

MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY



Dekompilačné nástroje založené na LLVM

BAKALÁRSKA PRÁCA

Vincent Mihalkovič

Brno, jar 2019

Vyhlásenie

Vyhlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Vincent Mihalkovič

Vedúci práce: RNDr. Petr Ročkai, Ph.D.

Podakovanie

Ďakujem Petrovi Ročkaiovi za profesionálny prístup, zrozumiteľné a jasné vytyčovanie čiastkových cieľov. Lukášovi Koreňikovi ďakujem za ochotu a pomoc pri odstraňovaní prekážok, ktoré stáli v ceste a tatovi za rady pri spisovaní práce.

Zhrnutie

Porovnanie dvoch voľne dostupných dekompilečných nástrojov, založených na LLVM - RetDec a McSema prebehlo prostredníctvom súboru rôznorodých prevažne monofunkčných programov. Hlavným kritériom testovania bola možnosť výstup dekompilátoru preložiť a získať tak sémanticky ekvivalentný program voči pôvodnému. Vzhľadom na toto zvolené kritérium je optimálnym nástrojom McSema.

Klíčové slová

RetDec, McSema, LLVM, dekompilácia, analýza binárnych súborov,
reverzné inžinierstvo, jazyk C, porovnanie

Obsah

Úvod	2
1 Základné pojmy	3
1.1 Procesor a inštrukcie	3
1.2 Preklad (kompilácia)	3
1.3 LLVM a LLVM IR	6
1.4 Dekompilácia	12
1.4.1 Význam	12
1.4.2 Problémy	13
1.4.3 Proces dekompilácie	13
2 RetDec	15
2.1 Základná charakteristika	15
2.2 Proces dekompilácie	16
2.2.1 Fáza predspracovania	17
2.2.2 Vnútoraná fáza	18
2.2.3 Výstupná fáza	19
3 McSema	21
3.1 Základná charakteristika	21
3.2 Proces dekompilácie	22
3.2.1 mcsema-disass	22
3.2.2 mcsema-lift	24
4 Testovacia sada	27
5 Zhrnutie	30
6 Záver	32
Bibliografia	33
A Elektronické prílohy	35

Zoznam tabuliek

1.1	Typy v LLVM IR	7
4.1	Výsledky testovania	28

Zoznam obrázkov

1.1	prekladač	4
1.2	AST výrazu: $a = b < c ? *p + b * c : 1 \ll d()$. obrázok	
	prebraný z [4]	5
1.3	Ilustrácia jazykovo nezávislého prekladača prebratá z	
	[5]	6
1.4	trojadresný kód	8
1.5	Faktoriál v jazyku C	8
1.6	Ilustrácia základných blokov kódu z obrázka 1.5	11
2.1	predspracovanie, prebrané z [9]	17
2.2	prevod binárnych reťazcov do LLVM IR. Obrázok	
	prebraný z [9]	18
2.3	Obrázok prebraný z [9]. Riadiace štruktúry (zľava): if,	
	if-else, while/for-loop, break, continue, switch.	19
3.1	CFG súbor, prebraný z [11]	26

Úvod

Proces kompilácie pozostáva z viacerých fáz. Pri každej z nich sa nenávratne stráca logická a abstraktná štruktúra pôvodného zdrojového kódu (napríklad propagáciou funkcií a konštánt, rozvinutím makier, odstránením vyšších riadiacich konštrukcií a podobne) pričom je zachovaná funkčnosť vyjadrená zdrojovým kódom. Po poslednom štádiu kompilácie je množstvo stratenej informácie také veľké, že inverzný proces - rekonštrukcia pôvodného zdrojového kódu je v striktnom zmysle nemožná. Vzhľadom na to, že strojový kód je zbavený logickej a abstraktnej štruktúry, neexistuje deterministický, automatický postup rekonštrukcie zdrojového kódu. Proces *dekompilácie*^[1] preto musí zahŕňať strategické a tvorivé postupy, ktoré sú špecifickou charakteristikou daného dekompilečného nástroja.

Dekompilácia je forma reverzného inžinierstva, proces transformácie binárneho (spustiteľného) súboru do vyššej formy reprezentácie. Vo vyššej forme reprezentácie by mal byť kód nezávislý na platforme a v čitateľnejšej forme než napríklad program v assembleri. Účelom môže byť napríklad odhalenie bezpečnostných rizík, optimalizácia kódu, verifikácia programu, alebo zmena cieľovej architektúry.

Cieľom mojej práce je porovnanie úspešnosti a efektívnosti dvoch existujúcich dekompilečných nástrojov na báze LLVM. Jedná sa o voľne dostupný nástroj vyvíjaný firmou AVAST - RetDec a nástroj McSema, ktorého jedinou komponentu s komerčnou licenciou úspešne nahradil študent fakulty informatiky *voľne dostupným* nástrojom Dyninst [1]. Určujúcim hľadiskom pri porovnávaní týchto nástrojov bola možnosť rekompilácie - výstup z dekompilečného nástroja má mať rovnakú funkcionálnosť ako pôvodný program a malo by byť možné ho znova preložiť. Toto je zároveň základná požiadavka ak chceme nejaký z dekompilečného nástroja použiť v nástroji DIVINE [2] - *software* vyvíjaný na FI vo výskumnom laboratóriu ParaDiSe, určený na formálnu verifikáciu.

Na porovnanie týchto nástrojov som zoskupil a vytvoril množstvo rôznorodých programov predovšetkým napísaných v jazyku C. Jedná sa napríklad o programy, ktoré pracujú s poľami, súbormi, globálnymi premennými, ukazovateľov aritmetikou a podobne. Postupujem

1. Spisovne spätný preklad (anglicky *decompilation*)

nasledovne: zvolený program je preložený (získam binárny súbor), následne dekompilovaný, výstup dekompilátora opäťovne preložený a tieto dva spustiteľné súbory porovnávam z hľadiska funkčnosti a správania. Ak sa chovanie dekompilovaného programu líši od správania pôvodného kódu, treba analyzovať zdroj rozdielného alebo chybného správania a pokiaľ je to možné navrhnúť metódu automatickej opravy chýb rovnakého druhu.

V nasledujúcej kapitole vysvetľujem základné pojmy, proces kompilácie, dekompilácie, projekt LLVM a jazyk LLVM IR. V druhej a tretej kapitole charakterizujem obidva dekompilačné nástroje. Štvrtá kapitola uvádza niekoľko programov použitých pri testovaní, dôvod ich použitia, akú konkrétnu schopnosť dekompilátora testujú. Posledná kapitola zdôvodňuje prečo je nástroj McSema vhodnejším kandidátom pre použitie v súčinnosti s nástrojom DIVINE.

1 Základné pojmy

Cieľom tejto kapitoly je vysvetliť pojmy, ktoré súvisia s dekompiláciou a samotný proces dekompilácie.

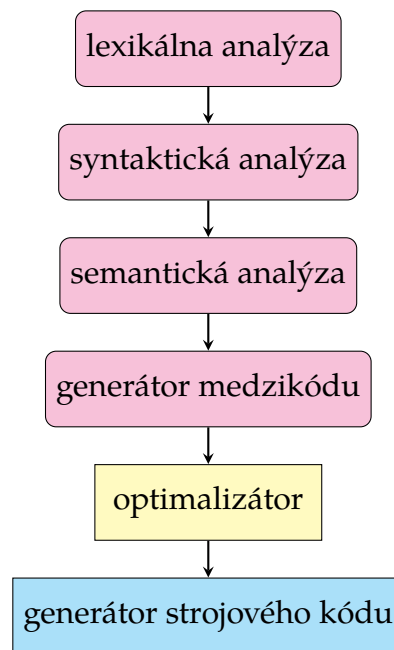
1.1 Procesor a inštrukcie

- **Procesor** (anglicky *CPU*) je hlavná výpočtová jednotka v počítači. Jeho účelom je vykonávanie inštrukcií načítaných z pamäte. Každý procesor disponuje vlastnou sadou (elementárnych) inštrukcií. To znamená, že program preložený pre procesor *x86* nebude fungovať na procesore *PowerPC*. Procesory delíme podľa architektúry do dvoch základných skupín:
 1. *Harvardská architektúra* procesory z tejto skupiny majú uložené inštrukcie a dáta na dvoch oddelených miestach. Ich výhodou je simultánny prístup ku inštrukciám a dátam. Nevýhoda je komplexnosť. Používa sa napríklad v mikrokontroleroch.
 2. Skupina procesorov s *architektúrou von Neumann* ukladá inštrukcie a dáta do rovnakého pamäťového priestoru. Výhodou je jednoduchosť a nevýhodou sekvenčný prístup. Používa sa vo väčšine počítačov.
- **Inštrukčná sada** je zoznam všetkých inštrukcií, ktoré dokáže konkrétny typ procesora rozpoznať a vykonať.
- **Strojový kód** je súbor pozostávajúci z inštrukcií a dát v binárnej forme. Tento súbor je možné na cieľovej architektúre (napríklad *x86*) spustiť.

1.2 Preklad (kompilácia)

Zdrojový kód je program napísaný v bežnom programovacom jazyku (napríklad C++), má logickú a abstraktnú štruktúru. Ak stačí program bez úprav zdrojového kódu preložiť a spustiť (a program funguje ako

má), hovoríme o zdrojovom kóde nezávislom na platforme¹. Závislosť súvisí napríklad s použitím nejakej knižnice, ktorá je závislá na konkrétnej platforme. Aby sme mohli zdrojový kód spustiť, je nutné ho najprv preložiť. Túto činnosť vykonáva prekladač (anglicky *compiler*) a prekladá zdrojový kód do **strojového** kódu. Strojový kód obsahuje elementárne inštrukcie konkrétneho procesora a preto je spustiteľný súbor (strojový kód) závislí na cieľovej architektúre. Proces prekladu



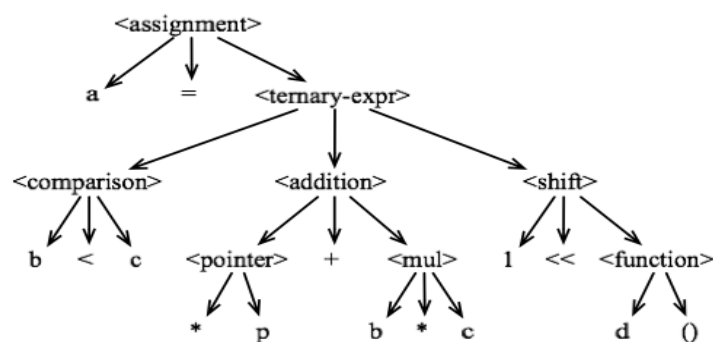
Obr. 1.1: prekladač

pozostáva z viacerých častí (ilustrujem obrázkom 1.1), ktoré sú podrobne rozpísané tu [3], vo všeobecnosti ich môžeme rozdeliť na analýzu (lexikálnu, syntaktickú, sémantickú) a syntézu (generovanie strojového kódu).

1. Lexikálna analýza - výstupom sú znaky (anglicky *tokens*) alebo lexémy. V tejto časti sú odstránené biele znaky (napríklad medzery) a komentáre.

1. Pojem platforma nereprezentuje len procesor, ale aj operačný systém.

2. Syntaktická analýza (anglicky *parser*) kontroluje gramatickú správnosť vstupnej postupnosti a neberie ohľad na význam znakov. Pri syntaktickej kontrole sa používa bezkontextová gramatika a vzniká AST (anglicky *abstract syntax tree*) - abstraktný syntaktický strom.
3. Sémantická analýza zahŕňa kontrolu typov (anglicky *type checking*). Prekladač navštevuje postupne jednotlivé uzly v AST strome a rozhoduje či je s nimi súvisiaca operácia sémanticky správna. Ilustrujem na obrázku 1.2.



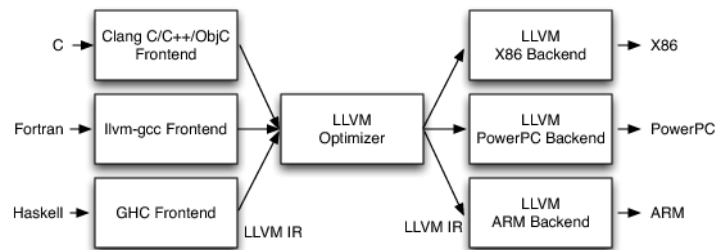
Obr. 1.2: AST výrazu: $a = b < c ? *p + b * c : 1 \ll d()$. obrázok prebraný z [4]

4. Generátor medzikódu² (anglicky *intermediate code*), v tejto časti prevedie kompilátor kód do reprezentácie, ktorá ešte nie je závislá na cieľovej architektúre, ale už neobsahuje konštrukcie vysoko-úrovňového jazyka.
5. Pri optimalizácii³ sa kompilátor snaží medzikód vylepšiť tak aby bol výsledný strojový kód napríklad rýchlejší, neobsahoval nadbytočné inštrukcie alebo bol menej energeticky náročný. Pri optimalizácii sa nemení sémantika kódu.

2. Kompilátory používajú tento kód z dôvodu efektivity (optimalizačné nástroje pracujú s medzi-kódom) a znova-použitelnosti (druhý dôvod bude objasnený v nasledujúcej sekcii)

3. Existujú dva typy optimalizácii: na platforme závislé (snažia sa využiť vlastnosti danej architektúry) a nezávislé

6. Generovanie strojového kódu je posledná časť, v ktorej sú premenným priradené pamäťové adresy, a kód je z internej reprezentácie prevedený do sekvencie inštrukcií daného procesora.



Obr. 1.3: Ilustrácia jazykovo nezávislého prekladača prebratá z [5]

1.3 LLVM a LLVM IR

- LLVM [6] je projekt zastrešujúci komplexnú sadu nástrojov a knižníc, ktoré slúžia predovšetkým na implementáciu prekladačov. Obrázok 1.3 znázorňuje modularitu nasledujúcich troch zložiek, ktorá je dosiahnutá prekladom do vnútornej reprezentácie - medzikódu, jazyka LLVM IR.

1. analýza a preklad zdrojového kódu do LLVM IR,
2. analýzy a optimalizácie,
3. preklad optimalizovaného LLVM IR kódu do jazyka cieľovej architektúry.

Medzi najznámejšie projekty pod záštitou LLVM patrí *Clang* - prekladač pre jazyky C a C++. Čoskoro začali vznikať prekladače pre ďalšie jazyky: Ada, Fortran, Haskell a iné. Vďaka modularnosti prekladaču bolo možné využiť už existujúce (vyššie spomenuté) komponenty a stačilo implementovať časť, ktorá preloží zdrojový kód do jazyka LLVM IR.

Okrem vývoja modulárnych prekladačov LLVM zastrešuje aj mnoho iných projektov ako napríklad ladiaci nástroj *LLDB*, ktorý

	Typy
základné	void, boolean, label, token, metadata, ...
celočíselné	i1, i2, ..., i8, ..., iN
desatinné	half, float, double, x86 fp80, fp128, ppc fp128
zložené	array, vector, structure, function, pointer..

Tabuľka 1.1: Typy v LLVM IR

je rýchlejší a menej pamäťovo náročný ako konkurenčný GDB, alebo optimalizované knižnice *libc++* pre jazyk C++.

- **LLVM IR** (skratka IR znamená *Intermediate Representation*) LLVM IR je nízkoúrovňový programovací jazyk - na platforme a jazyku nezávislá inštrukčná sada (podobná sade procesorov RISC), ktorú používajú nástroje projektu LLVM. LLVM IR je dostatočne nízkoúrovňová z dôvodu možnosti reprezentovať čo najväčšie množstvo programovacích jazykov. Táto forma je vhodná nielen pre veľké množstvo užitočných optimalizácií, ktoré by inak vyžadovali dodatočné (komplikované) analýzy, ale aj rýchle (mechanické) generovanie strojového kódu. Nižšie uvádzam základné vlastnosti LLVM IR:
 - **SSA** forma (*Single Static Assignment*), znamená, že do každej premennej je priradená hodnota práve raz (pri jej vzniku),
 - inštrukcie sú v troj-adresnej forme znázorenej na obrázku 1.4,
 - miesto konečného počtu registrov, LLVM IR poskytuje neobmedzený počet typovaných globálnych (označených prefixom „@“) a lokálnych (označených prefixom „%“) premenných,
 - LLVM IR je typovaný jazyk, kde každá premenná má svoj typ a inštrukcie kontrolujú korektnosť typov pri operandoch. Základné dátové typy sú uvedené tu 1.1.

výsledek := operátor operand1 operand2

Obr. 1.4: trojadresný kód

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

Obr. 1.5: Faktoriál v jazyku C

```
define i32 @factorial(i32) #0 {
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 %0, i32* %3, align 4
    %4 = load i32, i32* %3, align 4
    %5 = icmp eq i32 %4, 0
    br i1 %5, label %IfEqual, label %IfNotEqual

IfEqual:
    store i32 1, i32* %2, align 4
    br label %Return

IfNotEqual:
    %6 = load i32, i32* %3, align 4
    %7 = load i32, i32* %3, align 4
    %8 = sub nsw i32 %7, 1
    %9 = call i32 @factorial(i32 %8)
    %10 = mul nsw i32 %6, %9
    store i32 %10, i32* %2, align 4
    br label %Return

Return:
    %11 = load i32, i32* %2, align 4
    ret i32 %11
}
```

Ukážku LLVM IR kódu znázorňenú na 1.5, získanú prekladom⁴ programu v jazyku C som mierne upravil (premenovanie návěstí): Kód je členený do funkcií, ktoré sa skladajú zo základných blokov (ilustrujem obrázkom 1.6), a tie obsahujú sekvencie inštrukcií. Základný blok (anglicky *basic block*) je postupnosť inštrukcií, ktoré sú vždy vykonané v tomto poradí. Začiatok každého základného bloku je buď

- prvá inštrukcia programu,
- nasledujúca inštrukcia po podmienenom, nepodmienenom skoku, volaní alebo návrate z procedúry (po inštrukciách *br*, *ret*, *call*, *<podmienka>*),
- nasledujúca inštrukcia po návesti.

Na prvom riadku kľúčovým slovom *define* začína definícia funkcie *faktorial*, *i32* je návratový typ funkcie - 32 bitové celé číslo, názov funkcie, a v zátvorkách vstupné argumenty funkcie. Nasleduje:

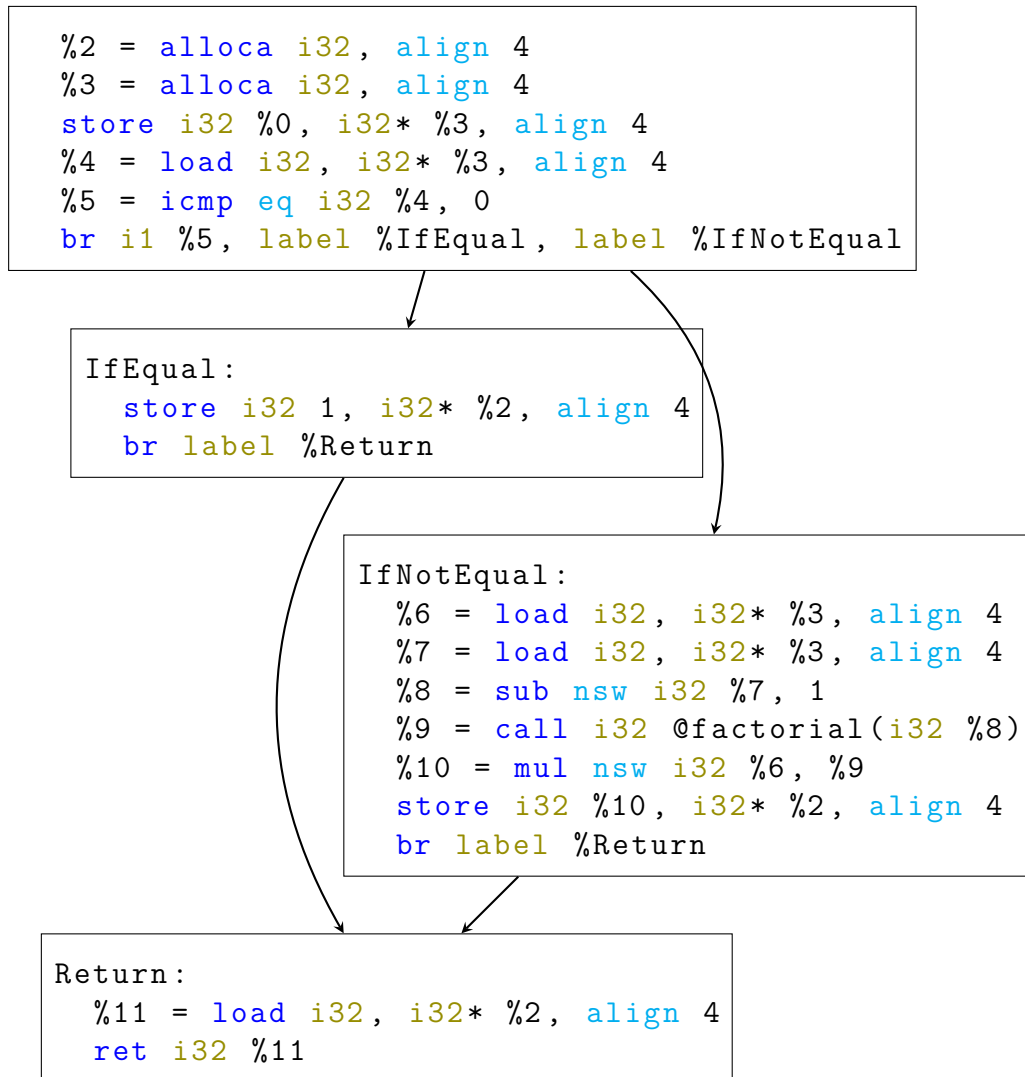
- Na prvých dvoch riadkoch alokuje inštrukcia *alloca* v pamäti priestor pre 32 bitové celé číslo. Ukazovatele na priradené neinicializované kusy pamäte sú uložené v premenných označených *%2* a *%3*.
- Nasleduje inštrukcia *store*, ktorá uloží hodnotu premennej *%0* na adresu ukazovateľa v premennej *%3* (Túto premennú si môžeme predstaviť ako ukazovateľ na premennú *n* z pôvodného programu v jazyku C).
- Inštrukcia *load* nahrá z adresy uloženej v ukazovateli *%3* do novej premennej *%4* hodnotu typu *i32*.
- Inštrukcia *icmp* uloží výsledok porovnania 32 bitového celého čísla v premennej *%4* s konštantou 0, do novej premennej *%5*. Spôsob porovnania určuje kľúčové slovo. *eq*⁵ (anglicky *equal*).

4. príkazom `$ clang -S -emit-llvm fact.c`

5. okrem *eq* sú k dispozícii aj iné možnosti porovnania napríklad *ugt* - *unsigned greater than*, *sge*: *signed greater or equal*.

- Inštrukcia *br* (skratka znamená anglicky *branch*) je poslednou v tomto základnom bloku. Ak bola podmienka (hodnota v premennej %4 rovná 0) splnená, presúvame sa na návesť označenú *IfEqual*, ak nie, presúvame sa na návesť označenú *IfNotEqual*.
- Inštrukcia *store* uloží do pamäte na adresu uloženú v ukazovateli %2 32 bitovú celočíselnú konštantu 1.
- Poslednou inštrukciou tohto základného blok je nepodmiernený variant inštrukcie *br*, ktorým sa presúvame na návesť označenú *Return*.
- Nasledujúci základný blok označený návesťou *IfNotEqual*, začína dvoma inštrukciami *load*, ktoré vytvoria dve nové premenné a nahrávajú do nich hodnotu z pamäte uloženú v ukazovateli %3 (premenná %3 je ukazovateľom na vstupný argument).
- Inštrukcia *sub nsw* (skratka znamená anglicky *No Signed Wrap*) uloží do novej premennej označenej %8 výsledok odčítania konštanty jedna od hodnoty v premennej %7.
- Inštrukcia *call* reprezentuje volanie funkcie *factorial*, ako argument je predaná hodnota premennej %8. Do novej premennej %9 je uložené 32 bitové celé číslo, ktoré funkcia *factorial* vráti.
- Inštrukcia *mul* (anglicky *multiplication*) vynásobí hodnoty oboch operandov, v tomto prípade hodnoty premenných %6 a 9 a výsledok operácie uloží do premennej %10.
- Ešte pred volaním skoku (inštrukcie *br*) nahráme obsah premennej %10 do pamäte na adresu uloženú v premennej %2.
- Posledný základný blok: inštrukcia *store* nahrá do novej premennej %11 hodnotu uloženú v %2, a inštrukcia *ret* (anglicky *return*) túto novú hodnotu vráti.

Okrem spomenutej priamo čitateľnej textovej reprezentácie, existujú ešte dve ekvivalentné formy LLVM IR kódu: binárny kód uložený na disku a reprezentácia v pamäti počítača.



Obr. 1.6: Ilustrácia základných blokov kódu z obrázka 1.5

LLVM IR reprezentácia programu v pamäti počítača je tvorená nasledovnými komponentami [7]:

1. Trieda `Module` obsahuje zoznam funkcií, tabuľku symbolov, zoznam globálnych premenných.
2. Trieda `Function` obsahuje všetky objekty, ktoré súvisia s definíciou alebo deklaráciou funkcie. Napríklad zoznam vstupných argumentov funkcie, zoznam základných blokov.
3. Trieda `BasicBlock` obsahuje zoznam LLVM inštrukcií daného bloku. Rozhranie tejto triedy umožňuje iterovať cez inštrukcie, alebo získať poslednú (terminálnu) inštrukciu.
4. Trieda `Instruction` reprezentuje jednotlivé inštrukcie jazyka LLVM IR, a obsahuje základné informácie napríklad názov inštrukcie (napríklad `Add`), informácie o operandoch, typ inštrukcie (terminálna, binárna, unárna) alebo vlastnosti inštrukcie (idempotentná, komutatívna, asociatívna).

1.4 Dekompilácia

Dekompilátor (spätný prekladač) je nástroj, ktorého cieľ je rekonštrukcia (transformácia) vstupného binárneho súboru do zdrojového kódu (reprezentácie priamo čitateľnej človekom) so zachovaním pôvodného významu (správania) programu.

1.4.1 Význam

Dekompilácia sa uplatňuje v prípadoch, kedy je zdrojový kód nedostupný alebo neexistuje a k dispozícii je jeho binárna reprezentácia. Používa sa napríklad z nasledujúcich dôvodov:

- obnovenie nedostupného zdrojového súboru z jeho binárnej reprezentácie,
- portabilita - dekompiláciou získaný binárny súbor je možné preložiť na inej platforme,

- porozumenie programu (sémanticky). Dôvodom môže byť napríklad analýza (potenciálneho) škodlivého softvéru,
- verifikácia programu - vďaka dekompilácii je možné využiť verifikačné nástroje (ktoré pracujú zo zdrojovými kódmi) na overenie korektnosti/správnosti existujúceho programu (v binárnej reprezentácii),
- optimalizácia programu - pomocou dekompilácie môžeme aplikovať na binárny súbor optimalizačné nástroje, ktoré nemuseli alebo nemohli (lebo neexistovali) byť použité.

1.4.2 Problémy

Podstatou problému dekompilácie je neurčitosť pri interpretácii binárnych dát a nedostatok informácií (tie sú odstránené pri procese prekladu). Problém existencie generického (vstupom by mohol byť ľubovoľný binárny súbor) dekompilátora súvisí s rôznorodosťou procesorov (tie majú odlišné inštrukčné sady, rozdiely pracujú s desatinnými číslami, rozlišujeme napríklad 64/32-bitové procesory a podobne), rôzne ABI, rôzne formáty binárnych súborov (ELF, PE, Mach-O a iné), rôzne programovacie jazyky, rôzne prekladače (optimalizácie zdrojového súboru sťažujú proces dekompilácie), staticky zostavený kód a iné.

1.4.3 Proces dekompilácie

Táto časť vychádza z článku [8]. Štruktúra dekompilátora je podobná kompilátoru. Dekompilácia je proces pozostávajúci z fáz, ktoré postupne menia strojový kód do vyššej reprezentácie. Vstupom je binárny súbor.

1. Analýzou súboru zistíme rad základných informáciách napríklad: cieľovú architektúru (*x86*, *PowerPc* a podobne), vstupný bod (anglicky *entry point*), identifikácia knižničných funkcií (dekompilátor musí mať dané knižnice k dispozícii).
2. Prevod strojového kódu do jazyka symbolických inštrukcií robí *disassembler*, tento prevod je závislý na procesore, type inštrukčnej sady a nie je ani zďaleka jednoznačný - najväčším problémom

je rozhodnutie či sa jedná o adresu alebo hodnotu. Preto existujú rôzne heuristiky a algoritmy ako dosiahnuť čo najväčšiu úspešnosť. Okrem lineárnej metódy (prechádzanie kódu sekvenčne) existuje (efektívnejšia) rekurzívna metóda, ktorá začne v počiatočnom bode (anglicky *entry point*) a rekurzívne prechádza na ďalší dosiahnuteľný kód.

3. Sémantická analýza zbiera informácie o typoch a hľadá skupiny inštrukcií, ktoré reprezentujú idióm (príklad idiómu: delenie alebo násobenie dvojkou je realizované pomocou bitových posunov).
4. Transformácia do inštrukcií vyššieho jazyka využíva analýzu toku dáta a analýzu toku riadenie (anglicky *control flow graph*) (cieľom je napríklad obnova vyšších riadiacich štruktúr).

2 RetDec

Kapitola predstavuje dekompilačný nástroj RetDec, dôvody vzniku, ciele a základné vlastnosti. Stručne uvádzam proces dekompilácie a predstavujem použité komponenty. Obrázky a obsah kapitoly vychádza z prezentácie nástroja RetDec jeho autormi na konferencii RECon 2018, ktorá prebiehala v Montreali v roku 2018 [9].

2.1 Základná charakteristika

RetDec (skratka anglicky znamená *Retargetable Decompiler*) je voľne šíriteľný dekompilátor založený na LLVM. RetDec je vyvíjaný od roku 2011 s podporou firmy AVAST (predtým AVG), na jeho vývoji spolupracovalo niekoľko študentov fakulty informačných technológií VUT v Brne (autor RetDecu bol študentom VUT). Hlavnou črtou tohto dekompilátora je univerzálnosť - nie je viazaný na dekompiláciu binárnych súborov konkrétnej architektúry, operačného systému alebo formátu. Motivácia pre vznik, bola predovšetkým analýza škodlivého softvéru (anglicky *malware*) a bezpečnostných hrozieb. Použitie tohto dekompilátoru uľahčuje používateľovi analýzu binárnych súborov (nemusí čítať jazyk symbolických inštrukcií, ale napríklad kód v jazyku C).

V súčasnosti je RetDec veľký projekt (500 000 riadkov kódu), rozdelený medzi samostatné komponenty. Vďaka tomuto modulárnemu návrhu je možné používať jednotlivé komponenty¹ nezávisle na zvyšku.

V súčasnosti podporuje formáty binárnych súborov:

- ELF, PE, Mach-O, COFF, AR (archive), Intel HEX,
- nespracované (anglicky *raw*) binárne súbory.

Podporované architektúry:

- 32-bit: Intel x86, ARM, MIPS, PIC32, a PowerPC,
- 64-bit: x86-64.

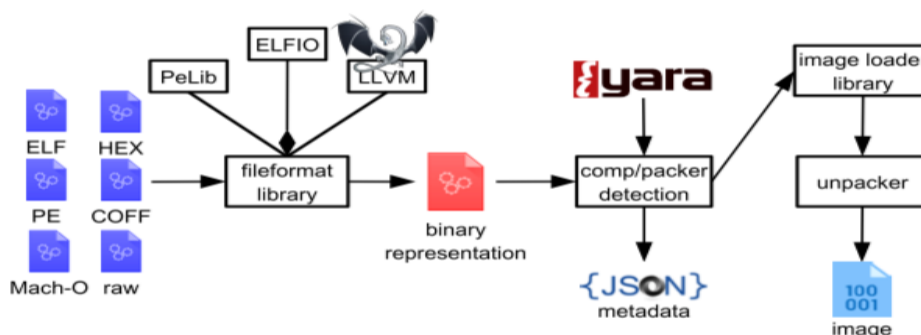
1. zoznam komponent: <https://github.com/avast/retdec/tree/master/src>

Základné vlastnosti (funkcionalita) RetDecu:

- statická analýza binárnych súborov,
- zistenie použitého prekladača a taktiež (ak bol použitý) nástroju určeného na kompresiu dát - *packer*,
- dekódovanie inštrukcií,
- rekonštrukcia funkcií, typov a vysoko-úrovňových konštrukcií,
- rozpoznanie staticky zostaveného kódu (napríklad z knižníc)
- využitie ladiacich informácií (priložených v súboroch typu PDB alebo DWARF),
- rekonštrukcia idiómov,
- rekonštrukcia konštrukcií jazyka C++: hierarchie tried (RTTI, virtuálne tabuľky), preťažených funkcií, šablónových funkcií a podobne,
- integrovaný disassembler,
- výstup v dvoch bežných programovacích jazykoch: C a jazyk podobný Pythonu,
- generovanie grafu volaní, CFG (anglicky *control flow graph*) a iné štatistiky týkajúce sa dekompilovaného programu.

2.2 Proces dekompilácie

Proces dekompilácie je podobný procesu kompilácie - skladá sa z troch častí (analýza, optimalizácie a syntéza). Na konci procesu je vstupný binárny súbor reprezentovaný zdrojovým kódom v jazyku C.



Obr. 2.1: predspracovanie, prebrané z [9]

2.2.1 Fáza predspracovania

Najprv je vstupný binárny súbor prevedený do uniformnej reprezentácie, potom sú získané základné informácie. Výstupom je dočasná reprezentácia a súbor obsahujúci základné informácie. Fázu predspracovania ilustruje graf na obrázku 2.1. Jednotlivé časti predspracovania sú uvedené nižšie.

1. Prevod binárnych súborov do uniformnej reprezentácie prebieha pomocou knižnice `fileFormat`². Podobná knižnica existuje aj pre spracovanie súborov obsahujúcich ladiace informácie (DWARF, PDB).
2. Pomocou knižnice YARA (ktorá hľadá dopredu definované vzory v binárnych súboroch) získame základné informácie o architektúre (názov, endianitu a či sa jedná o 32/64-bitovú architektúru), funkciách, globálnych premenných, ďalej adresu vstupného bodu, formát súboru, typ súboru, použitý prekladač, a podobne. Získané informácie sú uložené v novom súbore vo formáte json.
3. Nasledujúci krok, je posledným pred generovaním LLVM IR. Knižnica (s názvom `imageLoader`) vyskúša staticky simulovať nahranie programu do pamäte (dôvodom je riziko, že niektoré

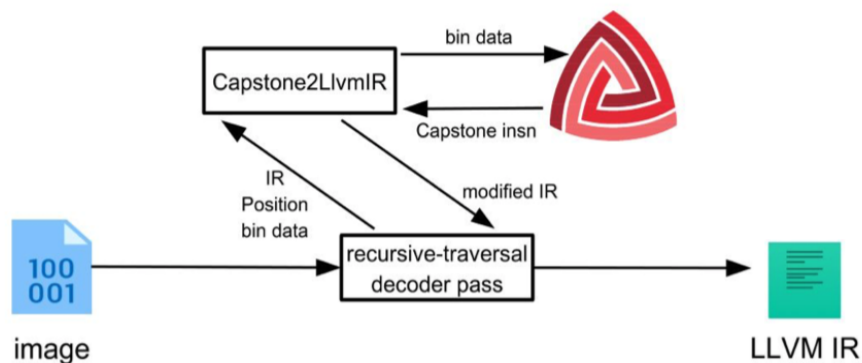
2. knižnica je súčasťou projektu

programy vyzerajú inak v pamäti počítača ako v binárnom súbore).

4. Výstupom tejto časti je obraz (anglicky *image*).

2.2.2 Vnútoraná fáza

V tejto časti je súbor prevedený pomocou rekurzívneho algoritmu, knižnice Capstone2LLVMIR a json súboru zo základnými informáciami z binárnej do LLVM IR reprezentácie. Tento proces ilustruje graf na obrázku 2.2.



Obr. 2.2: prevod binárnych reťazcov do LLVM IR. Obrázok prebraný z [9]

- Vstupom do tejto fázy je json súbor obsahujúci základné informácie, binárny súbor v uniformnej reprezentácii, (ak bol priložený) súbor obsahujúci ladiace informácie v uniformnej reprezentácii, obraz (anglicky *image* - výstup statickej simulácie), databázy obsahujúce špecifikácie rôznych ABI a signatúry knižničných funkcií.
- Prevod binárneho súboru do LLVM IR reprezentácie prebieha nasledovne: zo vstupného bodu rekurzívne prechádzame celý binárny súbor. Aktuálne analyzovaný binárny reťazec odovzdáme knižnici Capstone2LLVMIR³, tá vráti náležitú inštrukciu, inštrukcia je v ďalšom kroku prevedená do ekvivalentnej LLVM IR

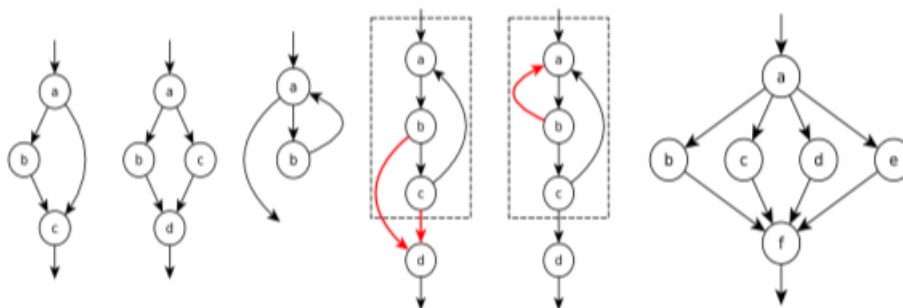
3. Capstone je disassembler [10]

formy. Rozhodnutie o dekódovaní nasledujúceho refazca závisí na práve obdržanej LLVM IR inštrukcii.

- Nasledujú rôzne transformácie kódu, ktoré nemenia sémantiku kódu. Jedná sa o optimalizácie z ktorých sa niektoré používajú bežne pri preklade (sú súčasťou napríklad prekladača Clang) a iné boli implementované pre potreby RetDecu: detekcia funkcie *main*, detekcia konštánt, odstránenie nedosiahnuteľných funkcií, detekcia idiómov, detekcia systémových volaní, transformácia (ne)podmienených skokov, detekcia systémových volaní, rekonštrukcia globálnych premenných, propagácia dátových typov, transformácia z globálnych do lokálnych premenných, rekonštrukcia C++ hierarchie a iné

2.2.3 Výstupná fáza

Nasleduje finálna fáza. Najprv je kód prevedený z LLVM IR do vnútornej reprezentácie BIR (účelom je rekonštrukcia vyšších riadiacich štruktúr a rôzne typy ďalších optimalizácií) a až následne do jazyka C.



Obr. 2.3: Obrázok prebraný z [9]. Riadiace štruktúry (zľava): if, if-else, while/for-loop, break, continue, switch.

- V tejto časti sa na reprezentáciu kódu nepoužíva LLVM IR ale BIR (anglicky *backend intermediate representation*) jedná sa o formu reprezentácie podobnú AST (abstraktný syntaktický strom, ktorého príklad som uviedol na obrázku 1.2).

- kód je transformovaný ďalšími optimalizáciami: detekcia riadiacich štruktúr (obrázok 2.3), redukcia premenných, zjednodušenie arit. výrazov ($a + -1 - (-4) \Rightarrow a + 3$), alebo zjednodušenie negácie ($if(!(a == b)) \Rightarrow if(a != b)$), zjednodušenie ukazovateľovej aritmetiky ($*(a + 4) \Rightarrow a[4]$), a podobne
- nasleduje inteligentné premenovanie napríklad: (*for* $i = 0; i < 10; ++i$), argumenty funkcie (*fun*($a1, a2, a3$)), ďalej (*return result, int len = strlen()*)
- výstupom tejto časti je súbor v jazyku C (súčasťou výstupu je CFG (anglicky *control flow graph*), CG (anglicky *call graph*) a súbory, ktoré vznikli pri procese dekompilácie).

3 McSema

Kapitola predstavuje dekompilačný nástroj McSema, dôvody vzniku, ciele a základné vlastnosti. Kapitola čerpá z konferencie REcon, ktorá prebehla v roku 2014 a z troch článkov publikovaných na stránke firmy *Trail of Bits* všetko dostupné z [11]. Síce je zdrojový kód nástroja McSema voľne dostupný, používa komerčné knižnice. Vo svojej bakalárskej práci som používal verziu nástroja McSema, kde bola komerčná komponenta nahradená voľne dostupnou alternatívou - implementácia bola predmetom diplomovej práce študenta fakulty informatiky v Brne [1].

3.1 Základná charakteristika

McSema (skratka znamená **M**achine code **S**emantics) je nástroj určený na dekompiláciu spustiteľných programov. Výstupný formát je LLVM bitkód (ten je možné jednoducho previesť na LLVM IR v textovej reprezentácii). Dôvodom vzniku a zároveň hlavným cieľom tohto nástroja je produkovať výstup v reprezentácii a kvalite, ktorú bude možné znova preložiť.

McSema a knižnica remill je vyvíjaná firmou *Trail of Bits* finančne podporovaná (a používaná) agentúrou DARPA (anglicky *Defense Advanced Research Projects Agency*) a Americkým ministerstvom obrany.

Podpora:

- 32 a 64 bitové (Linux) ELF a (Windows) PE binárne súbory,
- zdieľaných knižníc z Linux aj Windows,
- veľké množstvo x86 a x86-64 inštrukcií, vrátane X87, MMX, SSE, AVX,
- McSema funguje na operačných systémoch Windows (testované verzie 7 a 10) a Linux (testované verzie Ubuntu 14.04, 16.04, openSUSE).

Základné vlastnosti (funkcionalita) nástroja McSema:

- Výstup dekompilácie je možné analyzovať, preložiť a preložený kód bude mať rovnakú funkcionalitu ako pôvodný program.
- Dekompilácia kódu do LLVM IR umožňuje modifikáciu zdrojového kódu. Dôvody môžu byť od optimalizácie cez obfuskáciu (anglicky *obfuscation*) až po zmenu logickej štruktúry programu.
- Statická analýza zdrojového kódu je možná vždy pretože netreba kód prekladať, ani spustiť. Vďaka tomu, že McSema produkuje preložiteľné súbory je možné používať analytické metódy, ktoré vyžadujú preložiteľný zdrojový kód napríklad metódu *symbolickej exekúcie* a s ňou súvisiaci nástroje (*KLEE*, *PAGAI*, *LLBMC*).
- Kód v LLVM IR je možné využiť v rôznych nástrojoch založených na LLVM, napríklad *libFuzzer*.
- Výstup programu McSema umožňuje napríklad analýzu z hľadiska bezpečnostných rizík, validáciu komerčného programu alebo zistenie miery pokrytia kódu testami.

3.2 Proces dekompilácie

Pozostáva z dvoch častí. V prvej vznikne po analýze binárneho súboru CFG súbor, ten je vstupom do druhej časti. Druhá časť na základe CFG súboru vytvorí LLVM bitkód, ktorý by mal byť sémanticky ekvivalentný vstupnému programu.

3.2.1 mcsema-disass

Komponenta `mcsema-disass` pomocou jedného z nástrojov IDA Pro (komerčný nástroj, ktorý používa hlavná verzia), Binary Ninja (komerčný nástroj) alebo DynInst (voľne dostupná alternatíva k uvedeným disassemblerom) transformuje binárny súbor do formátu CFG (anglicky *control flow graph*) program `mcsema-disass` akceptuje nasledujúce argumenty:

- cestu ku disassembler-u (napríklad cesta ku programu DynInst),

- argumentom `arch` odovzdáme informáciu o akú architektúru sa jedná (x86, amd64 alebo aarch64),
- argumentom `os` odovzdáme typ operačného systému,
- argumentom `log_file` odovzdáme cestu k súboru kam sa majú zapísať informácie o procese (ak sa objavia, tak prípadne chyby),
- argumentom `binary` odovzdáme cestu k vstupnému binárnemu súboru,
- argumentom `entrypoint` definujeme vstupný bod odkiaľ má disassembler začať (napríklad `main`),
- voliteľný argument `std-defs` slúži pre uvedenie cesty k súboru, ktorý obsahuje definície externých funkcií.

DyninstAPI je voľne dostupný projekt. Nástroje a knižnice ktoré sú jeho súčasťou slúžia na analýzu alebo modifikáciu súborov v binárnej reprezentácii. Pomocou troch nástrojov z uvedeného projektu (SymtabAPI, ParseAPI a InstructionAPI) implementoval študent fakulty informatiky vo svojej diplomovej práci [1] alternatívu, ktorou nahradil v komponente `mcsema-disass` (pôvodne) použité komerčné knižnice - nasledujúci text z tejto práce plne vychádza [1]. Implementácia v súčasnosti podporuje iba 64-bitové ELF (Linux) binárne súbory napísané v jazyku C. Jej účelom je vyrobiť z binárnej reprezentácie CFG súbor. Tento cieľ je uskutočnený v troch krokoch:

1. Pomocou knižnice SymtabAPI sú rozpoznané externé symboly a externé funkcie, ktoré sa nachádzajú v binárnom súbore.
2. V nasledujúcom kroku sú spracované referencie [2] zo sekcií. Sekcie sú prechádzané lineárne po 64-bitových reťazcoch, výnimkou sú sekcie `.data` a `.rodata`, ktoré obsahujú väčšinu referencií pochádzajúcich z pôvodného kódu, prechádzané sú po 32-bitových reťazcoch. Rozpoznanie všetkých referencií z binárneho súbor

1. predovšetkým zo strán 53 až 56.

2. Referencia je spôsob akým sa v binárnej reprezentácii odkazuje kód na inú časť kódu v tom istom programe. Tento spôsob (forma adresy) je závislý na platforme. Referencie vznikajú napríklad ak sa v zdrojovom kóde nachádza nejaký ukazovateľ.

je pri dekompilácii kľúčové. Ak sa nepodarí niektoré referencie rozpoznať alebo sú rozpoznané nesprávne, výstup dekompilácie sa bude sémanticky líšiť od pôvodného binárneho súboru.

3. Posledná časť má za cieľ spracovanie funkcií definovaných v binárnom súbore (interných), ak nie je tabuľka symbolov dostupná, použije sa heuristika knižnice ParseAPI.

Štruktúra CFG je definovaná nasledovne:

- `Function` tu sa nachádzajú funkcie ktoré sú implementované v binárnom súbore. Každá funkcia má atribút `ea` (anglicky *effective address*) a pole základných blokov. Základné bloky obsahujú atribút `ea` a pole inštrukcií.
- `Segment` obsahuje dáta z binárneho súboru. Napríklad globálne premenné alebo premenné označené kľúčovým slovom `static` v jazyku C/C++
- `ExternalFunction` sa nachádzajú funkcie, ktoré sa v binárnom súbore používajú ale nie sú tam definované (napríklad funkcie z `libc` ako `malloc` alebo `strlen`).
- `ExternalVariable` obsahuje dáta, ktoré sa v kóde používajú ale nie sú tam definované.

3.2.2 mcsema-lift

Komponenta `mcsema-lift` (s využitím knižnice `remill`) preloží CFG do LLVM IR. Výstupom je LLVM bitkód (je možné triválne preložiť do LLVM IR textovej reprezentácie) obsahujúci všetky funkcie zo súboru CFG.

- každá funkcia z pôvodného (binárneho) súboru je v LLVM IR kóde reprezentovaná ako funkcia, ktorá akceptuje 3 argumenty. Ukazovateľ na dátovú štruktúru `State` (táto štruktúra obsahuje registre - simuluje tak stav pamäte), číslo reprezentujúce PC (anglicky *program counter*), ukazovateľ na dátovú štruktúru `Memory` (táto štruktúra slúži na zabezpečenie poradia inštrukcií). Návratová hodnota každej funkcie je ukazovateľ na dátovú štruktúru

Memory. Chovanie každej funkcie je vyjadrené spôsobom ako mení dátové štruktúry State a Memory.

- Externé funkcie sú definované pomocou samostatného súboru, v ktorom sa nachádzajú extrahované základné informácie o API známych (Windows, Linux) knižničných funkcií.
- Volania funkcií a návraty z funkcií (anglicky *callbacks*) vyžadujú heuristiky so spomenutými štruktúrami, ktoré simulujú reálny stav pamäte v CPU.
- Potom ako sú funkcie v LLVM IR kóde deklarované sa vkladajú dátové segmenty.
- Pred výstupom je kód optimalizovaný existujúcimi nástrojmi na optimalizáciu LLVM IR kódu (tie sú súčasťou LLVM).

```

Module {
  funcs = [
    Function {
      ea = 0x804b7a3;
      blocks = [
        Block {
          ea = 0x804b7a3;
          block_follows = [ ];
          insts = [
            // mov    eax, 0x1
            Instruction {
              ea = 0x804b7a3;
              bytes = "\xb8\x01\x00\x00\x00";
            },
            // push   ebx
            Instruction {
              ea = 0x0804b7a8;
              bytes = "\x53";
            },
            // mov    ebx, dword [esp+0x8]
            Instruction {
              ea = 0x0804b7a9;
              bytes = "\x8b\x5c\x24\x08";
            },
            // int     0x80
            Instruction {
              ea = 0x0804b7ad;
              bytes = "\xcd\x80";
              local_noreturn = true;
            }
          ];
        }
      ];
    }
  ];
}

```

Obr. 3.1: CFG súbor, prebraný z [11]

4 Testovacia sada

V tejto kapitole uvádzam spôsob akým testovanie prebiehalo a tabuľku, ktorá ilustruje skupiny testovaných súborov, počet a výsledky testov. Súbory a testovacie skripty sú prílohou tejto bakalárskej práce.

Mnou vytvorený testovací skript fungoval nasledovne:

1. preložil súbor (vznikol spustiteľný program),
2. do jednej premennej uložil štandardný výstup programu,
3. do druhej premennej uložil návratovú hodnotu programu,
4. nasledovala dekompilácia programu,
5. výstup dekompilátora bol preložený (vznikol nový spustiteľný program),
6. analogická činnosť ako v krokoch 2 a 3,
7. dekompilátor bol úspešný ak sa správanie programu ani v jednom prípade (porovnanie štandardných výstupov a návratových hodnôt) nelíšilo.

Tabuľka (4.1) ilustruje výstupy testovania skupín súborov, z väčších skupín uvádzam niekoľko súborov pre predstavu čo bolo predmetom testovania:

- skupina posix: `fork.c`, `fs-rename.c`, `socketpair.c`, `syscall.c`, `fs-stat.c`, `kill-zero.c`,
- skupina pracujúca s vláknami používa knižnicu `pthread`: `detach.c`, `barrier-init.c`, `mutex-unlock.c`, `deadlock-cycle.c`, `cond-init-global.c`,
- skupina `libc`: `errno.c`, `exit.c`, `getenv.c`, `sprintf.c`,
- skupina operácie s ukazovateľmi: `ptr-arith.c`, `ptr-fragments.c`,
- skupina testujúca operácie s číslami: `float-arithmetic.c`, `complex-long-double.c`, `float-cast.c`,

skupina testov	počet testov	RetDec	McSema
posix funkcie	60	40	60
operácie s vláknami	31	7	27
funkcie z libc	15	7	14
operácie s ukazovateľmi	4	2	4
jednoduché operácie s desatinnými číslami	3	0	3
zložité operácie s desatinnými a celými číslami	3	0	1
pole ako globálna premenná	4	0	4
pole ako lokálna premenná	2	0	2
quicksort a bubblesort	2	1	1
variadické argumenty	2	0	2
jednoduché testy	4	4	4
funkcie pracujúce so súborami	2	1	2
operácie s konštantami (číslo a reťazec)	2	0	2
rôzne	33	18	31
fibonacci	1	1	1
program pracujúci s triedami (struct)	1	0	1
program so zdieľanou knižnicou	1	1	0
celkovo	170	82	159

Tabuľka 4.1: Výsledky testovania

- skupina jednoduché testy: `simple-for-loop.c`, `simple-main.c`, `simple-switch.c`,
- skupina označená *rôzne* obsahuje súbory, ktoré testujú: operácie s globálnymi premennými (`global-overflow.c`, `globals-and-io.c`, `global-initloop.c`), (ne)podmienené skoky (`setjmp.c`), `long-jmp-complex.c`, násobenie matíc (`matrix-vector-mult.c`), atomické operácie (`atomic-op-and-fetch.c`) a iné (napríklad aj program `helloworld.c`).

5 Zhrnutie

Testy uvedené v minulej kapitole jasne preukázali, že ak ide o možnosť prekladať výstup dekompilátora, McSema je jednoznačne vhodnejší kandidát.

Najčastejšie problémy pri testovaní súborov získaných dekompilátorom RetDec:

- komponenta `bin2llvmir` produkuje množstvo deklarácií funkcií, ktoré nie sú definované. Dôvody môžu byť napríklad nasledujúce:
 - ak sa v binárnom súbore nenachádza implementácia funkcie (funkcia je definovaná v dynamicky linkovanej knižnici)
 - telo funkcie môže byť odstránené pri detekcii staticky linkovaného kódu (funkcia zo staticky linkovanej knižnice)
 - aby LLVM optimalizácie z nejakých dôvodov neodstránili niektoré premenné, RetDec deklaruje množstvo funkcií bez definície.
- RetDec nie je schopný dekompilovať ukazovatele na funkcie.
- Problém sú aj staticky alokované dáta, takže ak bolo v pôvodnom kóde napríklad globálne pole nejakej veľkosti (`T buffer[size]`), vo výstupe dekompilácie je to neinicializovaný ukazovateľ.
- RetDecu chýba napríklad podpora pre MMX (220 inštrukcií) a SSE (70 nových inštrukcií).
- Niekedy zlyhá rekonštrukcia typov.
- Ak obsahoval pôvodný kód nepodmienené skoky (`goto`), tak je možné, že výstup dekompilátora nebude obsahovať niektoré návesty alebo ich bude obsahovať niekoľko (duplikácie).

Dôvody prečo existuje toľko problémov (ktoré McSema vyriešila) pri dekompilácii RetDecom možno priblížiť tvrdenia jeho vývojárov: *"Frankly, we never thought of recompiling LLVM IR as the LLVM IR that*

we generate serves only as a middle layer in the whole decompilation process (binary to C)."

*"It has never been our goal to produce LLVM IR or C that could be re-compiled. The goal was to produce something, that could be analyzed by a human."*¹

McSema uspela v podstate vo všetkých testoch.

1. <https://github.com/avast/retdec/issues/45>

6 Záver

Prvá časť práce predstavuje úvod do tématiky dekompilácie. Okrem uvedenia všeobecného procesu, cieľov a problémov dekompilácie, približujem proces prekladu a jeho vzťah k dekompoliácii. Spolu s jazykom LLVM IR opisujem projekt LLVM na ktorého báze sú porovnané dekompilátory založené. V prílohe sa nachádzajú monofunkčné programy, ktoré boli spolu so skriptami použité pri testovaní.

Vlastným jadrom práce je porovnanie dvoch voľne dostupných dekompilačných nástrojov RetDec a McSema vzhľadom na perspektívu ich uplatnenia vo formálnom verifikátore (ktorý používa metódu overenia modelu) DIVINE - nástroji vyvíjanom v laboratóriu ParaDiSe na fakulte informatiky v Brne. Práve použitie jazyka LLVM IR, ktorým DIVINE vnútorne reprezentuje vstupný program, umožňuje širokú podporu rôznych programovacích jazykov, vrátane C a C++. Obidva testované dekompilátory používajú LLVM IR a v blízkej budúcnosti by mala byť práve prostredníctvom vhodného dekompilátoru rozšírená podpora nástroja DIVINE tak aby dokázal verifikovať nielen zdrojové kódy, ale aj spustiteľné programy v binárnej reprezentácii. Systematické testy preukázali, že na daný účel je nástroj McSema vhodnejší.

Bibliografia

1. KORENČIK, Lukáš. *Decompiling Binaries into LLVM IR Using McSema and Dyninst* [online]. [cit. 2019-05-24]. Dostupné tiež z: <https://is.muni.cz/th/pxe1j/>. Diplomová práca. Masarykova univerzita, Fakulta informatiky, Brno. Vedúci práce Petr ROČKAI.
2. BARANOVÁ, Zuzana; BARNAT, Jiří; KEJSTOVÁ, Katarína; KUČERA, Tadeáš; LAUKO, Henrich; MRÁZEK, Jan; ROČKAI, Petr; ŠTILL, Vladimír. Model Checking of C and C++ with DIVINE 4. In: *Automated Technology for Verification and Analysis*. Springer, 2017, zv. 10482, s. 201–207. LNCS. Dostupné z DOI: [10.1007/978-3-319-68167-2_14](https://doi.org/10.1007/978-3-319-68167-2_14).
3. ULLMAN, Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston: Addison-Wesley Longman Publishing Co., 2006. ISBN 0-201-10088-6.
4. *This figure shows a possible derivation tree for the expression $a = b < c ? *p + b * c : 1 \ll d()$ written in the C language* [online]. Wikimedia Commons, 2018 [cit. 2019-08-05]. Dostupné z: https://commons.wikimedia.org/wiki/File:Arithmetic_C_IPL.png.
5. TEAM, The Clang. *Static analysis tools: using Clang in CppDepend*. 2017 [cit. 2019-01-05]. Dostupné z: <http://blog.llvm.org/2013/04/static-analysis-tools-using-clang-in.html>.
6. *LLVM Project* [online]. LLVM Language Reference Manual, 2003 [cit. 2019-01-05]. Dostupné z: <http://llvm.org/docs/LangRef.html>.
7. *LLVM Project* [online]. LLVM Namespace Reference, 2003–2019 [cit. 2019-01-05]. Dostupné z: <http://llvm.org/doxygen/namespacellvm.html>.
8. CIFUENTES, Cristina. *Reverse Compilation Techniques*. 1994.
9. KŘOUSTEK, J.; MATULA, P. *RetDec: An Open-Source Machine-Code Decompiler* [[talk]]. 2018. Presented at Pass the SALT 2018, Lille, FR.
10. *Capstone is a lightweight multi-platform, multi-architecture disassembly framework*. [online]. Capstone, 2018 [cit. 2019-08-05]. Dostupné z: <https://www.capstone-engine.org>.

11. *Static Translation of X86 Instruction Semantics to LLVM with McSema* [online]. McSema, 2018 [cit. 2019-01-05]. Dostupné z: <https://github.com/trailofbits/publications/tree/master/presentations/McSema%20-%20Static%20Translation%20of%20x86%20instructions%20to%20LLVM%20IR>.

A Elektronické přílohy

Prílohy:

- súbor tests
- súbor scripts