
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# ZPĚTNÝ PŘEKLAD APLIKACÍ PRO ARCHITEKTURU X86-64 V NÁSTROJI RETDEC

**DECOMPILATION OF X86-64 BINARIES IN RETDEC DECOMPILER**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **PETER KUBOV**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Doc. Dr. Ing. DUŠAN KOLÁŘ**
**SUPERVISOR**



**BRNO 2019**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

# Zadání bakalářské práce


Student: **Kubov Peter**
Program: Informační technologie
Název: **Zpětný překlad aplikací pro architekturu x86-64 v nástroji RetDec**
**Decompilation of x86-64 Binaries in RetDec Decompiler**
Kategorie: Překladače

Zadání:

1. Studujte problematiku reverzního inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se s x86-64 (x64) architekturou procesorů.
3. Seznamte se se zpětným překladačem RetDec společnosti Avast a s technologiemi v něm použitými (LLVM, Capstone, Keystone atd.).
4. Analyzujte současné překážky a nedostatky zpětného překladu x86-64 aplikací. Pro vylepšení podpory překladu těchto binárních souborů identifikujte komponenty které je nutné rozšířit, nebo nově implementovat.
5. Navrhněte nutná rozšíření a nové komponenty pro dosažení cíle z bodu 4.
6. Po konzultaci s vedoucím a konzultantem implementujte návrhy z předchozího bodu.
7. Vytvořené řešení důkladně otestujte sadou testů, včetně reálných programů pro danou architekturu, nebo vzorků potenciálně škodlivých programů.
8. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- Intel x86-64 Architecture Software Developer Manual.
- Dokumentace k projektům RetDec, LLVM, Capstone, Keystone atd.
- Dále dle doporučení vedoucího či konzultanta

Pro udělení zápočtu za první semestr je požadováno:

- Prvních pět bodů zadání a rozpracování šestého bodu.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Kolář Dušan, doc. Dr. Ing.**
Konzultant: Matula Peter, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 24. října 2018

# Abstrakt

Cieľom práce je implementovať podporu spätného prekladu binárnych súborov architektúry x64 pre spätný prekladač RetDec. Práca skúma prístupy reverzného inžinierstva, najmä z pohľadu informačných technológií. Zaoberá sa všobecným princípom spätných prekladačov a konkrétne prekladačom RetDec od spoločnosti Avast. Popisuje architektúru x86 a z nej odvovodenú architektúru x86-64. Výstupom práce je implementácia nových a rozšírenie existujúcich tried v jazyku C++, ktoré plnia chýbajúcu činnosť.



# Abstract

The goal of this thesis is to implement support for decompilation of x64 binary files in the RetDec decompiler. The thesis analyses different approaches to reverse engineering, mainly from the view of information technology. After a general classification of decompilers, thesis brings to attention one particular decompiler from Avast company–RetDec. The thesis also deals with the description of broadly used architecture x86, and it's descendant architecture x86-64. In result, the thesis provides new and extends existing classes in C++ to provide missing functionality.



# Kľúčové slová

Reverzné inžinierstvo, spätný preklad, prekladač, RetDec, LLVM IR, architektúra x86, x86-64, x64, ABI, volacie konvencie.



# Keywords

Reverse engineering, decompilation, compiler, RetDec, LLVM IR, architecture x86, x86-64, x64, ABI, calling conventions.



# Citácia

KUBOV, Peter. *Zpětný překlad aplikací pro architekturu x86-64 v nástroji RetDec*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Dr. Ing. Dušan Kolář

# Zpětný překlad aplikací pro architekturu x86-64 v nástroji RetDec

## Prehlásenie

Prehlasujem, že som túto bakalárskú prácu vypracoval samostatne pod vedením pána Doc. Dr. Ing. Koláře. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Peter Kubov
15. mája 2019


## Poďakovanie

Chcel by som poďakovať pánovi Doc. Dr. Ing. Kolářovi za jeho trpezlivosť, konzultácie a rady, ktoré mi pomohli pri vypracovaní tejto práce. Moja vďaka patrí aj Ing. Petrovi Matulovi za jeho odborné vedenie a pomoc pri riešení problémov.

# Obsah

# Kapitola 1

# Úvod

Exponenciálny rast počítačového výkonu premenil počítač na jednu z hlavných síl formujúcich ekonomiku a spoločnosť [17]. Výrobci počítačových architektúr súperia o to, koho architektúra ovládne svet počítačov. Zatiaľ čo trhu mobilných zariadení dominuje architektúra ARM, vo svete osobných počítačov si už dlho neporazene drží prvé miesto architektúra x86 od spoločnosti Intel. Táto architektúra procesorov stále podlieha vývoju a jej najnovším predstaviteľom je 64 bitová architektúra x86-64. Architektúra procesorov x86-64 sa nachádza vo väčšine moderných počítačov, a preto zaujala vývojarov zo spoločnosti Avast, avšak mierne z iného pohľadu.

Spoločnosť Avast už niekoľko rokov vyvíja spätný prekladač RetDec, ktorého cieľom je zjednodušiť prácu analytikom škodlivého softvéru. RetDec je nástroj schopný preložiť binárny súbor rôznych architektúr a vytvoriť z neho čitateľný súbor vo vyššom programovacom jazyku. Aktuálne chýba v spätnom prekladači podpora spracovania binárnych súborov architekúry x86-64. Cieľom tejto práce je vytvoriť a otestovať pridanie schopnosti spracovania binárnych súborov nepodporovanej architektúry x86-64.

Táto práca je rozdelená do niekoľkých logicky radených celkov. V prvej časti je zhrnutý úvod do problematiky práce a po nej nasleduje opis reverzného inžinierstva so zameraním na informačné technológie v kapitole 2. Pozornosť je venovaná oboznámeniu s nástrojmi reverzného inžinierstva a spôsobmi ich využitia v obore informačných technológií. Hlavne sa však zameriava na spätný preklad binárnych súborov.

Cieľom kapitoly 3 je oboznámiť čitateľa o existencii procesorovej architektúry x86 a z nej vychádzajúcej architektúry x86-64. V prvej časti tejto kapitoly sa nachádza všeobecný opis architekúry x86, jej registrov a sady inštrukcií. V druhej časti kapitoly je podrobnejšie rozobraná architektúra x86-64, kde je okrem základných prvkov priblížená téma aplikačného binárneho rozhrania.

V kapitole 4 je základný opis štruktúry spätného prekladača RetDec. Opisuje princíp jeho činnosti a štruktúru hlavných komponent. Po tejto kapitole nasleduje detailný opis častí, ktoré sú dôležité z hľadiska pridania podpory novej architektúry v kapitole 5. Zhrňuje prednosti a nedostatky jednotlivých častí v čase písania tejto práce. Po opise nedostatkov nástroja RetDec nasleduje kapitola 6 s návrhom potrebných rozšírení k vytvoreniu podpory architektúry x86-64. Venuje sa návrhu riešenia problémov vychádzajúcich z analýzy nedostatkov spätného prekladača.

Kapitola 7 obsahuje zobrazenie zaujímavých častí implementácie a nových rozšírení spätného prekladača, ktoré boli zavedené pri implementovaní návrhu riešenia. Zhŕňa výsledky a analyzuje nedostatky jednotlivých riešení problému. Na ňu je naviazaná kapitola 8, v ktorej je priblížený proces testovania spojený s vývojom spätného prekladača.

V závere práce je zhrnutie dosiahnutých výsledkov a zhodnotenie stavu spätného prekladača po implementovanej podpore novej architektúry. Následne je priblížená vízia možnosti ďalšej práce na spätnom prekladači.

# Kapitola 2

# Reveverzné inžinierstvo a spätný preklad

Táto kapitola oboznamuje čitateľa s existenciou reverzného inžinierstva. Dôraz je kladený na súvis reverzného inžinierstva s informačnými technológiami. Skutočnosti popisované v kapitole sú získané z [13].

Reverzné inžinierstvo je proces, v ktorom je predmet skúmania (auto, stíhačka, počítač, apod.) dekonštruovaný do takých detailov, aby bolo možné odkryť jeho vnútornú štruktúru. Účelom je odhaliť dizajn, architektúru, prípadne získať nejaké dôležité poznatky o skúmanom objekte. Prístup vykonávaný z pohľadu reverzného inžinierstva možno prirovnať k skúmaniu princípov prírodného javu. Princípy reverzného inžinierstva sú teda využívané v rôznych vedeckých disciplínach, kde sa skúmajú vlastnosti umelého, prípadne biologického objektu. Táto kapitola sa však ďalej sústreďuje na opis reverzného inžinierstva z pohľadu informačných technológií, konkrétne vo využití pri analýze softvéru.

## 2.1 Reverzné inžinierstvo v informatike

Reverzné inžinierstvo je z pohľadu informačných technológií možné pokladať za sadu dôležitých techník a nástrojov umožňujúcich zistiť skutočnú funkcionalitu skúmaného softvéru. Formálne je ho možné definovať podľa [9] ako:

> „Reverzné inžinierstvo je proces analýzy skúmaného systému pre identifikáciu komponent tohto systému a ich vzájomných vzťahov kvôli vytvoreniu reprezentácie systému vo vyššej úrovni abstrakcie.“

Vďaka tomuto procesu dokážeme lepšie vizualizovať štruktúru softvéru a odhaliť črty riadiace jeho chovanie. Reverzné inžinierstvo má v oblasti informačných technológií širokú škálu využití. V následujúcej časti kapitoly sú zhrnuté niektoré z hlavných oblastí, kde sa tento druh reverzného inžinierstva využíva najčastejšie. Následne sa venuje opisu hlavných typov nástrojov reverzného inžinierstva v informatike.

### 2.1.1 Využitia reverzného inžinierstva

Reverzné inžinierstvo je z pohľadu informačných technológií možné uplatniť v širokej škále oblastí. Následujúce príklady predstavujú jedny z hlavných a častých prípadov použitia reverzného inžinierstva v praxi.

#### Škodlivý softvér – malvér

Techniky reverzného inžinierstva sú využívané ako zo strany vývojárov malvéru, tak aj analytikov skúmajúcich tieto škodlivé programy. Vývojári malvéru využívajú tieto techniky za účelom zistenia slabín programu, ktoré by mohli využiť napríklad pre nahradenie časti kódu vlastným, alebo pre získanie moci nad systémom používateľa. Analytik na druhú stranu môže tieto techniky využiť na získanie znalostí o chovaní malvéru, ktoré bude možné využiť napríklad pre účely prevencie pred infekciou daným softvérom.

#### Kryptografia

Kryptografické algoritmy je možné rozdeliť na dve skupiny. U prvej skupiny je tajomstvom hodnota šifrovacieho kľúču a u druhej je tajomstvom samotný algoritmus. Algoritmus patriaci do druhej skupiny stráca bezpečnosť, akonáhle je odhalený jeho princíp. Práve na tieto účely je možné využiť techniky reverzného inžinierstva a rekonštruovať šifrovací algoritmus. Na druhú stranu, pri algoritmoch prvej skupiny je utajovaná hodnota kľúča, ktorého hodnota bola využitá pre šifrovanie správy. Privátne kľúče musia byť teda bezpečne uschované a algoritmy využívané pre šifrovanie správy sú zvyčajne prístupné verejnosti. Tento prístup šifrovania správ takmer úplne znemožňuje využiť techniky reverzného inžinierstva na dešifrovanie správy. Je však možné tieto techniky uplatniť pre uistenie sa, či pri implementácii algoritmu nebola do programu zanesená chyba, ktorá by ho spravila zraniteľným.

#### DRM – Správa digitálnych práv

V dnešnej dobe je väčšina obsahu s autorským právom (filmy, hudba, knihy, ...) dostupná v digitálnej forme na internete. Aby sa poskytovatelia mediálneho obsahu ochránili pred jeho neoprávneným šírením, sú vytvárané pre tento účel špecializované technológie. Tieto technológie sú označované ako správa digitálnych práv, so skratkou DRM (angl. Digital Rights Management). O tom ako DRM funguje je možné si prečítať viac v [8]. Ľudia, ktorí sa snažia prelomiť ochranu tvorenú technológiami DRM, musia najprv pochopiť ako technológie DRM fungujú. K tomu využívajú princípy reverzného inžinierstva.

#### Vývoj konkurenčných produktov

Aj napriek tomu, že reverzné inžinierstvo je možné uplatniť v oblasti informačných technológií na tvorenie konkurenčných produktov, nie je to v praxi zvykom. Takéto riešenie by pravdepodobne vyžadovalo totiž oveľa viac času a finančných prostriedkov ako vývoj nového produktu.

### 2.1.2 Nástroje reverzného inžinierstva

Nástroje sú základným prvkom reverzného inžinierstva. V tejto sekcii sú spomenuté základné typy nástrojov, ktoré sú v praxi využívané.

#### Monitorovanie systému

Pre informácie o programe na úrovni operačného systému sú využívané nástroje, ktoré ukazujú aké služby operačného systému tento program využíva. Získavajú sa tak informácie napríklad o sieťovej aktivite programu, jeho otvorených súboroch, prístup k registrom.

#### Disassembler

Predstavuje základný, najčastejšie využívaný nástroj reverzného inžinierstva. Vstupom tohto nástroju je binárny súbor, z ktorého vygeneruje jeho reprezentáciu v jazyku symbolických inštrukcií (assembler). Poväčšine sa jedná o pomerne nenáročný proces, nakoľko assembler je len textové mapovanie objektového súboru[1].

#### Debugger

Debugger nebol vytvorený pre účely reverzného inžinierstva, ale pre účely ladenia programu. Je to nástroj využívaný pre odhalenie chýb v programe, avšak z pohľadu reverzného inžinierstva je využívaný na pozorovanie behu konkrétnych častí programu, typicky v jazyku assemblera (získaný disassemblerom).

#### Spätný prekladač

Spätné prekladače sú vyššou formou disassemblerov. Sú to nástroje, ktoré sú schopné transformácie vstupného binárneho súboru do reprezentácie vyššieho programovacieho jazyka.

## 2.2 Spätný preklad

Spätný preklad je technika, pri ktorej je binárny súbor transformovaný špecifickými postup na jeho reprezentáciu vo vyššom programovacom jazyku. Nástroj, ktorý túto činnosť vykonáva je označovaný ako spätný prekladač, prípadne dekompilátor (z angl. decompiler). Pri spätnom preklade programu nie je typicky možné dosiahnuť rovnaký kód ako ten, ktorý bol napísaný pre daný program. Vychádza to už len z princípu akým fungujú prekladače. Veľa informácií o programe je prekladačom zahodených, pretože pre vykonávanie programu počítačom nie sú potrebné. Jednou z úloh spätného prekladača môže byť teda aj správne odhadnutie toho, aké informácie boli pri preklade stratené.

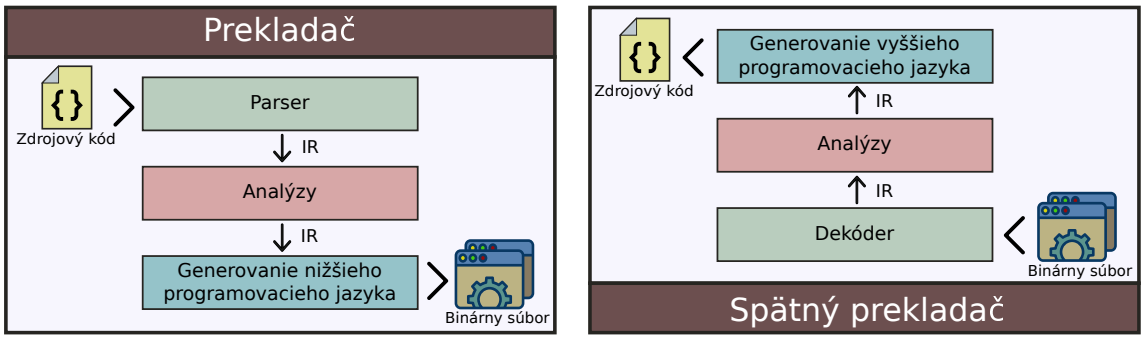
### 2.2.1 Všeobecná architektúra spätného prekladača

Architektonicky sú spätné prekladače veľmi podobné obyčajným prekladačom programovacích jazykov. Vstupom spätných prekladačov je však binárny súbor, ktorý je dekódovaný do vnútornej reprezentácie. Táto reprezentácia predstavuje medzikód, nad ktorým je možné vykonávať rôzne optimalizácie za účelom vyradenia nepodstatných informácií a vyzdvihnutia dôležitých informácií o programe. Posledná časť spätného prekladača má za úlohu transformovať vnútornú reprezentáciu na program vo vyššom programovacom jazyku. Na obrázku 2.1 je porovnanie činnosti prekladača a spätného prekladača.

### 2.2.2 Dostupné spätné prekladače

V následujúcej časti textu sú zmienené niektoré známe spätné prekladače, ktoré sú aktuálne dostupné. Okrem základných informácií o týchto nástrojoch je poskytnuté krátke zhrnutie možností a nedostatkov využitia daných spätných prekladačov.

[1] Z angl. object code – súbor obsahujúci strojový kód, spolu s ďalšími informáciami.




Obr. 2.1: Porovnanie činnosti prekladača a spätného prekladača.

### Hex-Rays Decompiler

Spätný prekladač od spoločnosti Hex-Rays, zameranej na softvér pre binárnu analýzu. Je dostupný ako doplnok k ich nástroju IDA disassember, nad výstupom ktorého vytvára spätný preklad. Momentálne nástroj podporuje dekompiláciu prekladačom generovaného kódu pre procesory x86, x64, ARM32, ARM64 a PowerPC. Skutočnosti o spätnom prekladači Hex-Rays boli čerpané z [3], kde je možné nájsť o nástroji viac informácií.

Výhodou dekompilátoru je to, že podporuje aktuálne najrozšírenejšie architektúry – ARM64, x64. Taktiež poskytuje API pre možnosť pridania programových doplnkov pracujúcich nad jeho výstupom. Jedná sa však o platený nástroj, ktorý je určený pre profesionálneho užívateľa. Ďalšími nevýhodami sú absencia podpory menej rozšírených architektúr a nemožnosť ľahko meniť funkcionalitu nástroja.

### Ghidra

Ghidra je softvér poskytujúci rámcové riešenie pre reverzné inžinierstvo. Spätný prekladač predstavuje jednu z možností, ktoré sú v Ghidre zahrnuté. Tento nástroj je vyvíjaný americkou agentúrou NSA (The National Security Agency), ktorá ho voľne poskytuje verejnosti aj so zdrojovými kódmi. Nástroj obsahuje podporu pre spätný preklad rozšírených architektúr x64 a ARM64, avšak aj menej rozšírených architektúr ako MIPS, microMIPS, SPARC a ďalších. Informácie o tomto nástroji boli čerpané z [2].

Ghidra je open source riešenie podporujúce spätný preklad veľkého množstva architektúr. Nástroj bol zverejnený iba nedávno a jeho komunita užívatelov stále rastie. Jednou z veľkých výhod nástroja je možnosť tímovej kolaborácie na projektoch. Ghidra je však robustný nástroj a jednotlivé časti využiteľné pre reverzné inžinierstvo nie je možné využívať samostatne.

### RetDec

RetDec je open source spätný prekladač vyvíjaný spoločnosťou Avast. RetDec podporuje preklad aplikácií 32 bitových architektúr x86, MIPS, ARM+Thumb a PowerPC. Implementácia RetDecu sa opiera o nástroje využívané pre tvorbu prekladačov. Nástroj RetDec je bližšie skúmaný v kapitole 4. Informácie o tomto nástroji boli čerpané z [4].

RetDec poskytuje jednoduché terminálové rozhranie. Jednotlivé časti, ktoré tvoria štruktúru spätného prekladača je možné využiť samostatne. V prekladači momentálne chýba podpora na trhu najrozšírenejších 64 bitových architektúr ARM a x86-64. Tento fakt spôsobuje, že RetDec zaostáva za konkurenciou.

# Kapitola 3

# Architektúra x86 a x86-64

Označenie x86 zastrešuje rodinu inštrukčných sád procesorov architektonicky vychádzajúcich z mikroprocesoru Intel 8086, viď. [12]. Všetky vytvorené architektúry procesorov patriace do tejto rodiny sú spätne kompatibilné a sú typicky predstaviteľmi CISC architektúry počítačov. Viac o type architektúry CISC je možné nájsť v [14]. V praxi sa pod termínom x86 zvykne označovať konkrétne 32-bitová architektúra IA-32, a preto aj ďalej v tejto práci je pod týmto termínom označovaná práve táto architektúra procesorov.

Architektúra x86 je príliš rozsiahla na to, aby bolo možné pokryť všetky informácie o nej v jednej kapitole. Primárnym účelom kapitoly je oboznámiť čitateľa s existenciou tejto architektúry a jej prvkami dôležitými pre túto prácu. Informácie o architektúrach x86 a x86-64 boli získané z manuálu pre vývoj softvéru pre architektúru x86 [17], kde je možné získať viac informácií o daných architektúrach.

## 3.1 Architektúra x86

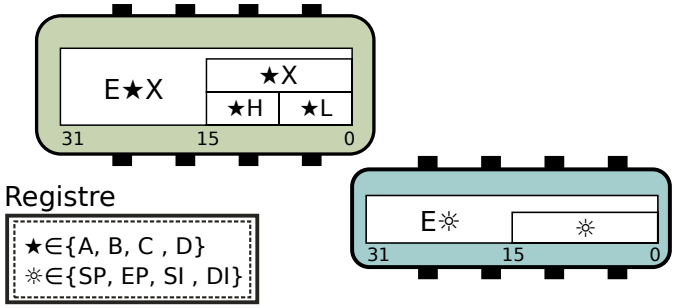
Jedná sa o 32-bitové rozšírenie originálne 16 bitových procesorov od spoločnosti Intel. Zavádza nové prvky a zachováva spätnú kompatibilitu s predošlými architektúrami. V nasledujúcej časti tejto kapitoly sú zhrnuté dôležité prvky architektúry x86 a jej rozšírenia z pohľadu tejto práce.

### 3.1.1 Sada registrov

Architektúra x86 definuje 16 základných registrov pre riadenie programu. Podľa [11] je tieto registre možné rozdeliť do následne popísaných kategórií.

1. **Všeobecne využiteľné registre**

   Typ registrov, ktorý je označovaný aj skratkou GPR (z angl. General Purpose Registers). Programátor má k dispozícií osem registrov GPR pre ukladanie dočasných dát. Pre každý z registrov definuje Intel odporúčané využitie, ktoré je možné nájsť v prílohe A.1. V rámci zachovania spätnej kompatibility s predošlou generáciou procesorovej architektúry je možné u každého z týchto registrov pristupovať k spodným 16 bitom, ako je zobrazené na obrázku 3.1. Okrem toho je možné si na danom obrázku všimnúť, že u registrov A, B, C a D je možné pristúpiť k dolným a horným ôsmim bitom príslušného registru X.




Obr. 3.1: Prekrytie GPR registrov pri 32-bitovej architektúre x86.

2. **Segmentové registre**

   Registre, ktoré slúžia ako selektory segmentov. Selektor segmentov je špeciálny ukazovateľ, ktorý určuje pamäťové segmenty. Každý zo selektorov odkazuje buď na pamäť programu, dát alebo zásobníka. Pre 32-bitovú architektúru x86 bolo definovaných šesť takýchto registrov: CS, DS, SS, ES, FS, GS.

3. **Stavové a riadiace registre**

   Stavové registre slúžia pre zistenie stavu vykonávaného programu. Dovoľujú programátorovi získať napríklad rozšírené informácie o výsledku vykonanej inštrukcie. Do kategórie stavových registrov patrí register EFLAGS. Medzi riadiace registre patrí register EIP. Register EIP obsahuje 32-bitový ukazovateľ na následujúcu inštrukciu, ktorá sa má vykonať. Tento register nie je možné modifikovať priamym zápisom, iba inštrukciami pre zmenu riadenia programu.

### 3.1.2 Inštrukčný súbor

Architektúra x86 umožňuje veľkú mieru flexibility, čo sa týka presunu dát medzi registrami a pamäťou. Podľa zdroju [11] je presun dát možné rozdeliť do piatich metód:

- priamy presun konštantnej hodnoty do registra,
- presun hodnoty medzi dvoma registrami,
- priamy presun konštantnej hodnoty do pamäte,
- presun hodnoty z registra do pamäte a naopak,
- presun hodnoty z pamäte do pamäte.

Prvé štyri metódy sú podporované všetkými modernými architektúrami, avšak posledná ostáva špecifická pre architektúru x86. Inštrukcie procesorových architektúr RISC, ako je napríklad ARM, môžu dáta z pamäte výlučne čítať alebo do nej dáta výlučne zapisovať. Ďalšou z dôležitých charakteristík je to, že architektúra x86 využíva premenlivú dĺžku inštrukcií. Najdlhšia inštrukcia môže mať dĺžku 15 bajtov, zatiaľ čo najkratšia inštrukcia má dĺžku jedného bajtu. [11]

Architektúra x86 patrí medzi predstaviteľov CISC typu architektúr procesorov, čo je možné pozorovať na počte inštrukcií definovaných v inštrukčnom súbore. Okrem základných inštrukcií boli vyvinuté rôzne špecializované rozšírenia inštrukčného súboru o nové inštrukcie. V tejto sekcii sú ďalej opísané základné typy inštrukcií architektúry x86. Ďalšie dôležité rozšírenia sú opísané v sekcii 3.1.3.

#### Inštrukcie pre všeobecné využitie

Skupina zahŕňa štandardné inštrukcie dostupné v každom procesore x86. Inštrukcie patriace do tejto skupiny vykonávajú niektorú z následujúcich činností:

- presun dát,
- logické operácie,
- binárna celočíselná aritmetika,
- desatinná aritmetika,
- posuny a rotácie,
- riadenie stavových príznakov,
- predanie riadenia behu programu alebo operácie s reťazcami znakov,
- operácie nad segmentovými registrami.

Konkrétne inštrukcie s popisom ich operandov je možné nájsť v manuáli [17].

### 3.1.3 Rozšírenia

V priebehu vývoja nových procesorov x86 a riešenia, s tým súvisiacich problémov vznikli rôzne rozšírenia základnej inštrukčnej sady, doplňujúce špecifickú funkcionalitu procesorom rady x86. Z pohľadu tejto práce sú dôležitými rozšíreniami koprocesor x87 a SSE.

#### Koprocesor x87

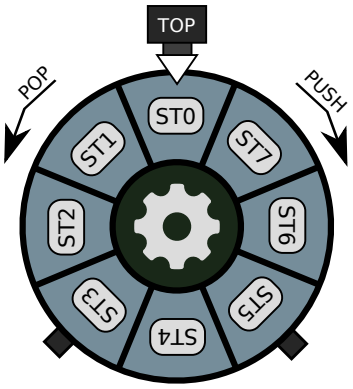
Rozšírenie prišlo zároveň s vytvorením špeciálnej mikroprocesorovej jednotky pre vysokorýchlostné spracovanie čísel s pohyblivou desatinnou čiarkou. Umožnilo získať presnejšie výpočty a uplatnilo sa v oblasti spracovania grafiky, vo vedeckých, inžinierskych a ekonomických aplikáciách. Definuje nové inštrukcie, ktoré umožňujú výpočty s číslami s pohyblivou desatinnou čiarkou, celými číslami a binárne kódovanými decimálnymi číslami. Pre urýchlenie výpočtov boli vytvorené špecializované inštrukcie implementujúce niektoré matematické funkcie. Konkrétne inštrukcie, ktoré pribudli s týmto rozšírením je možné nájsť v [17, Kapitola 8].

Rozšírenie prinieslo okrem nových inštrukcií nový typ registrov pre prácu s číslami s pohyblivou desatinnou čiarkou. Bol vytvorený rotačný zásobník obsahujúci 8 registrov. Činnosť rotačného zásobníka je abstrahovaná na obrázku 3.2. Ako je zobrazené, na vrchol zásobníka ukazuje špeciálny ukazovateľ. V prípade, že sa vykoná inštrukcia uloženia hodnoty do registra na vrchole, presunie sa vrchol zásobníka a ukazovateľ bude ukazovať na následujúci register. Po priradení hodnoty do posledného voľného registra sa nastaví príznak a ukazovateľ bude ukazovať na prvý register zásobníka.

#### Rozšírenie SSE

Motiváciou pre vytvorenie rozšírenia bolo zlepšiť výkon procesorov x86 pri náročnejších spracovaniach 2-D a 3-D grafiky, videí a obrázkov. Taktiež však vznikla snaha vytvoriť podporu procesora pri rozpoznávaní reči, syntéze zvuku, telefónii a videokonferenciách.

Rozšírenie `SSE` prinieslo novú inštrukčnú sadu typu SIMD (z angl. Single Instruction Multiple data – jedna inštrukcia, viac dát). Základná inštrukčná sada `SSE` obsahovala nových 70 inštrukcií pre prácu ako so skalárnymi, tak vektorovými hodnotami. Novšie generácie procesorov však priniesli nové verzie tohoto rozšírenia, z ktorých každá verzia priniesla nové inštrukcie. Konkrétne inštrukcie je možné nájsť v [17, Kapitola 10].




Obr. 3.2: Zásobník registrov x87.

Rozšírenie `SSE` takisto zaviedlo nových 8 registrov s označením `XMM` a veľkosťou 128 bitov. Tieto registre boli pôvodne určené pre držanie hodnôt typu štvorica čísel, s pohyblivou desatinnou čiarkou v základnej presnosti, viď [16]. Postupným zavádzaním nových inštrukcií v nových verziách rozšírenia `SSE` sa rozšíril účel, ale aj počet registrov `XMM`. Momentálne je definovaných 16 registrov v rozsahu 0 až 15. Každý z týchto registrov môže obsahovať hodnoty jedného z následujúcich dátových typov:

- štvorica 32-bitových čísel s pohyblivou desatinonou čiarkou,
- dvojica 64-bitových čísel s pohyblivou desatinnou čiarkou,
- štvorica 32-bitových celých čísel,
- osmica 16 bitových celých čísel,
- šestnástica 8 bitových celých čísel.

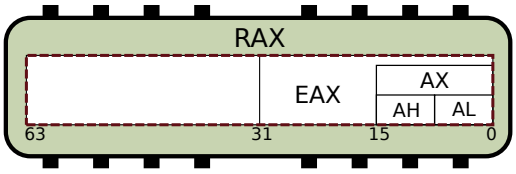
## 3.2 Architektúra x86-64

Architektúra x86-64 vznikla ako rozšírenie základnej 32-bitovej inštrukčnej sady x86. Vznikom x86-64 bol vyriešený problém architektúry x86, a to ten, že 32-bitová adresa neumožňuje adresovať viac ako štyri gigabajty pamäte. To pochopiteľne s rozvojom počítačov znamenalo limit, ktorý brzdil ďalší vývoj. Rozšírenia, ktoré boli zavedené sa týkajú širokej škály oblastí procesorovej architektúry. V tejto časti kapitoly sú bližšie popísané iba rozšírenia z pohľadu registrov a inštrukčnej sady. Architektúra x86-64 sa zvykne v praxi označovať aj x64, x86_64, AMD64 a Intel 64. V rámci textu je ďalej pre označenie tejto architektúry využívaný názov x64.

### 3.2.1 Rozšírenie sady registrov

V prvom rade architektúra x64 priniesla rozšírenie stavových, riadiacich a GPR registrov z 32 bitov na 64-bitovú veľkosť. Kvôli spätnej kompatibilite s predošlou architektúrou zachováva štýl, akým sa pristupuje k registrom. Na obrázku 3.3 je pre demonštráciu zobrazený rozšírený register akumulátora (`A`). Ako je možné vidieť, u všetkých GPR registrov oproti pôvodnej štruktúre pribudol iba prístup k 64-bitovej hodnote. U príznakového registra vznikol prístup `RFLAGS` a ukazovateľ na následujúcu inštrukciu bol rozšírený na register `RIP`. Okrem rozšírenia veľkosti registrov prináša nová architektúra aj systematické rozšírenie počtu registrov GPR. Pridáva 8 nových registrov, číselne označených 8 – 15 (začínajúc číslom

8 vzhľadom na originálnych 8 GPR registrov). Jedna z výhod, ktoré väčší počet registrov prináša, je napríklad možnosť pomocou nich predávať parametre pri volaní subrutín.




Obr. 3.3: Prekrytie GPR registrov v architektúre x64.

### 3.2.2 Rozšírenie inštrukčnej sady

V rámci spätnej kompatibility programov napísaných pre 32-bitové procesory boli zachované pôvodné inštrukcie. V 64-bitovom režime sú však tieto inštrukcie schopné pracovať naviac s 64-bitovými registrami. Okrem tohto rozšírenia však architektúra x64 pridala k základnej inštrukčnej sade nové inštrukcie špecifické pre danú architektúru. Niektoré z nových inštrukcií boli pridané najmä kvôli tomu, že niektoré inštrukcie pracujú s 32-bitovými registrami implicitne. Zoznam väčšiny pridaných inštrukcií je možné vidieť v prílohe A.2. Úplný zoznam aj s bližším popisom architektúry je možné nájsť v [17, Kapitola 5]

## 3.3 Aplikačné binárne rozhranie

Aplikačné binárne rozhranie je rozhranie medzi dvoma binárnymi programovými modulmi. Jedným z modulov býva často napríklad knižnica a druhým modulom býva program, ktorý je spustený užívateľom. Oproti aplikačnému programovému rozhraniu (API) je rozdiel v tom, že ABI definuje spôsob komunikácie modulov na hardvérovej úrovni, zatiaľčo API definuje komunikáciu na úrovni programovacieho jazyka.

Jedna z dôležitých vecí, ktoré ABI definuje, sú volacie konvencie udávajúce spôsob, akým sú predávané parametre určitej funkcii, a akým spôsobom táto funkcia vráti výsledok. Volacie konvencie udávajú informácie o tom, či budú všetky informácie predané na zásobníku, prípadne aké z parametrov budú predané, v ktorých registroch. V prípade predania parametrov na zásobníku, určuje ABI zároveň či budú parametre predané postupne v poradí počnúc prvým parametrom, alebo odzadu počnúc parametrom posledným.

Ďalej bude nasledovať opis dôležitých ABI definovaných pre architektúru x64 v rôznych operačných systémoch. Pre účely práce je dôraz pri ich opise kladený konkrétne na volacie konvencie, ktoré sú pre dané ABI definované.

### 3.3.1 System V ABI

System V ABI je rozhranie, ktoré využívajú pre komunikáciu na hardvérovej úrovni všetky moderné systémy podobné systému UNIX (Linux, MacOS, FreeBSD, apod.). Toto rozhranie ako jednu z mnohých vecí definuje konvenciu predávania parametrov pri volaní funkcií. Po vyhodnotení parametrov sú parametre predané volanej funkcií v šiestich až štrnástich registroch alebo na zásobníku. System V ABI definuje spôsob, akým budú parametre predané volanej funkcii, a ktoré registre budú na predávanie parametrov využité. Kvôli účelu názornosti bol algoritmus zjednodušený do následujúcej podoby. Celý algoritmus aj s obšírnejším opisom ABI je možné nájsť v [20]. Pre vyhodnocovanie spôsobu, akým budú predané

parametre, sú využívané triedy zobrazené v tabuľke 3.1. Konkrétny spôsob vyhodnocovania, ako budú predané parametre je následujúci:

1. Ak je trieda `MEMORY`, predaj hodnotu na zásobníku.
2. Ak je trieda `INTEGER`, vlož do prvého voľného registera z poriadia `RDI`, `RSI`, `RDX`, `RCX`, `R8` a `R9`.
3. Ak je trieda `SSE`, vlož do prvého voľného registera z poradia `XMM0` až `XMM7`.

Výnimku tvorí 128 bitový celočíselný typ, kedy sa hodnota rozdelí na dve a predá v dvoch registroch. Ak však je voľný len jeden, celá hodnota je predaná na zásobníku a posledný register ostane nevyužitý. Takisto ak pri vkladaní hodnoty parametra do registrov nie je voľný žiaden register, hodnota je predaná na zásobníku, a s ňou aj všetky následujúce hodnoty.

Pre návratové hodnoty má volacia konvencia takisto definovaný postup, akým vyhodnotiť miesto vloženia návratovej hodnoty. Ak je trieda návratovej hodnoty `MEMORY`, hodnota je navrátená na zásobníku na mieste, ktoré alokoval volajúci a vložil volanej funkcií ako prvý parameter v registri `RDI`. Pri triede návratovej hodnoty `INTEGER` je hodnota navrátená v registri `RAX`, prípadne v páre registrov `RAX:RDX`, ak je hodnotu možné predať v maximálne 128 bitoch. V prípade triedy `SSE` je návratová hodnota uložená v registri `XMM0`, ale ak to nie je možné je navrátená v registri `XMM1`.

| | |
|---|---|
| `INTEGER` | Celočíselné typy, ktorých veľkosť nepresahuje 128 bitov. |
| `SSE` | Rôzne číselné typy, ktoré nepatria do skupiny `INTEGER`. |
| `MEMORY` | Všetky typy, ktoré nepatria do kategórií `INTEGER` ani `SSE`. |

Tabuľka 3.1: Rozdelenie typov parametrov do tried.

### 3.3.2 Microsoft x64 ABI

Pre operačný systém Windows od spoločnosti Microsoft bolo definované vlastné aplikačné binárne rozhranie. Pre predávanie parametrov sú využívané vždy práve štyri registre a zásobník volaní. Toto ABI striktne definuje izomorfný vzťah medzi parametrami predanými funkčnému volaniu a registrami využitými pre predanie týchto parametrov. Každý parameter, ktorého veľkosť je väčšia ako 8 bajtov, musí byť volanej funkcií predaný odkazom. Na rozdiel od ABI v systémoch UNIX nie je vynaložená žiadna snaha rozdeliť jeden parameter do viacerých registrov. Pre operácie s pohyblivou desatinnou čiarkou je používaných 16 `XMM` registrov a zásobník registrov x87 sa nevyužíva. Tento zásobník môže byť využívaný volanou funkciou, ale musí byť považovaný za nestály medzi volaniami. Celočíselné hodnoty sú predávané podľa poradia parametrov postupne v registroch `RCX`, `RDX`, `R8` a `R9`. Na druhú stranu parametre s hodnotami v pohyblivej desatinnej čiarke sú predávané postupne v registroch `XMM0`, `XMM1`, `XMM2` a `XMM3`. V prípade, že sa predáva hodnota, ktorej veľkosť je väčšia ako je veľkosť registru GPR, je táto hodnota vložená na zásobník a funkcii je predaný odkaz na túto hodnotu obvyklým spôsobom, akým je predaný ukazovateľ.

Návratová hodnota je v prípade celočíselných typov navrátená v registri `RAX`, a v prípade čísel s pohyblivou desatinnou čiarkou v registri `XMM0`. V prípade, že je navrátená hodnota typu s veľkosťou väčšou ako je veľkosť registru `RAX`, musí volajúci funkcie pre výsledok alokovať miesto v pamäti a predať odkaz na toto miesto ako prvý parameter funkcie. Viac informácií o Microsoft x64 ABI je možné nájsť v [22].

# Kapitola 4

# Spätný prekladač RetDec

RetDec je názov open-source spätného prekladača vyvíjaného spoločnosťou Avast[1]. Nástroj je vyvíjaný tak, aby nebol spätný preklad závislý na architektúre vstupného binárneho súboru. Unikátnosť tohto nástroja spočíva v tom, že architektúra spätného prekladu je založená na infraštruktúre LLVM, navrhnutej pre tvorbu prekladačov. Táto kapitola je venovaná bližšiemu priblíženiu štruktúry spätného prekladača RetDec a infraštruktúry LLVM. Informácie o častiach a inšpirácie k schémam spätného prekladača RetDec boli získané z [18].

## 4.1 LLVM

LLVM predstavuje riešenie infraštruktúry prekladača, ktoré je možné využiť pre jednoduchú výstavbu prekladačov, optimalizácií a ďalších, s prekladačom súvisiacich programov. V rámci infraštruktúry bol navrhnutý univerzálny jazyk symbolických inštrukcií LLVM IR. Základné informácie o LLVM a LLVM IR boli získane z [19] a [6].

### 4.1.1 LLVM Intermediate Representation

LLVM IR je jazyk symbolických inštrukcií navrhnutý pre jednoduché predávanie kódu medzi modulmi prekladačov, a zároveň ako univerzálna vnútorná reprezentácia určená na aplikovanie rôznych sofistikovaných algoritmov optimalizácie programu. Cieľom LLVM je, aby bol jazyk dostatočne odľahčený a na nízkej úrovni abstrakcie. Zároveň však kladie požiadavku na expresívnosť, typovosť a rozšíriteľnosť.

### 4.1.2 Priechody

Termínom priechody sa v LLVM označujú optimalizácie kódu. Úlohou optimalizácií je prechádzať úseky programu za účelom kolekcie informácií alebo transformácie programu. LLVM rozdeľuje priechody do následujúcich kategórií:

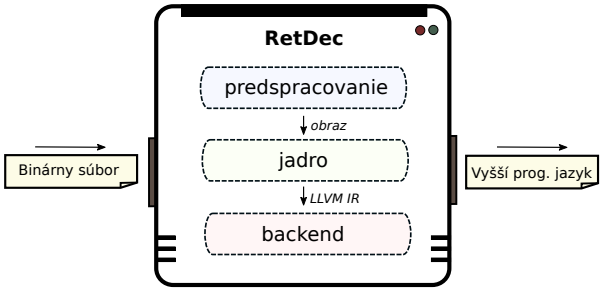
- **Analyzačné priechody** prechádzajú program na vstupe za účelom kolekcie informácií. Tieto informácie poskytujú na výstupe transformačným, alebo pomocným priechodom.
- **Transformačné priechody** sú schopné modifikácie programu na vstupe. Ich cieľom je pomocou špecializovaných algoritmov transformovať programy alebo ich časti.

[1] https://www.avast.com/about

- **Pomocné priechody** implementujú výpomocnú funkcionalitu, zvyčajne pre účely ladenia pri vývoji priechodov.

## 4.2 Štruktúra spätného prekladača

Základný koncept architektúry je zobrazený na obrázku 4.1. Ako je možné vidieť, na vstupe spätného prekladača je binárny súbor. Ten je transformovaný na výstupnú reprezentáciu vo vyššom programovacom jazyku procesom, ktorý je podobný princípu prekladačov. Tento proces sa odohráva v troch krokoch – predspracovanie, jadro a zadná časť (ďalej označovaná angl. termínom – backend). V nasledujúcich sekciách je zhrnutý princíp jednotlivých krokov spätného prekladu.




Obr. 4.1: Štruktúra spätného prekladača RetDec.

### 4.2.1 Predspracovanie

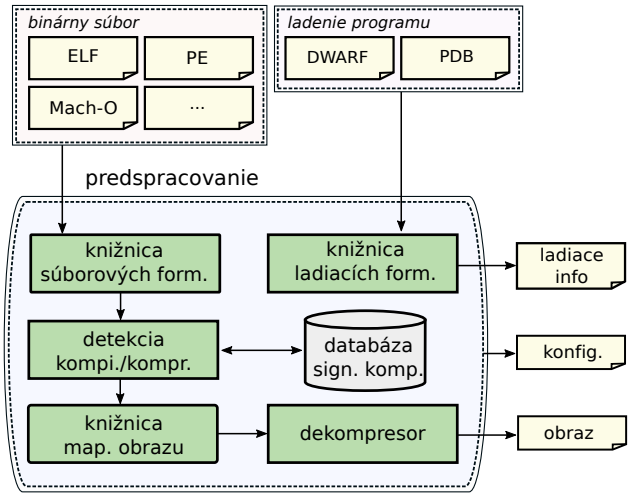
Cieľom fáze predspracovania je analýzou extrahovať zo vstupného binárneho súboru špecifické vlastnosti formátu objektového súboru. Výsledkom tejto analýzy je uniformná interná reprezentácia rôznych typov binárnych súborov pre ďalšie štádia dekompilácie. Fáza predspracovania je zobrazená na obrázku 4.2. Činnosť predspracovania je možné popísať v nasledujúcich krokoch:

1. S využitím knižnice súborových formátov je vytvorený objekt závislý na formáte platformy (Mach-O, PE, ELF, RAW dáta, a pod.).
2. Na vytvorený objekt sú aplikované YARA[2] signatúry, ktorými sú odhadované rôzne informácie o vstupnom súbore. Medzi dôležité informácie, ktoré je potrebné odhadnúť patrí detekcia kompilátora a kompresora.
3. Odhadnuté informácie sú využité pre vytvorenie konfiguračného súboru vo forme perzistentnej databázy obsahujúcej všetky informácie potrebné pre spätný preklad.
4. Nakoniec je interná reprezentácia transformovaná na súbor mapovaný do pamäte. Vďaka tomu dokáže spätný prekladač pracovať so vstupmi štýlom, akým by s nimi pracoval operačný systém. O súboroch mapovaných do pamäte je možné si prečítať viac v [24].

Binárny súbor na vstupe môže obsahovať ladiace informácie (DWARF), prípadne odkaz na separátny súbor obsahujúci tieto informácie (PDB). V takomto prípade sa využije

[2] https://virustotal.github.io/yara/

knižnica spracovania ladiacich informácií na vytvorenie vnútornej reprezentácie ladiacich informácií pre ďalšie časti prekladu.




Obr. 4.2: Časť predspracovania.

### 4.2.2 Jadro

Úlohou jadra spätného prekladu je transformovať načítaný obraz súboru do kódu LLVM IR. Jadro pre tento účel využíva modulárny nástroj pre optimalizáciu a analýzu s názvom `opt` od LLVM[3]. Tento nástroj slúži pre spúšťanie LLVM priechodov a predávanie stavu medzi nimi. V jadre spätného prekladača sú implementované tri skupiny modulov spolupracujúcich pre vygenerovanie výsledného LLVM IR – poskytovatelia, optimalizácie a analýzy.

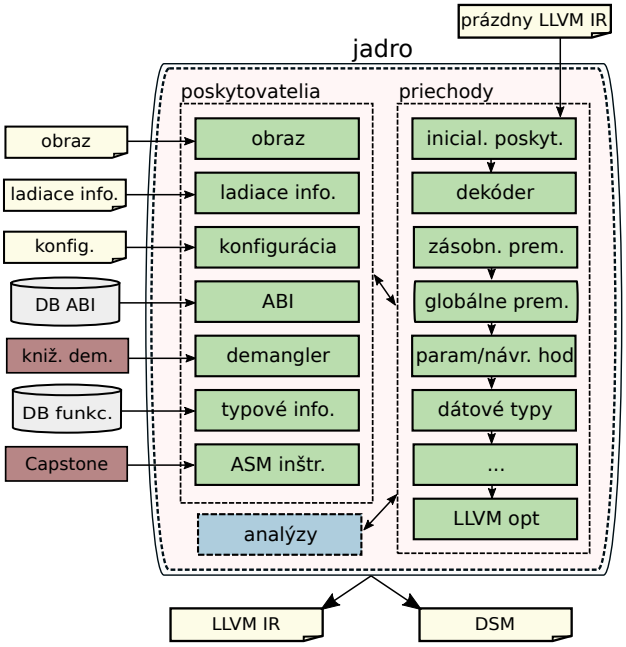
1. **Poskytovatelia** sú statické (globálne) objekty, ktoré slúžia ako rozhranie pre externé súbory, knižnice a databázy. Vytvárajú rozhrania, ktoré poskytujú abstrakciu od konkrétnych technológií. K poskytovateľom pristupujú počas svojej činnosti LLVM priechody.

2. **Optimalizácie** sú transformačné LLVM priechody implementované v spätnom prekladači. Sú to moduly, ktoré tvoria základ spätného prekladača. Na vstupe každej optimalizácie je modul LLVM, prípadne iba funkcia či blok, ktorý určitým spôsobom transformuje.

3. **Analýzy** predstavujú analyzačné priechody, ktoré skúmajú LLVM objekt na vstupe. Analýzy počas svojej činnosti vstupný objekt žiadnym spôsobom nemodifikujú. Výstupom analýz sú informácie, ktoré je možné využiť v optimalizáciach.

Štruktúra jadra spätného prekladača RetDec je zobrazená na obrázku 4.3. Činnosť jadra spätného prekladača je modulárne rozdelená do zreťazených LLVM priechodov a jej výsledkom je vytvorený LLVM modul obsahujúci preložený vstupný obraz na LLVM IR. Počiatočný priechod má za úlohu vytvoriť prázdny LLVM modul a inicializovať jednotlivých LLVM poskytovateľov. Po inicializačnom priechode nasleduje priechod dekódujúci vstupný obraz súboru na počiatočný LLVM IR. Kód LLVM IR je ďalej upravovaný následujúcimi

[3] http://llvm.org/docs/CommandGuide/opt.html




Obr. 4.3: Jadro spätného prekladača.

priechodmi, ktorý každý z nich implementuje špecifické analýzy reverzného inžinierstva. Príkladom takýchto analýz je rekonštrukcia globálnych premenných a premenných zásobníka, analýza parametrov funkcií a ich návratovej hodnoty alebo odvodenie a propagácia dátových typov. Po poslednom priechode upravujúcom vytvorený modul LLVM sú nad týmto modulom vykonané rôzne optimalizácie pomocou priechodov implementovaných v LLVM. Po týchto optimalizáciach je vygenerovaný výstup jadra spätného prekladača (LLVM modul, disasemblovaný kód) pripravený na ďalšie spracovanie.

### 4.2.3 Backend

Táto časť spätného prekladača má za úlohu konvertovať optimalizovaný kód LLVM IR na výstupný kód v jazyku C, prípadne graf volaní funkcií. V prvom kroku preloží vstupný LLVM IR do vnútornej reprezentácie, ktorá má podobu abstraktného syntaktického stromu, označovaného skratkou BIR (z angl. backend intermediate representation). Nad vytvoreným BIR následne vykoná rôzne optimalizácie, ako propagácia kópií alebo zjednodušenie výrazov. Po optimalizáciach je nakoniec vygenerovaný požadovaný výstup, napríklad kód v jazyku C.

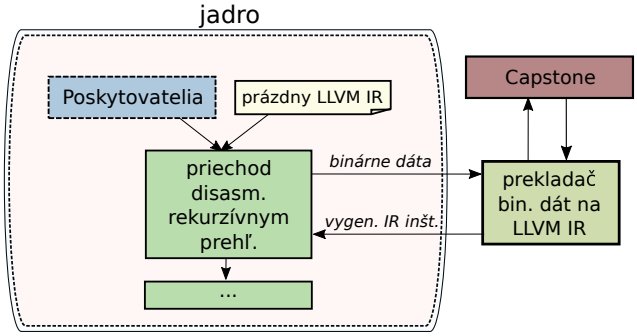
# Kapitola 5

# Analýza nedostatkov spätného prekladaču

RetDec momentálne podporuje spätný preklad aplikácií 32 bitových architektúr x86, MIPS, ARM+Thumb a PowerPC. V následujúcej časti textu sa nachádza opis princípu činnosti modulov spätného prekladača, ktoré je potrebné modifikovať pre pridanie novej architektúry x64. Okrem toho sú tieto časti analyzované z pohľadu požiadaviek na funkcionalitu, a ich aktuálnych nedostatkov.

## 5.1 Dekodér

Na vstupe priechodu dekódera je binárny súbor, obsahujúci postupnosť inštrukcií vo forme binárnych dát. Úlohou dekódera je pre každú inštrukciu vygenerovať sémanticky zhodnú reprezentáciu v LLVM IR. Štruktúra dekodéra je zobrazená na obrázku 5.1. Činnosť dekódera je rozdelená medzi dve časti: knižnica prekladača a priechod pre disasemblovanie.




Obr. 5.1: Činnosť dekodéra.

1. **Priechod pre disasemblovanie**

   Jedná sa o priechod, ktorého úlohou je riadenie algoritmu dekódovania. Pre svoju činnosť využíva algoritmus disasemblovania rekurzívnym prehľadávaním, viď [23]. Prechádza binárny súbor a má k dispozícií všetky informácie o doteraz zistených blokoch, ukončujúcich inštrukciách, funkciách a pod. Na základe získaných informácií vyberá následujúce binárne dáta, ktoré dáva preložiť knižnici prekladača.

2. **Knižnica prekladača**

   Účelom knižnice prekladača je generovanie kódu LLVM IR. Knižnica obsahuje metódu, ktorej vstupom je pozícia v LLVM module a binárne dáta. Túto metódu môžu volať používatelia tejto knižnice pre transformáciu binárnych dát na sémanticky zhodné bloky programu LLVM IR. Pre získanie informácií o inštrukcii z binárneho vstupu využíva knižnica nástroj Capstone disasembler engine [1]. Činnosť knižnice je možné rozdeliť do následujúcich krokov:

   (a) Knižnica prekladača vezme prázdny modul LLVM, do ktorého vygeneruje pomocné prostredie (globálne premenné a pomocné funkcie).

   (b) Zo získaných informácií o vstupných binárnych dátach nájde obslužnú rutinu pre preklad inštrukcie do LLVM IR.

   - V prípade, že knižnica prekladača nenájde rutinu pre obsluhu inštrukcie na vstupe, vygeneruje inštrukciu volania špecifickej pseudofunkcie. Prítomnosť takéhoto volania v spätne preloženom programe užívateľovi povie, že na danom mieste bola zaznamenaná inštrukcia, ktorej sémantika nebola modelovaná.

   (c) Vykoná obslužnú rutinu, ktorá pre vstupné dáta vygeneruje do modulu LLVM sémenaticky zhodný program LLVM IR.

### 5.1.1 Pridanie podpory architektúry x64

Priechod dekódera je miesto, ktoré je obvykle potrebné upraviť ako prvé pre pridanie podpory novej architektúry. Na základe definície objektov a sémantického opisu inštrukcií vytvorí program LLVM IR, s ktorým väčšina následujúcich priechodov pracuje uniformne pre všetky architektúry. Podmienkou pre pridanie sémantického opisu inštrukcií do knižnice prekladača je, aby bola pridávaná architektúra podporovaná nástrojom Capstone disasembler. Architektúra x64 je týmto nástrojom podporovaná a priechod dekódera je teda nutné rozšíriť o popis sémantiky inštrukcií a registrov špecifických pre x64.

#### Modelovanie inštrukčnej sady x64

Pre pridanie novej architektúry je potrebné pridať potrebné obslužné rutiny inštrukcií do knižnice prekladača. Tieto rutiny musia vedieť vygenerovať pre každú inštrukciu sémanticky zhodný úsek programu LLVM IR. Architektúra x64 má veľa inštrukcií spoločných s 32 bitovou x86. Pri modelovaní architektúry x86 bolo myslené na budúcu podporu architektúry x64, a preto boli základné inštrukcie x86 doplnené o prácu s 64-bitovými registrami. Príkladom nepodporovaných inštrukcií sú inštrukcie z inštrukčnej sady SSE, ktoré sú využívané pre prácu s číslami s pohyblivou desatinnou čiarkou. Vzhľadom na to, že základná inštrukčná sada bola pre RetDec modelovaná, je možné ukázať aktuálny výstup spätného prekladu a analyzovať jeho nedostatky. V ukážke 5.2 je zobrazený stav spätného prekladu aplikácií x64, pred rozšírením dekompilátora. Naľavo v ukážke sa nachádza pôvodný zdrojový kód v jazyku C, z ktorého bol vytvorený vstup pre RetDec. Napravo sa na porovnanie nachádza výstup spätného prekladu v rovnakom jazyku.

Ako je možné na obrázku vidieť, výstup spätného prekladača nie je kvalitný. V prvom rade je vidieť, že knižnica prekladača rozpoznala základné inštrukcie, no pri inštrukciách pracujúcich s číslami s pohyblivou desatinnou čiarkou boli vygenerované inštrukcie volania pseudofunkcií (`__asm_movsd`, `__asm_divsd`, a pod.). Okrem toho je možné vidieť, že

```c
int a = 0, b = 0;
scanf("%d %d", &a, &b);

double c = (a+b)/2.0;
printf("Avg(%d,%d) = %g\n", a, b, c);
```

```c
int64_t v1 = g7 - 8;
*(int64_t *)v1 = g6;
*(int32_t *)(v1 - 24) = 0;
*(int32_t *)(v1 - 20) = 0;
function_1050();
int128_t v2 = __asm_cvtsi2sd(
                  *(int32_t *)(v1 - 20)
                  + *(int32_t *)(v1 - 24));
int128_t v3 = __asm_movsd(
                  0x4000000000000000);
*(int64_t *)(v1 - 16) = __asm_movsd_1(
                  __asm_divsd(v2, v3));
__asm_movsd(*(int64_t *)(v1 - 16));
function_1040();
```

Obr. 5.2: Spätný preklad (vpravo) testovacieho programu (vľavo).

analýza parametrov neidentifikovala parametre volania funkcie, ani korektnú funkciu, ktorá je na danom mieste volaná. Pre skvalitnenie výstupu spätného prekladača, pri preklade binárnych súborov x64, bude najmä potrebné upraviť analýzu parametrov, viď. sekcia 5.2. Užívateľský dojem však takisto spríjemní, ak sa namiesto volania pseudofunkcií objavia na výstupe operácie pracujúce s hodnotami. Preto sa bude ďalej práca zaoberať aj problémami, spojenými s vytvorením podpory pre inštrukčnú sadu SSE.

### Pridanie sady registrov

Na začiatku svojej činnosti vygeneruje knižnica prekladača základné objekty architektúry – modelované registre. Všetky existujúce registre v RetDecu majú formu globálnych premenných. V prípade, že je možné na register nazerať rôznymi pohľadmi (viď. sekcia 3.1.1) vygeneruje sa len najväčší z nich. Každá modelovaná inštrukcia pracujúca s takýmto registrom vygeneruje v prípade potreby prácu s podčasťou získanou bitovými operáciami, viď ukážka 5.3. Vľavo na ukážke sa nachádza úsek programu v jazyku symbolických inštrukcií a vpravo z neho vygenerovaný program LLVM IR.

```
mov eax, 0xff
mov al, 0x02
```

→

```
; mov eax, 0xff
store i32 255, i32* @eax
; mov al, 0x02
%1 = load i32, i32* @eax
%2 = and i32 %1, -256
%3 = or i32 %2, 2
store i32 %3, i32* @eax
```

Obr. 5.3: Generovaná práca s pohľadmi na register.

Aktuálne riešenie je možné uplatniť len na prípady, kedy sú všetky náhľady na register rovnakého dátového typu. Pre pridanie podpory architektúry x64 je potrebné vytvoriť definície nepodporovaných registrov SSE, u ktorých majú jednotlivé pohľady na register rôzny dátový typ (viď. sekcia 3.1.3).

## 5.2 Optimalizácia parametrov a návratových hodnôt

Cieľom tohto priechodu je na základe analýzy volaní a definícií zistiť parametre funkcií a ich návratové hodnoty. Vstupom priechodu je program LLVM IR. Rekonštrukcia parametrov a návratových hodnôt je vykonaná následne v štyroch krokoch.

1. **Zber funkcií**

   Pre každú definíciu funkcie vytvorí záznam. Prejde všetky inštrukcie modulu LLVM, a každé nájdené volanie priradí k názname s jeho definíciou. Ak pre volanie nebola nájdená definícia funkcie, vytvorí preň nový záznam bez definície funkcie.

2. **Zber potenciálnych parametrov a návratových hodnôt**

   Prejde každý záznam a v prípade, že záznam obsahuje aspoň jedno volanie funkcie, vykoná:

   (a) Pre každé volanie prejde program od inštrukcie volania smerom dozadu a vyhľadá všetky inštrukcie zápisu.

   (b) Ak cieľový operand inštrukcie zápisu môže byť parametrom, uloží si záznam o operande.

   - V analýze sú napevno zabudovované informácie o tom, či môže byť operand inštrukcie parametrom, v závislosti na architektúre vstupného súboru.

   (c) Prejde program od inštrukcie volania smerom dopredu a vyhľadá všetky inštrukcie čítania.

   (d) Ak je operand čítania register, uloží si záznam o tomto registri.

   Prehľadávanie v oboch smeroch ukončí v následujúcich prípadoch:

   - Nie je žiadna predošlá/následujúca inštrukcia.
   - Narazí na miesto delenia základných blokov a na výber je viac ako jeden predchádzajúci/následujúci blok, ktorý sa má prezrieť.
   - Narazí na inštrukciu volania.

   Ak bola k záznamu priradená definícia funkcie, vykoná následujúce:

   (a) Prejde danú definíciu funkcie od začiatku do konca a vyhľadá inštrukcie čítania alebo návratu.

   (b) Ak operand inštrukcie čítania môže byť parametrom, a v definícií sa pred inštrukciou nenachádza zápis na toto pamäťové miesto, uloží si o operande záznam.

   (c) Pre každú inštrukciu návratu prehľadá program smerom dozadu a vyhľadá všetky inštrukcie zápisu.

   (d) Ak cieľový operand inštrukcie zápisu je register, uloží si záznam o registri.

   Pre overenie, či obsahuje pamäťové miesto vo funkcií definičný zápis, využíva analýzu *Reaching Definition Analysis* [7, Kapitola 9.2.4].

3. **Filtrácia hodnôt**

   Pre každé volanie a definíciu funkcie vytvorí prienik získaných pamäťových miest z kroku 2, ktoré predstavujú potenciálne parametre a návratové hodnoty. Následne vylúči tie, ktoré nie sú vhodné v závislosti na konkrétnej architektúre.

4. **Modifikácia vstupného programu**

   Pre každú funkciu vytvorí inštrukcie čítania hodnôt z pamäťových miest potenciálnych parametrov, ktoré ostali po prieniku v kroku 3. Načítané hodnoty predá ako parametre danej funkcii. Výsledok funkcie uloží na pamäťové miesto pre návratovú hodnotu, ktoré ostalo po kroku filtrácie.

### 5.2.1 Pridanie volacích konvencií x64

Pri aktuálnom návrhu optimalizácie parametrov a návratových hodnôt nie je možné jednoducho pridať podporu novej volacej konvencie. Zber hondôt je závislý na architektúre a pri filtrácii nevhodných hodnôt sa nepredpokladá, že architektúra môže mať viacero volacích konvencií. Taktiež informácie o vlastnostiach parametrov sú vložené priamo do implementácie algoritmu. Konkrétne pri architektúre x86 sa predpokladá, že každá funkcia využíva volaciu konvenciu `cdecl`[1]. Naviac sa pri každej architektúre predpokladá, že všetky parametre sú predávané smerom sprava doľava. Chyba je takisto v algoritme zberu parametrov, ktorý nedokáže nájsť všetky potenciálne parametre. Pri zbere hodnôt je využívaný jednoduchý algoritmus prechodu predošlého kódu, kde sa neráta s tým, že sa zápis parametru alebo návratovej hodnoty môže nachádzať v rôznych vetvách štruktúry programu.

Na obrázku 5.2 je možné vidieť, že okrem parametrov a správnych návratových hodnôt nie je pri architektúre x64 taktiež správne detekované volanie funkcie štandardnej knižnice. Okrem toho je stav optimalizácie možné lepšie demonštrovať na porovnaní výstupu dekompilácie rekurzívnej funkcie na ukážke 5.4.

Pre pridanie podpory x64 bude ako prvé nutné vytvoriť novú optimalizáciu parametrov a návratových hodnôt, ktorá odstráni nedostatky aktuálnej a bude ju možné parametrizovať pomocou poskytovateľa informácií ABI, popísaného v sekcii 5.3.

## 5.3 Poskytovateľ ABI

Poskytovateľ, ktorý podáva rozšírené informácie o rôznych častiach architektúry. Pre každú podporovanú architektúru je vytvorený samostatný objekt, ktorý obsahuje informácie o danej architektúre. Aktuálne dokáže objekt poskytovateľa ABI predať rôzne informácie o registroch (typ, veľkosť, využitie, a pod.), avšak nie je možné zistiť, či je daný register využívaný na predávanie parametrov alebo návratových hodnôt. Poskytovateľa je nutné rozšíriť tak, aby bol schopný overiť možnosť využitia objektu (lokálna, globálna premenná) ako parameter. Rozhranie je nutné rozšíriť na základe analýzy volacích konvencií architektúr, ktoré sú aktuálne podporované alebo ich bude možné v budúcnosti podporovať. Zo skúmania rôznych spôsobov predávania parametrov boli identifikovaná následujúce prípady:

- **Parametre funkcie na zásobníku.** Jedná sa o štandardný prípad predávania parametrov, kedy pre každú funkciu procesor alokuje/uvoľní pamäť zásobníkového rámca. V tomto prípade nie je možné pri predávaní parametrov spoliehať na meno (ofset) parametru, nakoľko je odlišný v definícii a pri volaní. Architektúra x64, aj každá z aktuálne podporovaných architektúr využíva tento spôsob predávania parametrov.
- **Parametre funkcie v registroch.** Taktiež ide o štandardný prípad predávania parametrov. Na rozdiel od predávania parametrov na zásobníku, je možné pri analýze spoliehať na označenie registru. Pri pristupovaní k registru sa využíva rovnaké meno

[1] https://docs.microsoft.com/cs-cz/cpp/cpp/cdecl

```c
int ack(int m, 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));
}
```

```c
int64_t ack(void) {
    int64_t v1 = g4;
    int64_t v2 = v1 - 8;
    *(int64_t *)v2 = g1;
    g1 = v2;
    int64_t v3 = v1 - 24;
    *(int32_t *)(v1 - 12) = (int32_t)g2;
    *(int32_t *)(g1 - 8) = (int32_t)g3;
    int64_t v4 = g1;
    uint32_t v5 = *(int32_t *)(v4 - 4);
    uint32_t v6 = *(int32_t *)(v4 - 8);
    int64_t v7;
    if (v5 == 0) {
        v7 = g1;
        g1 = *(int64_t *)v7;
        g4 = v7 + 16;
        return v6 + 1;
    }
    int64_t result;
    if (v6 != 0) {
        g3 = (int64_t)v6 - 1;
        g2 = v5;
        g4 = v3 - 8;
        g3 = ack();
        g2 = (int64_t)(*(int32_t *)(g1 - 4) - 1);
        g4 -= 8;
        result = ack();
    } else {
        g3 = 1;
        g2 = v5 - 1;
        g4 = v3 - 8;
        result = ack();
    }
    v7 = g1;
    g1 = *(int64_t *)v7;
    g4 = v7 + 16;
    return result;
}
```

Obr. 5.4: Spätný preklad (vpravo) testovacieho programu (vľavo).

pri volaní, aj v definícii. Tento spôsob je využívaný najmä u architektúr s väčšou sadou registrov, ako je ARM32, MIPS alebo aktuálne pridávaná architektúra x64. Niektoré volacie konvencie u x86 však taktiež podporujú predávanie prvých parametrov v registroch, typicky však ide iba o malý počet z nich.

- **Parametre ako okno registrov.** Jedná sa o spôsob, akým sú predávané parametre pri architektúre SPARC. Ide o architektúru, ktorá je podporovaná v knižnici Capstone disassembler, a teda má možnosť byť podporovaná spätným prekladačom RetDec. Pre každú funkciu je alokované okno obsahujúce pevný počet vstupných a výstupných registrov. V prípade volania funkcie sa premenujú výstupné registre volajúceho na vstupné registre volaného. Pri návrate z funkcie sa na druhú stranu vykoná opačný postup. Pri tejto možnosti nie je možné spoliehať na označenie registrov, však index vstupných registrov je mapovaný priamo na index výstupných registrov.

- **Parametre na zásobníku registrov.** Zásobník registrov je vlastnosť jedinečná pre architektúru Intel Itanium (IA-64). Podobne ako u okna registrov je pre každú funkciu alokovaná skupina registrov. Okno sa však v tomto prípade dynamický mení a index registrov využívaných pre predávanie parametrov je závislí na kontexte volania funkcie. Aktuálne však architektúra IA-64 nie je podporovaná knižnicou Capstone.

Po rozšírení rozhrania ABI bude pre pridanie podpory architektúry x64 nutné vytvoriť definíciu nového objektu, ktorý bude poskytovať potrebné informácie o jej registroch a spôsobe ich využitia.

# Kapitola 6

# Návrh podpory architektúry x64

Architektúra x64 je aktuálne najrozšírenejšou architektúrou na trhu osobných počítačov. Fakt, že RetDec nepodporuje preklad binárnych súborov architektúry x64, obmedzuje množinu prípadov v akých je možné využiť tento spätný prekladač. Následujúci text zhŕňa návrh pre pridanie podpory spätného prekladu aplikácií architektúry x64 z pohľadu rôznych častí spätného prekladača RetDec.

## 6.1 ABI

Pred vytvorením nového návrhu optimalizácie a návratových hodnôt je potrebné upraviť rozhranie ABI tak, aby jeho objekty boli schopné poskytovať rozšírené informácie o volacích konvenciách. Vznikli požiadavky na rozšírenie rozhrania základného ABI, aby bolo schopné poskytovať informácie o možnosti predania hodnoty parametrom a o registroch pre návratové hodnoty. Tieto informácie vychádzajú z definície konkrétnej volacej konvencie.

Pre každú architektúru je definovaná volacia konvencia a pre niektoré je ich definovaných hneď niekoľko. Jednou z možností je, že všetky tieto informácie budú uložené v každom závyslom objekte ABI, čo však neprináša žiadnu výhodu a výsledné riešenie bude neprehľadné a nie veľmi ľahko rozšíriteľné. Vzhľadom na túto skutočnosť som vytvoril návrh nového poskytovateľa volacích konvencií, popísaného ďalej. Výhodou tohto návrhu je to, že každý objekt ABI podávajúci informácie o architektúre definuje výčtom druhy konvencií, ktoré podporuje.

### 6.1.1 Poskytovateľ volacej konvencie

Jedná sa o sprostredkovateľa objektov pre poskytovanie informácií o volacích konvenciách. Návrh poskytovateľa vychádza z návrhového vzoru *jedináčik*, viď. [15, Kapitola 3]. Objekt poskytovateľa slúži pre konštruovanie špecializovaných objektov implementujúcich rozhranie `CallingConvention`, popísané ďalej v sekcii. Pre každý objekt volacej konvencie má pod jednoznačným identifikátorom registrovanú metódu, ktorou je daný objekt možné skonštruovať. Na miestach, kde je nutné získať informácie o volacej konvencii funkcie, sa zavolá konštrukčná metóda poskytovateľa s príslušným identifikátorom volacej konvencie.

#### Rozhranie CallingConvention

Predstavuje všeobecné rozhranie, pre poskytovanie všetkých dôležitých informácií o volacích konvenciách z pohľadu spätného prekladu. Každý objekt, ktorý rozhranie implementuje,

špecifikuje potrebné informácie a predstavuje práve jednu volaciu konvenciu. Informácie, ktoré objekty poskytujú, sú už následne závisle na konkrétnej architektúre, pre ktorú bola volacia konvencia definovaná. Informácie poskytované týmto rozhraním vychádzajú z analýzy rôznych volacích konvencií, spísanej v kapitole 5.3.

Z pohľadu pridania podpory architektúry x64 vznikli dve nové triedy, ktoré implementujú rozhranie `CallingConvention`. Informácie, na základe ktorých sú triedy implementované, boli identifikované a spísané v kapitole 3.3.

## 6.2 Optimalizácia parametrov a návratových hodnôt

Po analýze aktuálnych nedostatkov v prechode optimalizácie parametrov a návratových hodnôt (viď. kapitola 5.2), vznikla požiadavka na vytvorenie novej optimalizácie, ktorá nahradí existujúcu. U novej optimalizácie je žiadané, aby ju bolo možné parametrizovať na základe špecifických informácií od poskytovateľov informácií. Oproti aktuálnej situácií je takéto riešenie výhodnejšie práve z dôvodu, že pre pridanie novej architekúry už nebude potrebné upraviť implementáciu analýzy parametrov. Na to bude stačiť pridať nový popis ABI, konkrétne pri architektúre x64 popis ABI System V a Microsoft (viď. kapitola 3.3). Takéto riešenie bude ľahšie udržateľné a rozšíriteľné.

Vzhľadom na štruktúru akou sú v RetDecu vytvárané prechody, rozhranie starej optimalizácie ostane nezmenené. Na vstupe dostane optimalizácia modul LLVM obsahujúci kód LLVM IR vygenerovaný predošlými prechodmi. Po vykonaní odhadu parametrov aplikuje táto optimalizácia zmeny do kódu LLVM IR a predá modul ďalšiemu prechodu na vstup.

Najväčšie zmeny v optimalizácií parametrov sa týkajú zberu a filtrácie hodnôt. Vzhľadom na zložitosť týchto činnosti som sa rozhodol vytvoriť samostatné analýzy, vykonávajúce požadovanú činnosť. U novej optimalizácie ostane taktiež zachované poradie krokov činnosti, popísané v kapitole 5.2. Zmeny sa týkajú princípov krokov a jednotlivých algoritmov. Pre svoju činnosť využíva nová optimalizácia špecializované objekty kolektoru a filtru, ktoré sú popísané v následujúcej časti textu. Potenciálne parametre môžu byť aktuálne reprezentované lokálnymi, alebo globálnymi premennými. Prezentované algoritmy sú zovšeobecnené na akékoľvek odkazovateľné pamäťové miesta, prípadne objekty LLVM IR, ktoré budú ďalej označované ako hodnoty.

### 6.2.1 Kolektor hodnôt

Pre účely zberu potenciálnych parametrov a návratových hodnôt bolo vytvorené rozhranie kolektora. Objekty implementujúce rozhranie kolektora využívajú pre svoju činnosť algoritmy zberu hodnôt, ktoré je možné v prípade potreby modifikovať pre špecifické účely niektorých architektúr. Objekt navrhnutého kolektoru je možné využiť na kolekciu hodnôt predstavujúcich parametre a návratové hodnoty pri následujúcich prípadoch:

- bola nájdená definícia funkcie,
- v programe sa nachádza volanie funkcie,
- v programe sa nachádza volanie hodnoty objektu, ktorý nie je funkciou. Jedná sa napríklad o adresy uložené v lokálnych premenných.

Vzhľadom na požiadavku všeobecnosti bol vytvorený návrh algoritmov schopných zberu potenciálnych parametrov a návratových hodnôt z volaní a definícií funkcií. Pod všeobecnosťou je myslené, že algoritmy vykonávajú svoju činnosť nezávisle od konkrétnej architektúry. V následujúcej časti textu sa nachádza opis činnosti navrhnutých algoritmov.

### Zber hodnôt z inštrukcií zápisu v základnom bloku

Jedná sa o pomocný algoritmus, ktorý je využívaný pre zber hodnôt v bloku inštrukcií. Algoritmus je všeobecný a je možné ho využiť ako na zber parametrov volania, tak na zber zápisu návratovej hodnoty v definícii funkcie. Vstupom je inštrukcia základného bloku, ktorý sa má prejsť pre zber. Výstupom je množina hodnôt, ktoré je možné využiť pre predanie parametrov, alebo navrátenie hodnoty. Celý postup, akým sú získavané vhodné hodnoty je zobrazený v algoritme 1.

**Algoritmus 1** Kolekcia hodnôt parametrov, alebo návratových hodnôt v základnom bloku.

**Vstup:** $I$ – inštrukcia, od ktorej sú spätne hľadané hodnoty.
**Výstup:** $(Values, HasPred)$ – nájdené hodnoty a informácia o možnosti pokračovania.

```
function CollectInInstrucionBlock(I)
    Excluded ← ∅, Values ← ∅
    Block ← GetBasicBlock(I)
    while HasPrevInst(I) do
        I ← GetPrevInst(I)
        if IsCallInst(I) or IsReturnInst(I) then
            return (Values, False)
        if IsStoreInst(I) then
            (Dest, Val) ← GetStoreOperands(I)
            if not ABI.IsRegister(Dest) and not ABI.IsStackVar(Dest) then
                Excluded ← Excluded ∪ {Dest}
            if IsLoadInst(Val) and LoadedValue(Val) is not Dest then
                Excluded ← Excluded ∪ {LoadedValue(Val)}
            if Dest not in Excluded then
                Excluded ← Excluded ∪ {Dest}
                Values ← Values ∪ {Dest}
        if I is FirstInstOf(Block) then
            return (Values, True)
    return (Values, False)
```

### Rekurzívny zber inštrukcií zápisu

Pomocný algoritmus určený pre zber hodnôt, ktorý je nezávislý od štruktúry zreťazenia základných blokov. Algoritmus je možné využiť pre zber možných parametrov volania, tak aj zber návratových hodnôt v definícii funkcie. Návrh algoritmu je možné popísať matematickou funkciou z rovnice 6.1. Funkcia dostane na vstupe inštrukciu základného bloku, od ktorej prehľadá základný blok algoritmom 1. Následne rekurzívnymi volaniami zistí vhodné hodnoty v predchádzajúcich základných blokoch. Výstupom je množina hodnôt potenciálnych parametrov alebo návratových hodnôt, získaná ako zjednotenie množniny hodnôt aktuálneho základného bloku s množinou získanou ako prienik potenciálnych parametrov nájdených v predchádzajúcich základných blokoch.

$$Rc(I) = Bc(I) \cup (Rc(Pre(B_I, 1)) \cap Rc(Pre(B_I, 2)) \cap \cdots \cap Rc(Pre(B_I, N_{B_I}))) \quad (6.1)$$

Kde:

| | |
|---|---|
| $B_I$: | základný blok, v ktorom sa nachádza inštrukcia $I$. |
| $Rc(I)$: | funkcia, ktorá vráti množinu možných parametrov, pred inštrukciou $I$. |
| $Bc(I)$: | funkcia, implementujúca algoritmus 1. |
| $Pre(X, y)$: | funkcia, ktorá pre blok $X$ vráti poslednú inštrukciu $y$-tého bloku, ktorý ho bezprostredne predchádza. |
| $N_{B_I}$: | Počet základných blokov bezprostredne predchádzajúcich blok $B_I$. |

### Zber parametrov v definícii funkcie

V prípade, že sa v analyzovanom programe nachádza definícia funkcie uplatní sa pre nájdenie potenciálnych parametrov algoritmus 2. Tento algoritmus sa snaží odhaliť tie premenné, s ktorými sa v definícii funkcie pracuje ako s parametrami funkcie. Pre parametre je typické, že sa jedná buď o zásobníkové premenné, alebo registre, s ktorými sa pracuje bez toho, aby bola ich hodnota inicializovaná. Na vstupe algoritmu je analyzovaná definícia funkcie. Pre svoju činnosť využíva objekt, ktorý je označený ako `RDA`. Objekt `RDA` implementuje analýzu `Reaching Definition Analysis`, o ktorej sa je možné dozvedieť viac v [7, Kapitola 9.2.4]). Táto analýza dokáže poskytnúť informácie o miestach, kde sa definuje hodnota objektu s ktorým sa pracuje. Príkladom definície hodnoty premennej v definícii funkcie je inštrukcia zápisu.

**Algoritmus 2** Kolekcia možných parametrov v definícii funkcie.

**Vstup:** $F$ – definícia funkcie; $RDA$ – inicializovaný objekt analýzy.
**Výstup:** $Values$ – nájdené možné parametre.

```
function CollectInDefinition(F, RDA)
    Values ← ∅
    I ← FirstInstOf(F)
    while I is not LastInstOf(F) do
        if IsLoadInst(I) then
            Val ← LoadedValue(I)
            if not ABI.IsRegister(Val) and not ABI.IsStackVar(Val) then
                continue
            if not RDA.isUsed(Val) then
                continue
            if not RDA.isDefined(Val) then
                Values ← Values ∪ {Val}
        I ← GetPrevInst(I)
    return Values
```

#### Zber návratovej hodnoty v definícií funkcie

Vstupom algoritmu je definícia funkcie. Na výstupe je zoznam, obsahujúci dvojice zložené z inštrukcie návratu a množiny potenciálnych návratových hodnôt. Algoritmus prebieha následujúcim spôsobom:

1. Započne prechádzanie všetkých inštrukcii v definícii funkcie.
2. Vezme následujúcu inštrukciu a zistí jej typ.
3. V prípade, že je aktuálne skúmaná inštrukcia inštrukciou návratu, vytvorí nový záznam.
4. Pre novo vytvorený záznam vykoná algoritmus *Rekurzívneho zberu hodnôt*. Nájdené hodnoty predstavujú potenciálne návratové hodnoty funkcie.
5. V prípade, že existuje následujúca inštrukcia, vráti sa na bod 2. V opačnom prípade ukončí vykonávanie.

#### Zber parametrov a návratovej hodnoty volania

Pre inštrukciu volania v LLVM IR je typické, že sa jedná o volanie funkcie, alebo hodnoty premennej. Kolekciu potenciálnych parametrov a návratových hodnôt volania však táto skutočnosť neovplyvňuje. Potenciálne parametre je pre každé volanie možné získať vykonaním algoritmu *rekurzívneho zberu hodnôt*. Návratové hodnoty následne nájde tak, že prehľadá program za inštrukciou volania a nájde všetky čítania registrov. Podmienky ukončenia prehľadávania sú následovné:

- bola nájdená inštrukcia volania,
- bola nájdená inštrukcia návratu,
- nie je následujúca inštrukcia.

### 6.2.2 Filter hodnôt

Fáza filtrácie je určená pre elimináciu hodnôt takých nájdených potenciálnych parametrov funkcie, ktoré nespĺňajú podmienky definované volacou konvenciou. Pre úlohu filtrácie som navrhol nové rozhranie filtru. Každý objekt, ktorý implementuje rozhranie filtru, vykonáva špeciálne postupy pre elimináciu hodnôt, ktoré nemôžu byť parametrom určitej funkcie. Každý objekt filtru dostane pre svoju činnosť informácie o volacej konvencie funkcie od objektu implementujúceho rozhranie `CallingConvention`.

Pre splnenie podmienky všeobecnosti novej analýzy som navrhol spoločné algoritmy pre aktuálne podporované, aj pridávané architektúry opísané ďalej v texte. Každú časť algoritmu je však možné ľahko doplniť o nové postupy pre splnenie špecifických potrieb cielených architektúr.

#### Roztriedenie hodnôt na vstupe

Nezávisle od toho, či sa jedná o hodnoty potenciálnych parametrov, alebo návratových hodnôt, je potrebné hodnoty na vstupe pre účely filtrácie roztriediť do následujúcich skupín:

- zásobníkové premenné,
- registre GPR,

- registre pracujúce s vektorovými číslami (ďalej označované VR).
- registre pre ukladanie čísel s pohyblivou desatinnou čiarkou (ďalej označované ako FPR).

Registre zaraďuje do skupín na základe ich typu, o ktorom dáva informáciu objekt ABI. Tento objekt dáva taktiež informácie o tom, či je hodnota premennou na zásobníku.

#### Eliminácia registrov

Algoritmus slúži pre filtrovanie nevhodných registrovov. Je možné ho využiť na registre, ktoré predstavujú parametre ale aj návratové hodnoty. Algoritmus je všeobecný, to znamená že nie je závyslí na konkrétnom type registrov. Vstupom algoritmu je možina registrov a šablóna s výčtom očakávaných registrov. Pre každú skupinu registrov (GPR, FPR, VR) špecifikuje šablónu objekt volacej konvencie.

1. Ak je šablóna na vstupe prázdna, eliminuje všetky registre a ukončí algoritmus.
2. Registre zoradí podľa šablóny na vstupe.
   - V prípade, že sa niektorý z registrov nenachádza v šablóne, zahodí daný register.
3. Postupne prejde všetky registre na vstupe a zistí či poradie registru je zhodné s poradím daného registru v šablóne. Ak poradie registru nesedí, je tento register zahodený aj so všetkými ostatnými registrami, ktoré ešte neboli skontrolované.

#### Eliminácia zásobníkových premenných

Algoritmus, ktorý dostane na vstupe množinu zásobníkových premenných a eliminuje tie, ktoré nemohli byť využité na predanie parametra. Eliminácia prebieha v následujúcich krokoch:

1. Zoradí zásobníkové premenná na vstupe, na základe ich ofsetu.
   - Využije informácie z objektu `CallingConvention` a zistí, či volacia konvencia predáva hodnoty sprava doľava, alebo naopak. To ovplyvní, či budú pre účely eliminácie zásobníkové premenné zoradené podľa ofsetu vzostupne, alebo zostupne.
   - Na získanie ofsetu premennej, využije objekt konfigurácie.
2. Prejde všetky hodnoty na vstupe a skúma či medzi každými dvoma zásobníkovými premennými je rozdiel ofsetov menší ako hodnota, ktorú definuje objekt ABI pre každú architektúru.
   - Veľkosť ofsetu podlieha veľkosti hodnoty, ktorá je ukladaná na zásobník. Každá architektúra však definuje limit veľkosti hodnoty, ktorú vie na zásobník uložiť niektorá z jej inštrukcií. Tento limit sa môže líšiť na základe dátového typu ukladanej hodnoty.
3. V prípade, že je veľkosť rozdielu ofsetov väčšia ako stanovená hodnota, eliminuje všetky následujúce ofsety. Filtračná metóda teda vytvorí najdlhšiu sekvenciu zásobníkových premenných, ktoré mohol prekladač využiť pre predanie parametrov.

Na základe predpokladu, že k pridávaniu hodnôt na zásobník sa prejde až po tom, čo sú všetky registre aspoň jednej skupiny registrov na predávanie parametrov obsadené, je možné eliminovať zásobníkové premenné aj následujúcim spôsobom:

1. Zistí či objekt volacej konvencie špecifikuje, že veľké objekty sú predávané odkazom. Ak nie, ukončí algoritmus, v opačnom prípade pokračuje.

    - U konvencií, ktoré nepredávajú veľké objekty odkazom, ako `fastcall`[1], sa prejde na predávanie parametrov pomocou zásobníka aj v prípade, že sú ešte dostupné registre.

2. Zistí, či je aspoň jedna skupina registrov využívaných na predávanie parametrov zaplnená (na predávanie sa vyžívajú všetky dostupné registre danej skupiny). Ak taká skupina existuje, eliminuje všetky hodnoty zásobníkových premenných na vstupe.

### Filtrácia definície funkcie

Vstupom algoritmu pre filtrovanie definície funkcie je záznam danej definície funkice, ktorý obsahuje:

- množinou potenciálnych parametrov,
- zoznam dvojíc obsahujúcich inštrukciu návratu a množinu návratových hodnôt nájdených v tele funkcie

Hodnoty potenciálnych parametrov a jednotlivé množiny návratových hodnôt roztriedi využitím algoritmu pre *roztriedenie hodnôt na vstupe*, popísaného vyššie. Po roztriedení možných parametrov a návratových hodnôt vykoná:

1. Pre hodnoty parametrov eliminuje všetky možné zásobníkové premenné, ktorých ofset nie je kladný. Na registre uplatní algoritmus *eliminácie registrov* a na zásobníkové premenné algoritmus *eliminácie zásobníkových premenných*.

2. Pre všetky záznamy dvojíc vykoná prienik nájdených možných návratových hodnôt. Hodnoty, ktoré ostali po prieniku vyfiltruje pomocou algoritmu *eliminácie registrov*.

### Filtrácia volanií

Algoritmus pre filtrovanie hodnôt volaní funkcií. Svoju činnosť vykonáva nezávisle na definícii funkcie. Vstupom algoritmu je zoznam inštrukcií volania, s ktorých má každá:

- množinu nájdených hodnôt parametrov,
- množinu nájdených návratových hodnôt.

Činnosť algoritmu prebieha v následujúcich krokoch:

1. Prejde zoznam inštrukcií a roztriedi parametre a návratové hodnoty využitím algoritmu *roztriedenia hodnôt na vstupe*, opísaného vyššie.

2. Zoznam prejde znovu ponechá len množiny spoločných hodnôt. V každej množine registrov ostanú len tie registre, ktoré sú pre všetky inštrukcie volania spoločné. Pre hodnoty zásobníkových premenných zistí najmenší spoločný počet a odstráni zvyšné.

[1] https://docs.microsoft.com/en-us/cpp/cpp/fastcal

3. Z hodnôt parametrov a návratových hodnôt vyfiltruje tie, ktoré nie sú vhodné v závyslosti od volacej konvencie. Na hodnoty registrov uplatní algoritmus *eliminácie registrov* a na zásobníkové premenné algoritmus *eliminácie zásobníkových premenných*

## 6.3 Dekodér

Jednou z funkcií dekódera je schopnosť generovať pre každú inštrukciu podporovanej architektúry sémanticky zhodný úsek programu LLVM IR. Pre túto funkciu je potrebné rozšíriť knižnicu prekladača a doplniť príslušné obslužné rutiny, ktoré sa vykonajú v prípade že sa narazí na danú inštrukciu v binárnom súbore. Základná inštrukčná sada bola pre RetDec pokrytá pri vytváraní podpory architektúry x86. Modelované inštrukcie, jedinečné pre architektúru x64 sú v prílohe prílohe A.2. Sémantický popis inštrukcií v LLVM IR je vytvorený na základe algoritmu ich činnosti získaného zo zdroja [10].

V analýze nedostatkov spätného prekladača (viď. sekcia 5.1) pre podporu architektúry x64 bola vytvorená požiadavka na modelovanie sémantiky podmnožiny inštrukčnej sady SSE. Konkrétne podmnožiny, pracujúcej s číslami s pohyblivou desatinnou čiarkou. Pre pridanie popisu SSE inštrukcií som vytvoril návrh, akým budú modelované pohľady na registre tejto inštrukčnej sady a spôsob akým budú tieto pohľady využívané v generovanom LLVM IR.

### 6.3.1 Pohľady na register

Inštrukčná sada `SSE` využíva registre `XMM`, na ktoré je možné nazerať rôznymi pohľadmi (viď. sekcia 3.1.3). Typ `XMM` registrov by bolo možné modelovať v jazyku C pomocou dátového typu `union`[2]. Podpora dátového typu `union` v jazyku LLVM IR nie je, obsahuje však prostriedky pre simulovanie tejto podpory. Pri vytváraní návrhu som vychádzal zo spôsobu, akým generuje prácu s dátovým typom `union` prekladač `clang`[3]. Na ukážke 6.1 je zobrazený dátový typ `union` (vľavo), ktorý je transformovaný po preklade na LLVM IR (vpravo). Pre najväčšiu položku dátového typu je vygenerovaný globálny dátový typ. V prípade, že sa v programe pracuje s iným pohľadom, vygeneruje prekladač pretypovanie na štruktúru o rovnakej veľkosti, obsahujúcu daný dátový typ. Týmto spôsobom je možné riešiť problém generovania sémantiky SSE inštrukcií aj v spätnom prekladači.

```
typedef union {
  int i32[4];
  double d65[2];
  float d32[4];
  char i8[16];
} XMM;
.
.
.
XMM a;
scanf("%d", a.i32);
```

→

```
%union.XMM = type { [2 x double] }
.
.
.
%1 = alloca %union.XMM, align 8
%2 = bitcast %union.XMM* %1 to [4 x i32]*
%3 = getelementptr inbounds [4 x i32],
            [4 x i32]* %2, i32 0, i32 0
```

Obr. 6.1: Výstup prekladača `clang` pri práci s dátovým typom `union`.

[2] https://en.cppreference.com/w/c/language/union
[3] https://clang.llvm.org/

### 6.3.2 Inštrukcie pracujúce s pohľadmi

Inštrukčná sada SSE obsahuje množstvo inštrukcií. Pre podporu architektúry x64 sú však dôležité iba tie, ktoré nahliadajú na registre ako štruktúru čísel s pohyblivou desatinnou čiarkou. Súhrn týchto inštrukcií je v prílohe A.3. Každá z následných inštrukcií je modelovaná tak, že pri práci s registrami SSE pretypuje globálny register na vhodný pohľad. Takéto riešenie je možné vidieť na ukážke 6.2. V prípade potreby implementácie vektorových inštrukcií pracujúcich s registrami SSE ako štvoricou celých čísel, bude možné využiť rovnaký postup.

```
; XMM xmm0, xmm1
%2 = alloca %union.XMM
%3 = alloca %union.XMM

; &xmm0.i32[0]
%4 = bitcast %union.XMM* %2 to [4 x i32]*
%5 = getelementptr inbounds [4 x i32],
                     [4 x i32]* %4, i64 0, i64 0

; &xmm1.d32[0]
%6 = bitcast %union.XMM* %3 to [4 x float]*
%7 = getelementptr inbounds [4 x float],
                     [4 x float]* %6, i64 0, i64 0

%8 = call i32 (i8*, ...) @scanf(..., i32* %5, float* %7)
```

Obr. 6.2: Príklad generovania práce s rôznymi pohľadmi registrov XMM.

# Kapitola 7

# Implementácia podpory architektúry x64

Pre implementáciu špecializovaných analýz sa v spätnom prekladači RetDec využíva knižnica prekladačovej infraštruktúry LLVM, napísaná v programovacom *jazyku C++*. Kapitola je určená pre opis zaujímavých častí implementácie vychádzajúcej z návrhu v kapitole 6. Okrem toho sa tu nachádza opis rozšírení, ktoré boli vytvorené na základe priebežného testovania a skúmania výstupov prekladu binárnych súborov.

## 7.1 Poskytovateľ volacích konvencií

Na základe návrhu poskytovateľa volacích konvencií v sekcii 6.1.1 bola implementovaná trieda, schopná vytvárať špecializované objekty poskytujúce informácie o volacích konvenciách. Okrem toho boli definované volacie konvencie pre podporované a pridávané architektúry. Počas pridávania informácii o volacích konvenciách bolo zistené, že veľa skutočností o podporovaných architektúrach bolo zanedbávaných a správnosť spätného prekladu veľa krát spočívala na náhode. V následujúcom zhrnutí je zoznam všetkých volacích konvencií, ktoré bolo nutné analyzovať a implementovať pre spätný prekladač:

- ARM 32/64 ABI,
- MIPS 32/64 ABI, PSP ABI,
- PowerPC 32/64 ABI,
- x64 – Microsoft x64 ABI, System V ABI,
- x86 – cdecl, stdcall, thiscall, Pascal, Watcom.

## 7.2 Optimalizácia parametrov a návratových typov

Na základe návrhu novej optimalizácie v sekcii 6.2 bola vytvorená implementácia, pre ktorú bolo možné ľahko vytvoriť popis volacích konvencií architektúry x64. Na ukážke 7.1 je možné vidieť príklad aktuálneho výsledku dekompilácie aplikácie x64. Vľavo na obrázku je výstup dekompilácie pred vytovrením novej optimalizácie a vpravo výstup po jej vytvorení. V následujúcich častiach sekcie sú opísané niektoré zaujímave časti a zhrnutie nedostatkov aktuálnej implementácie.

```
int64_t ack(void) {
    int64_t v1 = g4;
    int64_t v2 = v1 - 8;
    *(int64_t *)v2 = g1;
    g1 = v2;
    int64_t v3 = v1 - 24;
    *(int32_t *)(v1 - 12) = (int32_t)g2;
    *(int32_t *)(g1 - 8) = (int32_t)g3;
    int64_t v4 = g1;
    uint32_t v5 = *(int32_t *)(v4 - 4);
    uint32_t v6 = *(int32_t *)(v4 - 8);
    int64_t v7;
    if (v5 == 0) {
        v7 = g1;
        g1 = *(int64_t *)v7;
        g4 = v7 + 16;
        return v6 + 1;
    }
    int64_t result;
    if (v6 != 0) {
        g3 = (int64_t)v6 - 1;
        g2 = v5;
        g4 = v3 - 8;
        g3 = ack();
        g2 = (int64_t)(*(int32_t *)(g1 - 4) - 1);
        g4 -= 8;
        result = ack();
    } else {
        g3 = 1;
        g2 = v5 - 1;
        g4 = v3 - 8;
        result = ack();
    }
    v7 = g1;
    g1 = *(int64_t *)v7;
    g4 = v7 + 16;
    return result;
}
```

```
int64_t ack(int64_t a1, int64_t a2) {
    if ((int32_t)a1 == 0) {
        return a2 + 1 & 0xffffffff;
    }
    int64_t result;
    if ((int32_t)a2 != 0) {
        result = ack(a1 + 0xffffffff
                     & 0xffffffff,
                 ack(a1, a2 - 1));
    } else {
        result = ack(a1 + 0xffffffff
                     & 0xffffffff,
                 1);
    }
    return result;
}
```

Obr. 7.1: Porovnanie výstupu dekompilácie po zmenách v optimalizácii parametrov.

### 7.2.1 Rozšírené informácie o funkciách

V optimalizácii parametrov a návratových hodnôt je možné využiť informácie, ktoré boli o funkciách zistené vo fáze predspracovania. Jedná sa o informácie o typoch parametrov a návratovej hodnoty funkcie, ktoré je následne možné využiť na filtráciu nájdených hodnôt. Tieto informácie je možné získať z následujúcich zdrojov:

- **Ladiace informácie.** V prípade, že sa na vstupe nachádza súbor, u ktorého sú známe ladiace informácie, tak sa vo fáze predspracovania extrahujú knižnicou pre ladiace formáty.
- **Konfigurácia.** U niektorých funkcií je možné na základe YARA pravidiel určiť, že sa jedná o knižničné funkcie, ktorých signatúry sú známe.

- **IDA disassembler.** RetDec podporuje získavanie informácii o vstupnom súbore IDA disassemblerom. Ten dokáže poskytnúť informácie o funkciách daného súboru.

Taktiež je však možné využiť odhadnuté informácie o zistenom prekladači pre určenie implicitnej volacej konvencie. Pre architektúru x86 má prekladač `gcc` implicitnú volaciu konvenciu `cdecl`, avšak prekladač `fpc` využíva konvenciu `pascal` a prekladač `watcom` definuje vlastnú.

### Filtrácia na základe známych typov

Na základe známych informácii o typoch parametrov a návratových hodnôt je možné upraviť spôsob, akým sa eliminujú hodnoty. Bol implementovaný algoritmus, ktorý simuluje spôsob, akým by konvencia predala parametre. Zistí registre, ktoré by mali byť využité a počet zásobníkových premenných. Algoritmus sa komplikuje v možnostiach, akými jednotlivé architektúry predávajú veľké typy, objekty, čísla s pohyblivou desatinnou čiarkou, apod. Po vytvorení filtračnej šablóny overí, či dané hodnoty boli nájdené v programe LLVM IR, a ak nie, pokúsi sa dané hodnoty dohľadať. Na ukážke 7.2 je možné vidieť rozdiel keď sa aplikujú známe informácie o type funkcie, oproti ukážke 7.1, kde vstupný binárny ladiace informácie neobsahoval.

```
1  int32_t ack(int32_t m, int32_t n) {
2      if (m == 0) {
3          return n + 1;
4      }
5      int32_t result;
6      if (n != 0) {
7          result = ack(m - 1, ack(m, n - 1));
8      } else {
9          result = ack(m - 1, 1);
10     }
11     return result;
12 }
```

Obr. 7.2: Výstup dekompilácie pri známych typoch parametrov.

### Filtrácia parametrov variadických funkcií

V prípade, že je zistený typ funkcie a tá je variadická, je potrebné každé volanie filtrovať zvlášť, bez aplikácie prieniku hodnôt. Každé volanie môže mať totiž rôzny počet parametrov. Okrem toho sa implementovaný algoritmus pokúsi vyhľadať formátovací reťazec, prehľadaním inštrukcií zápisov do hodnôt parametrov. Pre tieto účely bolo nutné rozšíriť implementáciu kolektora, aby k hodnotám parametrov uložil aj inštrukcie zápisu, na základe ktorých boli objavené. V prípade zisku formátovacieho reťazca využije knižnicu pre jeho spracovanie. Objavené typy parametrov volania sú následne využité pre elimináciu hodnôt rovnakým štýlom, akým sú filtrované hodnoty na základe známych typov, popísane vyššie. Ak formátovací reťazec variadickej funkcie nie je známy, využije pre filtráciu hodnôt každého volania algoritmy pre *elimináciu registrov* a *elimináciu zásobníkových premenných* opísané v sekcii 6.2.2.

### 7.2.2 Zohľadnenie filtrovania alternatívnymi registrami

Na základe analýzy volacej konvencie definovanej pre Microsoft x64 ABI v sekcii 3.3.2 vznikla požiadavka pre možnosť eliminácie hodnôt na základe alternatívnych sád registrov pre parametre. Vzhľadom na jedinečnosť povahy tohto problému bola vybraná možnosť vytvoriť pre túto architektúru samostatný objekt filtra. Tento objekt však len rozširuje elimináciu hodnôt o nový algoritmus. Aj keď by bolo možné zakomponovať do návrhu možnosť alternatívnych registrov, dané riešenie by vytvorilo ťažšie udržateľný a menej čitateľný kód. Nový objekt filtru je následne využitý len v prípade, že bol zistený prekladač od spoločnosti Microsoft. V ostatných prípadoch sa pre filtráciu využije objekt implementujúci všeobecný algoritmus.

### 7.2.3 Predávanie a navrátenie objektov veľkých typov

Nová implementácia zohľadňuje možnosť predávania parametrov veľkých typov. Niektoré volacie konvencie predávajúce parametre v registroch, predajú objekt v prípade že má typ väčší ako veľkosť registra na zásobníku a všetky následujúce parametre taktiež. Pri filtrácii na základe známych typov simuluje spôsob predania veľkých typov u rôznych volacích konvencií. Niektoré volacie konvencie predávajú veľké typy odkazom, teda sa pre parameter vyhradí jedno pamäťové miesto. Na druhú stranu, iné konvencie predávajú veľké typy tak, že predajú jednotlivé podčasti osobitne. Všetky tieto informácie o konvenciách sú podávané objektom volacej konvencie.

### 7.2.4 Analýza nedostatkov

Aj napriek tomu, že sa v novej implementácii ráta s možnosťami predania veľkých objektov, nie sú tieto objekty ešte plne rekonštruované nakoľko by to vyžadovalo rozsiahlejší zásah do spätného prekladača. Napríklad v prípade výskytu hodnoty, ktorá by bola predaná v dvoch pamäťových jednotkách (registroch, zásobníkových premenných), sa vygeneruje práca len s hlavnou časťou (spodné bajty). Aj napriek tomu, že sa vygeneruje práca len s časťou skutočnej hodnoty, neznamená to v každom prípade nekvalitný výstup. Rovnaký problém sa týka aj rekonštrukcie návratu veľkých objektov.

Z analýzy nedostatkov v sekcii 5.3 vznikla požiadavka na schopnosť filtrácie registrov, ktoré sa nachádzajú v registrovom okne. Pre vyriešenie tohto nedostatku bolo rozhodnuté, že nebude ovplyvnený všeobecný algoritmus filtrácie. Aktuálne architektúra SPARC nie je podporovaná spätným prekladačom a v prípade pridania jej podpory bude potrebné upraviť algoritmus eliminácie registrov, aby dokázal rátať s touto možnosťou.

Na druhú stranu, návrh filtra je dosť všeobecný na to, aby ho bolo možné rozšíriť a doplniť o požadovanú funkcionalitu. V prípade potreby je napríklad možné ľahko implementovať filtráciu užívateľmi definovaných konvencií.

## 7.3 Generované registre SSE

Na základe návrhu opísaného v sekcii 6.3.1, boli vytvorené objekty registrov. Analýzou výstupov spätného prekladu však bolo zistené, že dané riešenie generuje príliš komplikovaný program LLVM IR, ktorý nie je možné ľahko spracovať ďalšími prechodmi.

Vzhľadom na túto skutočnosť bol implementovaný návrh mierne upravený. Pre podčasti pohľadov registrov XMM, ktoré sú dôležite z pohľadu sémantiky potrebných inštrukcií, sú generované globálne premenné. Tieto globálne premenné majú typ ukazovateľa na

požadovaný typ a sú inicializované na pamäťové miesto podčasti požadovaného pohľadu. V tomto návrhu stále ostáva jedna globálna premenná registru, ktorý obsahuje pamäť, na ktorú sa nahliada rôznymi pohľadmi. Generované pohľady registrov XMM je možné vidieť na ukážke 7.3.

```
@xmm0 = internal global i128 0

@xmm0_f3 = internal global float*
               bitcast (i8* getelementptr (
                   i8,
                   i8* bitcast (i128* @xmm0 to i8*),
                   i64 12
               ) to float*)

@xmm0_d1 = internal global double*
               bitcast (i8* getelementptr (
                   i8,
                   i8* bitcast (i128* @xmm0 to i8*),
                   i64 8
               ) to double*)
```

Obr. 7.3: Generovanie globálnych pohľadov registrov XMM

Oproti návrhu tento spôsob implementácie registrov zjednodušil popis sémantiky požadovaných inštrukcií. Inštrukcie už nemusia totiž generovať pretypovanie a pristupovanie na požadované pamäťové miesto. Okrem toho sa takisto zjednodušil popis podčastí v optimalizáciach. S každou podčasťou je možné v optimalizáciach pracovať ako so samostatným registrom. Výstup dekompilácie po rozšírení implementácie je možné vidieť na ukážke 7.4. Okrem výstupu je na ukážke zobrazené porovnanie s výstupom pred zavedením zmien v implementácii.

Dané riešenie takisto otvorilo nové možnosti ako spracovávať jednotlivé podčasti v ďalších fázach spätného prekladača. Napríklad je možné vytvoriť špeciálnu detekciu generovaných vzorov práce s týmito registrami a nahradiť ich v zadnej časti spätného prekladu prácou s dátovým typom `union`.

```
int64_t v1 = g7 - 8;
*(int64_t *)v1 = g6;
*(int32_t *)(v1 - 24) = 0;
*(int32_t *)(v1 - 20) = 0;
function_1050();
int128_t v2 = __asm_cvtsi2sd(
                  *(int32_t *)(v1 - 20)
                  + *(int32_t *)(v1 - 24));
int128_t v3 = __asm_movsd(
                  0x4000000000000000);
*(int64_t *)(v1 - 16) = __asm_movsd_1(
                  __asm_divsd(v2, v3));
__asm_movsd(*(int64_t *)(v1 - 16));
function_1040();
```

```
int32_t v1 = 0;
int32_t v2 = 0;
scanf("%d %d", &v1, &v2);
uint32_t v3 = v1;
uint32_t v4 = v2;
*g8 = (float64_t)(v4 + v3);
*g10 = 2.0;
*g8 = *g8 / *g10;
printf("Avg(%d,%d) = %g\n",
           (int64_t)v3,
           (int64_t)v4,
           *g8);
```

Obr. 7.4: Spätný preklad pred (vľavo) a po (vpravo) zavedení nových zmien v dekóderi.

# Kapitola 8

# Testovanie vytvorenej podpory

Cieľom vývojárov spätného perkladača RetDec je vytvoriť kvalitný a spoľahlivý produkt, ktorý bude reprezentovať spoločnosť, ktorá ho vyvíja. Vzhľadom na rozšírenosť projektu je žiadané, aby sa minimalizoval rozsah chýb, ktoré sa dostanú do produkcie k užívateľovi, pomocou testovania. Podľa zdroja [21] je testovanie možné opísať ako:

> „Proces, alebo skupina procesov, navrhnutých pre zaistenie, že program robí to, na čo bol navrhnutý, a že nevykoná nič neočakávané.“

Pre overenie funkčnosti implementovanej funkcionality sa v RetDecu využíva niekoľko metód testovania. V následujúcej kapitole sa nachádza opis jednotlivých spôsobov testovania spätného prekladača, najmä z pohľadu pridania podpory architektúry x64.

## 8.1 Jednotkové testy

Jednotkové, niekedy tiež označované modulové, testovanie je proces, pri ktorom sa testujú individuálne podprogramy, subrutiny, alebo procedúry programu v izolovanom prostredí [21, Kapitola 5]. Cieľom je overiť, či vyvíjaný modul plní požadovanú funkcionalitu.

Pre každý modul spätného prekladača je vytvorený ideálny vstup a overuje sa, či modul splnil svoju činnosť na základe porovnania jeho výstupu s očakávaným výstupom. V následujúcej časti textu sú opísané princípy vytvorených jednotkových testov pre priechod dekódera a optimalizácie parametrov a návratových hodnôt.

### 8.1.1 Optimalizácia parametrov a návratových hodnôt

Pre otestovanie funkčnosti novej optimalizácie bola rozšírena základná testovacia sada pre daný priechod o nové testy. Princíp testov však ostal zachovaný. Pre overenie funkčnosti analýzy sú vytvorené špeciálne programy v LLVM IR, ktoré sú predávané optimalizácii na vstup. Zároveň sa nastavia všetky potrebné objekty nutné pre priechod, ako `ABI`, `config`. Pre každý test je vytvorený očakávaný výstup – upravený program LLVM IR. Očakávaný a skutočný výstup sú následne porovnané a užívateľovi je zobrazený výsledok porovnania. Príklad jednotkového testu je možné vidieť v prílohe B.

Nové testy, zahŕňajú overenie funkčnosti pridaných volacích konvencií, pre existujúce, aj pridávané architektúry, ktorých podpora ešte nie je v spätnom prekladači. Tým bolo overené, že ak by podpora daných architektúr v budúcnosti vznikla, modul optimalizácie parametrov by pracoval podľa očakávania. Volacie konvencie boli testované na základe nastavenia volacej konvencie do konfiguračného súboru.

### 8.1.2 Priechod dekodéra

Pri jednotkovom testovaní dekodéra sa vytvorí test, ktorý nastaví v jazyku symbolických inštrukcií registre, inicializuje miesta v pamäti a naplánuje vykonanie inštrukcie. Špecifikovaná postupnosť v jazyku symbolických inštrukcii je predaná dekóderu, ktorý z nej vygeneruje modul LLVM IR. Kód vo vygenerovanom module je interpretovaný a v teste sa špecifikuje očakávaný stav objektov, ktoré by mal interpretovaný modul využiť. Testom budú ovlyvnené napríklad registre alebo mieta v pamäti. Taktiež je možné zistiť aké hodnoty boli volané interpretom, prípadne aké hodnoty boli načítané a zapísané počas vykonávania programu.

## 8.2 Regresné testy

Jedná sa o testovanie, ktoré je vykonávané po každom vylepšení, prípadne oprave programu. Významom regresného testovania je odhaliť, či vytvorená zmena neovplyvnila iné aspekty programu. Obvykle je testovanie vykonávané spustením programu na určitej podmnožine testovacích prípadov, ktorých prevedenie bolo pred zavedením zmeny úspešné. Regresné testovanie je dôležité, pretože zmeny a opravy chýb zvyknú byť viac náchylné na chyby ako pôvodný kód programu. [21, Kapitola 6]

Pre spätný prekladač RetDec bola vytvorená kolekcia regresných testov založených na dekompilácii binárnych súborov. Sadu testovacích súborov je možné rozdeliť do nasledujúcich kategórií:

1. Binárne súbory, ktorých zdrojový kód je známy.
2. Súbory (napr. zaslané užívateľmi), na základe ktorých bol objavený a vyriešený určitý problém spätného prekladača.
3. Vzorky škodlivého softvéru, ktoré boli analyzované spätným prekladačom.

Regresné testy sú uložené v stromovej štruktúre, na základe kategórie do ktorej spadajú. Príklad úseku štruktúry je zobrazený na ukážke 8.1. Konkrétne testy sú uložené v moduloch `test.py`, kde sú špecifikované testovacie triedy, nastavenia a metódy overenia, či test prešiel úspešne alebo nie.

```
regression-tests/
├── bugs/
│   └── invalid-function-name/
│       ├── input.exe
│       └── test.py
└── integration/
    ├── ack/
    │   ├── ack.exe
    │   └── test.py
    └── ackermann/
        ├── ackermann.exe
        └── test.py
```

Obr. 8.1: Príklad štruktúry usporiadania testov.

Pre testovanie jednotlivých kategórii je využívaný nástroj pre písanie a spúšťanie regresných testov, viď. [5]. Nástroj umožňuje vytvárať testy, ktoré pracujú so spätným prekladačom za účelom testovania následujúcimi spôsobmi:

- Analýza návratovej hodnotu spätného prekladu.
- Rekompilácia výstupu spätného prekladača – programu v jazyku C. Následne overenie úspešnosti, spustenie programu s parametrami a overenie výstupu.
- Analýza programu na výstupe spätného prekladača – mená, typy, parametre funkcií, globálne premenné, prítomnosť reťazcov, apod.

### 8.2.1 Testy pre architektúru x64

Vzhľadom na pridanie podpory architektúry x64, bolo nutné rozšíriť sadu regresných testov o nové binárne súbory. Pre generovanie binárnych súborov pre účely testovania boli využité následovné prekladače:

- OS Windows
  - `gcc 6.1.0`
  - `clang 3.5.0`
- OS Linux
  - `gcc 4.7.2`
  - `clang 3.8.9`

Nástroje na vytvorenie binárnych súborov boli zvolené tak, aby výsledky regresných testov boli po spätnom preklade rekompilovateľné vo väčšine prípadov. To je zabezpečené tým, že RetDec má vytvorenú pre tieto prekladače detekciu staticky linkovaného kódu. Pre účely regresného testovania bolo vygenerovaných nových 72 binárnych súborov architektúry x64.

## 8.3 Nočné testy

Jedná sa o obsiahlu množinu testov, ktorých vykonanie zaberie niekoľko hodín. Vzhľadom na túto skutočnosť sa testy väčšinou spúšťajú v noci. Tento druh testovania je vykonávaný v prípade zavedenia veľkých, potenciálne nebezpečných zmien v implementácii. V prvom rade je však žiadané, aby takého zmeny boli úspešne skontrolované regresnými testami. Počas testovania RetDec dekompiluje desiatky tisíc binárnych súborov, z ktorých sa zbierajú rôzne informácie, napríklad:

- návratové hodnoty prekladu,
- výstupy spätného prekladača,
- čas vykonávania každej dekompilácie,
- množstvo využitej pamäte.

Všetky získane dáta sú agregované v MySQL databáze, analyzované a nakoniec zobrazené užívateľovi pomocou webového rozhrania. Cez webové rozhranie je možné porovnať výsledky testovania so staršími behmi nočných testov. Vďaka tomu je možné zistiť, či zavedená zmena nezaniesla nepredpokladané chyby, prípadne spozorovať vylepšenia spätného prekladu. Užívateľské rozhranie nočných testov je možné vidieť v prílohe C.1.

Vzhľadom na rozsah novej implementácie optimalizácie parametrov a návratových hodnôt ju bolo nutné podrobiť nočným testom. Nevznikli výrazné zhoršenia, a teda nová analýza dokázala plne nahradiť predchádzajúcu.

Výsledok nočných testov rozšírených o binárne súbory x64 je možné vidieť na obrázku C.1. Vo výsledku nočných testov je možné vidieť veľké zhoršenia, nakoľko je výsledok porovnávaný s výsledkom bez súborov x64. V rozhraní je možné vidieť, že zhoršenia súvisia najmä s počtom dekompilácii, ktorý sa zvýšil o 13973. To ovlpyvnilo dobu potrebnú pre vykonanie testov, ktorá sa zvýšila o hodinu a 13 minút.

V prílohe C.2 je pohľad na nové neúspešné dekompilácie ukončené zlyhaním v jadre spätného prekladača. Pri bližšom skúmaní bolo zistené, že dekompilácie boli ukončené inými časťami spätného prekladača, než tými ktoré boli rozšírené v rámci tejto práce. Jedná sa však o výnimočné prípady, ktoré nemajú vplyv na všeobecnú funkcionalitu. V rámci budúcej práce na spätnom prekladači však bude možné detailne analyzovať príčiny neúspechu, a tým vylepšiť výsledky spätného prekladu.

# Kapitola 9

# Záver

V rámci tejto práce som sa zoznámil s princípmi reverzného inžinierstva a jeho uplatnením v praxi. Analyzoval som nástroje využívane pre reverzovanie v oblasti informačných technológii a zaoberal sa významom a princípom jednej z najvýznamnejších precesorových architektúr, architektúrou x86.

Študoval som činnosť spätných prekladačov a bližšie sa zameral na architektúru spätného prekladača vyvíjaného spoločnosťou Avast – RetDec. Venoval som sa analýze nedostatkov tohto prekladača z pohľadu rozšírenia podpory spätného prekladu o novú architektúru. Konkrétne som znalosti uplatnil pre pridanie podpory architektúry x64.

Pri analýze spätného prekladača som objavil chyby v preklade, týkajúce sa analýzy volacích konvencií. Pre vytvorenie návrhu opravy nájdených nedostatkov som sa venoval princípom rôznych volacích konvencií. Na základe získaných poznatkov som navrhol riešenie vo forme všeobecnej analýzy parametrov a vytvorený návrh som implementoval. Implementáciu som priebežne testoval a po dokončení som jej spoľahlivosť overil na veľkej sade binárnych súborov.

Aktuálne je podpora prekladu aplikácii x64 v RetDecu začlenená do produkcie pre experimentálne vyskúšanie užívateľom[1]. Nová optimalizácia je schopná generovania kvalitnejšieho kódu u volacích konvencií, ktoré neboli podporované. Implementované riešenie takisto otvorilo nové možnosti a vylepšenia, ktoré je teraz možné pridať do spätného prekladača. Napríklad návrh filtrácie parametrov umožňuje ľahké rozšírenie analýzy funkcií na základe užívateľom definovaných volacích konvencií. Taktiež podporuje generovanie práce s objektami v časti optimalizácie parametrov v prípade, že sa pre RetDec posilní práca s objektami.

Okrem vytvorenia novej optimalizácie parametrov a návratových hodnôt sa práca zaoberala pridaním podpory inštrukčnej sady SSE. Pre inštrukcie tejto sady sa v RetDecu generujú aktuálne pseudovolania, ktoré užívateľovi povedia, že sa na danom mieste nachádza nepodporovaná inštrukcia. Bol vytvorený návrh riešenia skúmaním rôznych výstupov prekladačov. Vytvorené riešenie bolo implementované a testované. Napriek tomu však riešenie vyžaduje podporu nových optimalizácií, ktoré skvalitnia výstup pre užívateľa. Riešenie sa opiera o náročnú prácu s ukazovateľmi, ktorá ešte nie je plne podporovaná aktuálnymi optimalizáciami v spätnom prekladači.

[1] https://github.com/avast/retdec/releases/tag/v3.3

# Literatúra


[1] Capstone disassembler engine. [viď. 16. 1. 2018].
URL http://www.capstone-engine.org/

[2] Ghidra. [viď. 28. 4. 2019].
URL https://www.nsa.gov/resources/everyone/ghidra/

[3] Hex-Rays Decompiler. [viď. 28. 4. 2019].
URL https://www.hex-rays.com/products/decompiler/

[4] Retargetable Decompiler. [viď. 28. 4. 2019].
URL https://retdec.com/

[5] RetDec Regression Tests Framework. [viď. 2. 5. 2019].
URL https://retdec-regression-tests-framework.readthedocs.io/en/latest/

[6] LLVM's Analysis and Transform Passes. [online], Apríl 2019, [viď. 29. 4. 2019].
URL https://llvm.org/docs/Passes.html

[7] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006, ISBN 978-0321486813.
URL https://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811

[8] Arnab, A.; Hutchison, A.: Digital rights management-An overview of current challenges and solutions. 01 2004.

[9] Chikofsky, E. J.; Cross, J. H.: Reverse engineering and design recovery: a taxonomy. *IEEE Software*, ročník 7, č. 1, Január 1990: s. 13–17, ISSN 0740-7459.

[10] Cloutier, F.: x86 and amd64 instruction reference. [online], December 2018, [viď. 23. 12. 2018].
URL https://www.felixcloutier.com/x86/

[11] Dang, B.; Gazet, A.; Bachaalany, E.; aj.: *Practical Reverse Engineering*. John Wiley & Sons Inc, 2014, ISBN 1118787315.

[12] Dass, L. B.: *The X86 Microprocessors: Architecture and Programming (8086 to Pentium) (Old Edition)*. Pearson, 2010, ISBN 9788131732465.
URL https://www.amazon.com/X86-Microprocessors-Architecture-Programming-Pentium-ebook/dp/B00G4YDRM4

[13] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons Inc, prvé vydanie, 2005, ISBN 978-0764574818.

[14] George, A. D.: An overview of RISC vs. CISC. In *[1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory*, March 1990, ISSN 0094-2898, s. 436–438, doi:10.1109/SSST.1990.138185.

[15] Helm, R.; Johnson, R.; Vlissides, J.; aj.: *Design Patterns*. Prentice Hall, 1995, ISBN 978-0201633610.
URL https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612

[16] Hollasch, S.: IEEE Standard 754 Floating Point Numbers. [online], August 2018, [vid. 22. 1. 2019].
URL https://steve.hollasch.net/cgindex/coding/ieeefloat.html

[17] Intel Corporation: *Intel®64 and IA-32 Architectures Software Developer's Manual*.
URL https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[18] Křoustek, J.; Matula, P.; Zemek, P.: RetDec: An Open-Source Machine-Code Decompiler. [talk], December 2017, presented at Botconf 2017, Montpellier, FR.

[19] Lattner, C.; Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[20] Matz, M.; Hubička, J.; Jaeger, A.; aj.: *System V Application Binary Interface*. November 2014.

[21] Myers, G. J.; Sandler, C.; Badgett, T.: *The Art of Software Testing*. WILEY, 2011, ISBN 1118031962.
URL https://www.amazon.com/Art-Software-Testing-Glenford-Myers-ebook/dp/B005PETXRM

[22] Robertson, C.: x64 calling convention. 2018, [vid. 16. 1. 2019].
URL https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention

[23] Schwarz, B.; Debray, S.; Andrews, G.: Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, Nov 2002, ISSN 1095-1350, s. 45–54, doi:10.1109/WCRE.2002.1173063.

[24] Wenzel, M.: Memory-Mapped Files. [online], Marec 2017, [vid. 22. 1. 2019].
URL
https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files

# Príloha A

# Inštrukcie a registre architektúry x64

| | |
|---|---|
| EAX/RAX | Akumulátor, slúži pre operandy a výsledky operácií. |
| EBX/RBX | Ukazovateľ na dáta. |
| ECX/RCX | Počítadlo pre reťazce znakov a cyklické operácie. |
| EDX/RDX | Ukazovateľ na vstup/výstup. |
| ESI/RSI | Ukazovateľ na zdrojový operand pri operáciách nad reťazcami znakov. |
| EDI/RDI | Ukazovateľ na cieľ[1] uloženia dát pri operáciách nad reťazcami znakov. |
| ESP/RSP | Ukazovateľ na vrchol zásobníka dát. |
| EBP/RBP | Ukazovateľ na dáta v zásobníku dát. |

Tabuľka A.1: Registre GPR x86 a ich odporúčané použitie.

| | |
|---|---|
| CDQE | Znamienkové rozšírenie EAX na RAX. |
| CQO | Znamienkové rozšírenie RAX na RDX:RAX. |
| CMPSQ | Porovnanie dvoch 64-bitových hodnôt. |
| CMPXCHG16B | Porovná a zamení 16 bajtov. |
| IRETQ | 64-bitový návrat z prerušenia. |
| JRCXZ | Vykoná skok ak hodnota v RCX je nula. |
| LODSQ | Načíta 64-bitový reťazec. |
| MOVSXD | Inštrukcia MOV so znamienkovým rozšírením 32-bitov na 64-bitov. |
| POPFQ | Načíta a odstráni hodnotu RFLAGS zo zásobníku. |
| PUSHFQ | Vloží hodnotu RGLAGS na zásobník. |
| RDTSCP | Prečíta čas a procesorové ID. |
| SCASQ | Porovná 64-bitov v RAX s hodnotou odkazovanou pomocou RDI. |
| STOSQ | Uloží 64-bitovú hodnotu v RAX na adresu v RDI. |
| SWAPGS | Nahradí hodnotu v GS registri[2]. |

Tabuľka A.2: Nové inštrukcie architektúry x64.

[1] Pri niektorých inštrukciách môže byť register EDI využitý aj ako ukazovateľ na zdroj.
[2] https://www.felixcloutier.com/x86/swapgs

| | |
|---|---|
| CVTSI2SS | Konverzia 32 alebo 64 bitového čísla do FP[3] reprezentácie. |
| CVTSS2SI | Inštrukcia komplementárna k inštrukcii CVTSI2SS. |
| CVTSI2SD | Chovanie ako inštrukcia CVTSI2SS, však FP v dvojnásobnej presnosti. |
| CVTSD2SI | Inštrukcia komplementárna k inštrukcii CVTSI2SD. |
| CVTSS2SD | Konverzia čísla FP do dvojnásobnej presnosti. |
| CVTSD2SS | Inštrukcia komplementárna k inštrukcii CVTSS2SD. |
| MOVSS | Presun FP čísla medzi registrami XMM, alebo pamäťou. |
| MOVSD | Podobne ako MOVSS, však FP v dvojnásobnej presnosti, |
| ADDSS | Súčet dvoch FP čísel. |
| ADDSD | Súčet dvoch FP čísel v dvojnásobnej presnosti. |
| SUBSS | Rozdiel dvoch FP čísel. |
| SUBSD | Rozdiel dvoch FP čísel v dvojnásobnej presnosti. |
| MULSS | Súčin dvoch FP čísel. |
| MULSD | Súčin dvoch FP čísel v dvojnásobnej presnosti. |
| DIVSS | Podiel dvoch FP čísel. |
| DIVSD | Podiel dvoch FP čísel v dvojnásobnej presnosti. |
| CMPSS | Porovnanie dvoch FP čísel. |
| CMPSD | Porovnanie dvoch FP čísel v dvojnásobnej presnosti. |

Tabuľka A.3: Inštrukcie SSE pracujúce s číslami v FP reprezentácii.

.

[3]FP – číslo s pohyblivou desatinnou čiarkou.

# Príloha B

# Tvorba jednotkových testov

V tejto prílohe sa nachádza príklad jednotkového testu, ktorým sa overuje funkčnosť optimalizácie parametrov a návratových hodnôt. Príkladom je cielené demonštrovať proces, ktorým sú vytvárané jednotkové testy. Okrem toho je však možné príkladom demonštrovať činnosť optimalizácie parametrov a návratových hodnôt. Príklad je založený na jednotkovom teste, ktorý je reálne využívaný.

1. Je potrebné vytvoriť program LLVM, ktorý obsahuje volanie, definíciu a prácu s hodnotami ktoré môžu, prípadne hodnotami ktoré nemôžu byť parametre. Pre účely testu majme modul LLVM, ktorý by bolo možné získať pri architektúre x64:

```
@rsi = global i64 0
@rcx = global i64 0
@rdx = global i64 0
@r8 = global i64 0
@r9 = global i64 0
@rax = global i64 0

declare void @print()

define void @fnc() {
  %stack_-8 = alloca i64
  %stack_-16 = alloca i64
  store i64 1, i64* @r9
  store i64 1, i64* @r8
  store i64 1, i64* @rsi
  store i64 2, i64* %stack_-8
  store i64 1, i64* @rdx
  store i64 2, i64* %stack_-16
  store i64 1, i64* @rcx

  call void @print()

  ret void
}
```

Obr. B.1: Príklad vstupného modulu jednotkového testu priechodu parametrov a návratových hodnôt.

2. Vytvorený testovací program je potrebné načítať vo forme reťazca a predať ako parameter metóde pre spracovanie LLVM modulov. Tá vytvorí obraz, ktorý je možné predať na vstup testovanej optimalizácii.

3. Inicializuje sa objekt konfigurácie zo špecifikácie vo formáte JSON. Konfigurácia podáva dôležité informácie o objektoch modulu LLVM a taktiež o vstupnom binárnom súbore spätného prekladača. Ďalej bude predpokladané, že súbor z ktorého modul LLVM vznikol mal formát `PE`. To znamená, že bol vytvorený pre spustenie na operačnom systéme Windows. V konfigurácii sú nastavené aj ďalšie informácie ako veľkosť slova, zistený prekladač a zásobníkové premenné. Zásobníkové premenné sú špecifikované štýlom, akým by ich funkciám priradil priechod pre ich analýzu. Pre účely testu bude potrebný konfiguračný súbor z ukážky B.2.

```
{
  "architecture": {
    "bitSize": 64,
    "endian": "little",
    "name": "x86"
  },
  "functions": [
    {
      "name": "fnc",
      "locals": [
        {
          "name": "stack_-8",
          "storage": {
            "type": "stack",
            "value": -8
          }
        },
        {
          "name": "stack_-16",
          "storage": {
            "type": "stack",
            "value": -16
          }
        }
      ]
    }
  ],
  "fileFormat": "pe64",
  "tools": [
    {
      "name": "gcc"
    }
  ]
}
```

Obr. B.2: Príklad konfiguračného súboru vo formáte JSON.

4. Nastaví sa objekt ABI tak, aby vedel špecifikovať aké registre predstavujú jednotlivé globálne premenné v module LLVM. Obyčajne je pri spätnom preklade objekt ABI inicializovaný vo fáze dekódovania, kde sú generované registre pre každú architektúru.

Pri jednotkovom testovaní sa však testujú izolované moduly, preto musí test nastaviť objekt ABI samostatne.

5. Je potrebné určiť očakávaný výstup. Pre architektúru x64 a formát `PE` by sa mala na analýzu parametrov využiť volacia konvencia Microsoft x64, popísaná v sekcii 3.3.2. Je možné očakávať, že registre budú využité pre predávanie parametrov v poradí `RCX`, `RDX`, `R8`, `R9` a po nich budú využité zásobníkové premenné. Register `RSI` nebude využitý vôbec. Po prevedení priechodu bude na vástupe očakávaný modul na ukážke B.3.

```
@rsi = global i64 0
@rcx = global i64 0
@rdx = global i64 0
@r8 = global i64 0
@r9 = global i64 0
@rax = global i64 0

declare i64 @print(i64, i64, i64, i64, i64, i64)
declare void @0()

define i64 @fnc() {
  %stack_-8 = alloca i64
  %stack_-16 = alloca i64
  store i64 1, i64* @r9
  store i64 1, i64* @r8
  store i64 1, i64* @rsi
  store i64 2, i64* %stack_-8
  store i64 1, i64* @rdx
  store i64 2, i64* %stack_-16
  store i64 1, i64* @rcx
  %1 = load i64, i64* @rcx
  %2 = load i64, i64* @rdx
  %3 = load i64, i64* @r8
  %4 = load i64, i64* @r9
  %5 = load i64, i64* %stack_-16
  %6 = load i64, i64* %stack_-8
  %7 = call i64 @print(
                  i64 %1, i64 %2, i64 %3,
                  i64 %4, i64 %5, i64 %6)
  store i64 %7, i64* @rax
  %8 = load i64, i64* @rax
  ret i64 %8
}
declare void @1()
```

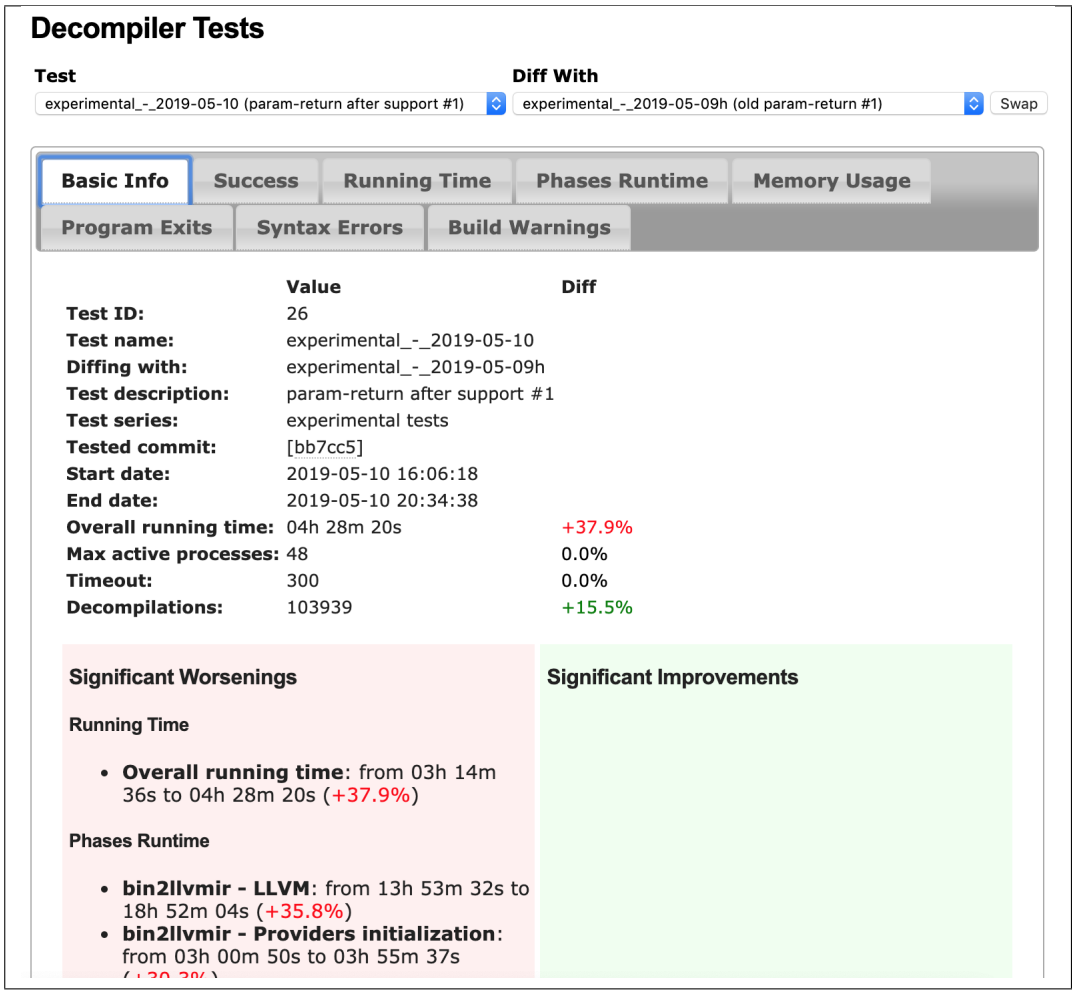
Obr. B.3: Predpokladaný výstup optimalizácie parametrov a návratových hodnôt pri vstupe modulu z ukážky B.1

6. Zavolá sa metóda pre spustenie priechodu a porovná sa výstupný modul LLVM IR s očakávaným.

V ukážke B.3 je možné vidieť, že sa očakáva vygenerovanie deklarácii funkcií s číselným názvom 0 a 1. Jedná sa o artefakt modulu pre modifikovanie LLVM IR. Pri zmene signatúry funkcie sa pre jednoduchosť vytvorí nová, ktorá obsahuje požadované vlastnosti. Od menej funkcie sa následne vezme názov, prípadne telo definície, a presunie sa k novej funkcii.

# Príloha C

# Výsledky nočných testov




Obr. C.1: Výsledok nočných testov po pridaní x64 binárnych súborov.

| bin2llvmir: | RC | Name | Count | Diff | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | success | 103754 | +15.3% | | | | |
| | 1 | - | 155 | +5066.7% | [Toggle Details] | | | |
| | **Category** | | **Bar** | | **Percentage** | **Count** | **Diff** | |
| | web-service (binary) - x86-64/pe | | | | 34.2% | 53 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86-64/elf | | | | 32.9% | 51 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86-64/macho | | | | 21.9% | 34 | +100.0% | [Toggle Cases] |
| | other | | | | 5.2% | 8 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86-64/coff | | | | 3.2% | 5 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86/elf | | | | 1.3% | 2 | 0.0% | [Toggle Cases] |
| | malware (binary) - mips/elf | | | | 0.6% | 1 | 0.0% | [Toggle Cases] |
| | web-service (binary) - arm64/elf | | | | 0.6% | 1 | +100.0% | [Toggle Cases] |
| | 134 | abort | 0 | 0.0% | | | | |
| | 135 | memory | 0 | 0.0% | | | | |
| | 137 | timeout | 29 | +2800.0% | [Toggle Details] | | | |
| | **Category** | | **Bar** | | **Percentage** | **Count** | **Diff** | |
| | web-service (binary) - x86-64/macho | | | | 48.3% | 14 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86-64/pe | | | | 31.0% | 9 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86-64/elf | | | | 17.2% | 5 | +100.0% | [Toggle Cases] |
| | web-service (binary) - x86/pe | | | | 3.4% | 1 | 0.0% | [Toggle Cases] |
| | 139 | sigsegv | 1 | +100.0% | [Toggle Details] | | | |
| | **Category** | | **Bar** | | **Percentage** | **Count** | **Diff** | |
| | web-service (binary) - x86-64/pe | | | | 100.0% | 1 | +100.0% | [Toggle Cases] |

Obr. C.2: Výsledok testovania jadra spätného prekldača po pridaní nových x64 súborov.

# Príloha D

# Obsah CD

Priložené CD obsahuje následujúci obsah:

- text práce vo formáte PDF – **`bp-xkubov06.pdf`**
- adresár **`tex`** so súbormi LaTeX,
- adresár **`examples`** s príkladmi, na ktorých je možné spätný preklad vyskúšať,
- adresár **`retdec`** obsahujúci repozitár so zdrojovými kódmi spätného prekladača a jednotkovými testami,
- súbor **`README.md`**.