BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2020
http://excel.fit.vutbr.cz

Spätný preklad špecializovaných a pokročilých inštrukčných sád nástrojom RetDec

Juraj Holub*

Abstrakt
V dnešnej dobe je proces analýzy nebezpečného softvéru dôležitou súčasťou informačných technológií. Jedna z kľúčových techník je spätný preklad škodlivých binárnych programov. Spätný preklad je komplexný proces, ktorým sa zaoberá niekoľko projektov. Projekt RetDec sa zameriava na flexibilný návrh a riešenie spätného prekladača s možnosťou znovupoužiteľnosti. Ide o open-source projekt vedený firmou Avast. Tento článok sa zaoberá návrhom nového rozšírenia pre spätný prekladač RetDec v oblasti podpory špecializovanej inštrukčnej sady pre jednotku FPU, ktorá je súčasťou procesorovej architektúry x86.

Kľúčové slová: RetDec, decompiler, Avast, reverse engineering, x86, FPU

Priložené materiály: RetDec Demonstration Video — Downloadable Code

*xholub40@stdu.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

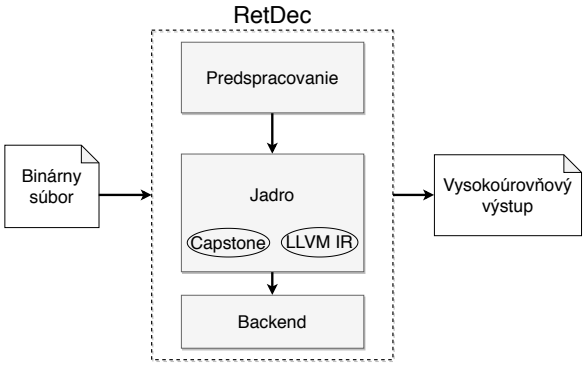
Táto práca sa zaoberá využitím reverzného inžinierstva v oblasti softvérových technológií. Reverzné inžinierstvo je všeobecne metóda získavania informácií alebo plánov o akýchkoľvek objektoch vytvorených človekom. V oblasti informačných technológií je význam tejto disciplíny najmä v rámci kybernetickej bezpečnosti. Táto technika je využívaná tvorcami škodlivého softvéru (tzv. malvér). Malvér využíva reverzné inžinierstvo na získavanie citlivých informácií o operačnom systéme s potenciálnym cieľom získať kontrolu nad zariadením. Ďalšia rozšírená oblasť je softvérové pirátstvo, kedy sa útočník snaží prelomiť ochranu komerčného digitálneho obsahu ako sú knihy, filmy, hudba, hry alebo rôzne platené programy. Na druhej strane môže pomôcť práve pri analýze malvéru za účelom zvýšenia bezpečnosti voči danému softvéru. Jedna z kľúčových techník je analýza malvéru pomocou programu všeobecne nazývaného spätný prekladač. Spätný prekladač je program, ktorý analyzuje spustiteľné binárne súbory a zrekonštruuje vysoko úrovňový výstup, napríklad v podobe grafu alebo kódu v programovacom jazyku. [1]

V dnešnej dobe existuje niekoľko projektov spätných prekladačov. Projekt RetDec sa zameriava na vytvorenie open-source nástroja, ktorý je rozdelený na viacero knižníc. Takýto návrh má za cieľ umožniť znovupoužiteľnosť jednotlivých nástrojov spätného prekladača. Cieľom tejto práce je navrhnúť optimalizačnú komponentu pre tento spätný prekladač. Navrhovaná komponenta optimalizuje preklad inštrukčnej sady jednotky FPU pre architektúru x86.

2. Spätný prekladač RetDec

RetDec (Retargetable Decompiler[1]) si dáva za cieľ vykonávať spätný preklad nezávisle od procesorovej

[1] Projekt RetDec: https://retdec.com/



Obrázok 1. Schéma architektúry spätného prekladača RetDec a technológie, ktoré používa.

architektúry, operačného systému a formátu spustiteľných binárnych súborov. Obrázok 1 zobrazuje architektúru prekladača a technológie použité v jednotlivých častiach prekladača. Kľúčovú technológiu tvorí LLVM a Capstone[2]. Prekladač je štruktúrovaný do troch hlavných blokov a každý blok sa skladá z množstva menších knižníc. Tieto bloky sú zreťazené v nasledujúcej sekvencii:

1. **Predspacovanie** zjednotí a zanalyzuje binárne súbory a extrahované metadáta sú vstupom pre jadro prekladača.
2. **Jadro** prekladača vytvára vnútornú reprezentáciu kódu v medzi jazyku (anglicky Intermediate Representation, ďalej len IR). Program aplikuje desiatky optimalizácií, ktoré transformujú IR. Optimalizovaná IR je výstupom tohto bloku.
3. **Backend** aplikuje rôzne optimalizácie s cieľom zvýšiť zrozumiteľnosť kódu a na záver vygeneruje výstup v jazyku C.

Knižnica LLVM

Pre prácu s IR v jadre prekladača sa používa knižnica LLVM. LLVM definuje bežnú, nízkoúrovňovú reprezentáciu podobnú assamblerovskej rodine inštrukčnej sady RISC. Zároveň ale poskytuje informácie užitočné pre efektívne analyzovanie a optimalizovanie IR s cieľom vyššej abstrakcie. Knižnica napríklad umožňuje jazykovo-nezávislý typový systém, graf toku riadenia alebo typovanú množinu registrov. LLVM IR neposkytuje vlastnosti vysokoúrovňových programovacích jazykov ako sú triedy, dedičnosť alebo spracovanie výnimiek. Cieľom LLVM IR nie je byť univerzálny IR ale je naopak doplnkom vysokoúrovňových virtuálnych strojov ako sú Smalltalk VM alebo Self VM. Benefity LLVM IR sa plne využijú pri reprezentácii staticky prekladaných jazykov ako je C a C++. [2]

[2]Capstone projekt. http://www.capstone-engine.org/

Základný koncept syntaktickej reprezentácie LLVM IR sa skladá z nasledujúcich datových štruktúr (viď [3]):

- **Modul** definuje obsah celého LLVM IR súboru.
- **Funkcia** je koncept syntakticky podobný procedúre v programovacom jazyku C. Telo funkcie je definované sekvenciou základných blokov.
- **Základný blok** predstavuje jeden uzol v grafe toku riadenia (anglicky *Control Flow Graph*, ďalej len *CFG*) v rámci jeho funkcie. Blok reprezentuje sekvenciu inštrukcií s jedným vstupným bodom a jedným výstupným bodom.
- **Inštrukcia** je najnižšia úroveň abstrakcie, ktorá tvorí typicky trojadresný kód s maximálne dvoma vstupmi a jedným výstupom. LLVM IR obsahuje inštrukcie na aritmetické operácie, logické operácie, bitové operácie, zápis a čítanie z pamäte, porovnávanie hodnôt, volanie funkcií a ďalšie.

Príklad 1 zobrazuje základné syntaktické prvky v LLVM module. Tento modul obsahuje jednu definíciu funkcie s globálnym názvom `@sum`. Taktiež definuje jednu globálnu premennú `@GLOBAL_VAR`, ktorej dátový typ je 32-bitový integer. Telo funkcie obsahuje jediný základný blok s označením `entry`. Vyjadrovacia schopnosť inštrukcií v základnom bloku je veľmi podobná assemblerovským inštrukciám, avšak navyše explicitne definuje dátové typy.

```
1 @GLOBAL_VAR = external global i32
2
3 define i32 @sum(i32 %a, i32 %b) {
4 entry:
5     %add = add i32 %b, %a
6     ret i32 %add
7 }
```

Príklad 1. Príklad syntaktických prvkov LLVM IR.

Jadro RetDec

Navrhované rozšírenie je súčasťou jadra spätného prekladača, preto nasledujúca podsekcia priblíži princíp jeho fungovania. Jadro spätného prekladača v prvej fáze mapuje assemblerovské inštrukcie získané spätným prekladom na LLVM IR. Cieľom mapovania nie je preložiť tieto inštrukcie s plným sémantickým významom. Zámerom prekladača je vygenerovať jednoducho pochopiteľný výstup v C/C++. Takýto výstup je následne efektívne analyzovateľný ľuďmi. Preto sémantické prekladanie inštrukcií do LLVM IR prebieha pre konkrétne inštrukcie v štyroch módoch:

1. **Plný preklad**: Inštrukcie, ktorých plný sémantický význam sa dá vyjadriť jednoduchou

sekvenciou LLVM IR.Ide tipycky o aritmetické inštrukcie a inštrukcie na transformáciu dát).

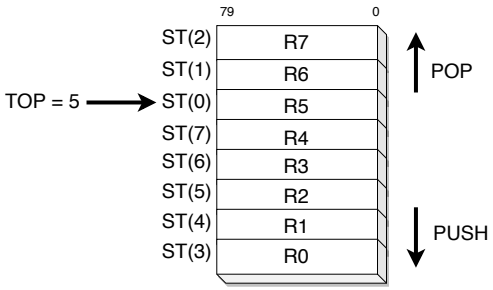
2. **Pseudo assembler funkcie**: Niektoré nízkoúrovňové inštrukcie nie je možné reprezentovať pomocou LLVM IR. Napríklad inštrukcia `FWAIT` sleduje prebiehajúce floating-point výnimky. Tento typ inštrukcií je reprezentovaný samovysvetľujúcou pseudo funkciou (napríklad `@__asm_fwait()`).
3. **Čiastočný preklad**: Inštrukcia je opäť reprezentovaná volaním pseudo funkcie. Navyše je ale explicitne poskytnutá informácia o riadení toku dát. Napríklad inštrukciu `FXSAVE [addr]`, ktorá uloží stav FPU, MMX, XMM jednotiek a ich registrov do 512 bajtov v pamäti na adresu `addr`. Preklad bude vyzerať ako volanie pseudo funkcie a následné uloženie 512 bajtov.
4. **Ignorovanie inštrukcie**: Inštrukcia `NOP` je typ inštrukcie, ktorej výskyt vo vysokoúrovňovom výstupe jazyka C je už nepotrebný a prekladač takéto inštrukcie ignoruje.

Po preklade inštrukcií je táto reprezentácia transformovaná množstvom optimalizačných priechodov, ktorých cieľom je zvýšiť abstrakciu a priblížiť IR výstupnému vysokoúrovňovému jazyku. V tejto časti spätného prekladu prebieha navrhované optimalizačné rozšírenie.

3. Preklad inštrukčnej sady FPU

Táto sekcia popisuje inštrukčnú sadu hardvérovej jednotky FPU (anglicky Floating Point Unit, viď [4]), ktorá slúži na výpočty pre čísla s plávajúcou desatinnou čiarkou (anglicky floating point, ďalej len FP). Sekcia ďalej vysvetľuje problémy vznikajúce pri mapovaní FPU inštrukcií do LLVM IR v spätnom prekladači RetDec.

FPU má vlastnú sadu inštrukcií a registrov. Obsahuje vlastné registre pre špeciálne účely ako sú riadiaci register alebo stavový register. Ďalej má osem datových registrov, ktoré sa nazývajú `R0` až `R7`. Tieto registre tvoria dátovú štruktúru zásobník a prístup k ním je podriadený princípom tejto dátovej štruktúry. Zásobník registrov zobrazuje Obrázok 2. Hardvérovým registrom je vždy relatívne priradené označenie `ST0` až `ST7`, kde `ST0` odkazuje na register, ktorý je aktuálne na vrchole zásobníka. Hodnota `TOP` určuje, ktorý hardvérový register je aktuálne vrcholom zásobníka. Zmena vrcholu zásobníka prebieha pomocou operácií *push* a *pop*. Inštrukcia *push* (alternatívne *load*) dekrementuje `TOP` a presunie obsah operandu inštrukcie do registru `ST0`. Inštrukcia *pop* (alternatívne *store*) presunie obsah registra `ST0`



Obrázok 2. Abstrakcia hardvérových dátových registrov FPU pomocou dátovej štruktúry zásobník.

do operandu inštruckie a inkrementuje `TOP`. Hodnota `TOP` je reprezentovaná 3-bitovou hodnotou v riadiacom registri a pri pretečení alebo podtečení vždy ukazuje na validný register. [4]

Mapovanie FP inštrukcií do LLVM IR

Relatívne indexovanie dátových registrov jednotky FPU vedie k problémom pri mapovaní FP inštrukcií na LLVM IR. Jednoduché mapovanie pomocou šablón pre jednotlivé inštrukcie nie je možné. Mapovanie vyžaduje pokročilejšiu analýzu. Uvažujme Príklad 2, na ktorom je časť možného assemblerovského kódu, ktorý manipuluje s FPU zásobníkom.

```
1 FADD ST0, ST1
2 FLD1
3 FADD ST0, ST1
```

Príklad 2. Príklad manipulácie s FPU zásobníkom v jazyku assembler.

Prvá inštrukcia manipuluje s operandami `ST0` a `ST1`, ktoré odkazujú na konkrétne hardvérové registre. Druhá inštrukcia `FLD1` dekrementuje vrchol zásobníka a uloží konštantu na nový vrchol zásobníka. V dôsledku dekrementácie už nebudú označenia `ST0` a `ST1` odkazovať na rovnaké hardvérové registre ako v prvej inštrukcii. Tretia inštrukcia je identická s prvou, ale jej operandy sú v skutočnosti už iné hardvérové registre. Toto chovanie vedie k problému, pretože bez kontextu získaného z hodnoty `TOP` je mapovanie chybné. Z tohto dôvodu nemôže LLVM IR reprezentovať Príklad 2 sekvenciou kódu zobrazenou v Príklade 3 (uvažujem, že `@st0` a `@st1` sú globálne premenné reprezentujúce dátové registre `ST0` a `ST1`).

```
1 %op0 = load x86_fp80, x86_fp80* @st0
2 %op1 = load x86_fp80, x86_fp80* @st1
3 %res = fadd x86_fp80 %op0, %op1
4 store x86_fp80 %res, x86_fp80* @st0
```

Príklad 3. Chybná reprezentácia FPU registrov pomocou LLVM IR.

Analýza preto mapuje inštrukcie `load` a `store` na volanie pseudo inštrukcií, ktoré majú ako parameter aktuálnu hodnotu vrcholu zásobníka. Takéto

mapovanie predpokladá, že tieto pseudo funkcie budú v rámci ďalších optimalizácií nahradené inštrukciami zo správnymi operandami. Príklad 4 zobrazuje výsledok mapovania assembler kódu z Príkladu 2. Premenná `@fpu_stat_TOP` je globálna premenná, ktorá obsahuje aktuálny stav vrcholu zásobníka.

```
1 %0 = load i3, i3* @fpu_stat_TOP
2 %1 = sub i3 %0, 1
3 %op0 = call x86_fp80 @_loadFP(i3 %0)
4 %op1 = call x86_fp80 @_loadFP(i3 %1)
5 %res = fadd x86_fp80 %op0, %op1
6 call void @_storeFP(i3 %0, x86_fp80 %res)
```

Príklad 4. Správna reprezentácia FPU registrov pomocou LLVM IR.

4. Navrhované rozšírenie

Táto sekcia popisuje navrhovanú optimalizáciu, ktorá mapuje pseudo funkcie pre prístup k FPU zásobníku na inštrukcie s korektnými operandami.

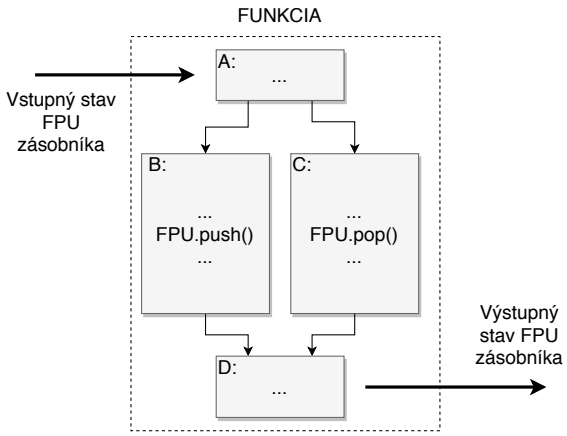
Rekonštrukcia vrcholu zásobníka prebieha vždy v rámci jednej funkcie. Na začiatku funkcie je zásobník vždy prázdny. Tento predpoklad vychádza z analýzy štandardov volania funkcií pre jednotlivé konvencie, architektúry a operačné systémy (viď [5]). Naprieč všetkými štandardami platí, že pri volaní funkcie je vždy zásobník prázdny, a to aj v prípade, že má funkcia parametre dátového typu FP. Takéto parametre sa predávajú ako ukazatele do pamäte, kde je FP hodnota uložená. Na konci volania funkcie je zásobník buď prázdny, alebo obsahuje jednu hodnotu, a to v prípade, že ide o funkciu s návratovou hodnotou typu FP. Avšak návratová hodnota typu FP sa ukladá na zásobník len v prípade, že ide o 32-bitovú architektúru. V prípade 16-bitovej architektúry sa návratová hodnota predáva ako referencia do pamäte a v prípade 64-bitovej architektúry sa návratová hodnota predáva cez registre jednotky SSE. Tabuľka 1 sumarizuje stav FPU zásobníka pre jednotlivé konvencie volania funkcií architektúry x86.

So znalosťou hodnoty vrcholu zásobníku na začiatku funkcie je možné sekvenčne prejsť celú funkciu a sledovať zmeny na zásobníku. Aktuálne volanie pseudo funkcií `load` a `store` je možné nahradiť za inštrukcie s konkrétnymi registrami. Avšak takýto postup ignoruje závislosť na CFG medzi jednotlivými základnými blokmi v rámci funkcie. Obrázok 3 reprezentuje závislosť vrcholu zásobníku na CFG. V bloku *B* sa dekrementuje zásobník a naopak v bloku *C* je zásobník inkrementovaný. Avšak tieto dva bloky tvoria alternatívne vetvy CFG, a preto bude vždy vykonaná len jedna z nich. Analýza neprihliadajúca

Architektúra	Konvencia volania	Register
16 bit	cdecl pascal fastcall	AX
	watcom	Nejednoznačné.
32 bit	cdecl stdcall pascal fastcall thiscall	ST(0)
	watcom	Nejednoznačné.
64 bit	Windows	SSE registre
	Linux, BSD, Mac OS	

Tabuľka 1. Konvencie volania funkcií pre rodinu architektúr x86 a sumarizácia využitia FPU registrov pre predávanie návratovej hodnoty s dátovým typom FP.

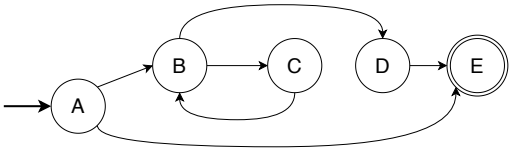
na tento fakt by teda nemohla vyhodnocovať vrchol zásobníka správne.



Obrázok 3. Ilustrácia nesprávnej manipulácie s FPU zásobníkom.

Analýza CFG funkcie

Navrhované riešenie analýzy CFG využíva základné pojmy a vedomosti numerickej lineárnej algebry a maticového počtu (bližšie informácie viď [6]). Navrhovaný algoritmus bude ilustrovaný na príklade CFG z Obrázka 4.



Obrázok 4. Príklad CFG pre funkciu.

Rozšírená analýza sa pokúsi získať stav vrcholu zásobníka na začiatku každého základného bloku. S touto znalosťou je ďalej možné prejsť jednotlivo každý základný blok a nahradiť pseudo funkcie za inštrukcie s konkrétnymi registrami. Na získanie hodnoty vrcholu zásobníka na začiatku každého

základného bloku sa podľa CFG zostaví sústava Rovníc (1), kde každý blok definuje dve neznáme: hodnota na začiatku (napríklad A_{in}) a konci bloku (paríklad A_{out}).

Každý základný blok je samostatne zanalyzovaný. Nezávisle na stave zásobníka pri vstupe do bloku je možné vyhodnotiť relatívny rozdiel vstupného a výstupného stavu vrcholu zásobníka. Napríklad pre blok A je táto hodnota označená ako A_Δ (obdobná konvencia označenia je použitá aj pre ostatné bloky). Výsledkom tejto analýzy je sústava Rovníc (2).

$$\begin{aligned} B_{in} - A_{out} &= 0 \\ C_{in} - B_{out} &= 0 \\ B_{in} - C_{out} &= 0 \\ D_{in} - B_{out} &= 0 \\ E_{in} - D_{out} &= 0 \\ E_{in} - A_{out} &= 0 \end{aligned} \quad (1) \qquad \begin{aligned} A_{out} - A_{in} &= A_\Delta \\ B_{out} - B_{in} &= B_\Delta \\ C_{out} - C_{in} &= C_\Delta \\ D_{out} - D_{in} &= D_\Delta \\ E_{out} - E_{in} &= E_\Delta \end{aligned} \quad (2)$$

Algoritmus získa Rovnice (3) na základe znalosti konvencie volania funkcie, ktorú dokáže spätný prekladač detekovať. V našom vzorovom príklade je A_{in} hodnota zásobníka na začiatku funkcie a C_{out} hodnota na konci funkcie. Ukončovacích blokov funkcie môže byť viacero, ale všetky musia mať rovnakú hodnotu. Zásobník je buď prázdny alebo obsahuje práve jednu hodnotu (v prípade, že návratová hodnota funkcie je typu FP).

$$A_{in} = 0 \qquad C_{out} = 0 \quad (3)$$

CFG môže obsahovať veľmi odlišné množstvo hrán. Výsledný systém bude preto mať typicky viac rovníc ako neznámych. Takýto systém sústavy lineárnych rovníc sa nazýva *preurčený* (anglicky *overdetermined*). Preurčený systém nemá vo väčšine prípadov riešenie. Avšak, v tomto prípade je veľká pravdepodobnosť, že existuje práve jedno riešenie. Celý spätný prekladač predpokladá, že analyzovaný binárny súbor je vytvorený štandardným prekladačom.

Takto získanú sústava lineárnych rovníc je možné reprezentovať pomocou matíc ako Rovnicu (4), kde $\mathbf{A}$ je matica koeficientov systému a $\bar{\mathbf{x}}$ je vektor neznámych.

$$\mathbf{A}\,\bar{\mathbf{x}} = \bar{\mathbf{b}} \quad (4)$$

Matica $(\mathbf{A}|\bar{\mathbf{b}})$ sa nazýva *rozšírená matica sústavy*. Analýza vyhodnotí *hodnosť* matice $\mathbf{A}$ a hodnosť rozšírenej matice $(\mathbf{A}|\bar{\mathbf{b}})$. Systém má práve jedno riešenie, ak sú tieto hodnosti rovné. Na riešenie preurčených lineárnych systémov sa najčastejšie používa aproximačná *metóda najmenších štvorcov* (anglicky *least squares*). Aproximácia vnáša do riešenia chybu avšak naša aplikácia nevyžaduje veľkú presnosť. Výsledky systému predstavujú indexy na registre a teda budú zaokrúhlené na celé čísla. Riešenie sústavy metódou najmenších štvorcov umožňujú rôzne numerické metódy. Tieto metódy *dekomponujú* maticu $\mathbf{A}$ na viacero matíc koeficientov, ktoré sa dajú riešiť efektívnejšie. Nepresnosť vnesená strojovým zaokruhľovaním sa znižuje lebo koeficienty sú substituované za skutočné hodnoty až spätne. Táto práca zhodnotila z hľadiska presnosti a efektívnosti nasledujúce tri metódy (zdroje porovnania [7, 8]):

- **Cholesky** dekompozícia je všeobecne najrýchlejšia ale aj najmenej presná. Malé odchýlky na vstupe vnášajú veľkú nepresnosť do výsledku.
- **QR** dekompozícia je numericky stabilná a teda aj presnejšia ale je náročnejšia na výpočet.
- **SVD** dekompozícia poskytuje všeobecne najpresnejšie riešenie a to aj pre väčšie nepresnosti na vstupe. Na druhej strane je značne náročnejšia na výpočet (približne 10 násobne).

Vypočítaný vektor $\bar{\mathbf{x}}$ obsahuje hodnoty FPU zásobníka na začiatku a konci každého bloku v rámci aktuálne analyzovanej funkcie. S touto vedomosťou optimalizácie prejde zvlášť každý základný blok, pričom už vie hodnotu zásobníku na jeho začiatku. Optimalizácia sleduje (teraz už skutočnú) hodnotu zásobníka sekvenčným prechodom cez blok a nahradí volania pseudo funkcií FPU registrami na ktoré aktuálne odkazuje vrchol zásobníka.

Navrhované rozšírenie bolo implementované v C++ a pre prácu s maticami bola použitá knižnica Eigen[3].

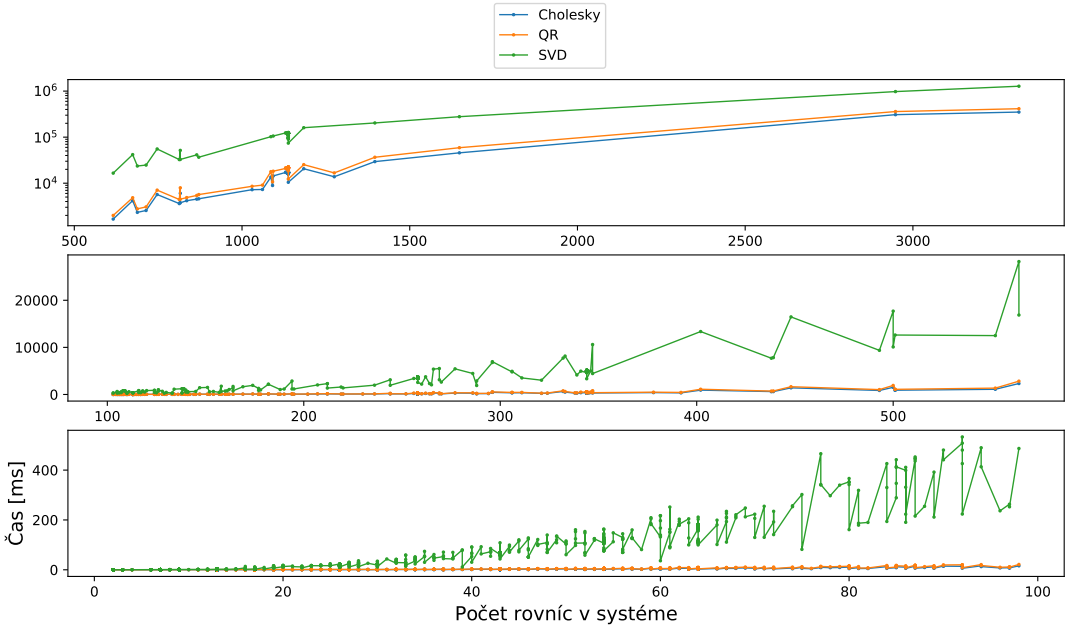
5. Výsledky experimentovania

Naimplementovaná optimalizácia bola otestovaná sadou integračných testov. Testovacie prostredie tvoril nástroj[4] vytvorený projektom RetDec. Tento nástroj umožňuje vyhodnocovať návratovú hodnotu spätného prekladu, analýzu vygenerovaného výstupu (mená funkcií, typov, zhoda na reťazce a podobne) a opätovný preklad a spustenie zrekonštruovaného programu (porovnávanie štandardného výstupu pôvodného a zrekonštruovaného programu).

Testovaciu sadu tvorilo 822 spustiteľných binárnych súborov, ktoré boli vytvorené rôznymi prekladačmi pre architektúru x86 (z toho 86%

[3] Free Software projekt Eigen: http://eigen.tuxfamily.org

[4] RetDec testovací nástroj: https://retdec-regression-tests-framework.readthedocs.io/en/latest/



Obrázok 5. Porovnanie výpočetnej náročnosti jednotlivých metód dekompozície lineárneho systému vzhľadom na jeho veľkosť.

Počet rovníc	Cholesky	QR	SVD
< 100	1.29 ms	1.88 ms	43.89 ms
$\langle 100\ ;\ 600 \rangle$	264.34 ms	345.11 ms	3 948.95 ms
$600 >$	18 741.14 ms	23 332.06 ms	125 831.64 ms

Tabuľka 2. Priemerný čas vyriešenia systému pomocou rôznych dekompozičných metód.

binárnych súborov je určených pre 64-bitovú architektúru, 12% pre 32-bitovú architektúru a zvyšok pre 16-bitovú architektúru). Tieto spustiteľné súbory sú súčasťou databázy testovacieho nástroja projektu RetDec.

Optimalizačné experimentovanie

V Sekcii 4 boli navrhnuté tri metódy riešenia preurčeného systému (Cholesky, QR, SVD). Algebraická knižnica Eigen podporuje všetky tri. Každá s týchto metód bola experimentálne naimplementovaná v novom rozšírení. V rámci troch experimentov boli jednotlivé metódy aplikované na testovaciu sadu a proces dekompozície bol meraný.

Testovacia sada obsahovala viac ako 4 000 funkcií, ktoré manipulujú s FPU. Výsledky merania pre jednotlivé metódy zobrazuje Obrázok 5. Cholesky a QR dekompozícia trvali zanedbateľne podobne. Na druhej strane, SVD dekompozícia trvala (a to najmä pre veľké systémy) neprijateľne dlho oproti ostatným metódam. Tabuľka 2 zobrazuje priemerné doby výpočtu pre jednotlivé metódy. Pre lepšiu analýzu sú výsledky merania rozdelené do troch skupín (podľa množstva rovníc v systéme).

Vyhodnotenie úspešnosti optimalizácie

Druhá časť experimentovania zhodnocuje úspešnosť optimalizácie. V rámci danej testovacej sady bolo celkovo analyzovaných 4 158 funkcií, ktoré pracovali s inštrukčnou sadou FPU. Išlo o funkcie vytvorené reálnymi prekladačmi. Preto sa očakávalo, že všetky systémy lineárnych rovníc z nich vytvorené budú mať riešenie (matica sústavy a rozšírená matica sústavy majú rovnakú hodnosť). Optimalizácia našla riešenie sústavy rovníc pre všetky testované funkcie. Pre tieto funkcie nahradila pseudo funkcie za konkrétne FPU registre zo 100 % úspešnosťou. Úspešnosť nahradenia vyhodnocoval testovací nástroj projektu RetDec. Tento nástroj detekuje deklaráciu volaných funkcií a v prípade neúspechu optimalizácie by teda detekoval nesubstituované pseudo funkcie.

6. Záver

Na základe výsledkov experimentovania vidím možnú potrebu zamerať sa na zefektívnenie optimalizácie pre funkcie s veľmi veľkým množstvom základných blokov. Pre funkcie, ktoré tvoria lineárne systémy s viac ako 1000 rovnicami je optimalizácia nezanedbateľne zaťažujúca na celú dekompiláciu. Pre moju testovaciu sadu u takto veľkých systémov trvala optimalizácia v priemere viac ako 23 sekúnd. Údaj je samozrejme len relatívny vzhľadom na konkrétne CPU, na ktorom experiment bežal.

Ďalšie potenciálne možnosti rozšírenia spätného prekladača RetDec vidím v oblasti jednotky SSE, ktorá v 64-bitových architektúrach nahradila jednotku FPU.

V rámci tejto jednotky by som chcel preštudovať a zistiť možné optimalizácie pri spätnom preklade tejto pokročilej inštrukčnej sady.

Poďakovanie

Chcel by som poďakovať vedúcemu mojej práce Zbyňkovi Křivkovi, a taktiež celému Avast tímu, ktorý mi poskytol profesionálnu pomoc. Menovite by som chcel poďakovať Petrovi Matulovi, Jakubovi Křoustkovi, a ostatným za ich konzultácie aa odnotné komentáre.

Literatúra

[1] Eldad Eilam. *Reversing secrets of reverse engineering*. Wiley, Indianapolis, 2005.

[2] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[3] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[4] Kip R Irvine. *Assembly language for x86 processors*. Prentice Hall, Upper Saddle River, 6th ed. edition, 2010.

[5] Agner Fog. *Calling conventions for different C++ compilers and operating systems*. Technical University of Denmark, 2019.

[6] H. Anton. *Elementary Linear Algebra: Applications Version*. Wiley Plus Products. Wiley, 2010.

[7] Pavel Cizek and Lenka Cizkova. Numerical linear algebra. 01 2004.

[8] Do Q Lee. *Numerically efficient methods for solving least squares problems*. The University of Chicago, 2012.