM32, MIPS32, PIC32, PPC32	ELF, PE, COFF

Tabuľka 2.1: Porovnanie najpopulárnejších spätných prekladačov.

Hex-Rays

Komerčný spätný prekladač Hex-Rays disponuje najväčšou podporou rôznych objektových formátov – zvláda všetky bežne vyskytujúce sa formáty ako PE, COFF, ELF, Mach-O a takmer 40 ďalších v rámci disassembleru IDA. Značnou nevýhodou je podpora architektúr limitovaná na architektúry x86, x64 a ARM. Výstupom prekladu je pseudo-kód založený na jazyku C [11].

REC Studio 4

Bezplatný closed-source spätný prekladač podporujúci architektúry x86, x64, MIPS, PowerPC a mc68k. Podporuje formáty PE, COFF, ELF a Mach-O. Podpora architektúry ARM je vo vývoji. Výstupom prekladu je pseudo-kód založený na jazyku C [5].

SmartDec

Spätný prekladač podporujúci architektúry x86 a x64. Samostatná verzia podporuje formáty ELF a PE, prekladač je však možné použiť spoločne s programom IDA, ktorý rozširuje podporu o desiatky ďalších formátov. Výstupom prekladu je kód v jazyku C s významnou podporou štruktúr jazyka C++ [20]. SmartDec bol neskôr rozšírený o podporu formátu Mach-O a architektúry ARM v rámci odvodeného projektu Snowman [21].

C4Decompiler

Jedná sa o komerčný spätný prekladač zameraný na operačné systémy rodiny Windows. Zaujímavosťou je podpora architektúry Intel Itanium (Intel 64). Jeho podpora objektových formátov je však obmedzená len na formát Windows PE, podpora formátu ELF je vo vývoji. Výstupom je kód v jazyku C [6].

Hopper

Komerčný dissasembler a spätný prekladač zameraný predovšetkým na operačné systémy Unix a OS X. Zvláda základné formáty binárnych súborov ELF, PE a Mach-O. Binárny kód prekladá do jazyka C s čiastočnou podporou pre jazyk Objective-C. Podporované architektúry sú x86, x64 a ARM [8].

Záver

Väčšina prekladačov sa obmedzuje na základné objektové formáty kľúčové pre jednotlivé operačné systémy. Jedná sa o PE (Windows), COFF, ELF (Unix) a Mach-O (OS X, iOS).

Rekonfigurovateľný spätný prekladač spoločnosti AVG, ktorý je podrobnejšie popísaný v kapitole 3, momentálne podporuje formáty COFF, PE a ELF. Zavedenie podpory pre formát Mach-O je kľúčové vzhľadom na relatívne vysoký podiel OS X a iOS na trhu operačných systémov [19]. Po pridaní podpory pre formáty Mach-O a Intel HEX sa rekonfigurovateľný spätný prekladač zaradí na prvé miesto v počte podporovaných objektových formátov bez závislosti od softvéru tretích strán, ako je to napr. v prípade programu Hex-Rays.

Kapitola 3

Rekonfigurovateľný spätný prekladač spoločnosti AVG

Rekonfigurovateľný spätný prekladač je nástroj vyvíjaný spoločnosťou *AVG Technologies CZ, s.r.o.* v spolupráci s *Fakultou informačních technologií VUT v Brne*. Podľa [4] sa jedná o spätný prekladač neviazaný ku žiadnej konkrétnej architektúre, operačnému systému alebo formátu binárnych súborov. Prekladač v dobe písania práce podporuje architektúry Intel x86, ARM + Thumb, MIPS, PIC32, a PowerPC (32-bit verzie), vstupné formáty ELF, PE a COFF a je schopný prekladať do jazyka C alebo upravenej formy jazyka Python. Nástroj je vývíjaný v jazyku C++.

3.1 Štruktúra rekonfigurovateľného spätného prekladača

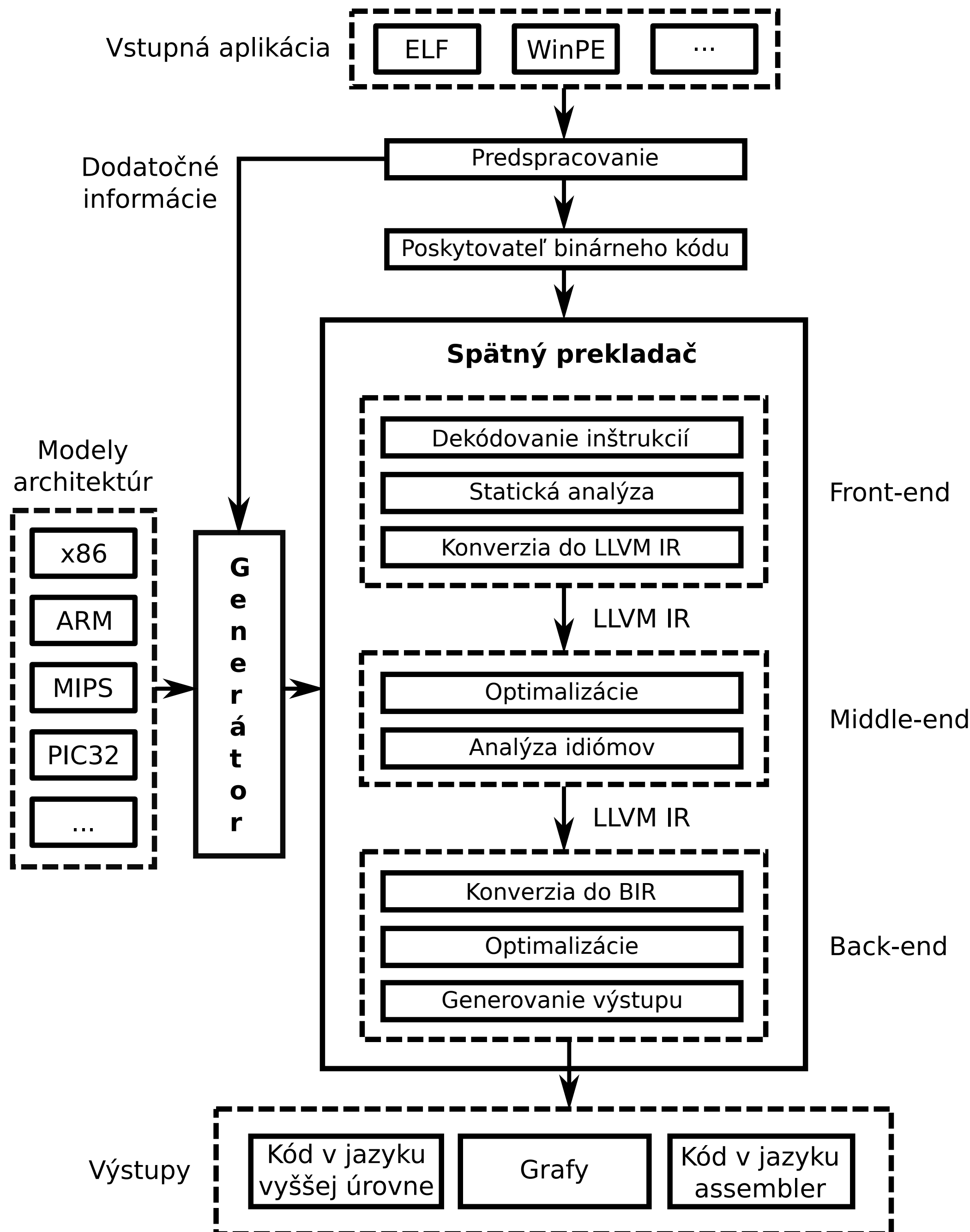
Táto podkapitola bola spracovaná na základe [13] a [22].

Štruktúra rekonfigurovateľného spätného prekladača je podobná štruktúre naznačenej v podkapitole 2.2. Nájdeme tu tri základné časti: front-end, middle-end a back-end. Vzhľadom nato, že zdrojový kód môže byť uložený vo forme rôznych objektových formátov, je nutné vstupný súbor pred samotným spätným prekladom najskôr predspracovať. Viaceré komponenty spätného prekladača spoločnosti AVG využívajú technológie LLVM.

3.1.1 Projekt LLVM

Projekt LLVM je kolekcia modulárnych a znovu-použiteľných technológií určená pre použitie v prekladačoch a ďalších nástrojoch [16].

Reprezentácia kódu LLVM IR (LLVM intermediate representation) poskytuje typovú bezpečnosť, operácie nízkej úrovne, flexibilitu a možnosť reprezentovať jazyky vyššej úrovne. LLVM kód môže byť použitý v troch rôznych formách a to ako reprezentácia v pamäti v rámci prekladača, bitový kód pre uloženie na disk a ako textová forma jazyka assembler určená pre interakciu s ľuďmi. Všetky tri reprezentácie sú si ekvivalentné. Projekt tak poskytuje efektívne prostriedky pre transformácie a analýzy vykonávané prekladačom a zároveň možnosť kód prirodzene ladiť a zobrazovať. Cieľom LLVM IR je poskytnúť odľahčenú, typovanú, rozšíriteľnú a univerzálnu reprezentáciu nízkej úrovne. Vzhľadom na prítomnosť typovej informácie je možné aplikovať veľké množstvo rozličných buď vstavaných alebo vlastných optimalizácií [15].



Obr. 3.1: Štruktúra rekonfigurovateľného spätného prekladača.

3.1.2 Predspracovanie

Predspracovanie (anglicky preprocessing) má za úlohu analyzovať vstupný zdrojový súbor a získať všetky potrebné dáta. Okrem kódu sa vo vstupnom súbore nachádzajú aj informácie o cieľovej architektúre, veľkosti kódu, bitovej šírke, prípadne o prekladači, o použitom jazyku a mnohé iné. Dáta sú v zdrojovom súbore v závislosti od jeho typu často rozdelené do viacerých sekcií a segmentov. Úlohou predspracovania je extrahovať tieto dáta a jednotlivé sekcie a segmenty previesť do vnútornej objektovej reprezentácie spätného prekladača.

Ďalším problémom, ktorý rieši predspracovanie, je prípadná kompresia a ochrana binárnych súborov, dáta je pred ďalším spracovaním nutné dekomprimovať. Ak spustiteľný súbor obsahuje informácie pre ladiace nástroje, tieto sú extrahované a pripadne použité pri samotnom spätnom preklade.

Základný prvok predspracovania tvorí knižnica *fileformatl*, ktorej úlohou je detekcia formátu a prevod vstupného súboru do vnútornej reprezentácie. Moduly pre prevod nových formátov, ktoré budú vytvorené v rámci tejto práce, budú umiestnené práve v tejto časti spätného prekladača. Knižnica je bližšie popísaná v kapitole 4.

3.1.3 Front-end

Úlohou tejto časti spätného prekladača je preložiť strojový kód závislý na vstupnej architektúre do vnútornej, od architektúry nezávislej, reprezentácie LLVM IR. V tejto časti ďalej dochádza k niekoľkým analýzam statického kódu ako napríklad oddelenie kódu od dát, dekódovanie inštrukcií, analýza toku riadenia programu a toku dát, prípadne rekonštrukcia niektorých konštrukcií vyššej úrovne.

Ladiace a symbolické informácie je možné použiť pre získanie informácií o moduloch, ktoré boli použité pri vzniku vstupného súboru, funkciách a lokálnych či globálnych premenných, prípadne o riadkovaní. Rozpoznávanie staticky linkovaných funkcií umožňuje tieto z procesu spätného prekladu vynechať. Pre užívateľa spätného prekladača väčšinou nie sú zaujímavé, keďže ich funkcionalita je známa. Ďalšími výhodami sú zvýšená prehľadnosť výsledného kódu a väčšia rýchlosť spätného prekladu. Takto upravený kód je pripravený pre dekódovanie inštrukcií, ktoré má za úlohu previesť inštrukcie strojového kódu do reprezentácie LLVM IR.

Analýza toku riadenia programu má za úlohu rozdelenie kódu do základných blokov a vytvorenie grafov toku riadenia programu, ktoré sú neskôr použité v ďalších fázach spätného prekladu. Ďalej sa analýza pokúsi vyhľadať a rozpoznať jednotlivé funkcie. Analýza toku dát rekonštruuje dátové typy, argumenty a návratové typy funkcií. Front-end ešte obsahuje niekoľko ďalších menej dôležitých analýz ako napr. analýza dátovej sekcie a analýza zásobníku.

Posledným krokom je generovanie LLVM IR v textovej forme. Najprv sa generujú deklarácie globálnych premenných a linkovaných funkcií, následne sa generuje IR kód rozdelený do funkcií, ktoré boli rozpoznané. Ako posledné sa odošlú pomocné dáta (počet a reálne mená funkcií a premenných) pre ďalšie časti spätného prekladača. Ďalším možným výstupom môže byť kód jazyka nízkej úrovne. Narozdiel od bežného disassembleru je možné výstup obohatiť o niektoré pomocné dáta spomenuté vyššie.

3.1.4 Middle-end

Táto časť je zodpovedná za optimalizáciu LLVM IR reprezentácie získanej ako výstup predošlej fázy. Motivácia pre tento krok je zjednodušenie kódu odstránením prebytočných inštrukcií a vytvorenie čo najvhodnejšieho kódu pre poslednú časť spätného prekladača. Využívaný je predovšetkým nástroj *opt*, ktorý je súčasťou LLVM frameworku. Nástroj *opt* obsahuje veľké množstvo vstavaných optimalizácií a zároveň umožňuje spúšťať vlastné optimalizácie.

Vstavané optimalizácie boli pôvodne navrhnuté pre optimalizáciu kódu pri klasickom preklade. Vzhľadom na rozdielne ciele spätného prekladu je nutné niektoré optimalizácie vynechať, napr. rozloženie štruktúr na obyčajné premenné by viedlo k zhoršeniu kvality výsledného kódu. Medzi najdôležitejšie vstavané optimalizácie pre spätný preklad patrí

eliminácia mŕtveho kódu, vyhľadávanie alternatívnych prístupov k premenným (tzv. aliasy), spojenie viacerých inštrukcií do menej jednoduchších inštrukcií, zmena poradia inštrukcií komutatívnych výrazov a iné.

Ďalšou dôležitou súčasťou je vyhľadávanie inštrukčných idiómov, sekvencie niekoľkých inštrukcií, reprezentujúce malú konštrukciu jazyka vyššej úrovne. Tieto idiómy vznikajú v rámci optimalizácie kódu pri klasickom preklade, kód sa však stáva ťažšie čitateľný. Typickým príkladom takéhoto idiómu je nahradenie inštrukcie pre uloženie nuly do registru inštrukciou logického exkluzívneho súčtu. Analýza inštrukčných idiómov sa snaží vrátiť tieto inštrukcie do pôvodného stavu. Nejedná sa o vstavanú optimalizáciu.

3.1.5 Back-end

Táto časť spätného prekladača konvertuje optimalizovaný prechodný kód na cieľový jazyk vyššej úrovne. Konverzia prebieha vo viacerých krokoch. V prvom kroku sa vstup v reprezentácii LLVM IR prevedie na reprezentáciu BIR (Back-end intermediate representation). BIR je interná reprezentácia LLVM IR používaná v zadnej časti prekldača a je možné ju udržať v pamäti bez akejkoľvek textovej reprezentácie. BIR ďalej umožňuje modelovať všetky konštrukcie jazyka LLVM IR a vytvárať konštrukcie jazykov vyššej úrovne (if-then-else, while, for atď.), ktoré LLVM IR nepodporuje. Ďalším vstupom sú ladiace informácie, ktoré sa neskôr použijú na premenovanie premenných.

Po získaní vstupov nasleduje dodatočná optimalizácia kódu v reprezentácii BIR. Medzi najdôležitejšie optimalizácie patrí zjednodušenie aritmetických výrazov, odstránenie prebytočných priradení, premenovanie premenných a konverzia numerických konštánt na konštanty symbolické.

Posledným krokom je generovanie kódu v jazyku vyššej úrovne, grafu toku riadenia programu a grafu volania funkcií. Spätný prekladač momentálne podporuje výstup formou dvoch jazykov, jazyka C alebo upraveného jazyka Python.

Kapitola 4

Knižnica fileformatl

Úlohou knižnice *fileformatl* je identifikácia formátu zdrojového súboru a prevod dát do vnútornej reprezentácie spätného prekladača. Moduly vytvorené v rámci tejto práce budú súčasťou práve tejto knižnice.

4.1 Identifikácia formátu

Identifikácia prebieha pomocou mapy signatúr, ktoré identifikujú jednotlivé formáty binárnych súborov. Pre pridanie podpory pre nový formát stačí pridať signatúry daného formátu do mapy signatúr. V prípade, že signatúra nie je úplne unikátna (napr. DOS MZ Executable a Portable Executable), je možné pridať dodatočnú analýzu. Signatúry väčšinou pozozstávajú z dvoch až štyroch bajtov a sú uložené hneď na začiatku súboru. Príkladom môže byť signatúra `0xCAFEBABE`, ktorá identifikuje formát Mach-O Universal Binary.

4.2 Vnútorná reprezentácia

Vnútorná reprezentácia pozostáva z tried a dátových typov reprezentujúcich informácie, ktoré sa bežne objavujú v binárnych súboroch. Základnou triedou vnútornej reprezentácie je trieda `FileFormat`. Táto trieda uchováva všetky ostatné triedy reprezentujúce jednotlivé časti spustiteľných súborov, definuje tiež základné rozhranie pre moduly.

4.2.1 Trieda Section

Najdôležitejšou informáciou potrebnou pre preklad sú samotné inštrukcie, tie sú vo vnútornej reprezentácii uložené formou aspoň jednej sekcie. V prípade, že formát žiadne sekcie neobsahuje, dáta musia byť reprezentované aspoň jednou logickou sekciou. Trieda okrem dát obsahuje ďalšie doplňujúce informácie – virtuálnu adresu, veľkosť v pamäti, veľkosť v súbore, pozíciu sekcie v súbore, typ sekcie, index a iné. Sekcie sú v rámci triedy `FileFormat` uložené v kontajnery typu `std::vector`.

4.2.2 Doplňujúce triedy

V spustiteľných súboroch sa objavujú mnohé ďalšie informácie, ktoré nie sú kritické pre samotný spätný preklad, ale môžu výrazne zlepšiť výsledný kód. Niektoré významné doplňujúce triedy:

- `SymbolTable` – reprezentuje jednu tabuľku symbolov. Vďaka tabuľke symbolov je možné použiť pôvodné mená funkcií alebo premenných tak, ako boli použité v pôvodnom zdrojovom súbore. Je tiež možné vyhnúť sa prekladu funkcií, ktorých význam je všeobecne známy, napr. funkcie štandardných knižníc.
- `ImportTable` – tabuľka importov. Táto tabuľka obsahuje mená a adresy symbolov, ktoré sa nachádzajú v iných moduloch alebo dynamických knižniciach. Obsahuje tiež mená knižníc a modulov, z ktorých tieto symboly pochádzajú.
- `ExportTable` – tabuľka exportov. Táto tabuľka obsahuje mená a adresy symbolov, ktoré sú definované v rámci tohto modulu a dostupné pre použitie v iných moduloch. Bežné predovšetkým pre dynamické knižnice.
- `Segment` – reprezentuje jeden segment vstupného súboru. Segmenty sa objavujú len v niektorých formátoch, napr. Mach-O alebo ELF.

Pridanie nových tried

V prípade, že sa v binárnom súbore nachádza užitočná informácia, ktorá nie je zatiaľ nijak reprezentovaná, je možné pomocou dedičnosti rozšíriť aktuálne triedy alebo pridať úplne novú triedu. Príkladom môže byť rozšírenie triedy `Section`, kde v základnej reprezentácii chýba podpora pre relokačné informácie či zarovnanie (formáty Mach-O, PE).

4.2.3 Interné dátové typy

Vzhľadom na rozdielnu reprezentáciu niektorých kľúčových informácií naprieč rôznymi formátmi je nutné zadefinovať vlastné dátové typy, ktoré bude možné používať nezávisle. Tieto typy sú:

- `Format` – predstavuje formát zdrojového súboru.
- `Architecture` – predstavuje cieľovú architektúru.
- `Endianness` – predstavuje informáciu o bajtovom usporiadaní.

4.3 Detekčné metódy

Úlohou týchto metód je poskytnutie najzákladnejších informácií o vstupnom súbore, ktoré sú nutné pre preklad alebo overenie, či je preklad vôbec možný. Implementácia týchto metód s výnimkou posledných dvoch je povinná. Tieto metódy sú:

- `isObjectFile` – overenie, či sa jedná o linkovateľný súbor.
- `isDll` – overenie, či sa jedná o dynamickú knižnicu.
- `isExecutable` – overenie, či sa jedná o spustiteľný súbor.
- `getMachineCode` – vráti konštantu reprezentujúcu cieľovú architektúru v rovnakej forme, akou je uložená v zdrojovom súbore.
- `getAbiVersion` – vráti verziu nízkoúrovňového rozhrania použitého v zdrojovom súbore (anglicky Application binary interface). Informáciu neobsahujú všetky formáty.

- `getImageBaseAddress` – vráti základnú adresu, na ktorú by mal byť obsah súboru nahraný. Informácia sa objavuje len v niektorých formátoch, napr. PE.
- `getEpAddress` – vráti adresu vstupného bodu programu. Informácia sa objavuje väčšinou len v spustiteľných súboroch, výnimočne v dynamických knižniciach.
- `getEpOffset` – vráti pozíciu vstupného bodu v súbore. Informácia je dostupná, len ak je dostupná adresa vstupného bodu.
- `getTargetArchitecture` – vráti cieľovú architektúru ako dátový typ `Architecture`.
- `getEndianness` – vráti spôsob usporiadania bajtov ako dátový typ `Endianness`.
- `getBytesPerWord` – vráti bajtovú šírku slova.
- `getDeclaredNumberOfSections` – vráti počet sekcií vo vstupnom súbore.
- `getDeclaredNumberOfSegments` – vráti počet segmentov vo vstupnom súbore.
- `getFileFormatName` – vráti reťazec obsahujúci meno daného formátu.
- `getDeclaredFileLength` – vráti udanú veľkosť vstupného súboru. Implementácia je nutná len v prípade nevyhovujúceho algoritmu pôvodnej funkcie.
- `areSectionsValid` – overenie, či sú sekcie načítané správne. Implementácia je nutná len v prípade nevyhovujúceho algoritmu pôvodnej funkcie.

4.4 Pridanie podpory nového formátu

Pre pridanie podpory nového formátu je nutné vytvoriť triedu, ktorá dedí základnú triedu `FileFormat`, implementovať virtuálne detekčné metódy a metódy pre načítanie informácií do vnútornej reprezentácie (sekcie, symboly, importy, exporty a iné). Ďalej je nutné pridať signatúry do mapy signatúr a novú hodnotu do dátového typu `Format`, ktorá bude reprezentovať nový typ formátu v rámci spätného prekladača. Nakoniec treba rozšíriť funkciu `createFileFormat`, ktorá na základe detekcie signatúry vytvorí inštanciu triedy, ktorá reprezentuje daný formát.

4.5 Nástroj fileinfo

Nástroj *fileinfo* stavia na knižnici *fileformatl* a slúži pre výpis informácií o vstupnom súbore. V základnom móde dôjde k výpisu informácií ako názov formátu, bitová šírka, typ súboru, prípadne pozícia v súbore a virtuálna adresa vstupného bodu programu. V rozšírenom móde dôjde k výpisu tabuliek sekcií a segmentov, symbolov, importov a exportov, prípadne mnohých ďalších informácií špecifických pre konkrétny formát alebo typ súboru. Nástroj podporuje výstup vo formáte plain text alebo JSON. V rámci tejto práce bude nutné rozšíriť program *fileinfo* o podporu nových formátov pre účely písania regresných testov.

Kapitola 5

Formáty objektových súborov

Formát objektového súboru popisuje, ako sú informácie nachádzajúce sa v objektových súboroch uložené a štruktúrované. Podľa [14] je možno informácie obsiahnuté v objektových súboroch rozdeliť do nasledujúcich sekcií:

- **Hlavičkové informácie** – poskytujú všeobecné informácie o súbore ako dátum vytvorenia, meno zdrojového súboru alebo veľkosť kódu a rôzne príznaky ako napr. cieľová architektúru alebo bajtové usporiadanie (anglicky endianness).
- **Objektový kód** – binárne inštrukcie a dáta vygenerované prekladačom.
- **Relokačné záznamy** – zoznam miest v objektovom kóde, ktoré musia byť upravené, ak linker zmení adresy objektového kódu.
- **Symboly** – globálne symboly definované v tomto module, symboly z ostatných modulov alebo symboly definované linkerom.
- **Ladiace informácie** – informácie určené pre ladiaci program, ako napr. číslovanie riadkov kódu, lokálne symboly a popis dátových štruktúr.

Nie všetky formáty obsahujú všetky vyššie spomenuté sekcie a niektoré formáty zase definujú svoje vlastné. Podľa [14] je možné rozdeliť objektové súbory na základe ich funkcie do nasledujúcich kategórií:

- **Spustiteľné súbory** – (anglicky executable files) obsahujú objektový kód, väčšinou bez symbolov a relokačných informácií. Objektový kód je buď jeden veľký segment alebo rozdelený do malej množiny segmentov, ktoré reflektujú danú architektúru. Tento typ súboru je možné spustiť ako samostatný program.
- **Linkovateľné súbory** – (anglicky linkable files) popri objektovom kóde obsahujú veľké množstvo informácii o symboloch a relokačných informácii potrebných pre proces linkovania. Objektový kód je rozdelený do veľkého množstva logických segmentov. Tieto súbory sú vstupom pre linker, ktorý vytvorí výsledný spustiteľný súbor.
- **Súbory ukladané do pamäte** – (anglicky loadable files) môžu byť uložené do pamäte spolu so spúšťaným programom, napr. knižnice. Tento typ súboru nie je možné spustiť samostatne.

Niektoré formáty je ďalej možné previesť na iný typ formátu, ktorý väčšinou slúži inému účelu, prípadne kóduje informácie rozdielnym spôsobom. Príkladom takýchto formátov je *Intel HEX* alebo *Motorola SREC*, ktoré ukladajú výsledné informácie formou ASCII textu. Ich výhodou je napr. možnosť ich použitia v spolupráci s nástrojmi, ktoré nepodporujú binárne data (textový edtior, posielanie pomocou SMTP).

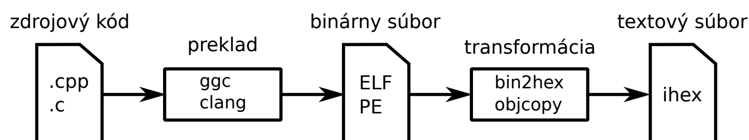
5.1 Intel HEX

Táto podkapitola bola spracovaná na základe [12].

Intel HEX je formát navrhnutý spoločnosťou Intel Corporation, ktorý ukladá binárne informácie formou ASCII textu – nejedná sa tak o binárny formát v úplnom zmysle slova. Využitie formátu Intel HEX nájdeme predovšetkým v nástrojoch pre programovanie mikrokontrolérov, zapisovanie informácií do ROM, PROM a EPROM pamätí, prípadne pre simulácie softvéru v rámci jednotlivých vývojových prostredí. Ďalší rozdiel oproti bežným binárnym súborom je, že formát neposkytuje informácie o tom, čo binárne informácie obsiahnuté v súbore predstavujú – nenájdeme tu informácie o bajtovom usporiadaní či cieľovej architektúre, tieto informácie musíme poznať alebo získať iným spôsobom. Chýbajú tiež segmenty a sekcie, ktoré nájdeme v objektových súboroch iných formátov.

5.1.1 Vytvorenie súboru s formátom Intel HEX

Intel HEX súbor nevzniká kompiláciu zdrojového súboru určitého jazyka, ale konverziou už kompilátorom vytvoreného binárneho súboru. Väčšina Unixových systémov poskytuje spolu s prekldačom nástroje na to určené, konktérne ide o programy *bin2hex* alebo *objcopy*. Postup ilustruje obrázok 5.1.



Obr. 5.1: Vznik súboru s formátom Intel HEX.

5.1.2 Štruktúra formátu Intel HEX

Formát Intel Hex ukladá binárne informácie ako hexadecimálne číslo v ASCII podobe (využíva znaky *'a'* až *'f'* alebo *'A'* až *'F'* a číslice *'0'* až *'9'*). Jeden bajt binárnej informácie je zakódovaný dvojicou ASCII znakov, výsledná reprezentácia dát tak zaberie dvakrát viac bajtov než pôvodná binárna reprezentácia. V poradí prvý znak jednej dvojice v súbore vždy predstavuje štyri najvýznamnejšie bity (anglicky most significant bits).

Súbor je rozdelený na záznamy, väčšinou každý uložený na práve jednom riadku (špecifikácia sa o riadkovaní nezmieňuje, väčšina nástrojov ho však pre lepšiu čitateľnosť používa). Začiatok nového záznamu je označený symbolom dvojbodky (ASCII #58), záznam ďalej obsahuje nasledujúce informácie:

- **Veľkosť dát** – je určená dvomi ASCII znakmi, maximálna veľkosť dátovej časti jedného záznamu je tak obmedzená na 255 bajtov binárnej informácie resp. 510 ASCII znakov. Prakticky je však kôli čitateľnosti toto číslo oveľa menšie.

- **Adresa** – je určená štyrmi ASCII znakmi, máme tak k dispozícií 16-bitovú adresu. Pre zápis 32-bitovej adresy potrebujeme špeciálny záznam, ktorý poskytne najvýznamnejších 16 bitov 32-bitovej adresy. Ak takýto záznam chýba, uvažuje sa nula. Adresa je vo formáte Intel HEX vždy uložená v bajtovom usporiadaní big endian.
- **Typ záznamu** – bajtová hodnota určená dvomi ASCII znamki identifikujúca typ záznamu. Môže nadobúdať hodnoty 0 až 5.
- **Dáta** – samotné dáta záznamu. Formát nepredpisuje žiadne usporiadanie, v prípade potreby tejto informácie je nutné použiť iný spôsob pre jej zistenie.
- **Kontrolný súčet** – bajtová hodnota slúžiaca pre overenie správnosti záznamu. Hodnotu vypočítame ako súčet jednotlivých bajtov záznamu s jeho následnou negáciou (prípadné pretečenie pri sčítaní neberieme do úvahy).

```
     adresa                  dáta
:10 8564 00 60000000B6FEFFFF190000000410E08 85
:10 8574 00 8502420D0555C50C0404000038000000 B6
:10 83E0 00 D0C9E979FFFFFF90E973FFFFFF5589E5 E9
velkosť     typ                           kontrolný súčet
```

Obr. 5.2: Ukážka formátu Intel HEX.

Formát Intel HEX definuje nasledujúce typy záznamov:

- **Dátový záznam** – obsahuje dáta a spodných 16 bitov ich adresy.
- **Záznam pre koniec súboru** – označuje koniec súboru, vyskytuje sa práve raz ako posledný záznam v súbore. Dátová časť je prázdna a adresa je typicky 0.
- **Záznam pre 20-bitové adresovanie** – poskytuje 16-bitovú segmentovú adresu pre *real mode* adresovanie architektúry 80x86.
- **Záznam pre zápis obsahu registrov CS:IP** – špecifikuje obsah registrov CS (Code Segment) a IP (Instruction Pointer). Jedná sa o vstupný bod programu pri použití 20-bitového adresovania.
- **Záznam pre rozšírenie lineárnej adresy** – umožňuje 32-bitové adresovanie. Adresová časť záznamu sa ignoruje a dátová časť obsahuje najvýznamnejších 16 bitov adresy. Adresa nasledujúcich záznamov tak vznikne súčtom poskytnutej hodnoty posunutej o 16 bitov doľava a hodnoty uloženej v adresovej časti nasledujúcich záznamov. Táto hodnota je platná, pokiaľ nie je prepísaná novým záznamom pre rozšírenie adresy. Ak sa tento záznam v súbore nevyskytne pred dátovým záznamom, uvažuje sa nula.
- **Záznam pre zápis obsahu registru EIP** – špecifikuje 32-bitovú hodnotu, ktorá bude nahraná do registru EIP. Register EIP (Extended Instruction Pointer) v architektúre x86 uchováva adresu nasledujúcej inštrukcie, v prípade Intel HEX formátu sa jedná o prvú vykonanú inštrukciu – vstupný bod programu.

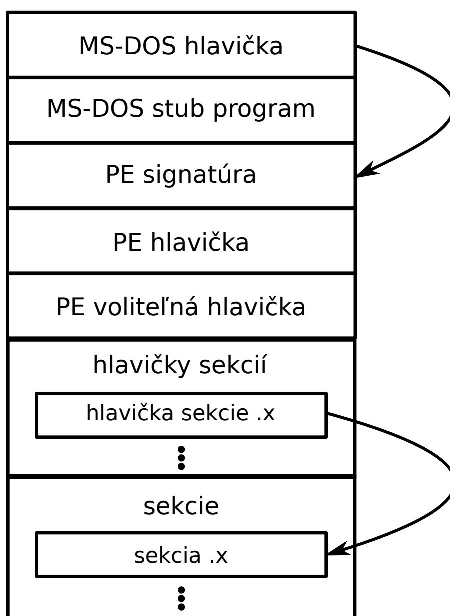
5.2 Portable Executable

Táto podkapitola bola spracovaná na základe [18].

Formát Portable Executable (PE) je formát pre spustiteľné súbory, linkovateľné súbory alebo dynamicky linkované knižnice (DLL). Formát bol vyvinutý spoločnosťou Microsoft a prvýkrát predstavený s operačným systémom Windows NT 3.1 a používa sa dodnes v tridsaťdva a šesťdesiatštyri bitových verziách OS Windows. Formát je založený na Unixovom formáte COFF a ako názov napovedá, formát je prenosný a teda nie je viazaný na žiadnu konkrétnu cieľovú architektúru.

5.2.1 Štruktúra formátu Portable Executable

Štruktúra formátu je podobná všeobecnej štruktúre spomenutej na začiatku kapitoly 5. Nájdeme tu hlavičku a dáta logicky rozdelené na viaceré časti – sekcie. Štruktúra formátu je zobrazená na obrázku 5.3.



Obr. 5.3: Štruktúra formátu Portable Executable. Prevzaté z [18], upravené.

MS-DOS stub

Prvá časť súboru pozostáva z MS-DOS hlavičky a MS-DOS stub programu, ktorý je schopný behu v reálnom móde operačného systému MS-DOS. Jeho účelom je vypísanie hlášky *„This program cannot be run in DOS mode“* alebo inej, pokiaľ bola špecifikovaná pri linkovaní súboru. Zabezpečuje tak kompatibilitu s OS MS-DOS. Lokácia 0x3C obsahuje posuv PE signatúry od začiatku súboru, umožňuje tak preskočiť MS-DOS stub program, ktorého dĺžka nie je pevná.

PE signatúra

Signatúra jednoznačne identifikujúca formát súboru ako Portable Executable. Pozostáva zo štyroch bajtov obsahujúcich sekvenciu 'PE\0\0'.

PE hlavička

PE hlavička sa nachádza na začiatku linkovateľného súboru alebo ihneď za PE signatúrou v prípade spustiteľného súboru. Informácie obshiahnuté v PE hlavičke popisuje tabuľka 5.1.

Pozícia	**Počet bajtov**	**Popis**
0	2	Identifikátor cieľovej architektúry.
2	2	Počet sekcií.
4	2	Časové razítko.
8	4	Ukazovateľ na tabuľku symbolov.
12	4	Počet symbolov v tabuľke symbolov.
16	2	Veľkosť voliteľnej hlavičky
18	2	Charakteristika (príznaky) súboru.

Tabuľka 5.1: Štruktúra PE hlavičky. Prevzaté z [18], upravené.

Voliteľná hlavička

Hneď za PE hlavičkou sa nachádza voliteľná hlavička. Voliteľnosť závisí od typu súboru, v spustiteľných súboroch je voliteľná hlavička povinná, v linkovateľných súboroch táto hlavička nemá význam. Veľkosť voliteľnej hlavičky nie je fixná, jej veľkosť je určená poľom *SizeOfOptionalHeader* PE hlavičky súboru. Voliteľná hlavička má vlastnú signatúru, ktorá určuje, či sa jedná o 32-bitový (*PE32*) alebo 64-bitový (*PE32+*) súbor. Voliteľnú hlavičku môžeme rozdeliť na tri hlavné časti:

- **Štandardné polia** – tieto polia sú definované pre všetky implementácie COFF hlavičky. Polia popisuje tabuľka 5.2.
- **Polia špecifické pre OS rodiny Windows** – dodatočné polia pre podporu špecifických funkcií OS Windows.
- **Definície špeciálnych tabuliek** – tieto polia tvoria páry adresa-veľkosť, ktoré určujú pozíciu a veľkosť špeciálnych tabuliek, ktoré sa nachádzajú v súbore. Príkladom môže byť tabuľka exportovaných symbolov alebo tabuľka importovaných symbolov.

Hlavičky sekcií

Hlavičky sekcií sa nachádzajú hneď za voliteľnou hlavičkou. Počet sekcií je daný poľom *NumberOfSections* PE hlavičky súboru. Jednotlivé hlavičky sú číslované (od 1) a ich poradie je dané procesom linkovania. Každá hlavička sekcie má veľkosť 40 bajtov a jej formát popisuje tabuľka 5.3.

Pozícia	Počet bajtov	Popis
0	2	Signatúra voliteľnej hlavičky.
2	1	Verzia použitého linker-u (veľké číslo).
3	1	Verzia použitého linker-u (malé číslo).
4	4	Veľkosť sekcie (sekcií) obsahujúcej kód.
8	4	Veľkosť sekcie (sekcií) pre inicializované dáta.
12	4	Veľkosť sekcie (sekcií) pre ne-inicializované dáta.
16	4	Adresa vstupného bodu po načítaní do pamäti.
20	4	Adresa začiatku kódovej sekcie po načítaní do pamäti.
24	4	Adresa začiatku dátovej sekcie po načítaní do pamäti.

Tabuľka 5.2: Štandardné polia voliteľnej hlavičky. Prevzaté z [18], upravené.

Pozícia	Počet bajtov	Popis
0	8	Názov sekcie.
8	4	Veľkosť sekcie po načítaní do pamäte.
12	4	Virtuálna adresa po načítaní do pamäte.
16	4	Veľkosť sekcie (alebo inicializovaných dát) na disku.
20	4	Ukazovateľ na výskyt sekcie v súbore.
24	4	Ukazovateľ na výskyt relokačných záznamov v súbore.
28	4	Ukazovateľ na výskyt záznamov riadkovania pre ladenie.
32	2	Počet relokačných záznamov.
34	2	Počet záznamov riadkovania.
36	4	Príznaky sekcie. Predovšetkým R/W práva a typ dát.

Tabuľka 5.3: Štruktúra hlavičky sekcie. Prevzaté z [18], upravené.

Sekcie

Inicializované dáta sekcií pozostávajú z blokov bajtov. V prípade, že je celá sekcia inicializovaná na nuly, tieto dáta nemusia byť zahrnuté v súbore. Veľkosť a pozícia konkrétnej sekcie v súbore sú uložené v hlavičke danej sekcie. Ak je veľkosť sekcie na disku menšia, než veľkosť sekcie po načítaní do pamäti, rozdiel je inicializovaný na nuly. V prípade spustiteľného súboru musí poradie sekcií zohľadňovať ich virtuálnu adresu a požiadavky na zarovnanie dané poľom *FileAlignment* voliteľnej hlavičky.

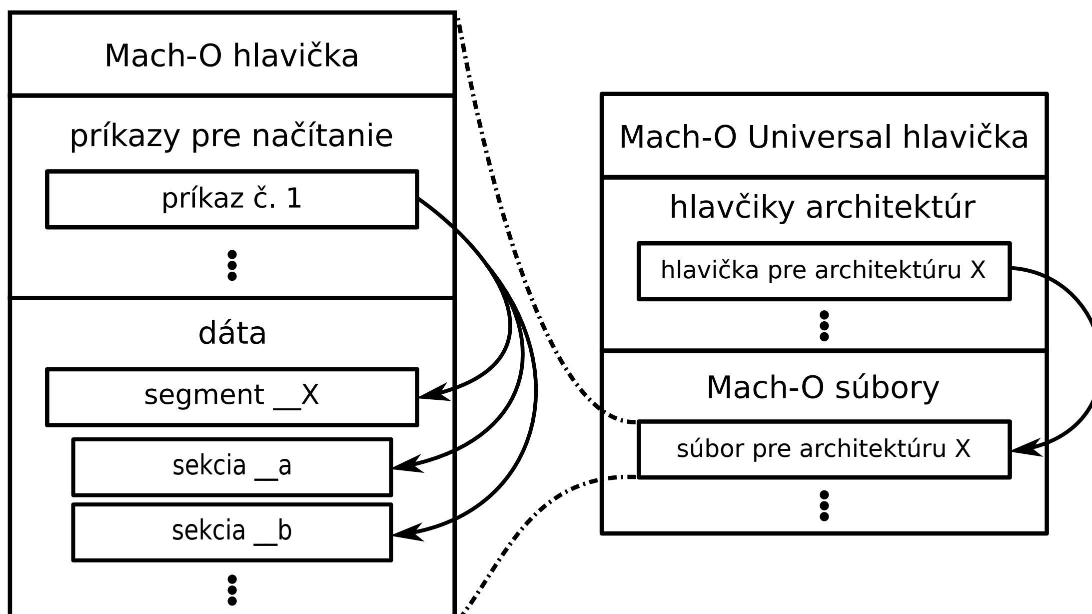
5.3 Mach-O

Táto podkapitola bola spracovaná na základe [1] a verejne dostupných zdrojových kódov.

Formát Mach-O je binárny formát vyvíjaný spoločnosťou Apple Inc. a používaný v operačných systémoch OS X a iOS. Formát je primárne použitý pre spustiteľné a linkovateľné súbory a dynamické knižnice, menej často pre uloženie ladiacich informácií, výpis pamäti (anglicky core dump), modulov (anglicky bundles), dynamických linkerov alebo OS X a iOS ovládačov (anglicky kernel extensions).

5.3.1 Štruktúra formátu Mach-O

Štruktúra formátu je mierne odlišná od bežnej štruktúry objektových formátov. Na začiatku súboru sa nachádza tradičná hlavička so signatúrou formátu, za ňou nasleduje vopred špecifikovaný počet príkazov pre načítanie binárneho súboru (anglicky load commands). Za týmito príkazmi sa nakoniec nachádzajú samotné dáta. Štruktúra formátu je zobrazená na obrázku 5.4.



Obr. 5.4: Štruktúra formátu Mach-O (vľavo) a Mach-O Universal Binary (vpravo).

Signatúra

Signatúra je súčasťou hlavičky a identifikuje formát súboru ako Mach-O alebo Mach-O Universal Binary. Existuje celkom päť signatúr popísaných v tabuľke 5.4.

Signatúra	Bitová šírka	Usporiadanie bajtov
`0xFEEDFACE`	32 bitov	big endian
`0xFEEDFACF`	64 bitov	big endian
`0xCEFAEDFE`	32 bitov	little endian
`0xCFFAEDFE`	64 bitov	little endian
`0xCAFEBABE`	univerzálna	univerzálne

Tabuľka 5.4: Mach-O signatúry

Hlavička

Hlavička ďalej poskytuje informácie o cieľovej architektúre, type a vlastnostiach súboru a o celkovom počte a veľkosti príkazov pre načítanie súboru.

Pozícia	Počet bajtov	Popis
0	4	Signatúra formátu.
4	4	Rodina cieľovej architektúry, napr. ARM.
8	4	Bližšia špecifikácia architektúry, napr. ARMv7.
12	4	Funkcia súboru (spustiteľný súbor, knižnica).
16	4	Počet príkazov pre načítanie.
20	4	Veľkosť príkazov pre načítanie súboru v bajtoch.
24	4	Príznaky (anglicky flags).
28	4	Rezervované (len 64-bit verzia formátu Mach-O).

Tabuľka 5.5: Štruktúra Mach-O hlavičky. Prevzaté z [1], upravené.

Príkazy pre načítanie súboru

Príkazy pre načítanie súboru (ďalej len príkazy) popisujú celkové rozloženie dát v binárnom súbore, napr. segmenty a sekcie, tabuľky symbolov, exportov a importov, vstupný bod, použité dynamické a statické knižnice, prípadne informácie o použitom zabezpečení a cieľovom operačnom systéme. Medzi najdôležitejšie príkazy patria:

- `LC_SEGMENT` a `LC_SEGMENT_64` – načítanie segmentov a sekcií. Počet príkazov tohto typu odpovedá počtu segmentov v binárnom súbore. Za príkazom pre načítanie segmentov môže nasledovať pole štruktúr obsahujúcich dáta pre načítanie sekcií. Informácia o počte sekcií daného segmentu je obsiahnutá v rámci samotného príkazu.
- `LC_UNIXTHREAD` – stav registrov pred štartom aplikácie (OS X v.10.5 a menej).
- `LC_MAIN` – pozícia vstupného bodu programu v súbore a veľkosť zásobníku pre hlavné vlákno programu (OS X v.10.6 a viac).
- `LC_SYMTAB` – pozícia tabuľky symbolov a tabuľky reťazcov vo vstupnom súbore a ich veľkosti.
- `LC_DYSYMTAB` – začiatočný index jednotlivých typov symbolov v tabuľke symbolov (lokálne symboly, exporty a importy) a pozícia a veľkosť mnohých ďalších, pre spätný preklad menej dôležitých, tabuliek. Tento príkaz je závislý na príkaze `LC_SYMTAB`, ktorý popisuje uchovanie samotných symbolov.
- `LC_DYLD_INFO` a `LC_DYLD_INFO_ONLY` – alternatívny príkaz pre načítanie importov a exportov (OS X v.10.8 a viac). Tento príkaz vie poskytnúť informácie nezávisle na existencii príkazov `LC_SYMTAB` a `LC_DYSYMTAB`.
- `LC_LOAD_DYLIB` – získanie názvov použitých dynamických knižníc. Načítanie importov a exportov je závislé na tomto príkaze. Počet príkazov tohto typu odpovedá počtu použitých dynamických knižníc.

Podľa enumerácie uvedenej v zdrojových súborov projektu LLVM existuje celkom 47, viac či menej dokumentovaných príkazov. Nemusí sa však jednať o všetky existujúce príkazy. S výnimkou niekoľkých prípadov, ktoré si vyžadujú špecifické usporiadanie, keď napr. príkaz `LC_DYSYMTAB` musí pre správne fungovanie predchádzať príkaz `LC_SYMTAB`, nie je poradie príkazov nijak stanovené. Všeobecný formát príkazu popisuje tabuľka 5.6. Zvyšná časť je rozdielna pre každý príkaz a závisí od typu príkazu.

Pozícia	Počet bajtov	Popis
0	4	Typ príkazu.
4	4	Celková veľkosť príkazu v bajtoch.

Tabuľka 5.6: Štruktúra Mach-O príkazu. Prevzaté z [1], upravené.

5.3.2 Mach-O Universal Binary

Formát Mach-O Universal Binary vznikol ako pomôcka pre prechod z architektúry PowerPC na architektúru x86, ktorý firma Apple Inc. uskutočnila v roku 2005. Súbor tohto formátu vie pojať ľubovoľný počet architektúr, väčšinou sa však jedná kombinácie architektúr PowerPC a Intel x86 alebo odlišné verzie ARM architektúr v prípade systému iOS. Formát môže obsahovať všetky typy Mach-O súborov spomenuté vyššie ako aj celé statické knižnice, väčšinou zabalené Unixovým formátom archiver. Štruktúra formátu je zobrazená na obrázku 5.4.

Hlavička

Hlavička obsahuje signatúru `0xCAFEBABE` (vždy big endian) a informáciu o počte architektúr obsiahnutých v tomto súbore.

Hlavičky architektúr

Za úvodnou hlavičkou nasleduje zoznam hlavičiek architektúr, ktoré špecifikujú pozíciu v súbore a celkovú veľkosť dát pre konkrétnu architektúru. Hlavičky architektúr sú uložené v bajtovom usporiadaní typu big endian.

Pozícia	Počet bajtov	Popis
0	4	Rodina cieľovej architektúry, napr. ARM.
4	4	Bližšia špecifikácia architektúry, napr. ARMv7.
8	4	Pozícia v súbore pre konkrétnu architektúru.
12	4	Vekosť dát v súbore pre konkrétnu architektúru.
16	4	Zarovnanie dát.

Tabuľka 5.7: Štruktúra Mach-O Universal Binary hlavičky. Prevzaté z [1], upravené.

Kapitola 6

Spätný preklad súborov formátu Intel HEX

Formát Intel HEX sa od ostatných formátov spustiteľných súborov zásadne líši tým, že binárne dáta sú rozdelené do viacerých záznamov a uložené formou zápisu pomocou ASCII znakov. Zároveň chýbajú informácie o polohách a veľkostiach prípadných sekcií a tabuliek symbolov, exportov a importov. Tieto vlastnosti znemožňujú použitie už implementovaných metód pre načítanie dát do vnútornej reprezentácie spätného prekladača. Načítanie dát a rekonštrukciu sekcií je tak nutné implementovať na strane samotného modulu.

6.1 Požiadavky

Pre úspešný spätný preklad je nutné dáta previesť z textu do binárnej formy a uložiť do aspoň jednej sekcie, s ktorou môžu ďalšie časti prekladača ďalej pracovať. Ďalej je nutné implementovať všetky metódy popísané v podkapitole 4.3.

Kvôli nemožnosti použiť vstupný súbor v rámci ďalších častí prekladača je tiež nutné implementovať metódu pre serializáciu dát. Jednotlivé sekcie budú usporiadané podľa ich virtuálnej adresy a zlúčené do jedného bloku binárnych dát. Takto pripravený blok bude slúžiť ako náhrada pôvodného vstupného súboru.

6.2 Návrh

V súčastnosti existuje niekoľko knižníc pre manipuláciu s formátom Intel HEX, väčšina však nevyhovuje buď pre použitý programovací jazyk, rozhranie knižnice alebo licenciu. Vzhľadom na jednoduchosť formátu Intel HEX sa tak ako najjednoduchšia možnosť javí napísať si vlastnú analýzu formátu, prispôsobenú požiadavkám spätného prekladača.

6.2.1 Lexikálna analýza

Spracovanie súboru lexikálnou analýzou prebieha po jednotlivých záznamoch, väčšinou uložených na jednom riadku. Analyzátor postupne načíta jednotlivé časti záznamu po skupine bajtov obsahujúcich ASCII hodnoty tak, ako sú popísané v podkapitole 5.1.2. Časti záznamu, ktoré sú nutné pre jeho úspešné prečítanie (veľkosť dát a kontrolný súčet), sú prevedené z ASCII zápisu do číselnej formy hneď. Dátové časti záznamu, adresa a samotné binárne dáta, sú ponechané v nezmenenom stave. Lexikálna analýza posiela spracované

dáta ďalej formou tokenu. Token je objekt obsahujúci typ, adresu a dáta práve jedného záznamu. Typ tokenu môže nadobúdať šesť hodnôt, prvých päť odpovedá jednotlivým typom záznamom, šiesty má za úlohu propagovať prípadnú chybu v rámci lexikálnej analýzy.

6.2.2 Interpretácia dát

Úlohou tejto časti je interpretácia dát získaných v lexikálnej analýze a ich uschovanie pre nasledujúce dotazy. V rámci prevodu je nutné interpretovať tri druhy informácii:

- **Úpráva adries nasledujúcich záznamov** – v tomto prípade je nutné previesť informáciu o úprave adresy uloženej formou ASCII textu do vnútornej reprezentácie, ktorá bude neskôr využitá pri interpretácii virtuálnych adries nasledujúcich záznamov. Jedná sa buď o záznam `0x04`, ktorý obsahuje horných 16 bitov adresy alebo záznam `0x02`, ktorý obsahuje segmentovú adresu (x86 register CS).

- **Získanie adresy vstupného bodu** – v tomto prípade dôjde k prevodu ASCII znakov reprezentujúcich obsah x86 špecifických registrov CS a IP alebo EIP do číselnej formy. Samotná adresa sa v prípade *real mode* adresovania vypočíta na základe obsahu registrov CS a IP až pri dotaze na vstupný bod pomocou vzorca 6.1. V prípade bežného adresovania (šírka adresy je 32 bitov) nie je ďalší výpočet nutný – záznam obsahuje priamo hodnotu registru EIP.

$$Adresa = (CS \times 16) + IP \qquad (6.1)$$

- **Interpretácia samotných dát** – v tomto prípade dochádza k interpretácii samotných binárnych dát a ich virtuálnej adresy. V prvom kroku dôjde k prevedeniu a prípadnej úprave adresy, ak predtým boli použité odpovedajúce záznamy. Následne dochádza k prevodu samotných dát a vytváraniu logických sekcií.

6.2.3 Rozdelenie vstupných dát na logické sekcie

Keďže sa medzi jednotlivými blokmi dát obsiahnutých v za sebou idúcich záznamoch môžu vyskytnúť medzery, je nutné dáta uložené vo vstupnom súbore rozdeliť na logické celky, sekcie, ktoré obsahujú len bezprostredne za sebou idúce dáta.

Na začiatku sa predpokladá jedna sekcia. V prípade, že adresa aktuálne spracovaného záznamu je väčšia oproti adrese posledného dátového bajtu predchádzajúceho záznamu o viac ako jedna, jedná sa o novú sekciu. Tento prístup spôsobuje, že viaceré pôvodné sekcie, ktorých dáta sú uložené v pamäti bezprostredne za sebou, sú spojené do jednej logickej sekcie. Takto vytvorené sekcie sú dočasne uložené v internej reprezentácií modulu. Po prečítaní celého súboru sú potom sekcie presunuté do vnútornej reprezentácie spätného prekladača.

6.2.4 Chýbajúce informácie

Vzhľadom na chýbajúce informácie o sekciách alebo ich prípadné spojenie do väčších sekcií nevieme rozhodnúť, aké informácie sekcia obsahuje a či vôbec dôjde k jej načítaniu do pamäti. U všetkých sekcií preto uvažujeme, že k načítaniu do pamäti dôjde. Typ dát jednotlivých sekcií zostáva nešpecifikovaný, čo môže spôsobiť nesprávne interpretovanie dát v ďalších fázach spätného prekladu (napr. sekcia obsahujúca dáta môže byť interpretovaná ako kód). Následkom je výrazné zníženie kvality prekladu.

Ďalšou chýbajúcou informáciou je cieľová architektúra. Vzhľadom na povinnosť implementácie detekčných metód súvisiacich s týmto druhom informácie musí tieto údaje (architektúra, bajtové usporiadanie a šírka adresy) poskytnúť samotný užívateľ.

6.2.5 Serializácia dát

Pri serializácii dát je nutné usporiadať sekcie podľa ich virtuálnej adresy a následne zlúčiť všetky dáta do jedného celku. Zároveň je nutné nastaviť pozície jednotlivých sekcií a vstupného bodu programu. Tie nebudú ukazovať na pozíciu v súbore, ako je tomu pri ostatných formátoch, ale na pozíciu v serializovanom bloku dát.

6.3 Implementácia

Implementácia modulu je rozdelená do celkom šiestich tried:

- `IntelHexToken` – reprezentuje jeden token v rámci lexikálnej analýzy bližšie popísaný v podkapitole 6.2.1. Obsahuje dve metódy `addStringByTwo` a `controlChecksum`, ktoré slúžia pre overenie kontrolného súčtu. Pre reprezentáciu ešte neinterpretovaných dát sa využíva trieda `std::string`.

- `IntelHexTokenizer` – má za úlohu otvoriť vstupný súbor a previesť jednotlivé záznamy na tokeny. Verejné rozhranie pozostáva z metód `openFile` a `setInputStream` pre nastavenie vstupu, ktorý bude spracovaný. Jednotlivé tokeny je možné získať volaním metódy `getToken`, ktorá postupne znak po znaku číta vstup. Privátna metóda `readN` rieši čítanie znakov a metóda `makeErrorToken` slúži pre prípravu tokenu v prípade chyby vo vstupnom súbore.

- `IntelHexSection` – dátová trieda reprezentujúca blok bezprostredne za sebou idúcich dát s určitou virtuálnou adresou. Trieda tiež implementuje operátor *menší než* pre účel zoradenia sekcií.

- `IntelHexParser` – trieda zodpovedná za interpretáciu dát obsiahnutých v tokenoch. Verejné rozhranie pozostáva z metód určených pre spustenie analýzy – `parseFile` a `parseStream`, metód pre získanie vstupného bodu programu – `hasEntryPoint` a `getEntryPoint`. Trieda ďalej obsahuje verejné statické metódy pre prevody potrebné v rámci spracovania ASCII textu – `strToInt` a `isHexadec`. Privátne metódy `handleData`, `setOffset`, `setSegment`, `setEIP` a `setCSIP` slúžia pre interpretáciu jednotlivých typov záznamov.

- `IntelHexFileFormat` – dedí základnú triedu `FileFormat` a implementuje povinné rozhranie pre potreby spätného prekladača. Nezvyčajným prvkom sú verejné metódy `setTargetArchitecture`, `setEndianness` a `setBytesPerWord`, ktoré riešia problém chýbajúcich informácií popísaný v predchádzajúcej kapitole. Pripravené serializované dáta je možné získať volaním metódy `getSerializedData`. Verejné rozhranie ešte dopĺňajú dva konštruktory, jeden pre otvorenie súboru, druhý pre vstup formou toku dát (anglicky input stream). Pre prípravu serializovaných dát a načítanie sekcií slúžia privátne metódy `initializeSectionOffsets`, `initializeSections` a `init`.

6.4 Testovanie

Pre otestovanie podpory formátu Intel HEX bola zvolená kombinácia dvoch prístupov. Jednotkové testy zamerané na rýchlu kontrolu správnosti pri vývoji alebo prípadnom rozširovaní či udržovaní modulu. Regresné testy pre odhalenie prípadnej regresie v rámci ďalšieho vývoja spätného prekladača a overenie správnosti spätného prekladu.

6.4.1 Jednotkové testy

Jednotkové testy sú implementované v jazyku C++ pomocou knižnice *Google Test*. Z dôvodu implementácie týchto testov bolo do testovaného modulu pridané rozhranie pre prácu s dátovými tokmi, testované vzorky sú tak súčasťou zdrojového kódu testov. Sada celkom 12 testov pokrýva príklady oboch typov adresovania, legálne kombinácie záznamov a tiež možné nesprávne vstupy. Testy pokrývajú takmer všetky riadky kódu s výnimkou práce so vstupnými súbormi – tie bude možné overiť pri regresných testoch. Referenčné výsledky boli získané za pomoci štandardného nástroja *objdump*.

6.4.2 Regresné testy

Regresné testy sú implementované v jazyku Python3 pomocou pripraveného interného rozhrania `regression_tests`. Testy spustia spätný preklad priložených testovacích súborov a následne kontrolujú kvalitu výstupu v jazyku C, predovšetkým správnu identifikáciu funkcií a graf ich volaní. Sada celkom 17 testov pokrýva rozdielne architektúry (PIC32, MIPS a x86), usporiadanie bajtov (little, big endian) a optimalizácie prekladača.

6.5 Dosiahnuté výsledky

Vytvorený modul je schopný spracovať vstupný súbor formátu Intel HEX a poskytnúť informácie pre spätný preklad alebo nástroj *fileinfo*. Všetky navrhnuté testy skončili úspechom.

6.5.1 Spätný preklad

Spätný prekladač je schopný preložiť vstupné súbory formátu Intel HEX. Testovacie súbory boli vytvorené pomocou bežného prekladu a následného prevodu pomocou štandardného programu *objcopy*. Pôvodný zdrojový kód programu pre výpočet Ackermannovej funkcie použitý pre testovanie je možné nájsť v prílohe B.1.

Výstup 6.3 obsahuje výsledok spätného prekladu z architektúry MIPS (big endian). Pri pohľade na funkciu `function_4006f0` je možné rozoznať algoritmus Ackermannovej funkcie. Funkcie `function_400a40` a `function_400a20` predstavujú volania štandardných funkcií `scanf` a `printf`. Vzhľadom nato, že testovací vstup bol vytvorený len zo sekcie `.text`, tieto volania odkazujú na prázdne miesto a neobsahujú preto žiadny spätne preložený kód. Vytvorenie programu zo všetkých sekcií by malo za následok spätný preklad častí neobsahujúcich kód a následný nezmyselný výstup, prípadne zlyhanie spätného prekladu. Kvôli absencii tabuľky symbolov tiež nie je možné rekonštruovať pôvodné mená funkcií. Preklad vstupného súboru prebehol bez optimalizácií.

Výstup 6.4 obsahuje výsledok spätného prekladu z architektúry MIPS (little endian). Program obsahuje staticky linkované štandardné funkcie. Funkcia `function_8900368` obsahuje algoritmus Ackermannovej funkcie. Staticky linkované funkcie sa podarilo úspešne identifikovať a zrekonštruovať. Preklad vstupného súboru prebehol s optimalizáciou O1.

Výstup 6.5 demonštruje preklad z architektúry PIC32. Rovnako ako v predchádzajúcom príklade bolo možné identifikovať a následne zrekonštruovať staticky linkované štandardné funkcie. Výstup 6.6 demonštruje preklad z architektúry x86.

6.5.2 Nástroj fileinfo

V ukážke výstupu 6.1 je vidno informácie o názve a type formátu a vstupnom bode programu. Ďalej je vidno tabuľku ôsmich logických sekcií vytvorených na základe ich adries a veľkostí. Výstup je možné porovnať so štandardným nástrojom *objdump* 6.2. Pri pohľade na jednotlivé výstupy je možné pozorovať dva rozdiely. Prvým rozdielom je odlišný generický názov pre sekcie. Druhý rozdiel je o niečo podstatnejší a je možné ho vidieť v stĺpcoch `offset` a `File off`. Program *objdump* používa skutočnú pozíciu v súbore, teda odkazuje na polohu prvého ASCII znaku prvého záznamu danej sekcie. Program *fileinfo* používa pozíciu v serializovanom bloku dát. Tento prístup je však nevyhnutný pre správne fungovanie spätného prekladača.

```
File format                : Intel HEX
File type                  : Executable file
Entry point address        : 0x400520
Warning: Unknown compiler or packer.

Declared number of sections              : 8

Section table
-------------

i    name              offset     fsize      address     memsize
-----------------------------------------------------------------
0    ihex_section_0    0          0x00d      0x400134    0x00d
1    ihex_section_1    0x00d      0x32e      0x400144    0x32e
2    ihex_section_2    0x33b      0x0a4      0x400474    0x0a4
3    ihex_section_3    0x3df      0x4cc      0x400520    0x4cc
4    ihex_section_4    0x8ab      0x050      0x400a00    0x050
5    ihex_section_5    0x8fb      0x028      0x410a50    0x028
6    ihex_section_6    0x923      0x014      0x410a80    0x014
7    ihex_section_7    0x937      0x01c      0x410aa0    0x01c
```

Výstup 6.1: Výstup programu *fileinfo*.

```
start address 0x00400520

Sections:
Idx Name     Size      VMA       LMA       File off  Algn
  0 .sec1    0000000d  00400134  00400134  00000011  2**0
  1 .sec2    0000032e  00400144  00400144  00000038  2**0
  2 .sec3    000000a4  00400474  00400474  00000945  2**0
  3 .sec4    000004cc  00400520  00400520  00000b1c  2**0
  4 .sec5    00000050  00400a00  00400a00  000018b7  2**0
  5 .sec6    00000028  00410a50  00410a50  000019a9  2**0
  6 .sec7    00000014  00410a80  00410a80  00001a3a  2**0
  7 .sec8    0000001c  00410aa0  00410aa0  00001a7c  2**0
```

Výstup 6.2: Výstup štandardného programu *objdump*.

```c
// -------------------- Global Variables --------------------

int32_t g1 = 0; // $fp

// ---------------------- Functions ------------------------

// Address range: 0x4006f0 - 0x4007c3
int32_t function_4006f0(int32_t a1, int32_t a2) {
    int32_t v1 = g1; // 0x4006f8
    int32_t v2;
    g1 = &v2;
    if (a1 == 0) {
        // 0x400724
        // branch -> 0x4007a8
        // 0x4007a8
        g1 = v1;
        return a2 + 1;
    }
    // 0x400738
    int32_t result; // 0x4007a81
    if (a2 == 0) {
        // 0x40074c
        result = function_4006f0(a1 - 1, 1);
        // branch -> 0x4007a8
    } else {
        // 0x40076c
        result = function_4006f0(a1 - 1, function_4006f0(a1, a2 - 1));
        // branch -> 0x4007a8
    }
    // 0x4007a8
    g1 = v1;
    return result;
}

// Address range: 0x4007c4 - 0x40087b
int main(int argc, char ** argv) {
    int32_t v1;
    g1 = &v1;
    int32_t v2 = 0; // bp-28
    function_400a40();
    function_4006f0(v2, 0);
    function_400a20();
    return &v2;
}

// Address range: 0x400a20 - 0x400a2f
int32_t function_400a20(void) {
    // 0x400a20
    return unknown_400a00();
}

// Address range: 0x400a40 - 0x400a4f
int32_t function_400a40(void) {
    // 0x400a40
    return unknown_400a00();
}
```

Výstup 6.3: Výsledok spätného prekladu programu pre výpočet Ackermannovej funkcie. Použitá architektúra: MIPS. Bajtové usporiadanie: big endian. Optimalizácie: žiadne.

```
// --------------------- Global Variables ---------------------

int32_t g1 = 0; // $ra
int32_t g2 = 0; // $s0
int32_t g3 = 0; // $s1
int32_t g4 = 0; // $s2
int32_t g5 = -1; // 0x8916f74
int32_t g6 = 0; // 0x8916f84
int32_t g7 = 1; // 0x8917a88
int32_t g8 = -1; // 0x8916f7c
int32_t g9 = 0; // 0x8916f80
int32_t g10 = 0; // 0x8916f88

// ------------------------ Functions -------------------------

// Address range: 0x8900368 - 0x89003c3
int32_t function_8900368(int32_t a1, int32_t a2) {
    // 0x8900368
    int32_t result;
    if (a2 == 0) {
        // 0x8900388
        result = function_8900368(0, 1);
        // branch -> 0x89003b8
    } else {
        int32_t v1 = function_8900368(a1, a2 - 1); // 0x89003a0
        result = function_8900368(g2 - 1, v1);
        // branch -> 0x89003b8
    }
    // 0x89003b8
    return result;
}

// Address range: 0x89003c4 - 0x8900427
int32_t function_89003c4(void) {
    int32_t v1 = 0; // bp-16
    int32_t v2 = 0; // bp-12
    scanf("%d␣%d", &v1, &v2);
    int32_t result = function_8900368(v1, v2); // 0x89003f0
    printf("ackerman(␣%d␣,␣%d␣)␣=␣%d\n", v1, v2, result);
    return result;
}

// Address range: 0x8900428 - 0x890043b
int32_t function_8900428(int32_t a1) {
    // 0x8900428
    __register_exitproc();
    return 0;
}
```

Výstup 6.4: Výsledok spätného prekladu programu pre výpočet Ackermannovej funkcie. Použitá architektúra: MIPS. Bajtové usporiadanie: little endian. Optimalizácie: O1.

```
// --------------------- Global Variables ---------------------

int32_t g1 = 0; // $fp
char g2[2] = "<"; // 0x9d001eac

// ------------------------ Functions -------------------------

// Address range: 0x9d001d5c - 0x9d001e0b
int32_t function_9d001d5c(int32_t a1, int32_t a2, int32_t a3) {
    int32_t v1 = g1; // 0x9d001d64
    int32_t v2;
    g1 = &v2;
    if (a1 == 0) {
        // 0x9d001d80
        // branch -> 0x9d001df0
        // 0x9d001df0
        g1 = v1;
        return a2 + 1;
    }
    int32_t v3 = a1 - 1; // 0x9d001da4
    int32_t result; // 0x9d001e04_11
    if (a2 == 0) {
        // 0x9d001d9c
        result = function_9d001d5c(v3, 1, a3);
        // branch -> 0x9d001df0
    } else {
        // 0x9d001dc0
        result = function_9d001d5c(v3, function_9d001d5c(a1, a2 - 1, a3), a3
           ↪ );
        // branch -> 0x9d001df0
    }
    // 0x9d001df0
    g1 = v1;
    return result;
}

// Address range: 0x9d001e0c - 0x9d001eab
void function_9d001e0c(int32_t a1, int32_t a2, int32_t a3) {
    int32_t v1;
    g1 = &v1;
    int32_t v2 = 0; // bp-20
    int32_t v3 = 0; // bp-16
    scanf("%d␣%d", &v2, &v3);
    int32_t v4 = function_9d001d5c(v2, v3, (int32_t)&v3); // 0x9d001e60
    printf("ackerman(␣%d␣,␣%d␣)␣=␣%d\n", v2, v3, v4);
}
```

Výstup 6.5: Výsledok spätného prekladu programu pre výpočet Ackermannovej funkcie. Použitá architektúra: PIC32. Bajtové usporiadanie: little endian. Optimalizácie: žiadne.

```c
// --------------------- Global Variables ---------------------

int32_t g1 = 0; // eax
int32_t g2 = 0; // ebp

// ------------------------ Functions -------------------------

// Address range: 0x8048370 - 0x804839f
int32_t entry_point(int32_t a1, int32_t a2) {
    // 0x8048370
    unknown_8048350();
    return g1;
}

// Address range: 0x80483a0 - 0x80483af
int32_t function_80483a0(void) {
    // 0x80483a0
    int32_t result;
    return result;
}

// Address range: 0x804846b - 0x80484c6
int32_t function_804846b(int32_t a1, int32_t a2) {
    // 0x804846b
    if (a1 == 0) {
        // 0x8048477
        // branch -> 0x80484c5
        // 0x80484c5
        return a2 + 1;
    }
    // 0x804847f
    int32_t result; // bp+123
    if (a2 == 0) {
        // 0x8048485
        result = function_804846b(a1 - 1, 1);
        // branch -> 0x80484c5
    } else {
        // 0x804849b
        result = function_804846b(a1 - 1, function_804846b(a1, a2 - 1));
        // branch -> 0x80484c5
    }
    // 0x80484c5
    return result;
}

// Address range: 0x80484c7 - 0x804853f
int32_t function_80484c7(void) {
    // 0x80484c7
    unknown_8048360();
    int32_t result = function_804846b(0, 0); // 0x8048510
    unknown_8048330();
    return result;
}
```

Výstup 6.6: Výsledok spätného prekladu programu pre výpočet Ackermannovej funkcie. Použitá architektúra: x86. Bajtové usporiadanie: little endian. Optimalizácie: žiadne.

Kapitola 7

Spätný preklad súborov formátu Portable Executable

Podpora formátu Portbale Executable už v rámci spätného prekladača existuje a je založená na knižnici *PeLib*. Nejdená sa teda o pridanie podpory doteraz nedpodporovaného formátu ale o znovu-implementáciu už existujúceho modulu pomocou novej knižnice *object*, ktorá je súčasťou projektu LLVM.

7.1 Požiadavky

Pre úspešný spätný preklad je nutné načítať sekcie, s ktorými môžu ďalšie časti prekladača ďalej pracovať, a implementovať všetky metódy popísané v podkapitole 4.3. Ďalej je pre zachovanie kvality doterajších výsledkov prekladu nutné spracovať tabuľky symbolov, importov a exportov. Tiež je nutné implementovať rozhranie, predovšetkým pre nástroj *fileinfo*, ktoré bolo vytvorené v rámci doterajšieho vývoja za pomoci pôvodnej knižnice.

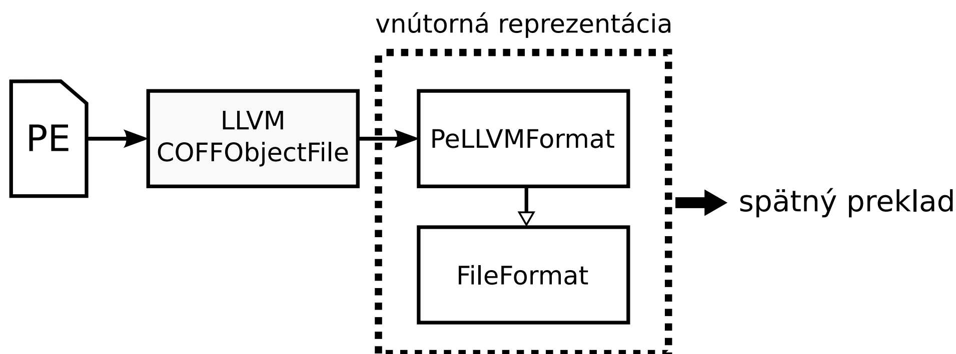
7.2 Návrh

Snaha implementovať podporu formátu PE pomocou LLVM triedy `COFFObjectFile` má dva hlavné dôvody:

- **Knižnica PeLib je už zastaralá a vývoj zastavený** – v knižnici sa často používajú staré konštrukcie jazyka C++, čo komplikuje údržbu či prípadný ďalší vývoj. Formát PE tiež prešiel od zastavenia vývoja knižnice niekoľkými drobnými zmenami, knižnica je tak zastaralá aj po funkčnej stránke. Projekt LLVM je aktívne vyvíjaný.
- **Je výhodné zjednotiť podporu formátov** – LLVM kód sa už používa v rámci podpory formátu COFF a bude použitý pre pridanie podpory formátu Mach-O. V prípade úspechu je tiež v budúcnosti možné prerobiť podporu formátu ELF, ktorý LLVM knižnica *object* rovnako podporuje. Toto umožňí odstránenie veľkej časti prebytočného kódu z projektu spätného prekladača.

7.2.1 Načítanie sekcií

Načítanie sekcií je možné realizovať iteráciou nad tabuľkou sekcií pomocou metód poskytnutých triedou `COFFObjectFile` LLVM knižnice *object*.



Obr. 7.1: Použitie LLVM triedy `COFFObjectFile` v rámci modulu.

Rozšírenie triedy PeCoffSection

Trieda `PeCoffSection` reprezentuje v rámci spätného prekladača jednu sekciu formátu PE alebo COFF. Jej rozšírenie je nutné pre potreby nástroja *fileinfo*. Ten v súčastnosti pre zistenie chýbajúcich informácií používa knižnicu *PeLib* nezávisle od knižnice *fileformatl*. Rozšírenie sa týka informácií o relokačných záznamoch a riadkovaní.

7.2.2 Načítanie symbolov, exportov a importov

Pre načítanie importov a exportov stačí použiť rozhranie LLVM triedy `COFFObjectFile`. Trieda poskytuje podporu aj pre tzv. oneskorené importy (anglicky delayed imports). Následne stačí previesť LLVM dátové typy do vnúturnej reprezentácie spätného prekladača.

7.2.3 Detekčné metódy

Väčšinu informácií potrebných pre implementáciu detekčných metód je možné nájsť v príslušných hlavičkách formátu PE alebo za pomoci príslušných metód triedy `COFFObjectFile`. Potrebné hlavičky je vhodné načítať hneď v konštruktore, detekčné metódy môžu následne z týchto štruktúr čerpať. Zaujímavosťou je použitie LLVM typov `ulittle16`, `ulittle32` a `ulittle64`, ktoré reprezentujú bezznamienkové čísla uložené s usporiadaním little endian. V prípade prevodu na štandardné typy použité v rámci spätného prekladača je tak nutné použiť explicitnú typovú konverziu.

Veľkosť tabuľky reťazcov

Pozíciu tabuľky reťazcov v súbore je možné získať pomocou odpovedajúcej metódy triedy `COFFObjectFile`. Následne je nutné načítať prvé štyry bajty uchovávajúce celkovú veľkosť tabuľky. Tie môžme načítať za pomoci funkcie pre interpretáciu binárnych dát rozhrania triedy `FileFormat`.

Zistenie počtu relokačných blokov a záznamov

Informácie o relokačných záznamoch sú uložené v rámci jednotlivých sekcií, celkový počet je možné získať iteráciou nad tabuľkou symbolov a sčítavaním počtu záznamov v jednotlivých sekciách. Relokačné bloky je možné spočítať pomocou odpovedajúcich iterátorov.

7.2.4 Rich Header

Rich Header je špeciálna štruktúra, ktorá sa vyskytuje medzi DOS a PE hlavičkami v niektorých súboroch formátu PE. Štruktúra popisuje použité nástroje *Windows VC++* a informácie je nutné najskôr dekódovať [9]. Štruktúra nie je oficiálne dokumentovaná a chýba aj podpora v rámci LLVM. Jej obsah je možné použiť napr. pre odhalenie škodlivého softvéru. Pre implementáciu analýzy Rich Header je možné použiť časť existujúceho kódu, ktorý je založený na knižnici *PeLib*, je však nutné nahradiť *PeLib* špecifické dátové typy a zastaralé funkcie za vhodné štandardné alternatívy.

7.3 Implementácia

Implementácia nového modulu sa nachádza v triede `PeLLVMFormat`. Ďalej bola pridaná trieda `RichHeaderLLVM` a upravená trieda `PeCoffSection`.

- `PeLLVMFormat` – dedí základnú triedu `FileFormat` a implementuje povinné rozhranie pre potreby spätného prekladača. Okrem metód, ktorých implementácia je povinná, tiež trieda kopíruje a implementuje rozhranie pôvodnej triedy `PeFormat`. Ďalej bolo pridaných niekoľko metód potrebných pre nástroj *fileinfo*, ktoré nahradia predchádzajúce priame použitie (mimo rozhrania `PeFormat`) knižnice *PeLib*.
- `RichHeaderLLVM` – implementuje načítanie štruktúry Rich Header. Jedná sa prevažne o pôvodný kód upravený pre fungovanie nezávislé na knižnici *PeLib*.
- `PeCoffSection` – bola rozšírená o členy pre uchovanie informácií o riadkovaní a relokačných záznamoch a o príslušné `get` a `set` metódy.

7.4 Testovanie

Pre testovanie podpory nie je nutné implementovať žiadne nové testy. Súčasťou pôvodnej implementácie je viac než tisíc regresných testov pokrývajúcich spätný preklad aj nástroj *fileinfo*. Výsledky týchto testov rozoberá nasledujúca podkapitola. V rámci implementácie podpory súborov formátu PE pomocou LLVM nebola pridaná žiadna nová funkcionalita, ktorá by vyžadovala vytvorenie nových testov.

7.5 Dosiahnuté výsledky

Vytvorený modul je schopný spracovať vstupný súbor formátu Portable Executable a poskytnúť informácie pre spätný preklad alebo nástroj *fileinfo*. Spätný preklad je možný v kvalite dostatočnej pre znovu-preloženie výsledku bežným prekladačom. Výstup 7.1 obsahuje ukážku spätného prekladu programu pre výpočet faktoriálu. Pôvodný zdrojový kód je možné nájsť v prílohe B.2. Z vyše 1200 dostupných testov skončí úspechom veľká väčšina – približne 95 %. Neúspech zvyšných testov (celkom 67) pramení z chýb a nedostatkov LLVM knižnice *object*. Nasledujúce podkapitoly rozoberajú nájdené problémy a chyby použitej knižnice a navrhujú možné riešenia.

```
#include <stdint.h>
#include <stdio.h>

// ------------------- Function Prototypes --------------------

int32_t _factorial(int64_t a1);

// ------------------------ Functions -------------------------

// Address range: 0x401560 - 0x40159f
int32_t _factorial(int64_t a1) {
    int32_t v1 = a1;
    int32_t result;
    if (v1 != 0) {
        // 0x40157b
        result = _factorial((int64_t)(v1 - 1)) * v1;
        // branch -> 0x401598
    } else {
        result = 1;
    }
    // 0x401598
    return result;
}

// Address range: 0x4015a0 - 0x40161f
int main(int argc, char ** argv) {
    int32_t v1;
    if (scanf("%d", &v1) != 1) {
        // 0x401607
        return 0;
    }
    printf("factorial(␣%d␣)␣=␣%d\n", v1, _factorial((int64_t)v1));
    while (scanf("%d", &v1) == 1) {
        // 0x4015d8
        printf("factorial(␣%d␣)␣=␣%d\n", v1, _factorial((int64_t)v1));
        // continue -> 0x4015d8
    }
    // 0x401607
    return 0;
}
```

Výstup 7.1: Výsledok spätného prekladu programu pre výpočet faktoriálu.

7.5.1 Chybné načítanie importov

LLVM metódy pre načítanie importov obsahujú hneď niekoľko rozličných chýb.

Prvým problémom je iterácia cez tabuľku importov (anglicky Import Directory Table), ktorá je súčasťou adresára importov (anglicky Import Directory). Táto tabuľka obsahuje zoznam štruktúr, ktoré slúžia pre načítanie importov a je ukončená so štruktúrou, ktorej polia sú nastavené na nulu [18]. LLVM knižnica však používa odlišný prístup. Na základe celkovej veľkosti adresára importov a veľkosti jedného vstupu tabuľky importov získa ich počet, pomocou ktorého potom vykonáva iteráciu. Tabuľka importov však nemusí byť jediným obsahom adresára importov – niektoré nástroje môžu miesto využiť napr. pre dáta samotných importov. Následkom je, že iterácia prejde za hranicu tabuľky importov, čo vo

väčšine prípadov skončí pádom aplikácie kvôli interpretácií nesprávnych dát a následnému neoprávnenému prístupu do pamäti. Dočasným riešením problému v rámci tejto práce bolo implementovanie vlastnej kontroly, ktorá odhalí terminálnu štruktúru a včasne zastaví iteráciu. Ideálnym riešením by bola oprava použitého algoritmu v rámci samotnej knižnice.

Druhým problém vzniká pri niektorých orezaných (anglicky stripped) alebo komprimovaných binárnych súboroch. V rámci jedného vstupu tabuľky importov okrem iného nájdeme odkazy na tabuľku vyhľadávania importov (anglicky Import Lookup Table) a tabuľku adries importov (anglicky Import Address Table). Pred načítaním binárnych súborov do pamäti musí byť obsah oboch tabuliek totožný. Pri načítaní dôjde k prepísaniu tabuľky adries novými dátami [18]. Vzhľadom na dvojitú redundanciu týchto dát môže dojsť k odstráneniu jednej z tabuliek, väčšinou tabuľky vyhľadávania importov. LLVM pri načítaní používa iba tabuľku vyhľadávania importov a nekontroluje správnosť odkazu na ňu, čo je zároveň aj príčina pádu aplikácie pri predošlom probléme. Ideálne riešenie by bolo rozšíriť funkcionalitu knižnice *object* tak, aby v prípade chýbajúcej tabuľky vyhľadávania bolo možné použiť tabuľku adries. Štruktúra tejto tabuľky je identická, ďalšie modifikácie tak nie sú nutné. Ďalším možným riešením je vlastná implementácia načítania importov.

Tretí problém vzniká pri určovaní názvu importov. Niektoré dynamicky linkované funkcie nie sú identifikované ich menom ale tzv. poradovým číslom (anglicky ordinal number). V takomto prípade import nemusí obsahovať vlastný názov. Ak import názov nemá, LLVM metóda vráti názov predchádzajúceho importu namiesto prázdneho reťazca, čo komplikuje spätný preklad a výrazne zhoršuje jeho kvalitu.

7.5.2 Chybné načítanie oneskorených importov

Rozhranie pre načítanie oneskorených importov je prakticky nepoužiteľné. Okrem rovnakej logickej chyby pri čítaní tabuľky oneskorených importov (anglicky Delay-Load Directory Table) tiež nefunguje operácia `pre-increment`. Pokus o použitie pripraveného C++ `range-based for` cyklu skončí pádom aplikácie kvôli neoprávnenému prístupu do pamäti. Možným riešením je rozsiahla oprava LLVM implementácie alebo vlastná implementácia načítania oneskorených importov.

7.5.3 Kontrola vstupných súborov

Pri konštrukcii objektu triedy `COFFObjectFile` dochádza ku kontrole niektorých kľúčových vlastností súboru, ktoré časť testovaných vzoriek nespĺňa. Tieto prípady je možné rozdeliť do dvoch skupín:

- **Nesprávne vyhodnotenie korektného súboru** – špecifikácia formátu je vo viacerých prípadoch voľnejšia a rôzne prekladače môžu zvoliť odlišný prístup. Príkladom takejto situácie môže byť špecifikácia obsahu adresára importov a súvisiaci problém spomenutý v podkapitole 7.5.1.
- **Správne vyhodnotenie nekorektného súboru** – toto síce nie je možné považovať za chybu, nejedná sa však o žiadúcu vlastnosť vzhľadom na zameranie spätného prekladača. Prekladač musí byť schopný analyzovať škodlivý softvér a tiež poškodené alebo zámerne modifikované binárne súbory.

7.5.4 Zhrnutie

Na základe vyššie uvedených informácii je možné usúdiť, že použitie LLVM knižnice *object* v jej aktuálnom stave nie je v prípade formátu PE úplne ideálne riešenie. Pre plnohodnotné použitie bude nutné vykonať rozsiahle opravy LLVM implementácie, prípadne implementovať vlastné riešenia vzniknutých problémov. Vzhľadom na niektoré ďalšie vlastnosti LLVM implementácie tiež pravdepodobne nebude úplne možné odstrániť použitie knižnice *PeLib*.

Kapitola 8

Spätný preklad súborov formátu Mach-O

Pridanie modulu pre formát Mach-O je významný krok pre projekt spätného prekladača, uzavrie sa tým podpora trojice najpoužívanejších formátov súčastnosti – PE, Mach-O a ELF. Návrh a implementáciu modulu však do značnej miery komplikuje nedostatok kvalitnej oficiálnej dokumentácie. Práca tak často vychádza z neoficiálnych zdrojov alebo dostupných zdrojových kódov spoločnosti Apple Inc.

8.1 Požiadavky

Pre úspešný spätný preklad je nutné načítať segmenty a sekcie a implementovať všetky metódy popísané v podkapitole 4.3. Ďalej je pre zlepšenie výsledkov prekladu vhodné spracovať tabuľku symbolov, importov a exportov.

8.2 Návrh

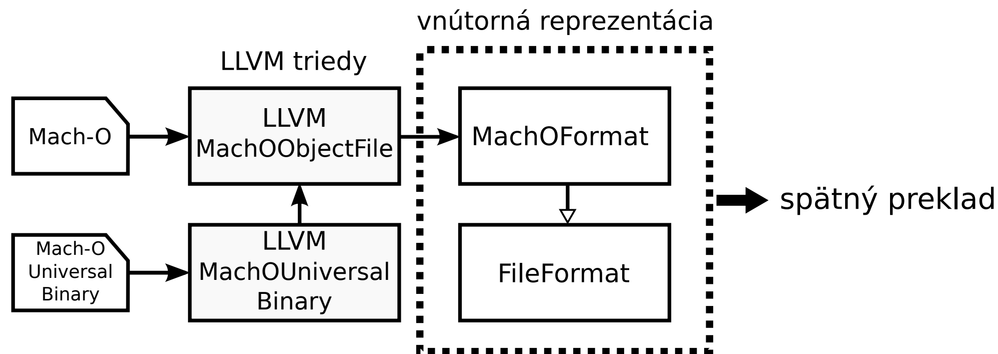
Pre pridanie podpory formátu Mach-O bola zvolená LLVM knižnica *object*, konkrétne triedy `MachOObjectFile` a `MachOUniversalBinary`. Toto rozhodnutie má hneď niekoľko výhod:

- **LLVM používa samotné Apple Inc.** – primárnym prekladačom používaným s operačným systémom OS X je Clang (založený na LLVM), prípadne LLVM GCC (LLVM prekladač s GCC front-end rozhraním). Apple Inc. sa tiež aktívne podiela na vývoji LLVM projektu [3].

- **Knižnica sa už v rámci spätného prekladača používa** – LLVM je v rámci predspracovania použité pre analýzu formátu COFF. V ostatných častiach spätného prekladača je LLVM zastúpené ešte väčším podielom.

- **Podpora Mach-O Universal Binaries** – LLVM poskytuje jednoduché rozhranie pre spracovanie univerzálnych binárných súborov.

Nevýhodou môže byť chýbajúca podpora pre konštrukcie objavujúce sa v starších formátoch binárnych súborov, ktoré už dnes nie sú využívané. Táto podpora však chýba aj v ostatných knižniciach. Podpora týchto konštrukcií je nutná vzhľadom na charakter vyvíjaného produktu – musí byť schopný spätne preložiť aj staršie binárne súbory.



Obr. 8.1: Použitie LLVM tried v rámci modulu pre spracovanie formátu Mach-O.

8.2.1 Zistenie bitovej šírky a usporiadania bajtov

Vzhľadom na to, že konštruktor triedy `MachOObjectFile` potrebuje poznať bitovú šírku (32 alebo 64 bitov) a usporiadanie bajtov (little alebo big endian), je nutné túto informáciu získať vopred. Na tento účel je možné použiť rozhranie rodičovskej triedy `FileFormat` a načítať prvé štyri bajty obsahujúce signatúru. Na základe signatúry dôjde k nastaveniu vnútorného stavu, ktorý je použitý pre správne volanie konštruktoru a niektorých detekčných metód.

8.2.2 Podpora univerzálnych binárnych súborov

V prípade, že sa jedná o univerzálny súbor, je nutné najskôr vytvoriť inštanciu triedy `MachOUniversalBinary`. Tá poskytuje možnosť iterácie nad hlavičkami dostupných architektúr a umožňuje vybrať dáta konkrétnej architektúry. Spätný prekladač nemôže prekladať viacero súborov súčasne, je tak nutné zvoliť práve jeden súbor. Vzhľadom na množinu spätným prekladačom podporovaných architektúr a možných analýz modul preferuje 32-bitové verzie architektúr v nasledujúcom poradí: x86, ARM, PowerPC.

8.2.3 Spracovanie príkazov pre načítanie vstupného súboru

V rámci LLVM verzie 3.8 je dostupný pred-pripravený zoznam príkazov obsiahnutých v súbore. Túto verziu LLVM však nie je možné momentálne nasadiť vzhľadom na prebiehajúce práce na iných častiach spätného prekladača, ktoré závisia na rozhraní verzie 3.6. Načítanie príkazov je tak nutné vykonať vlastnoručne. Pri čítaní akýchkoľvek štruktúr priamo zo súboru je nutné brať ohľad na bajtové usporiadanie a konvertovať ho pomocou nato určených makier, ktoré sú rovnako súčasťou projektu LLVM.

Načítanie segmentov a sekcií

Pre načítanie segmentov a sekcií slúžia príkazy `LC_SEGMENT` a `LC_SEGMENT_64` popísané v podkapitole 5.6. Príkazy obsahujú všetky potrebné informácie pre prevod segmentov a sekcií do vnútornej reprezentácie. Popis sekcií obsahuje tiež informácie, ktoré presahujú rozsah vnútornej reprezentácie sekcie a je preto vhodné pridať rozširujúcu triedu, ktorá tieto informácie uchová, napr. pre použitie v nástroji *fileinfo*.

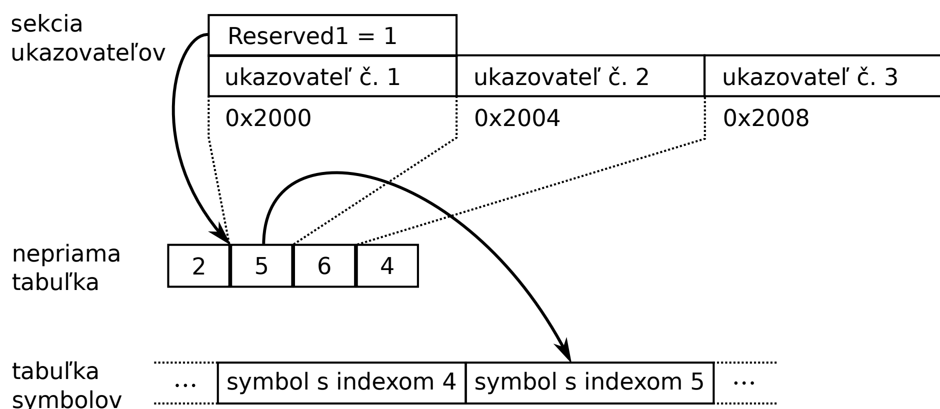
Načítanie tabuľky symbolov

Pre načítanie symbolov slúži príkaz LC_SYMTAB popísaný v podkapitole 5.6. Tabuľka symbolov obsahuje lokálne symboly ale aj exporty a importy uložené formou poľa štruktúr typu nlist. Pre reprezentáciu Mach-O symbolu je vhodné vytvoriť triedu, ktorej inštancie budú dočasne uchovávať informácie odpovedajúce práve jednému symbolu. Trieda tiež bude obsahovať metódy, ktorú budú schopné podľa potreby interpetovať symbol buď ako všeobecný symbol, import alebo export.

Načítanie importov a exportov

Postup pre načítanie importov a exportov sa líši podľa verzie operačného systému. Od verzie OS X 10.8 je možné využiť príkaz LC_DYLD_INFO, ktorý je preferovaným zdrojom. Pre spracovanie stačí použiť príslušné metódy triedy MachOObjectFile a previesť dáta do vnútornej reprezentácie spätného prekladača.

V prípade starších binárnych súborov je nutné použiť príkaz LC_DYSYMTAB. Ten popisuje, ktorá časť tabuľky symbolov obsahuje lokálne symboly, ktorá importy a ktorá exporty a tiež pozíciu a polohu tzv. nepriamej tabuľky (anglicky Indirect Table). Jednotlivé symboly je tak možné interpretovať z dočasnej reprezentácie navrhnutej v predchádzajúcej sekcii. Problém nastáva pri importoch, ktoré v tabuľke symbolov nemajú uloženú virtuálnu adresu. Bez adresy nebude možné správne pomenovať volané funkcie v spätne preloženom kóde. Ukazovatele na volané funkcie, ktorých adresu je potrebné získať, sa nachádzajú v špeciálnych sekciách __la_symbol_ptr resp. __nl_symbol_ptr, označené príznakovým bitom S_LAZY_SYMBOL_POINTERS resp. S_NON_LAZY_SYMBOL_POINTERS [1]. Počet ukazovateľov je možné zistiť pomocou veľkosti jedného ukazovateľa (32 alebo 64 bitov) a celkovej veľkosti sekcie. Každá takáto sekcia má nastavený člen Reserved1, ktorý označuje, od ktorého indexu sú záznamy nepriamej tabuľky paralelné s ukazovateľmi danej sekcie. Nepriama tabuľka obsahuje indexy do tabuľky symbolov pre daný ukazovateľ [17]. Túto schému ilustruje obrázok 8.2. Pre načítanie je teda nutné nájsť odpovedajúcu sekciu, vykonať iteráciu nad ukazovateľmi, ktoré obsahuje, a priradiť adresy jednotlivých ukazovateľov symbolom označených nepriamou tabuľkou.



Obr. 8.2: Spôsob uloženia importov v starších súboroch formátu Mach-O.

8.2.4 Návrh detekčných metód

V rámci funkcionality detekčných metód stačí väčšinou správne interpretovať informácie obsiahnuté v hlavičke súboru (typ formátu, architektúra) prípadne využiť vnútorný stav modulu (bitová šírka, usporiadanie bajtov). Výnimkou je získanie adresy a pozície vstupného bodu v prípade spustiteľného súboru a meno použitého formátu.

Tvorba mena formátu súboru

Komplikácia v tomto prípade nastáva, ak sa jedná o formát Mach-O Universal Binary. Všetky skúšané nástroje na určovanie typu súboru (napr. štandardný UNIX nástroj *file*), pridávajú k menu formátu aj zoznam architektúr, ktoré sa v súbore nachádzajú. V detekčnej metóde `getFileFormatName` tak dôjde k iterácií nad hlavičkami dostupných architektúr a pridaním ich názvu do výsledného reťazca.

Získanie vstupného bodu programu

Postup pre získanie vstupného bodu sa líši podľa verzie operačného systému. Od verzie OS X 10.6 je možné využiť príkaz `LC_MAIN`, ktorý priamo obsahuje informáciu o pozícii vstupného bodu v súbore, pre získanie adresy stačí konvertovať pozíciu pomocou už načítaných segmentov.

V prípade starších binárnych súborov je možné použiť príkaz `LC_UNIXTHREAD`. Jeden z údajov, ktorý poskytuje, je obsah registra Instruction Pointer resp. Program Counter v závislosti od architektúry. Tieto registre obsahujú adresu nasledujúcej inštrukcie. V prípade ešte nespusteného binárneho súboru sa teda jedná o vstupný bod programu. Pozíciu v súbore je možné získať prevodom adresy pomocou už načítaných segmentov. Pozícia týchto registrov závisí od cieľovej architektúry. Hlavičkové súbory obsahujúce štruktúry reprezentujúce rozloženie registrov obsahujú množstvo zbytočných informácii a nemá význam pridávať ich do projektu celé (každá architektúra má svoj vlastný hlavičkový súbor). Problém je možné riešiť výpočtom pozície kľúčového registra v rámci príkazu a vhodným uchovaním tejto hodnoty v zdrojovom kóde.

8.2.5 Detekcia zašifrovaných binárnych súborov

Zašifrované binárne súbory nie je možné preložiť, je ich preto nutné identifikovať skôr než preklad začne. K odhaleniu šifrovania je možné použiť príkaz `LC_ENCRYPTION_INFO` resp. `LC_ENCRYPTION_INFO_64`. Po nájdení príkazu je ešte nutné skontrolovať pole `id`. V prípade, že je pole `id` nastavené na nulu, šifrovanie nie je použité.

8.3 Implementácia

Implementácia nového modulu sa nachádza v triede `MachOFormat`. Ďalej boli pridané triedy `MachOSection` a `MachOSymbol`.

- `MachOFormat` – dedí základnú triedu `FileFormat` a implementuje povinné detekčné metódy a načítanie segmentov, sekcií, symbolov, exportov a importov. Uchováva tiež informácie o polohe registra IP alebo PC v príkaze `LC_UNIXTHREAD`.
- `MachOSection` – dedí triedu `Section` a rozširuje ju pre potreby reprezentácie Mach-O špecifických informácií.

- `MachOSymbol` – predstavuje jeden záznam Mach-O tabuľky symbolov. Implementuje metódy `getAsSymbol`, `getAsExport` a `getAsImport`, ktoré umožnia konverziu na jednotlivé triedy vnútornej reprezentácie.

8.4 Testovanie

Testovanie je zabezpečené sadou 38 regresných testov, ktoré pokrývajú rôzne kombinácie typov súborov a architektúr. Zároveň sa testujú oba spôsoby načítania importov a získania vstupného bodu programu. Niekoľko testov je tiež vyhradených univerzálnym či zašifrovaným binárnym súborom. Testy kontrolujú výstup programu *fileinfo*, ktorý bol rozšírený pre potreby testovania, alebo kvalitu kódu spätného prekladu. Testy sú napísané v jazyku Python 3 za použitia interného rozhrania `regression_tests`. Vstupom pre testy sú buď automaticky generované súbory alebo súbory získané od používateľov online služby pre spätný preklad.

8.4.1 Overenie správnosti načítania importov

Vzhľadom nato, že implementácia načítania importov v starších súboroch vychádza z neoficiálnych zdrojov, bolo nutné tento postup dôkladne overiť. Pre tento účel poslúžili novšie binárne súbory, ktoré umožňujú použiť oba postupy. Správnosť je možné overiť striedaním použitých postupov a porovnaním výsledkov, ktoré musia byť identické.

8.5 Dosiahnuté výsledky

Pomocou modulu je možné spätne preložiť súbory formátu Mach-O a niektoré súbory formátu Mach-O Universal Binary alebo poskytnúť potrebné informácie nástroju *fileinfo*. Všetky navrhnuté testy skončia úspechom.

8.5.1 Identifikácia zašifrovaných súborov

Modul je schopný úspešne identifikovať zašifrované vstupné súbory a poskytnúť potrebné varovanie ďalším nástrojom. V súboroch získaných z online služby pre spätný preklad bolo nájdených celkom 19 (2,52 %) zašifrovaných súborov. Jedná sa prevažne o univerzálne binárne súbory alebo aplikácie pre operačný systém iOS.

8.5.2 Výsledky spätného prekladu

Výstup 8.3 demonštruje úspech spätného prekladu z architektúry x86. Vo výstupe je možné vidieť telo funkcie `_ack`, ktorá počíta výsledok Ackermannovej funkcie čiastočne iteratívne, čo je spôsobené optimalizáciou použitého prekladača. Vo funkcii `function_1eee`, ktorú pravdepodobne vygeneroval prekladač, je vidieť postupné volania funkcií `scanf`, `_ack` a `printf`. Pôvodný zdrojový kód je možné nájsť v prílohe B.1.

Výstup 8.1 demonštruje úspech spätného prekladu z architektúry PowerPC. Vo výstupe je možné vidieť volanie funkcie `_NSApplicationMain`. Podľa [2] sa jedná o základnú inicializačnú funkciu pre spustenie Objective C aplikácií. V ukážke chýba správna rekonštrukcia argumentov funkcie, čo je spôsobené chýbajúcou podporou konštrukcií jazyka Objective C v ďalších fázach spätného prekladu. Jedná sa o súbor získaný z online služby pre spätný preklad, pôvodný zdrojový kód tak nie je k dispozícii.

```
#include <stdint.h>

// Address range: 0x34d0 - 0x36cf
int main(int argc, char ** argv) {
    // 0x34d0
    _NSApplicationMain();
    return argc;
}
```

Výstup 8.1: Výsledok spätného prekladu binárneho súboru s architektúrou PowerPC.

8.5.3 Univerzálne súbory

Za pomoci modulu je možné spätne preložiť univerzálne binárne súbory za predpokladu, že obsahujú súbory formátu Mach-O. Toto však neplatí u približne 60 % binárnych súborov získaných z online služby pre spätný preklad, ktoré obsahujú celé statické knižnice zabalené programom *archiver*. Problémom takto vytvorených statických knižníc okrem chýbajúcej podpory zo strany LLVM je, že môžu obsahovať hneď niekoľko objektových súborov vhodných pre preklad. Takýto archív bude nutné najskôr rozbaliť a postupne spätný preklad aplikovať na získané Mach-O súbory. Toto však nie je možné riešiť v rámci aktuálnej architektúry knižnice `fileformatl`, ktorá nie je pripravená reprezentovať viac súborov súčasne.

Výstup 8.4 obsahuje výsledok spätného prekladu univerzálneho binárneho súboru s programom pre výpočet faktoriálu. Vo výstupe je možné vidieť telo funkcie `_factorial`, ktorá počíta výsledok faktoriálu čísla `a1` rekurzívne. Vo funkcii `function_1eec`, ktorú pravdepodobne vygeneroval prekladač, je vidieť postupné volania funkcií `scanf`, `_factorial` a `printf`, čo takmer odpovedá volaniam funkcií v pôvodnom zdrojovom kóde. Rozdelenie pôvodne jedného cyklu na jednu podmienku a následný cyklus je spôsobené optimalizáciou použitého prekladača. Pôvodný zdrojový kód je možné nájsť v prílohe B.2. Vstupný súbor obsahuje architektúry x86 a x86-64. Príklad tak zároveň demonštruje správny výber preferovanej architektúry, keďže architektúra x86-64 zatiaľ nie je spätným prekladačom podporovaná.

Výstup 8.2 demonštruje spracovanie zašifrovaného univerzálneho binárneho súboru nástrojom *fileinfo*. Názov formátu obsahuje mená architektúr, ktoré je možné nájsť v súbore. Nasledujúce informácie sa však týkajú už konkrétnej zvolenej architektúry. Zaujímavé je predovšetkým posledné varovanie, ktoré upozorňuje na zašifrovaný obsah súboru.

```
File format               : Mach-O Universal Binary: [armv7] [armv7s] [arm64]
File class                : 32-bit
File type                 : Executable file
Architecture              : ARM (little endian)
Endianness                : Little endian
Entry point address       : 0x11655
Entry point offset        : 0x11655
Entry point section name  : __text
Entry point section index: 0
Overlay offset            : 0x3ed40
Overlay size              : 0x88140
Warning: Unknown compiler or packer.
Warning: Information about symbols, sections etc. is shown for one architecture.
Warning: This file is encrypted.
```

Výstup 8.2: Ukážkový výstup nástroja *fileinfo*.

```
int32_t g1 = 0; // eax
int32_t g2 = 0; // ebp
int32_t g3 = 0; // esi

// Address range: 0x1e90 - 0x1edf
int32_t _ack(int32_t a1, int32_t a2) {
    // 0x1e90
    if (a1 == 0) {
        // branch -> 0x1ed5
    } else {
        int32_t v1; // 0x1ecb
        while (true) {
            int32_t v2 = a1 - 1; // 0x1eb0
            g3 = v2;
            v1 = 1;
            if (a2 != 0) {
                // 0x1ebc
                _ack(a1, a2 - 1);
                v1 = 1;
                v2 = 0;
                // branch -> 0x1ec9
            }
            // 0x1ec9
            if (v2 == 0) {
                // break -> 0x1ed5
                break;
            }
            a1 = v2;
            a2 = v1;
            // continue -> 0x1eb0
        }
        // 0x1ed5
        a2 = v1;
        // branch -> 0x1ed5
    }
    // 0x1ed5
    int32_t v3;
    g2 = v3;
    return a2 + 1;
}

// Address range: 0x1ee0 - 0x1eed
int main(int argc, char ** argv) {
    int32_t v1;
    g2 = &v1;
    int32_t v2;
    function_1eee((char *)v2, 0, 0, 0, 0, 0, 0, g3, 0);
    return g1;
}

// Address range: 0x1eee - 0x1f56
void function_1eee(char * a1, int32_t a2, int32_t a3, int32_t a4, int32_t a5,
    ↪ int32_t a6, int32_t a7, int32_t a8, int32_t a9) {
    // 0x1eee
    int32_t v1;
    *(int32_t *)(g2 - 24) = v1;
    *(int32_t *)(g2 - 16) = 0;
    *(int32_t *)(g2 - 20) = 0;
    scanf((char *)(v1 + 160));
    int32_t v2 = g2; // 0x1f1c
    int32_t v3 = _ack(*(int32_t *)(v2 - 16), *(int32_t *)(v2 - 20)); // esi
    printf((char *)(*(int32_t *)(g2 - 24) + 166));
    g1 = v3;
}
```

Výstup 8.3: Výsledok spätného prekladu programu pre výpočet Ackermannovej funkcie. Použitá architektúra: x86. Optimalizácie: O1.

```
int32_t g1 = 0; // eax
int32_t g2 = 0; // ebp

// Address range: 0x1e90 - 0x1edf
int32_t _factorial(int64_t a1) {
    int32_t v1 = a1;
    int32_t result;
    if (v1 != 0) {
        // 0x1eb5
        result = _factorial((int64_t)(v1 - 1)) * v1;
        // branch -> 0x1ed5
    } else {
        result = 1;
    }
    // 0x1ed5
    int32_t v2;
    g2 = v2;
    return result;
}

// Address range: 0x1ee0 - 0x1eeb
int main(int argc, char ** argv) {
    int32_t v1;
    g2 = &v1;
    int32_t v2;
    function_1eec((char *)v2, 0, 0, 0, 0, 0);
    return g1;
}

// Address range: 0x1eec - 0x1f65
void function_1eec(char * a1, int32_t a2, int32_t a3, int32_t a4, int32_t a5,
    ↪ int32_t a6) {
    int32_t v1 = g2; // 0x1eed
    *(int32_t *)(v1 - 8) = 0;
    *(int32_t *)(g2 - 12) = *(int32_t *)(v1 + 8);
    *(int32_t *)(g2 - 16) = *(int32_t *)(v1 + 12);
    int32_t v2;
    *(int32_t *)(g2 - 28) = v2;
    if (scanf((char *)(*(int32_t *)(g2 - 28) + 168)) != 1) {
        // 0x1f5b
        g1 = 0;
        return;
    }
    int32_t v3 = _factorial((int64_t)*(int32_t *)(g2 - 20)); // 0x1f2c
    int32_t v4 = g2; // 0x1f31
    *(int32_t *)(v4 - 24) = v3;
    *(int32_t *)(g2 - 32) = printf((char *)(*(int32_t *)(v4 - 28) + 171));
    while (scanf((char *)(*(int32_t *)(g2 - 28) + 168)) == 1) {
        // 0x1f26
        v3 = _factorial((int64_t)*(int32_t *)(g2 - 20));
        v4 = g2;
        *(int32_t *)(v4 - 24) = v3;
        *(int32_t *)(g2 - 32) = printf((char *)(*(int32_t *)(v4 - 28) + 171));
        // continue -> 0x1f26
    }
    // 0x1f5b
    g1 = 0;
}
```

Výstup 8.4: Výsledok spätného prekladu súboru s formátom Mach-O Universal Binary. Použitá architektúra: x86. Optimalizácie: O1.

Kapitola 9

Záver

V tejto práci boli predstavené základy softvérového reverzného inžinierstva, predovšetkým spätný preklad binárnych súborov. Ďalej bol v práci predstavený rekonfigurovateľný spätný prekladač vyvíjaný spoločnosťou *AVG Technologies CZ, s.r.o.* s hlavným zameraním na spracovanie vstupných súborov. Následne boli predstavené formáty Intel HEX, Portable Executable, Mach-O a Mach-O Universal Binary, ktorých moduly boli následne navrhnuté, implementované a testované.

Do spätného prekladača tak bola úspešne pridaná podpora dvoch resp. troch nových formátov – Intel HEX, Mach-O a Mach-O Universal Binary. Tiež došlo k pridaniu alternatívnej implementácie modulu pre formát Portable Executable založenej na LLVM. Spätný prekladač sa tak môže chváliť najväčším počtom podporovaných formátov nezávisle od softvéru tretích strán a tiež podporou formátov troch na trhu najviac zastúpených rodín operačných systémov – Windows, Unix a OS X respektíve iOS. Nasadenie modulov pre formáty Mach-O a Intel HEX do užívateľského vydania produktu by sa malo uskutočniť v nasledujúcich týždňoch. Modul pre formát Portable Executable bude pred nasadením vyžadovať ešte niekoľko potrebných úprav.

Testovanie overilo schopnosť modulov spracovať súbory daných formátov a poskytnúť informácie potrebné pre spätný preklad alebo nástroj *fileinfo*. V prípade modulu pre formát Mach-O tiež bola testovaná schopnosť správne identifikovať zašifrované binárne súbory a schopnosť vybrať správnu architektúru v prípade formátu Mach-O Universal Binary. Testovanie bolo vykonané na umelo vytvorených vstupných súboroch ale aj reálnych vzorkách súborov získaných z online služby pre spätný preklad.

9.1 Budúci vývoj

Vzhľadom na jednoduchosť formátu Intel HEX prakticky neexistuje priestor pre pridanie novej funkcionality v rámci modulu. Ak by v budúcnosti prišla požiadavka na implementáciu podobného ASCII formátu (napr. SREC), bude možné použiť veľkú časť už napísaného kódu. Toto však bude vyžadovať určité úpravy tried, ktoré dnes nijako nepodporujú prípadné pridanie podpory nových formátov.

V prípade ďalšieho vývoja podpory formátu PE pomocou LLVM je najskôr nutné odpovedať na dve otázky. Prvou otázkou je, či vzhľadom na rozsiahle nedostatky knižnice *object* má vôbec význam vo vývoji pokračovať. Knižnica *PeLib* má síce tiež niekoľko nedostatkov a zastaralý kód, ale už implementovaná funkcionalita funguje správne a knižnica je úspešne nasadená v rámci spätného prekladača. V prípade použitia LLVM bude nutné venovať

množstvo času implementácii základných vecí, ako napr. načítanie importov do vnútornej reprezentácie. Výhodou je, že LLVM je aktívne vyvíjaný projekt. Druhá otázka sa týka prístupu k riešeniu samotných problémov. Vlastné riešenie bude krátkodobo efektívne, prinesie však množstvo nového kódu s potrebou údržby. Oprava LLVM knižnice *object* bude výrazne pomalšia a náročnejšia, ale v prípade prijatia navrhovaných opráv ľahko udržateľná a prínosná nielen pre projekt spätného prekladača. Finálne rozhodnutie nie je v kompetencii autora tejto práce, prípadné odporúčanie je však pokračovať vo vývoji formou opráv.

V rámci formátu Mach-O je dostupných niekoľko ďalších príkazov, ktoré je možné spracovať primárne pre výpis v rámci nástroja *fileinfo* či zlepšenie kvality prekladu niektorých binárnych súborov. Veľký problém predstavuje nedostatok oficiálnej kvalitnej dokumentácie popisujúcej štruktúru formátu. Už implementácia niektorých základných častí potrebných pre preklad či zlepšenie jeho kvality vychádza z neoficiálnych článkov, prípadne analýzy dostupných zdrojových kódov OS X alebo reverzného inžinierstva.

Keďže veľké percento univerzálnych binárnych súborov formátu Mach-O tvoria statické knižnice, bude vhodné v budúcnosti pridať podporu formátu *archiver*.

Literatúra

[1] Apple Computer, Inc.: *Mach-O Runtime Architecture*. 2004.

[2] Apple Inc.: *AppKit Functions Reference* [online]. 2012. [cit. 2016-05-12]. Dostupné z: https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ApplicationKit/Miscellaneous/AppKit_Functions/.

[3] Apple Inc.: *LLVM Compiler Overview* [online]. 2012. [cit. 2016-04-02]. Dostupné z: https://developer.apple.com/library/mac/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/.

[4] AVG Technologies CZ, s.r.o.: *Retargetable Decompiler* [online]. 2016. [cit. 2016-01-15]. Dostupné z: https://retdec.com/home/.

[5] Backer Street Software: *REC Studio 4 - Reverse Engineering Compiler* [online]. 2016. [cit. 2016-01-15]. Dostupné z: http://www.backerstreet.com/rec/rec.htm.

[6] C4IT Ltd.: *C4Decompiler, the C Decompiler for Windows x64* [online]. 2016. [cit. 2016-01-15]. Dostupné z: http://www.c4decompiler.com/.

[7] Cifuents, C.: *Reverse Compilation Techniques*. Dizertačná práca, Queensland University of Technology, 1994.

[8] Cryptic Apps SARL: *Hopper FAQ* [online]. 2016. [cit. 2016-01-15]. Dostupné z: http://www.hopperapp.com/faq.html.

[9] Daniel Pistelli: *Microsoft's Rich Signature (undocumented)* [online]. 2010. [cit. 2016-04-28]. Dostupné z: http://www.ntcore.com/files/richsign.htm.

[10] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. Indianapolis: Wiley, 2005. ISBN 0-7645-7481-7.

[11] Hex-Rays SA.: *Hex-Rays Decompiler: Support* [online]. 2016. [cit. 2016-01-15]. Dostupné z: https://www.hex-rays.com/products/decompiler/support.shtml.

[12] Intel Corporation: *Hexadecimal Object File Format Specification*. 1988.

[13] Křoustek, J.: *Rekonfigurovatelná analýza strojového kódu*. Dizertačná práca, Fakulta informačních technologií VUT v Brně, 2014.

[14] Levine, J. R.: *Linkers and Loaders*. San Francisco: Morgan Kaufmann, 2000. ISBN 1-55860-496-0.

[15] LLVM Project: *LLVM 3.9 documentation* [online]. 2015. [cit. 2015-12-11]. Dostupné z: http://llvm.org/docs/.

[16] LLVM Project: *The LLVM Compiler Infrastructure Project* [online]. 2015. [cit. 2015-12-11]. Dostupné z: http://llvm.org/.

[17] *Mach-O Binaries* [online]. 2015. [cit. 2016-03-31]. Dostupné z: http://www.m4b.io/reverse/engineering/mach/binaries/2015/03/29/mach-binaries.html.

[18] Microsoft Corporation: *Microsoft Portable Executable and Common Object File Format Specification*. 2013.

[19] Net Applications: *Operating System Market Share* [online]. 2016. [cit. 2016-02-02]. Dostupné z: www.netmarketshare.com/operating-system-market-share.aspx.

[20] SmartDec Project: *About SmartDec* [online]. 2016. [cit. 2016-01-15]. Dostupné z: http://decompilation.info/about.

[21] *Snowman* [online]. 2016. [cit. 2016-01-15]. Dostupné z: https://derevenets.com/.

[22] Ďurfina, L.; Křoustek, J.; Matula, P.; aj.: A Novel Approach to Online Retargetable Machine-Code Decompilation. *Journal of Network and Innovative Computing*, ročník 2, č. 1, 2014: s. 224–232.

Prílohy

Zoznam príloh

A Obsah CD **55**

B Pôvodné zdrojové kódy **56**

Príloha A

Obsah CD

CD obsahuje:

- Zdrojové kódy vytvorené v rámci práce.
- Text práce vo formáte PDF.
- Vytvorenú testovaciu sadu.
- Súbor README.
- Makefile.

Príloha B

Pôvodné zdrojové kódy

```
#include <stdio.h>
#include <stdlib.h>

unsigned int ack(unsigned int m, unsigned int n) {
    if (m == 0) {
        return n + 1;
    }
    else if (n == 0) {
        return ack(m - 1, 1);
    }
    else {
        return ack(m - 1, ack(m, n - 1));
    }
}

int main(int argc,char *argv[])
{
    unsigned int res = 0, x = 0, y = 0;

    scanf("%d␣%d", &x, &y);
    res = ack(x ,y);
    printf("ackerman(␣%d␣,␣%d␣)␣=␣%d\n", x, y, res);
    return res;
}
```

Zdrojový kód B.1: Pôvodný zdrojový kód programu pre výpočet Ackermannovej funkcie.

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0)
        return 1;
    return n*factorial(n-1);
}


int main(int argc,char *argv[])
{
    int x;
    while(scanf("%d", &x) == 1)
    {
        int res = factorial(x);
        printf("factorial( %d ) = %d\n", x, res);
    }
    return 0;
}
```

Zdrojový kód B.2: Pôvodný zdrojový kód programu pre výpočet faktoriálu.