VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

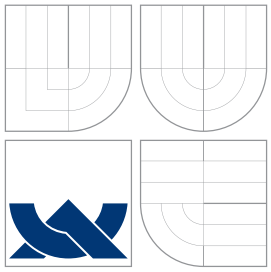
OPTIMALIZACE LLVM IR PRO ÚČELY ZPĚTNÉHO PŘEKLADU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

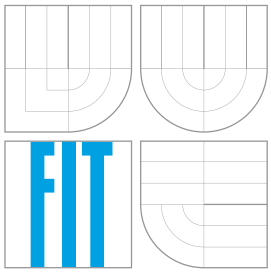
AUTOR PRÁCE
AUTHOR

Bc. JAROSLAV KOLLÁR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE LLVM IR PRO ÚČELY ZPĚTNÉHO PŘEKLADU

LLVM IR OPTIMIZATIONS FOR DECOMPILATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAROSLAV KOLLÁR

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETER MATULA

BRNO 2015

Abstrakt

Tato práce se zabývá návrhem a implementací optimalizací ve střední části zpětného překladače vyvíjeného firmou AVG Technologies. Úlohou těchto optimalizací je zlepšit čitelnost produkovaného zdrojového kódu a současně vylepšit některé navržené optimalizace v zadní části zpětného překladače. V úvodu jsou poskytnuty základní informace o reverzním inženýrství a zpětných překladačích, které slouží pro účel uvedení do dané problematiky. Poté se nachází popis stavu zpětného překladače před zahájením této diplomové práce. Dále následuje hlavní část práce, která se věnuje popisu návrhu a implementace jednotlivých navržených optimalizací. Poté následuje popis testování optimalizací a shrnutí dosažených výsledků. V závěru práce je diskutován budoucí možný vývoj.

Abstract

This master's thesis describes the design and implementation of optimizations in the middle-end part of a retargetable decompiler developed by AVG Technologies. The purpose of these optimizations is to improve readability of the produced source code and improve existing optimizations in the back-end part. In the introduction, basic information about reverse engineering and decompilation is provided. Then, a description of state of the retargetable decompiler before this work is given. After that, the main part of this work is presented, which describes the design and implementation of the proposed optimizations. Then, the techniques that were used for testing are described. This description is followed by a summary of the achieved results. The present work is concluded by a discussion of possible future development.

Klíčová slova

Reverzní inženýrství, zpětný překladač, zpětný překlad, střední část zpětného překladače, optimalizace, čitelnost zdrojového kódu.

Keywords

Reverse engineering, decompiler, decompilation, middle-end part of a decompiler, optimization, source code readability.

Citace

Jaroslav Kollár: Optimalizace LLVM IR pro účely zpětného překladu, diplomová práce, Brno, FIT VUT v Brně, 2015

Optimalizace LLVM IR pro účely zpětného překladu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně za odborného vedení Ing. Petra Matuly a konzultanta Ing. Petra Zemka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jaroslav Kollár
25. května 2015

Poděkování

Děkuji svému vedoucímu Ing. Petru Matulovi a konzultantovi Ing. Petru Zemkovi, Ph.D. za odborné vedení, za poskytnuté rady a za čas, který mi při tvorbě práce věnovali.

Obsah

Kapitola 1

Úvod

V dnešnej dobe je v rámci informačných technológií často kladený dôraz na bezpečnosť. Medzi jedno z najväčších rizík je možné zaradiť rôzne druhý škodlivého softvéru. Typickými príkladmi sú klasické počítačové vírusy, trójske kone, internetové červy a iné. Všeobecné označenie týchto škodlivých programov je možne pomocou pojmu malvér [21]. V súčasnosti dochádza k veľkému nárastu škodlivých softvérov, ako aj k vzniku nových druhov. Veľký vzostup malvéru je možné vidieť pri mobilných zariadeniach a tabletoch. Ide predovšetkým o operačný systém Android [4]. Typickými úlohami škodlivého softvéru je najčastejšie získavanie citlivých dát od užívateľov, poškodzovanie súborov, znepríjemňovanie práce na počítači a podobné [21]. Jednou z možných techník obrany je nespúšťanie a neotváranie nedôveryhodných materiálov. Populárna a osvedčená technika je využívanie antivírusových programov. Antivírusové programy prehľadávajú súbory, sledujú správanie aplikácií a aktivitu systému na pozadí. Pri týchto činnostiach súčasne porovnávajú či napríklad chovanie niektorej aplikácie neodpovedá správaniu sa napadnutého programu. V súboroch zase hľadajú sekvenciu inštrukcií, ktorá odpovedá známemu malvéru [8]. Schopnosť detekovať malvér je však limitovaná kvalitou antivírusového programu. Preto je dôležité dbať na vývoj techník, ktoré umožnia vylepšiť kvalitu programov bojujúcich s malvérom. Jednou z techník, ktorú môžu použiť tvorcovia aplikácii proti malvéru, je softvérové reverzné inžinierstvo. Ako príklad je možne uviesť využitie spätného prekladu nad získaným binárnym súborom. O spätný preklad sa postará spätný prekladač, pričom jeho úlohou je z tohto súboru vygenerovať program vo vysokoúrovňovom jazyku. Následne je možné tento kód preskúmať a vyhodnotiť jeho činnosť [13]. Táto technika je výhodná v tom, že daný súbor nie je potrebné ani spustiť.

Na základe uvedeného je zrejmé, že spätný prekladač v rukách tvorcov antivírusových programov môže znamenať veľký prínos. Za týmto účelom je vyvíjaný aj spätný prekladač spoločnosťou AVG Technologies. Prostredníctvom verejne prístupnej webovej služby si je možné tento spätný prekladač vyskúšať [1]. Spätný prekladač vyvíjaný spoločnosťou AVG Technologies je rozdelený na tri časti. Ide o prednú časť, strednú časť a zadnú časť. Úlohou prednej časti je prevod binárneho kódu platformovo závislej aplikácie do platformovo nezávislého kódu. Výstupom prednej časti je kód v jazyku LLVM IR [15], ktorý spĺňa uvedenú vlastnosť [24]. LLVM IR predstavuje prechodnú reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízkoúrovňové operácie, flexibilitu a schopnosť prevodu z množstva vysokoúrovňových jazykov. Cieľom jeho tvorcov bolo dosiahnuť jednoduchý nízkoúrovňový jazyk, ktorý je typovaný a rozšíriteľný [15]. V strednej časti dochádza k optimalizovaniu

tohto LLVM IR kódu, pričom cieľom je ho upraviť do čo najvhodnejšej podoby pre zadnú časť spätného prekladača. Toto optimalizovanie je vykonané prostredníctvom vstavaných optimalizácií nachádzajúcich sa vo frameworku LLVM [24]. LLVM framework bol navrhnutý ako inovatívny framework pre prekladače a pre tvorbu nástrojov na nich založených. Obsahuje množinu jazykovo nezávislých inštrukcií, veľké množstvo vstavaných optimalizačných algoritmov a analýz a už spomínanú prechodnú reprezentáciu LLVM IR [15]. Stredná časť je vo veľkej miere závislá na tomto frameworku. Zadná časť spätného prekladača zabezpečuje generovanie vysokoúrovňového kódu. Nachádzajú sa tu aj optimalizácie, ktorých úlohou je zvýšiť čitateľnosť produkovaného kódu. Aktuálne sú podporované dva vysokoúrovňové jazyky a to jazyk C a modifikovaná verzia jazyka Python [24].

Ako už bolo zmienené, v zadnej časti spätného prekladača sa nachádzajú optimalizácie, ktorých úlohou je zvýšiť čitateľnosť produkovaného kódu. Niektoré tieto optimalizácie sú však príliš pomalé na veľkých binárnych súboroch (súbory o veľkosti niekoľkých MB). Hlavnou úlohou tejto diplomovej práce je presun týchto optimalizácií do strednej časti a prípadné rozšírenie ich funkčnosti. Pri návrhu tohto presunu sa vychádzalo z toho, že LLVM framework poskytne lepšiu podporu pre tieto optimalizácie, a tak dôjde k ich zrýchleniu.

V rámci tejto diplomovej práce bolo navrhnutých a implementovaných niekoľko optimalizácií. Ich detailný popis bude prezentovaný v neskorších častiach textu. Ide napríklad o optimalizáciu riešiacu usporiadanie PHI uzlov. Jej úlohou je riešiť problém spojený s paralelným vyhodnocovaním inštrukcie `phi` nachádzajúcej sa v jazyku LLVM IR [25]. Výstup spätného prekladu je však v jazyku C, ktorý je vyhodnocovaný sekvenčne. Preto je potrebné aby táto optimalizácia transformovala kód LLVM IR do funkčne ekvivalentnej sekvenčnej podoby. Ďalšou optimalizáciou je optimalizácia odstraňujúca mŕtve priradenia do globálnych premenných. Pod pojmom mŕtve priradenie do globálnej premennej rozumieme situáciu, pri ktorej dôjde k priradeniu hodnoty do globálnej premennej a táto hodnota nie je nikde v kóde z nej prečítaná. Takéto priradenie môže byť odstránené. Optimalizácia konvertujúca globálne premenné na lokálne patrí medzi ďalšie navrhnuté a implementované optimalizácie v tejto diplomovej práci. Jej úlohou je konvertovať globálne premenné na lokálne všade tam, kde je to možné. To znamená vytvárať nové lokálne premenné vo funkciách a nahrádzať použitia globálnych premenných pomocou týchto lokálnych premenných. Ďalšou optimalizáciou je optimalizácia odstraňujúca nedosiahnuteľné funkcie. Táto optimalizácia odstraňuje funkcie, ktoré nie sú nikdy volané. Poslednou navrhnutou a implementovanou optimalizáciou je optimalizácia odstraňujúca kód za volaniami funkcií, ktoré sa nevracajú. Jej úlohou ako už z názvu vyplýva je zabezpečiť odstránenie kódu za takýmito funkciami. Jednou z takých funkcií je napríklad `exit()` zo štandardnej knižnice jazyka C.

Práca je štruktúrovaná nasledovne. Kapitola 2 obsahuje základné informácie o reverznom inžinierstve a jeho použití v rámci informačných technológii, ako aj informácie o spätnom preklade. Detailný popis spätného prekladača vyvíjaného spoločnosťou AVG Technologies sa nachádza v kapitole 3. V tejto kapitole je možné nájsť aj základné informácie o frameworku LLVM. V kapitole 4 sa nachádza popis prechodnej reprezentácie LLVM IR. Stav spätného prekladača pred zahájením tejto diplomovej práce a uvedenie jej cieľov obsahuje kapitola 5. Návrhu a popisu jednotlivých optimalizácií sa venuje kapitola 6. Kapitola 7 popisuje implementáciu navrhnutých optimalizácií. Popis techník využitých pri testovaní týchto optimalizácií obsahuje kapitola 8. V kapitole 9 sa nachádza zhodnotenie dosiahnutých výsledkov navrhnutých a implementovaných optimalizácií. Záverečná kapitola 10 obsahuje zhodnotenie práce a diskutuje možný budúci vývoj.

Kapitola 2

Reverzné inžinierstvo a spätný preklad

Táto kapitola je založená na [7, 9].

Pod pojmom reverzné inžinierstvo chápeme proces, ktorého úlohou je získať chýbajúce znalosti o skúmanom predmete. Najčastejšie ide o znalosti, ktoré nie sú voľne dostupné. V niektorých prípadoch sú to zničené alebo stratené informácie. Tento pojem sa datuje už od dôb priemyselnej revolúcie a nie je striktne viazaný na modernú dobu. Reverzné inžinierstvo je veľmi často spájané s armádou. Tá ho využíva pri skúmaní technológií nepriateľa a následne sú potom tieto získané informácie využité pre tvorbu nových vlastných technológií. Počas studenej vojny a v obidvoch svetových vojnách existuje veľa príkladov upozorňujúcich práve na použitie reverzného inžinierstva. Moderná doba však prináša novú formu reverzného inžinierstva a to softvérové reverzné inžinierstvo. Pri reverznom inžinierstve je dôležitá aj otázka právnej legálnosti. V niektorých štátoch sú určité aktivity spadajúce pod reverzné inžinierstvo považované za nezákonné [7].

V tejto kapitole je možné nájsť základné informácie o problematike reverzného inžinierstva. Konkrétne podkapitola 2.1 sa venuje softvérovému reverznému inžinierstvu a jeho použitiu v praxi. Ďalej sa v tejto kapitole nachádza popis spätného prekladu, pretože spätný preklad predstavuje jednu z techník, ktoré reverzné inžinierstvo využíva. Spätnému prekladu sa venuje podkapitola 2.2. V tejto podkapitole sa nachádza aj popis výhod využitia spätného prekladača oproti disassembleru.

2.1 Softvérové reverzné inžinierstvo a jeho použitie

Softvérové reverzné inžinierstvo (ďalej len SRE z angl. *software reverse engineering*) a jeho použitie môžeme rozdeliť na dva prípady. V prvom prípade je v praxi používané ako analýza určitého softvérového systému s cieľom získať implementačné detaily, prípadne jeho návrh. Môže byť aplikovaná buď iba na jeho určitú časť alebo na celý systém [5].

V druhom prípade ide o prevod informácie z nízkoúrovňového formátu (napr. aplikačného binárneho kódu) na čitateľnejšiu formu na určitej úrovni abstrakcie. Takýto kód na vyššej úrovni abstrakcie je lepšie čitateľný a zrozumiteľnejší [3].

V dnešnej dobe sa často používajú tieto techniky na detekciu a analýzu malvéru. Rovnako ich je možné využiť pri odhaľovaní bezpečnostných chýb a slabín kreditných kariet. SRE je však často používané tak, že porušuje niektoré zákony. Napríklad ide o odstraňovanie ochrany proti kopírovaniu z hier, CD nosičov a podobne. Zneužívanie SRE k získavaniu cudzích zdrojových kódov a implementačných detailov je tiež časté a nepatrí medzi ojedinelé prípady jeho využitia. Tieto získané informácie sú potom využívane pri tvorbe novej technológie, prípadne dochádza len k šíreniu získaných informácií [22].

Typickými príkladmi nástrojov využívaných pri SRE sú disassembler a spätný prekladač. Disassembler prevádza strojový kód do jazyka symbolických inštrukcií [19]. Cieľom spätného prekladača je preklad binárneho kódu závislého na určitej platforme do zdrojového kódu vysokoúrovňového jazyka. Spätný prekladač je možné chápať oproti dissasembleru ako program na vyššej úrovni. Je to z toho dôvodu, že výstupom spätného prekladača je kód na vyššej úrovni, než kód produkovaný dissasemblerom [6]. Ku skomplikovaniu využitia SRE je možné použiť tzv. obfuskátor. Ten slúži k prevodu zdrojového kódu do zdrojového kódu v tom istom jazyku, avšak vykoná na ňom niekoľko zmien tak, aby bol takýto kód ťažko čitateľný. Obidva kódy sú však funkčne ekvivalentné [20]. Ďalšou možnosťou je využitie tzv. protektorov. Tie na výsledný binárny kód použijú kompresiu alebo ho pomocou šifrovacích techník zašifrujú a vložia kód na jeho automatické rozšifrovanie pri spustení [18].

2.2 Spätný preklad

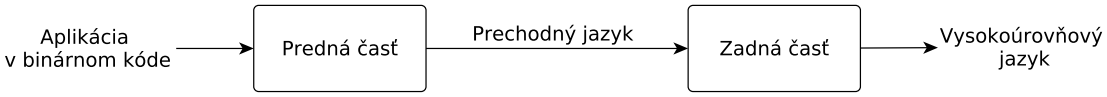
Spätný preklad je reverzný proces bežného prekladu, pričom jeho cieľom je vytvorenie vysokoúrovňového kódu z binárneho kódu. Pre tento vysokoúrovňový kód by však malo platiť, že je funkčne ekvivalentný s kódom v binárnej podobe. Hlavná motivácia spätného prekladu je tá, že tento kód na vyššej úrovni abstrakcie je v porovnaní s tým na nižšej úrovni čitateľnejší [6].

Spätný preklad vykonáva spätný prekladač. Pri spätnom prekladači je však potrebné uviesť, že binárny kód na ktorom je vykonaný spätný preklad je platformovo závislý. Spätný prekladač vykonáva reverzný proces prekladača [6]. Tento reverzný proces nie je ale možné chápať ako vrátenie krokov, ktoré vykonal prekladač. Takéto vrátenie krokov prekladača nie je možné z toho dôvodu, že v binárnom kóde sa nenachádzajú informácie o tom, aké kroky pri preklade vykonal prekladač. S procesom spätného prekladu je však spojený jeden významný problém. Tento problém súvisí s tým, že kód v binárnej podobe obvykle neobsahuje explicitné informácie akými sú napríklad definície dátových štruktúr, názvy premenných, prípadne informácie o iných dátových typoch. Často sa vedú polemiky o tom či je vôbec možné zostrojiť kvalitný spätný prekladač [7]. Vytvorenie zdrojového kódu vysokoúrovňového jazyka, ktorý identicky vykonáva funkcie pôvodného programu, je však pre niektoré jazyky možné [6]. Tento kód sa však vo väčšine prípadoch nepodobá na originálny zdrojový kód a býva veľmi ťažko čitateľný. Vo výslednom kóde chýbajú napríklad komentáre, názvy pôvodných premenných, deklarácie pôvodných dátových štruktúr. Takisto jedným z dôvodov, prečo výsledný kód po spätnom preklade neodpovedá štruktúre pôvodného kódu, sú optimalizácie vykonávané prekladačom [7].

Spätný prekladač ako už bolo uvedené je možné považovať za ďalší vývojový stupeň po disassembleru. Úlohou spätného prekladača je vytvoriť kód na vyššej úrovni abstrakcie, než kód v jazyku symbolických inštrukcií. Takýto kód je jednoduchšie čitateľný a analyzovateľný. Hlavnou myšlienkou spätného prekladu je získať čo najpodobnejší výstupný kód s kódom,

ktorý bol preložený do binárnej podoby. Spätný prekladač je však možné použiť aj na vytvorenie výstupného kódu v inom jazyku, než v akom bol pôvodný kód napísaný [11].

Jedna z možných štruktúr spätného prekladača je uvedená na obrázku 2.1. Táto štruktúra je veľmi podobná prekladaču. Jednotlivé časti však pracujú v opačnom poradí. Úlohou prednej časti je dekódovať inštrukcie v nízkoúrovňovom jazyku a ich prevod do prechodnej reprezentácie. Prechodná reprezentácia predstavuje vyššiu formu abstrakcie, než pôvodný nízkoúrovňový jazyk. Nezávislosť reprezentácie na nejakej konkrétnej architektúre je možné chápať ako ďalšiu veľkú výhodu. Úlohou zadnej časti je z prechodnej reprezentácie vygenerovať zvolený vysokoúrovňový jazyk [7].



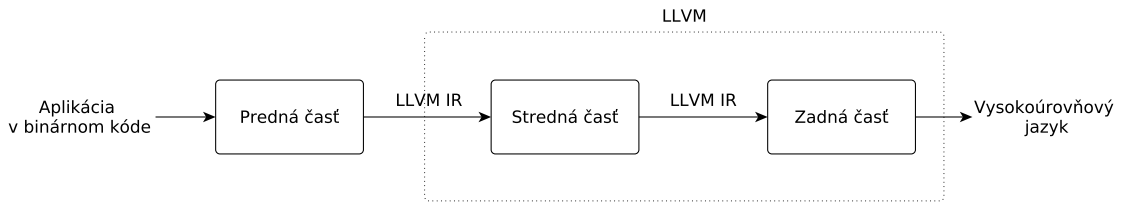
Obr. 2.1: Jedna z možných štruktúr spätného prekladača.

Kapitola 3

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies je platformovo nezávislý na architektúre (MIPS, ARM, Intel x86, atď.) a súborových formátoch (ELF, PE, Mach-O, atď.), generujúci zvolený vysokoúrovňový jazyk. Jeho využitie je plánované pri analýze malvéru napríklad pre mobilné zariadenia, tablety a iné podobné zariadenia. Momentálne sú podporované architektúry Intel x86, ARM, inštrukčné rozšírenie Thumb, MIPS, PIC32 a PowerPC. Medzi podporované súborové formáty patrí ELF a PE. Výstup je možný v jazyku C alebo v modifikovanej verzii jazyka Python [24]. Tento spätný prekladač a jeho funkcionalitu si je možné vyskúšať prostredníctvom verejne prístupnej služby [1].

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies sa skladá z troch častí. Z prednej, strednej a zadnej časti. Ich prepojenie demonštruje obrázok 3.1. Úlohou prednej časti je prevod binárneho kódu platformovo závislej aplikácie do platformovo nezávislého kódu. Kód v jazyku LLVM IR predstavuje takýto nezávislý kód, a preto je výstupom prednej časti. Stredná časť nad kódom získaným z prednej časti vykoná optimalizácie, pričom jej cieľom je získať kód čo najvhodnejší pre zadnú časť spätného prekladača. Tieto optimalizácie sú vykonávané nad kódom v jazyku LLVM IR a výstupom je optimalizovaný kód opäť v tomto jazyku. Úlohou zadnej časti je vyprodukovať vysokoúrovňový kód v čo najčitateľnejšej podobe [24].



Obr. 3.1: Štruktúra spätného prekladača vyvíjaného spoločnosťou AVG Technologies [24].

Táto kapitola je štruktúrovaná nasledovne. Podkapitola 3.1 obsahuje popis frameworku LLVM, na ktorom je založená stredná časť a čiastočne zadná časť. Podkapitola 3.2 sa venuje prednej časti. Strednú časť popisuje podkapitola 3.3. Táto časť spätného prekladača je z hľadiska tejto diplomovej práce najvýznamnejšia, pretože optimalizácie navrhnuté v rámci tejto

diplomovej práce sú určené práve pre túto časť spätného prekladača. Záverečná podkapitola 3.4 sa venuje zadnej časti.

3.1 LLVM

LLVM framework bol navrhnutý ako inovatívny framework pre prekladače a pre tvorbu nástrojov na nich založených. Obsahuje množinu jazykovo nezávislých inštrukcií, veľké množstvo vstavaných optimalizačných algoritmov a analýz a prechodnú reprezentáciu LLVM IR. LLVM IR spĺňa takzvanú *static single assignment* (ďalej len SSA) formu [15]. Pre SSA formu platí, že do každej premennej môže dôjsť iba práve k jednému priradeniu. Súčasne musí byť splnená podmienka, že každá premenná je definovaná pred jej použitím [23]. LLVM IR predstavuje prechodnú reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízkoúrovňové operácie, flexibilitu a schopnosť prevodu z množstva vysokoúrovňových jazykov. Cieľom jeho tvorcov bolo dosiahnuť jednoduchý nízkoúrovňový jazyk, ktorý je typovaný a rozšíriteľný [15]. Detailný popis jazyka LLVM IR sa nachádza v samostatnej kapitole 4. Pre nasledujúcu časť textu však nie je potrebná znalosť uvedenej kapitoly. Tá je potrebná až v neskorších častiach venujúcich sa navrhnutým a implementovaným optimalizáciám.

3.2 Predná časť

Predná časť spätného prekladača je jediná časť, ktorá je platformovo závislá. Jej hlavnou úlohou je prevod binárneho kódu platformovo závislej aplikácie do sekvencie LLVM IR inštrukcií [12].

V prednej časti dochádza k odstraňovaniu staticky linkovaného kódu z binárneho kódu, pretože tento kód zbytočne zvyšuje neprehľadnosť výsledného výstupného kódu. Týka sa to napríklad knižničných funkcií typu `printf`, kde je všeobecne známe, čo tieto funkcie vykonávajú. Takto upravený binárny kód je vstupom pre takzvaný inštrukčný dekodér. Jeho funkcionalita je podobná disassembleru, avšak výstupom nie je kód v jazyku symbolických inštrukcií, ale sekvencia inštrukcií odpovedajúca reprezentácii LLVM IR. Táto sekvencia inštrukcií je odlišná od LLVM IR, ale pripomína ju a predstavuje vnútornú reprezentáciu prednej časti, pričom platí, že je platformovo nezávislá [10].

V prednej časti spätného prekladača sa nachádza niekoľko analýz. Jednou z týchto analýz je analýza toku riadenia programu. Táto analýza je zodpovedná za rozpoznávanie základných blokov a vytvorenie takzvaného grafu toku riadenia. Ďalšou jej úlohou je rozpoznanie všetkých skokových inštrukcií a rozpoznanie cieľových adries týchto skokových inštrukcií. Ďalšia analýza, ktorá sa v tejto prednej časti nachádza je analýza funkcií. Jej základnou úlohou je rozpoznať funkcie. V prednej časti sa ešte nachádza analýza dátového toku. Táto analýza je zodpovedná za rekonštrukciu dátových typov, rekonštrukciu argumentov funkcií ako aj ich návratových hodnôt. Okrem týchto uvedených analýz existujú v prednej časti ďalšie analýzy. Tie však nemá zmysel popisovať, pretože tie vyššie uvedené patria medzi najpodstatnejšie [10].

Poslednou inštanciou prednej časti je generátor LLVM IR kódu. Tento generátor generuje LLVM IR kód z vnútornej reprezentácie tak, aby získaná LLVM IR reprezentácia bola čo najvhodnejšia pre ďalšie časti spätného prekladača [10].

3.3 Stredná časť

Vstupom strednej časti spätného prekladača je kód v jazyku LLVM IR. Rovnako je kód v tomto jazyku aj výstupom pre túto časť. Úlohou strednej časti je optimalizovať kód v jazyku LLVM IR tak, aby bol čo najvhodnejší pre zadnú časť a zvýšil tak schopnosť generovania čitateľnejšieho kódu zadnou častou spätného prekladača.

V strednej časti je využívaný nástroj `opt` [13]. Tento nástroj poskytuje LLVM framework a pomocou neho je možné spúšťať veľké množstvo vstavaných optimalizácií a analýz. Zároveň je možné pomocou neho spúšťať i vlastné implementované optimalizácie. Vďaka tomuto nástroju dochádza vo veľkej miere k optimalizovaniu kódu v jazyku LLVM IR [16].

V strednej časti sa nachádza aj analýza idiómov, ktorá prispieva k čitateľnejšiemu kódu. Táto analýza už bola vytvorená pred zahájením tejto diplomovej práce a zároveň platí, že nie je súčasťou LLVM frameworku. Je však spúšťaná pomocou nástroja `opt`. Analýza je potrebná z dôvodu, že prekladač sa pri preklade snaží niektoré inštrukcie nahradiť za iné inštrukcie, ktoré sú rýchlejšie vykonateľné počítačom. Často však vďaka tomu dôjde k použitiu viacerých inštrukcií miesto jednej. Z toho vyplýva, že takto dochádza k zhoršovaniu čitateľnosti kódu, a preto táto analýza idiómov sa snaží vrátiť zmeny vykonané prekladačom [10].

V strednej časti sa okrem vstavaných optimalizácií a analýzy inštrukčných idiómov nachádzajú navrhnuté a vytvorené optimalizácie v tejto diplomovej práci. Úlohou týchto optimalizácii je rovnako zlepšiť LLVM IR kód pre zadnú časť spätného prekladača, čo následne vedie k čitateľnejšiemu vysokoúrovňovému kódu. Tieto optimalizácie sú spúšťané rovnako ako tie vstavané pomocou nástroja `opt`. Ich detailný popis návrhu obsahuje kapitola 6.

3.4 Zadná časť

Poslednou časťou spätného prekladača je zadná časť. Úlohou tejto časti je produkovanie vysokoúrovňového kódu tak, aby bol tento kód čo najčitateľnejší. Vstupom zadnej časti je LLVM IR kód. Jej výstupom je kód vysokoúrovňového jazyka. V zadnej časti dochádza k prevodu LLVM IR do BIR. BIR je vnútornou reprezentáciou tejto časti spätného prekladača, pričom je vytvorený na základe kódu LLVM IR. Prechodná reprezentácia BIR umožňuje zachytiť všetky prvky do pamäti, pričom nie je potreba žiadnej textovej reprezentácie. Jej dôležitou schopnosťou je, že dokáže modelovať všetky podstatné konštrukcie jazyka LLVM IR z hľadiska spätného prekladača. Okrem toho je schopná zoskupovať viacero príkazov a výrazov do jedného, ktoré sú následne prevedené do vysokoúrovňového jazyka. Ďalšou vlastnosťou BIR je umožnenie štruktúrovania kódu. Tieto postupy nie sú možné v LLVM IR, pretože LLVM IR je nízkoúrovňový jazyk. BIR následne slúži ako vstup pre konkrétny generátor výsledného jazyka [24].

V zadnej časti spätného prekladača sa nachádzajú optimalizácie, ktorých úlohou je zlepšiť čitateľnosť produkovaného kódu. Ide napríklad o optimalizáciu zjednodušujúcu aritmetické výrazy. Táto optimalizácia nahrádza napríklad súčet dvoch čísel za výsledok tohto súčtu. Ďalej je to optimalizácia konvertujúca bitové posuny na násobenie a delenie. Odstraňovanie prebytočných zátvoriek v aritmetických výrazoch je takisto riešené optimalizáciou nachádzajúcou sa v zadnej časti spätného prekladača. Ďalšou optimalizáciou je optimalizácia odstraňujúca mŕtvy kód. Mŕtvy kód predstavuje taký kód, ktorý sa nikdy nevykoná.

Okrem uvedených optimalizácií sa nachádza v zadnej časti ešte veľké množstvo ďalších optimalizácií [2].

Výstupom zadnej časti je vysokoúrovňový kód. Aktuálne podporované jazyky sú jazyk C a modifikovaná verzia jazyka Python. Tento Python je rozšírený napríklad o ukazovatele a referenčné operátory z jazyka C. Keďže jazyk Python neobsahuje dátový typ pole ani štruktúra, tak sú tieto konštrukcie nahradené inými. Pole je nahradené dátovým typom list a štruktúra pomocou slovníku [24].

Kapitola 4

LLVM IR

Táto kapitola sa venuje popisom niektorých konštrukcií jazyka LLVM IR. Napríklad ide o funkcie, moduly, premenné a pod. Táto časť však nenahrádza dokumentáciu syntaxe týchto konštrukcií. Tá je dostupná na stránkach LLVM frameworku [15]. Miesto toho sa bude nasledujúca časť textu venovať jednoduchým príkladom, ktoré demonštrujú tieto konštrukcie. Rovnako bude obsahovať základné informácie, ktoré sú potrebné v neskorších kapitolách pre pochopenie danej problematiky a ukážok kódov v jazyku LLVM IR.

Programy v jazyku LLVM IR sú tvorené takzvanými modulmi. Moduly obsahujú funkcie, globálne premenné, definície dátových typov a pod [15]. V prípade spätného prekladača vyvíjaného spoločnosťou AVG Technologies je celý program v jednom module [10].

LLVM poskytuje veľké množstvo dátových typov. Tabuľka 4.1 zhrňuje tie najdôležitejšie typy, ktoré budú využívané v príkladoch v neskorších kapitolách.

Tabuľka 4.1: Príklady dátových typov v LLVM IR [15].

Typ	Popis
`i32`	32 bitová celočíselná hodnota
`i64`	64 bitová celočíselná hodnota
`iX`	X bitová celočíselná hodnota
`float`	32 bitová hodnota s pohyblivou rádovou čiarkou
`double`	64 bitová hodnota s pohyblivou rádovou čiarkou
`i32 *`	Ukazovateľ na 32 bitovú celočíselnú hodnotu
`[40 x i32]`	Pole obsahujúce štyridsať 32 bitových celočíselných hodnôt
`{ i32, i32, i32 }`	Štruktúra zložená z troch 32 bitových celočíselných hodnôt

V LLVM IR kóde sa nachádzajú aj funkcie [15]. Napríklad funkcia s menom `func`, návratovým typom `i32` a jedným parametrom `a` typu `i32` vyzerá v kóde LLVM IR ako na príklade 4.1. Na príklade je možné vidieť aj volanie tejto funkcie. Toto volanie je realizované prostredníctvom inštrukcie `call`.

Funkcia je zložená z takzvaných základných blokov. Bloky obsahujú inštrukcie, pričom každý blok musí byť ukončený ukončovacou inštrukciou. Ukončovacie inštrukcie spôsobia pri vyhodnocovaní bloku prechod do iného bloku. Ide napríklad o inštrukcie `br` (skok na iný blok), `ret` (návrat z funkcie) a pod. Každá funkcia ma takzvaný vstupný blok. Tento blok

```
define i32 @func(i32 %a) {
...
  ...
  call i32 @func(i32 3)
  ...
}
```

Obr. 4.1: Príklad zápisu funkcie a jej volania v jazyku LLVM IR.

je len jeden. Ide o prvý vyhodnocovaný blok vo funkcii, v ktorej sa nachádza [15]. Funkcia obsahujúca jeden vstupný blok s názvom `entry` a blokom `bb`, obsahujúcim inštrukciu `ret`, ktorá spôsobí ukončenie vyhodnotenia funkcie a návrat hodnoty 0 je znázornená na ukážke 4.2. Ako je možné vidieť, tak v prípade inštrukcie skoku je adresa skoku uvedená názvom bloku, na ktorý je tento skok vykonaný, pričom pred týmto názvom bloku je uvedený znak "`%`".

```
define i32 @func() {
entry:
  br label %bb

bb:
  ret i32 0
}
```

Obr. 4.2: Príklad obsahujúci základne bloky vo funkcii.

Premenné v jazyku LLVM IR je možné rozdeliť na lokálne a globálne. Rozsah platnosti týchto premenných odpovedá typickým konvenciám ako v iných jazykoch, napríklad jazyku C. Názvy globálnych premenných začínajú znakom "`@`". Napríklad premenná `@glob` predstavuje globálnu premennú s názvom `glob`. Lokálne premenné sa líšia od globálnych tak, že ich pomenovanie začína znakom "`%`". Lokálna premenná s názvom `loc` by bola v jazyku LLVM IR zapísaná ako `%loc` [15].

Uveďme si teraz pre ilustráciu príklad zložitejšieho kódu LLVM IR. Jeden z možných kódov zápisu výpočtu faktoriálu v jazyku LLVM IR je možné vidieť na príklade 4.3. Na riadku 3 dochádza k porovnaniu obsahu hodnoty predanej do funkcie s hodnotou 0. Na základe výsledku tohto porovnania dôjde na riadku 4 k skoku na blok `recurse.end` v prípade, že predávaná hodnota do funkcie je rovná 0. V opačnom prípade dôjde k skoku na blok `recurse`. V bloku `recurse` na riadkoch 7 a 8 je použitá inštrukcia `phi`. Jej detailný popis sa nachádza pri popise optimalizácie, ktorá usporiadava uzly PHI, konkrétne v oddiele `??`. Pre tento príklad však stačí uviesť, že inštrukcia `phi` vyberie hodnotu z množiny určených hodnôt na základe predchádzajúceho bloku, z ktorého došlo k skoku na tento blok. Napríklad pre skok z bloku `entry` to bude hodnota nachádzajúca sa v parametre `X`. Táto inštrukcia je potrebná z dôvodu zachovania SSA formy. Riadok 9 obsahuje inštrukciu pre sčítanie, riadok 10 inštrukciu pre násobenie. Na riadku 11 dochádza opäť k porovnaniu s nulou a na základe toho na riadku 12 dochádza k pokračovaniu výpočtu a skoku na blok `recurse`, alebo k ukončeniu a skoku na blok `recurse.end`. Riadok 16 vykonáva vrátenie výsledku z funkcie. Na uvedenom príklade je možné vidieť, že kód jazyka LLVM IR spĺňa SSA formu. Do každej premennej je hodnota priradená v kóde iba raz a súčasne platí, že vždy došlo k priradeniu hodnoty do premennej pred načítaním hodnoty z tejto premennej.

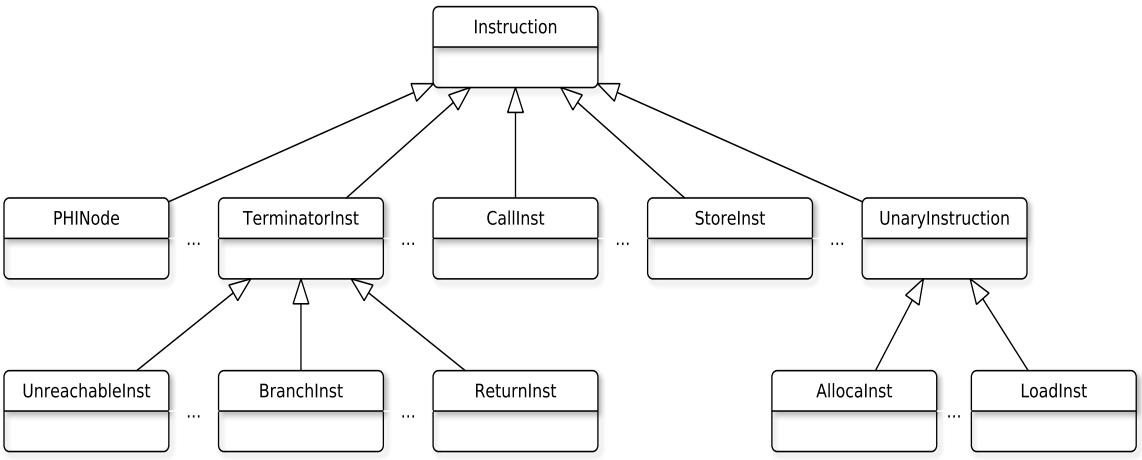
```
1  define i32 @factorial(i32 %X) {
2  entry:
3    %1 = icmp eq i32 %X, 0
4    br i1 %1, label %recurse.end, label %recurse
5
6  recurse:
7    %X2 = phi i32 [%2, %recurse], [%X, %entry]
8    %acc = phi i32 [%3, %recurse], [1, %entry]
9    %2 = add i32 %X2, -1
10   %3 = mul i32 %X2, %acc
11   %4 = icmp eq i32 %2, 0
12   br i1 %4, label %recurse.end, label %recurse
13
14 recurse.end:
15   %res = phi i32 [1, %entry], [%3, %recurse]
16   ret i32 %res
17 }
```

Obr. 4.3: Príklad výpočtu faktoriálu zapísaného v jazyku LLVM IR.

LLVM IR je možné vytvárať programovo. LLVM poskytuje rozhranie v jazyku C++ pre vytváranie LLVM IR kódu. Pre prístup k rozhraniu v jazyku C++ je možné použiť aj iné jazyky, ako napríklad jazyk Python [14].

LLVM obsahuje triedy slúžiace ako abstrakcia rôznych prvkov prechodnej reprezentácie LLVM IR a vďaka ním je možné vytvárať jednotlivé prvky tohto jazyka. Trieda `Module` reprezentuje modul. Funkciu reprezentuje trieda `Function`. Základné bloky predstavuje trieda `BasicBlock`.

Inštrukcie používané v LLVM IR sú reprezentované tiež triedami. Na UML diagrame 4.4 sú znázornené hlavne triedy, ktoré predstavujú využívané inštrukcie pri navrhnutých optimalizáciách. Bázovou triedou pre všetky inštrukcie je trieda `Instruction`. Bázovou triedou pre všetky inštrukcie ukončujúce základný blok je trieda `TerminatorInst`. Trieda `CallInst`



Obr. 4.4: UML diagram tried znázorňujúci vzťahy medzi triedami reprezentujúcich inštrukcie.

predstavuje konkrétnu triedu pre inštrukciu volania funkcie. Trieda `BranchInst` reprezentuje inštrukciu skoku a trieda `ReturnInst` inštrukciu ukončenia toku programu pre danú funkciu. Rovnako konkrétne triedy reprezentujúce inštrukcie sú triedy `PHINode`, `StoreInst`, `LoadInst`, `AllocaInst` a `UnreachableInst`. Bázovou triedou pre unárne inštrukcie je trieda `UnaryInstruction` [17]. Syntax a sémantika inštrukcií využívaných v konkrétnych optimalizáciách bude vysvetlená v kapitolách prislúchajúcim týmto optimalizáciám.

Kapitola 5

Stav spätného prekladača pred zahájením tejto práce

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 6

Návrh optimalizácií

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 7

Implementácia optimalizácií

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 8

Testovanie optimalizácií

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 9

Dosiahnuté výsledky a ich zhodnotenie

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 10

Záver

Hlavným cieľom tejto práce bol presun niektorých optimalizácií zo zadnej časti spätného prekladača do strednej časti. Celkovo bolo navrhnutých päť optimalizácií. Konkrétne ide o optimalizáciu riešiacu usporiadanie PHI uzlov, optimalizáciu odstraňujúcu mŕtve priradenia, optimalizáciu konvertujúcu globálne premenné na lokálne, optimalizáciu odstraňujúcu nedosiahnuteľné funkcie a optimalizáciu odstraňujúcu kód za volaniami funkcií, ktoré sa nevracajú. Súčasťou práce bolo preštudovať dostupné optimalizácie nachádzajúce sa vo frameworku LLVM a posúdiť ich prínos pre spätný prekladač. Pre testovanie navrhnutých optimalizácií boli vytvorené skripty umožňujúce ich testovanie pomocou referenčných testov. Ďalej bolo pre tieto optimalizácie vytvorené množstvo referenčných testov a jednotkových testov.

Nové navrhnuté optimalizácie je možné posúdiť z dvoch hľadísk. Prvým hľadiskom je posúdenie prínosu na čitateľnosť výstupného kódu. Vo väčšine prípadov išlo o presun optimalizácií nachádzajúcich sa v zadnej časti spätného prekladača do strednej časti. Vzhľadom na to, že tieto optimalizácie boli používané v spätnom prekladači a zlepšovali čitateľnosť výstupného kódu, tak ich presunom do strednej časti nedošlo k zhoršeniu tejto čitateľnosti. Samozrejme platí, že nové optimalizácie funkčne pokrývajú tie staré. Dokonca boli niektoré nové optimalizácie v tejto diplomovej práci funkčne rozšírené oproti tým nachádzajúcim sa v zadnej časti. Konkrétne ide o optimalizáciu odstraňujúcu mŕtve priradenia, optimalizáciu konvertujúcu globálne premenné na lokálne a optimalizáciu odstraňujúcu nedosiahnuteľné funkcie. Zároveň bolo vyriešených niekoľko chýb nachádzajúcich sa v optimalizáciách v zadnej časti pri novo navrhnutých optimalizáciách. Konkrétne ide o optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciou odstraňujúcu nedosiahnuteľné funkcie.

Druhým hľadiskom je urýchlenie doby behu spätného prekladača. Nočné testovanie bolo vďaka novým optimalizáciám urýchlené o cca 11 %. V prípade paralelného spúšťania testov na 24 procesoroch ide o zrýchlenie približne 1 hodinu a 45 minút. Pokiaľ by boli testy spúšťané sekvenčne na jednom procesore, tak zrýchlenie by bolo približne 42 hodín.

Cieľom tejto práce bolo urýchliť beh spätného prekladača, konkrétne presunom niektorých optimalizácií zo zadnej časti do strednej časti. Zároveň bolo potrebné nové navrhnuté optimalizácie dostatočne otestovať. Súčasťou práce bolo preštudovať dostupné optimalizácie nachádzajúce sa vo frameworku LLVM a posúdiť ich prínos pre spätný prekladač. Uvedené ciele boli v tejto diplomovej práci úspešne splnené.

Z hľadiska budúceho vývoja je možným rozšírením nových navrhnutých optimalizácií v tejto

práci implementácia analýzy, ktorá rieši problematiku ukazovateľov. Táto analýza by bola prínosná pre optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciu konvertujúcu globálne premenné na lokálne. V týchto optimalizáciách by bola využiteľná pre určenie toho, aké všetky ukazovatele ukazujú na globálnu premennú.

Literatúra

[1] AVG team: Retargetable Decompiler. [online], [cit. 2015-01-13].
URL https://retdec.com/

[2] AVG team: Soupis optimalizací v backendu. Interná wiki projektu zpětného překladače.

[3] backerstreet.com: Reverse Engineering Resources. [online], [cit. 2014-12-23].
URL http://www.backerstreet.com/cg/work.htm

[4] Caetano, L.: Mobile Malware in 2014. [online], Poslední modifikace 2014-03-25 [cit. 2014-12-29].
URL http://blogs.mcafee.com/consumer/mobile-malware-2014

[5] Cipresso, T.: *Software reverse engineering education*. Diplomová práce, San Jose state university, 2009.
URL http://scholarworks.sjsu.edu/etd_theses/3734

[6] DebugMode: Decompilation. [online], Poslední modifikace 2001-10-22 [cit. 2014-12-25].
URL http://www.debugmode.com/dcompile/

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

[8] Gerçek, B.: Jak funguje antivirový program? [online], [cit. 2014-12-23].
URL http://www.symantec.com/region/cz/resources/antivirus.html

[9] Kollár, J.: *Optimalizace v zadní části zpětného překladače*. Bakalářská práce, Brno university of technology, 2013.

[10] Křoustek, J.: *Retargetable Analysis of Machine Code*. Dizertační práce, Brno university of technology, 2015.

[11] Křoustek, J.; Ďurfina, L.; Zemek, P.: Generic Source Code Migration Using Decompilation. In *10th Annual Industrial Simulation Conference (ISC'2012)*, EUROSIS, 2012, ISBN 978-90-77381-71-7, s. 38–42.

[12] Křoustek, J.; Ďurfina, L.; Zemek, P.; aj.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. *International Journal of Security and Its Applications*, 2011: s. 91–106, ISSN 1738-9976.

[13] Křoustek, J.; Ďurfina, L.; Zemek, P.; aj.: Detection and Recovery of Functions and Their Arguments in a Retargetable Decompiler. In *19th Working Conference on Reverse Engineering (WCRE'2012)*, IEEE Computer Society, 2012, ISBN 978-0-7695-4891-3, s. 51–60.

[14] LLVM Project: The LLVM Compiler Infrastructure. [online], [cit. 2015-01-11].
URL http://llvm.org/

[15] LLVM Project: LLVM Language Reference Manual. [online], Poslední modifikace 2014-12-23 [cit. 2014-12-23].
URL http://llvm.org/docs/LangRef.html

[16] LLVM Project: opt – LLVM optimizer. [online], Poslední modifikace 2014-12-23 [cit. 2014-12-23].
URL http://llvm.org/docs/CommandGuide/opt.html

[17] LLVM Project: LLVM API Documentation. [online], Poslední modifikace 2014-12-29 [cit. 2014-12-29].
URL http://llvm.org/docs/doxygen/html/index.html

[18] Oreans Technologies: Themida Features. [online], [cit. 2014-12-25].
URL http://www.oreans.com/themida_features.php

[19] Rouse, M.: Disassemble. [online], Poslední modifikace 2005-09 [cit. 2014-12-29].
URL http://whatis.techtarget.com/definition/disassemble

[20] Russell, K. J.: Obfuscation. [online], Poslední modifikace 2006-10 [cit. 2014-12-29].
URL http://searchsoftwarequality.techtarget.com/definition/obfuscation

[21] TechTerms.com: Malware. [online], [cit. 2014-12-27].
URL http://www.techterms.com/definition/malware

[22] Wikipedia contributors: Reverse engineering. [online], Poslední modifikace 2014-04-19 [cit. 2014-12-21].
URL http://en.wikipedia.org/wiki/Reverse_engineering

[23] Wikipedia contributors: Static single assignment form. [online], Poslední modifikace 2014-12-10 [cit. 2013-12-27].
URL http://en.wikipedia.org/wiki/Static_single_assignment_form

[24] Zemek, P.: *Design of a Language for Unified Code Representation*. Interná technická správa projektu Lissom, 2012.

[25] Zhao, J.; Nagarakatte, S.; Martin, M. M. K.; aj.: Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *12 Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012: s. 427–440.
URL http://acg.cis.upenn.edu/papers/popl12_vellvm.pdf

Dodatok A

Obsah DVD

- Zdrojové súbory, obrázky a Makefile pre vygenerovanie textu práce do elektronickej podoby.
- Text práce v elektronickej podobe.
- Zdrojové kódy implementácie optimalizácií navrhnutých v tejto práci.
- Makefile pre preklad jednotkových testov a optimalizácií vytvorených v tejto diplomovej práci.
- Spustiteľná verzia spätného prekladača.
- Referenčné testy vytvorené v rámci tejto diplomovej práce.
- Skripty pre referenčné testovanie a generovanie referenčných testov, ktoré boli vytvorené v rámci tejto diplomovej práce.
- Zdrojové kódy jednotkových testov, ktoré boli vytvorené v rámci tejto diplomovej práce.
- Súbor README s pokynmi pre spustenie spätného prekladu, spustenie testov a informáciami o adresároch na DVD.
- Skript, ktorý zjednodušuje prácu pri zobrazovaní výsledkov jednotkových a referenčných testov.
- Externé knižnice.
- Rôzne balíčky ako sú gnuarm, pspsdk, clang a iné, potrebné k spätnému prekladu.