VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

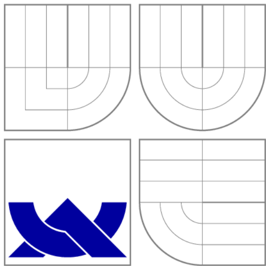
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERICKÁ OBFUSKACE NA ÚROVNI BAJTKÓDU

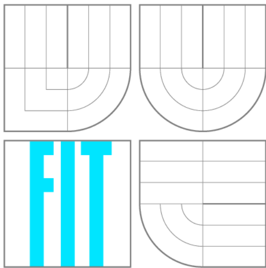
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
SAMUEL KOLLÁT

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERICKÁ OBFUSKACE NA ÚROVNI BAJTKÓDU

GENERIC OBFUSCATION ON THE BYTECODE LEVEL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — SAMUEL KOLLÁT
AUTHOR

VEDOUCÍ PRÁCE — Ing. LUKÁŠ ĎURFINA
SUPERVISOR

BRNO 2013

Abstrakt

V této práci je popsána definice obfuskace a metody její realizace. Následuje popis projektu LLVM a možnosti jeho využití při vytváření obfuskace na úrovni bajtkódu se zaměřením na generickost vzhledem k cílové architektuře. Jádro práce tvoří podrobný návrh metod obfuskace s cílem jejich implementace v zadní části překladače LLVM. Závěrečná sekce se věnuje ověření funkčnosti na různých architekturách pomocí automatizovaných testů.

Abstract

This work contains definition of obfuscation and methods of obfuscation. It is followed by description of LLVM Project and its suitability for obfuscation on the bytecode level for purpose of targeting different architectures. The core of the work is formed by detailed design of obfuscation methods aiming towards their implementation in back-end of LLVM compiler. Closing section is dedicated to verification of implemented functionality on different architectures by automated testing.

Klíčová slova

Obfuskace, deobfuskace, překladač, LLVM, reverzní inženýrství

Keywords

Obfuscation, deobfuscation, compiler, LLVM, reverse engineering

Citace

Samuel Kollát: Generická obfuskace na úrovni bajtkódu, bakalářská práce, Brno, FIT VUT v Brně, 2013

Generická obfuskace na úrovni bajtkódu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Lukáša Ďurfinu. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Samuel Kollát
7. května 2013

Poděkování

Na tomto mieste by som rád poďakoval svojmu vedúcemu Ing. Lukášovi Ďurfinovi za jeho odbornú pomoc a poskytnuté rady behom tvorby tejto práce.

© Samuel Kollát, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Obfuskácia **4**
2.1 Rozdelenie 4
2.1.1 Obfuskácia zdrojového kódu 4
2.1.2 Obfuskácia strojového kódu 4
2.2 Význam obfuskácie 5
2.2.1 Ochrana intelektuálneho vlastníctva 5
2.2.2 Zamedzenie detekcie malware 5
2.2.3 Testovanie spätných prekladačov 6

3 Metódy obfuskácie **7**
3.1 Vkladanie mŕtveho kódu 7
3.2 Substitúcia 7
3.3 Umelé cykly 8
3.4 Premiestnenie kódu 8
3.5 Klonovanie podprogramov 9
3.6 Nepriehľadné predikáty 10

4 Platforma LLVM **11**
4.1 História 11
4.2 Význam 11
4.3 LLVM IR 11
4.4 Výhody pre oblasť obfuskácie 12

5 Návrh **13**
5.1 Vkladanie mŕtveho kódu 13
5.2 Substitúcia 15
5.3 Dekompozícia kódu 16
5.4 Premiestnenie kódu 18
5.5 Klonovanie podprogramov 20
5.6 Nepriehľadné predikáty 22
5.7 Substitúcia pomocou genetického programovania 24

6 Implementácia **28**

7 Testovanie **31**
7.1 Cieľ testovania 31
7.2 Metóda testovania 31
7.3 Realizácia 32
7.4 Výsledky 32

8 Štatistiky **33**

9 Budúci vývoj **36**

10 Záver **38**

A Zdrojové kódy **41**
A.1 Ackermannova funkcia 41
A.2 Cyklický redundantný súčet 41
A.3 Dijkstrov algoritmus 43
A.4 Fouriérova transformácia 45
A.5 Quicksort 50

Kapitola 1

Úvod

Základnou témou tohoto dokumentu je analýza a implementácia generickej obfukácie na úrovni bajtkódu. Obfuskácia sa stala v informačných technológiach mocným nástrojom ako na ochranu intelektuálneho vlastníctva, tak aj na ochranu škodlivého software proti detekcii pomocou klasických nástrojov. Tieto skutočnosti poukazujú na fakt, že je nutné vytvoriť nástroj, ktorý umožní nielen vkladať jednotlivé obfuskácie s cieľom ochrany obsahu aplikácií, ale rovnako vkladať ich s cieľom testovania software, ktorého primárnym cieľom je odhaľovať škodlivé aplikácie. Z tohoto dôvodu je vhodné vytvoriť parametrizovateľnú obfuskáciu, ktorá by slúžila tvorcom nástrovoj pre reverzné inžinierstvo, medzi ktoré patria dekompilátory a deobfuskátory, ako zdroj testovacích dát, na ktorých by bolo možné testovať a overovať ich efektivitu a odolnosť voči jednotlivým obfuskačným metódam. Práca sa zameriava na platformu LLVM [4], pričom samotná obfuskácia je realizovaná ako niekoľko samostatných priechodov v prekladači LLVM. Prekladač LLVM umožňuje nielen efektívne vkladať jednotlivé obfuskačné priechody, ale aj intuitívne realizovať analýzy, ktoré sú potrebné na vkladanie jednotlivých obfuskačných metód.

Najprv bude definovaný pojem obfuskácie a jej ciele. Kapitola 2 postupne prechádza jednotlivými typmi obfuskácií a ich primárnymi využitiami. Budú spomenuté pozitívne aj negatívne aspekty využitia obfuskácie.

Kapitola 3 predstavuje hlavné metódy, ktoré sa používajú na realizáciu obfuskácie. Pri jednotlivých metódach je taktiež podstatné určiť, aká je ich odolnosť voči deobfuskácií.

Ďalej bude v kapitole 4 predstavená platforma LLVM s ohľadom na dôvody jej výberu pre realizáciu obfuskácie. Budú predstavené možnosti práce s LLVM a hlavne spôsob pridania novej funkcionality priamo do prekladača v podobe priechodov nad bajtkódom LLVM. Dôležitým prvkom práce je zameranie sa na generickosť implementácie vzhľadom na cieľovú architektúru. Tento dôvod bude podrobne opísaný v časti zameriavajúcej sa na bajtkód platformy [11] , ktorého kvalitná implementácia je jednou z významých vlastností LLVM.

Návrh jednotlivých obfuskačných priechodov a detailný popis ich funkcionality bude súčasťou kapitoly 5. Implementačné vlastnosti obfuskačných priechodov a aj samotného ich včlenenia do systému LLVM budú zhrnuté v kapitole 6.

Významnou súčasťou vývoja obfuskačných priechodov je testovanie, ktorému je venovaná kapitola 7. Keďže cieľom projektu je vytvoriť generickú obfuskáciu vzhľadom na cieľovú architektúru, je popísané aj testovanie obfuskačných priechodov na rôznych architektúrach. Zaujímavé štatistické údaje sú zhrnuté v kapitole 8.

Na záver sú v kapitole 9 predstavené myšlienky, ktoré by mohli určiť smer budúcemu vývoju projektu, pričom sa nezabúda na samotný budúci vývoj platformy LLVM.

Kapitola 2

Obfuskácia

V tejto časti bude predstavená obfuskácia z hľadiska softvérového inžinierstva, jej delenie, význam a využitie v praxi a taktiež metódy, ktorými je realizovaná.

Obfuskácia (zahmlievanie, zatemnenie) je cielené skrytie významu v komunikácií, čím sa komunikácia stáva náročnejšia na interpretovanie a následné pochopenie [13]. Formálne je obfuskácia definovaná nasledujúcim spôsobom [11].

Definícia 1 *Nech* $P \xrightarrow{T} P'$ *je transformácia zdrojového programu P na cieľový program* P'. $P \xrightarrow{T} P'$ *je obfuskačná transformácia, ak P a* P' *majú rovnaké pozorovateľné chovanie. Presnejšie, aby* $P \xrightarrow{T} P'$ *bola legálna obfuskačná transformácia musia byť splnené nasledujúce podmienky:*

1. *Ak sa P neukončí alebo sa ukončí s chybou, tak sa* P' *môže alebo nemusí ukončiť.*
2. *Inak sa* P' *musí ukončiť a vyprodukovať rovnaký výsledok ako P.*

Z tejto definície vyplýva, že P a P´ nemusia byť úplne ekvivalentné. Vo väčšine prípadov je P´ pomalšie, alebo využíva viac pamäte ako P [11].

2.1 Rozdelenie

Základným delením obfuskácie je delenie podľa typu kódu, nad ktorým je obfuskácia realizovaná. Týmto spôsobom sa obfuskácia delí na obfuskáciu zdrojového kódu a obfuskáciu binárneho (strojového) kódu.

2.1.1 Obfuskácia zdrojového kódu

Cieľom obfuskácie zdrojového kódu je lexikálna transformácia realizovaná nad samotným zdrojovým kódom s cieľom zhoršiť jeho čitateľnosť a porozumenie. Je to súbor jednocestných transformácií, ktoré realizujú odstránenie formátovania, komentárov, zmenu mien premenných a zmenu reprezentácie reťazcov, napríklad pomocou hexadecimálnych hodnôt [11].

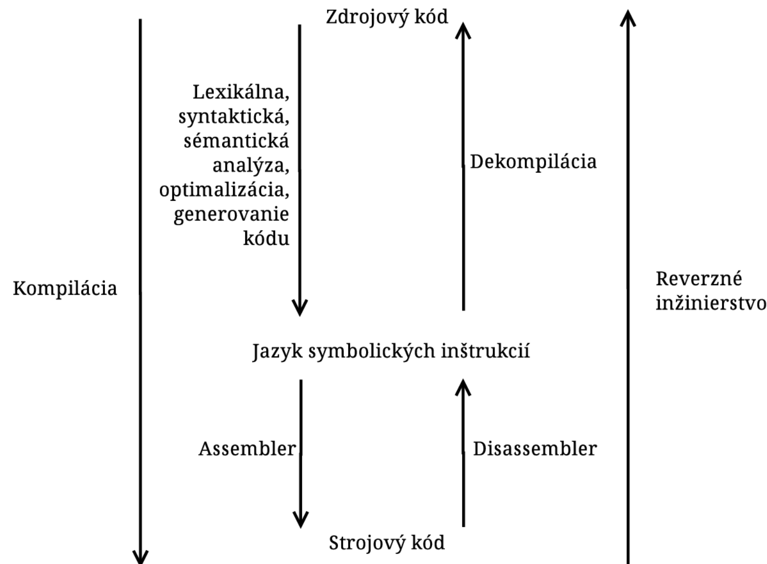
2.1.2 Obfuskácia strojového kódu

Obfuskácia nad binárnym kódom je základnou témou tejto práce a jej podrobnejší popis sa nachádza v nasledujúcich kapitolách. Ide o transformáciu formálne definovanú vyššie [4], ktorá spôsobí modifikáciu pôvodného programu, pričom sémantika úspešne realizovaného programu ostáva nemenná oproti programu pôvodnému.

2.2 Význam obfuskácie

Hlavným významom obfuskácie nad zdrojovým kódom je znemožnenie porozumenia obsahu programu pre človeka analyzujúceho takýto kód [12]. Keďže obfuskácia nad binármym kódom má omnoho širšie využitie, je táto podkapitola zameraná na tento druh obfuskácie. Od tohoto miesta budú termíny obfuskácia a obfuskácia nad binárnym (strojovým) kódom ekvivalentné.

Dôležitými pojmami spájajúcimi sa s obfuskáciou sú kompilácia, dekompilácia a reverzné inžinierstvo. Ich vzťah je zobrazený v grafe 2.1.



Obrázek 2.1: Vzťah kompilácie, dekompilácie a reverzného inžinierstva [14].

2.2.1 Ochrana intelektuálneho vlastníctva

Mnohé spoločnosti vyvýjajúce proprietárny softvér využívajú metódu obfuskácie ako lacný a efektívny spôsob ochrany ich intelektuálneho vlastníctva [10]. Základom je ochrana dátových štruktúr a algoritmov pomocou transformácie popísanej v [4]. Táto metóda poskytuje ochranu proti softvérovému pirátctvu, avšak nie je úplne bezpečná, pretože zatiaľ neexistuje obfuskácia, ktorá by bola úplne odolná voči deobfuskácií [10], teda spätnej trasformácií obfuskovaného kódu na kód neobfuskovaný.

2.2.2 Zamedzenie detekcie malware

Obfuskácia sa stala jedným z hlavných nástrojov tvorcov škodlivého softvéru na ochranu ich programov pred detekciou pomocou antivírových nástrojov [10]. Technika obfuskácie umožňuje malware transformovať svoj kód do nových generácií počas svojho šírenia. Takýto malware má označenie metamorfný [17]. Touto metódou sťažuje prácu antivírovým softvérom, protože každá generácia je odlišná od predchodzej, čím sa tradičné metódy vyhľadávania škodlivého softvéru, akou je napríklad vyhľadávanie pomocou signatúr v databázy vírusov, stávajú neúčinnými [17].

2.2.3 Testovanie spätných prekladačov

Ako už bolo spomenuté, obfuskácia je dôležitým prostriedkom pre zamedzenie získania obsahu programu, teda jeho dátových štruktúr a algoritmov, reverznými inžiniermi. Na druhú stranu je taktiež potrebné vyvýjať nástroje, ktoré umožňujú sofistikovane realizovať reverzné inžinierstvo nad obfuskovanými programami a to hlavne vzhľadom na detekciu škodlivého softvéru. Aby bolo možné efektívne a správne realizovať reverzné inžinierstvo nad už existujúcim obfuskovaným softvérom, je nutné nástroje, dekompilery, realizujúce túto činnosť, otestovať a odladiť. Na túto činnosť je vhodné vyvinúť obfuskátor, ktorý umožňuje realizovať jednotlivé druhy obfuskácií a jeho činnosť je možné riadiť pomocou určitých parametrov.

Kapitola 3

Metódy obfuskácie

V tejto kapitole budú postupne opísané jednotlivé najpoužívanejšie metódy obfuskácie. Predstavované budú v poradí od jednoduchšie realizovateľných a jednoduchšie deobfuskovateľných až po zložitejšie, poskytujúce lepšiu efektivitu pre vyššie zmienené účely.

3.1 Vkladanie mŕtveho kódu

Vkladanie mŕtveho kódu je technika, ktorá sa zameriava na zmenu vzhľadu výsledného kódu pomocou vkladania neúčinných inštrukcií, prípadne sekvencií inštrukcií do programu [17]. Môže sa jednať o rôzne druhy inštrukcií od jednoduchých ako je NOP, až po bloky realizujúce zmenu obsahu premenných a ich následnú zmenu späť do pôvodného stavu [10]. Takúto obfuskáciu znázorňuje nasledujúci úsek kódu 3.1.

...
a = 5
b = 4   // *stav S*
c = a + b
...
	...
	a = 5
	b = 4 // *stav S*
	swap(a,b) // *stav S´*
	swap(a,b) // *stav S*
	c = a + b
	...

Obrázek 3.1: Vloženie úseku mŕtveho kódu.

3.2 Substitúcia

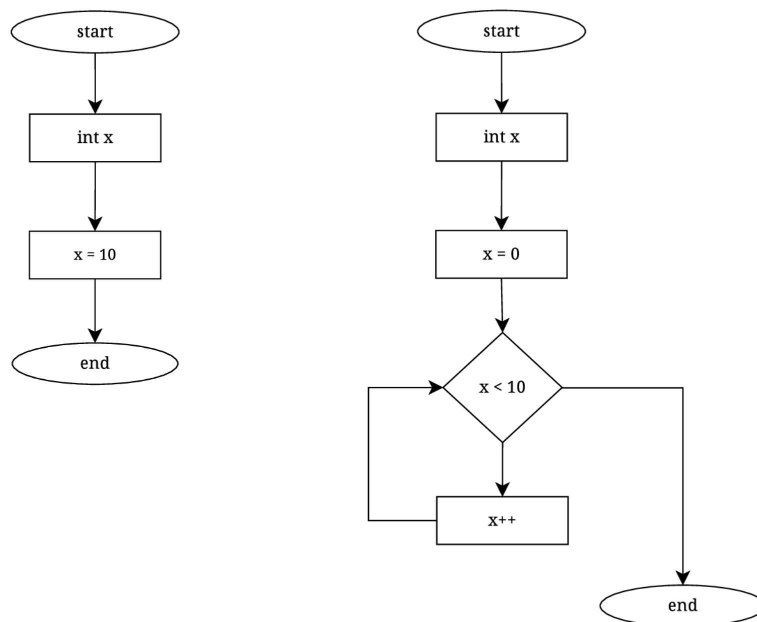
Základnou myšlienkou substitúcie je nahradiť pôvodné inštrukcie vyskytujúce sa v programe inými inštrukciami, prípadne blokom inštrukcií, ktoré sú sémanticky ekvivalentné s pôvodnými [17]. Táto metóda je náročnejšia na deobfuskáciu, a to hlavne v prípade, že nie je známa použitá množina ekvivalentných inštrukcií [10]. Z pohľadu obfuskácie je dôležitá substitúcia matematických výrazov inými matematickými výrazmi, prípadne substitúcia logických výrazov inými ekvivalentnými výrazmi. Ako príklad môže slúžiť nasledujúca tabuľka 3.2 ekvivalentých výrazov.

$x = 0$	$x = x \oplus x$
$x = 0$	$x = x - x$
$x = y - z$	$x = -z$ $x = y + x$
$x == 0$	$(x \&\& x) == 0$
$x == 0$	$(x \parallel x) == 0$
$x == 0$	$(x \oplus 0) == 0$

Obrázek 3.2: Tabuľka ekvivalentných výrazov.

3.3 Umelé cykly

Umelé cykly sú jednou z variánt substitúcie inštrukcií. Jedná sa o vloženie sekvencie inštrukcií realizujúcich cyklus na miesto, kde pôvodne tento cyklus potrebný nebol. Ako príklad môže slúžiť diagram 3.3, ktorý znázorňuje inicializáciu kladnej celočíselnej premennej v pôvodnom programe a v obfuskovanom programe.



Obrázek 3.3: Inicializácia kladnej celočíselnej premennej.

3.4 Premiestnenie kódu

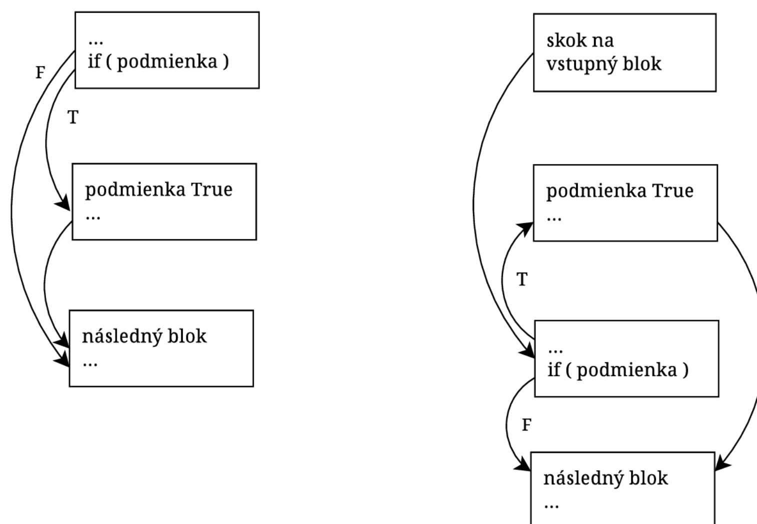
Ďalšou metódou obfuskácie je preskladanie jednotlivých častí programového kódu bez toho, aby to malo dopad na výsledné chovanie programu 3.4. Táto metóda najprv preskladá programový kód a následne pomocou nepodmienených skokov tieto časti prepojí, aby sa zachoval pôvodný sled vykonávania programu [17].

Ďalšou možnosťou je premiestnenie blokov kódu, ktoré už medzi sebou obsahujú väzby v podobe podmienených alebo nepodmienených skokov. Tento princíp je znázornený na

inštrukcia_1	inštrukcia_1
inštrukcia_2	jmp_návestie_1
inštrukcia_3	návestie_2:
inštrukcia_4	inštrukcia_4
inštrukcia_5	inštrukcia_5
inštrukcia_6	jmp_návestie_3
	návestie_1:
	inštrukcia_2
	inštrukcia_3
	jmp_návestie_2
	návestie_3:
	inštrukcia_6

Obrázek 3.4: Premiestnenie inštrukcií a ich prepojenie pomocou nepodmienených skokov.

diagrame 3.5 znázorňujúcom tok vykonávania programu v prípade neúplnej podmienky. Bloky kódu sú znázornené obdĺžnikmi a šípka zobrazuje inštrukciu skoku. Táto obfuskácia taktiež vyžaduje vloženie nového bloku, ktorý bude obsahovať skok na pôvodne prvý blok. Obsahom tohoto bloku môže byť taktiež časť inštrukcií z bloku obsahujúceho samotnú podmienku, čím sa dosiahne výraznejšia zmena obsahu jednotlivých blokov.



Obrázek 3.5: Premiestnenie blokov.

3.5 Klonovanie podprogramov

Ďalšou metódou, ktorá sťaží deobfuskáciu a následnú analýzu zdrojového kódu, je klonovanie podprogramov. Táto metóda je založená na vytvorení viacerých kópií jednej funkcie, na ktoré sú následne aplikované rôzne druhy obfuskácií [11]. Toto spôsobí, že výsledné funkcie sa javia ako odlišné nielen ich vzhľadom, ale aj samotnou sémantikou. Nutný krokom je taktiež pri každom volaní pôvodnej funkcie určiť, ktorá z naklonovaných funkcií sa na danom mieste použije.

3.6 Nepriehľadné predikáty

Nepriehľadné (nejasné, temné) predikáty sú v [10] definované následovne:

Definícia 2 *Nepriehľadné predikáty a premenné sú konštrukcie, ktorých hodnoty sú známe obfuskátoru, ale sú náročné na odvodenie pre deobfuskátor.*

Definícia v [11] dopĺňa:

Definícia 3 *Predikát P je nepriehľadný v bode p, ak jeho výsledok je známy v dobe obfuskácie. Píšeme P_p^F (P_p^T), ak sa P vždy vyhodnotí ako False (True), a $P_p^?$, ak sa P niekedy vyhodnotí ako True a niekedy ako False.*

Tabuľka 3.6 ponúka ukážku niekoľkých jednoduchých nepriehľadných predikátov a ich stálych pravdivostných hodnôt.

Predikát	**Hodnota**
x && 0	P_p^F
$x \parallel 1$	P_p^T
$x \oplus x$	P_p^F
$(x^2 * (x+1)^2)$ mod $4 == 0$	P_p^T

Obrázek 3.6: Ukážka nepriehľadných predikátov.

Kapitola 4

Platforma LLVM

LLVM (Low Level Virtual Machine) Projekt je infraštruktúra prekladača, ktorá je navrhnutá pre optimalizácie programov v čase kompilácie, zostavovania a behu [4]. Programovacím jazykom použitým pre jeho vývoj je jazyk C++. LLVM je vytvorený ako modulárny prekladač spojený s nástrojmi a utilitami, ktoré možno navzájom reťazovo prepájať.

4.1 História

Projekt LLVM vznikol ako výzkumný projekt v roku 2000 na Univerzite v Illinois [9]. Riaditeľmi projektu boli Vikram Adve a Chris Lattner. Cieľom bolo vytvoriť moderný prekladač, ktorý je založený na stratégií prekladania pomocou SSA (static single assignment form) formy a umožňuje ako statickú, tak aj dynamickú kompiláciu. Počiatočné vydanie vyšlo v roku 2003 pod University of Illinois/NCSA Open Source License [4]. V súčasnej dobe je poslednou stabilnou verziu prekladača verzia 3.2 .

4.2 Význam

LLVM má oficiálne plnú podporu pre programovacie jazyky C a C++ [2]. Podpora pre tieto jazyky je zaistená pomocou front-end prekladača Clang [1]. Keďže sa systém LLVM stal obľúbeným, vzniklo aj mnoho iných front-end prekladačov pre LLVM podporujúcich jazyky ako Objective-C, Fortran, Ada, Haskell, Python, Ruby a iné [4]. Ďalšou výhodou prekladača LLVM je podpora generovania strojového kódu pre veľkú škálu architektúr, medzi ktoré partia napríklad X86, X86-64, PowerPC, ARM, SPARC, MIPS, SystemZ, Xcore a ďalšie [5]. Jedným z cieľov projektu LLVM bolo taktiež vytvoriť podporu pre jednoduché vytváranie optimalizačných priechodov nad kompilovaným kódom. Hlavnou súčasťou, ktorá umožnila realizáciu tohoto cieľa je vhodne špecifikovaná reprezentácia kódu využívaná počas prekladu, nazývaná LLVM IR (intermediate representation) [4]. Táto súčasť prekladača bude bližšie opísaná v [11]. Súčasťou prekladača je taktiež manažér priechodov, ktorý automatický usporiadava priechody vzhľadom na ich vzájomné závislosti [5].

4.3 LLVM IR

Reprezentácia kódu LLVM (IR) je vytvorená na využívanie v troch formách [6]:

- Dátové štruktúry využívané prekladačom počas prekladu

- Bajtkód umožňujúci ukladanie na disk [3]
- Jazyk symbolických inštrukcií čitateľný človekom

Tieto tri formy sú navzájom ekvivalentné [6].

Jednou z najdôležitejších vlastností LLVM IR je typový systém. Silné typovanie umožňuje realizovať analýzy, transformácie a optimalizácie, ktoré by nemohli byť realizovateľné nad trojadresným kódom [7]. Typový systém sa skladá z troch hlavných častí [8] :

- Primitívne typy (celé číslo, desatinné číslo, ukazovateľ, ...)
- Odvodené typy (štruktúra, pole, vektor, ...)
- Mechanizmus doprednej deklarácie, ktorý umožňuje reprezentovať pomenované štrukturované typy, ktoré ešte nemajú definované telo.

Tým, že samotná reprezentácia kódu obsahuje typovú informáciu, je možné realizovať optimalizácie priamo nad touto reprezentáciou bez nutnosti uskutočnenia ďalších analýz [6].

4.4 Výhody pre oblasť obfuskácie

V predchádzajúcich sekciách bolo opísané ako sú vlastnosti systému LLVM vhodné na realizáciu analýz, transformácií a optimalizácií pri preklade zdrojového kódu na kód strojový. Práve schopnosť jednoducho vytvárať optimalizačné priechody nad LLVM IR je dôvodom, prečo využiť túto platformu na tvorbu obfuskácií. Obfuskáciu je možné považovať za optimalizačný priechod, ktorý má za úlohu uplatniť metódy opísané v [7].

Kapitola 5

Návrh

Obsahom tejto kapitoly bude podrobný popis návrhu priechodov pre zadnú časť prekladača LLVM a zároveň budú uvedené spôsoby implementácie jednotlivých priechodov.

Každá z nasledujúcich sekcií popisuje samostatný priechod v zadnej časti prekladača LLVM. Z pohľadu kapitoly 2 boli do jedného priechodu zlúčené metódy substitúcie 3.2 a vkladanie umelých cyklov 3.3. Je to z dôvodu, že vkladanie umelých cyklov je možné považovať za substitúciu sekvencie za iteráciu. Týmto sa obfuskačný priechod stáva robustnejší, pretože neobsahuje len substitúcie matematických výrazov, ale aj substitúcie riadiacich konštrukcií. Ďalšou zmenou je rozdelenie priechodu premiestňujúceho programový kód 3.4 do dvoch samostatných priechodov. Prvý priechod realizuje rozdelenie vybraných blokov kódu na menšie celky 5.3 a druhý priechod má na starosti samotné premiestnenie blokov 5.4. Takouto metódou sa dosiahne možnosti vhodnejšej parametrizácie obfuskácie, čo je vhodné hlavne pri testovaní nástrojov pre reverzné inžinierstvo.

V niektorých prípadoch budú pri jednotlivých obfuskáciach uvedené pravdepodobnosti, s akými dochádza k vloženiu konkrétnej obfuskácie. Tieto pravdepodobnosti sú modifikovateľné úpravou programového kódu.

Taktiež budú zmienené vlastnosti obfuskačných priechodov voči technikám reverzného inžinierstva, prípadne vhodnosť kombinácie jednotlivých vybraných obfuskačných priechodov. Aj keď boli jednotlivé obfuskačné priechody narhnuté ako samostatné celky pre potreby testovania nástrojov reverzného inžinierstva, sú v praxi tieto metódy kombinované a používané spoločne, aby sa dosiahlo čo najvyššej efektivity. A to ako z pohľadu ochrany intelektuálneho vlastníctva, tak aj z pohľadu ochrany malware pred antivírovými nástrojmi.

5.1 Vkladanie mŕtveho kódu

Ako uvádza sekcia 3.1, vkladaním mŕtveho kódu sa do programového kódu dostávajú inštrukcie alebo sekvencie inštrukcií, ktoré nemajú dopad na stav vykonávaného programu. Z tohoto dôvodu je pre tento priechod potrebné nájsť výrazy, prípadne postupy, ktoré budú v každej situácií neutrálne voči vykonávanej aplikácií.

Spomínaná sekcia taktiež uvádza, že základným spôsobom ako vložiť do programového kódu mŕtvy kód je použitie inštrukcie NOP, ktorá je súčasťou jazyka symbolických inštrukcií. V priechode pre prekladač LLVM nie je možné realizovať vloženie tejto inštrukcie, pretože táto inštrukcia sa nenachádza v sade inštrukcií, ktoré sú podporované systémom LLVM. Dôvodom je, že vnútorná reprezentácia kódu v prekladači LLVM pracuje na vyššej úrovni abstrakcie ako jazyk symbolických inštrukcií. Inštrukcia NOP avšak nepredstavuje

jediné riešenie ako vložiť do programu mŕtvy kód.

Jednou z možností je využitie neutrálnych prvkov jednotlivých operácií na množine celých čísiel, prípadne neutrality prvkov v Boolovej algebre. Princípy a výrazy implementované v obfuskačnom priechode sú zachytené v tabuľke 5.1.

$x + 0$
$x - 0$
$x * 1$
$x \parallel 0$
$x \oplus 0$
$x \&\& (-1)$

Obrázek 5.1: Implementové výrazy v priechode vkladania mŕtveho kódu.

Ďalší postup využitý v tomto obfuskačnom priechode je dvojnásobné vymenenie obsahu dvoch premenných. Pre túto operáciu boli vybrané dva algoritmy výmeny obsahu dvoch premenných, a to algorimus využívajúci operáciu exkluzívneho logického súčtu 5.1 a algoritmus využívajúci operáciu odčítania 5.2.

$$\begin{aligned} x &= x \oplus y \\ y &= x \oplus y \\ x &= x \oplus y \end{aligned} \tag{5.1}$$

$$\begin{aligned} x &= x + y \\ y &= x - y \\ x &= x - y \end{aligned} \tag{5.2}$$

Prvým krokom pri tejto výmene je nájdenie dvoch vhodných celočíslených premenných v bloku. V prípade, že sa pristúpi k vykonaniu tejto operácie, je pre každú z dvoch výmien náhodne vybraný jeden z algorimov, ktorý je následne aplikovaný.

Ďalším prvkom, ktorý je nutné zvážiť pri vkladaní mŕtveho kódu je miesto v programovom kóde, na ktoré budú jednotlivé inštrukcie mŕtveho kódu vložené. V prvom rade je nutné nájsť alokovanú premennú, prípade dvojicu premenných pri dvojitej výmene ich obsahu, ktorá je vhodná na to, aby bol pomocou nej vložený do programu mŕtvy kód. To znamená, že okrem toho, aby bola premenná v danom mieste programu platná, musí odpovedať aj svojim dátovým typom. Z toho dôvodu sú vždy inštrukcie mŕveho kódu pracujúce nad určitou premennou vkladané k skupinám pôvodných operácií nad touto premennou. Týmto sa dosiahne včlenenie mŕtveho kódu do pôvodných sekcií inštrukcií.

Tabuľka 5.2 uvádza pravdepodobnosti jednotlivých obfuskácií.

Nasledujúci úryvok vnútornej reprezentáciu kódu LLVM (LLVM IR) 5.3 ukazuje, že v skutočnosti nedochádza k zmene stavu vykonávaného programu. V prvom kroku sa načíta obsah vybranej premennej do registra a následne sa vykoná operácia násobenia, pričom sa využíva neutrality prvku 1 voči tejto operácií na množine celých čísiel. V poslednom príkaze dochádza k uloženiu výsledku operácie do pôvodne premennej.

Vkladanie mŕtveho kódu patrí k ľahšie odhaliteľným metódam obfuskácie. Táto metóda je nenáročná na deobfuskáciu, a to hlavne v prídade, že sa používajú jednoduché matematické výrazy, medzi ktoré patria napríklad výrazy z tabuľky 5.2. Je však nutné brať do

Každý z aritmetických a logických výrazov	14.3%
Dvojnásobná výmena obsahu premenných	66.6%

Obrázek 5.2: Pravdepodobnosti jednotlivých obfuskácií.

```
%load_dci = load i32* %x
%obf_dci = mul i32 %load_dci, 1
store i32 %obf_dci, i32* %x
```

Obrázek 5.3: Vložený mŕtvy kód

úvahy, že ak nástroj realizujúci reverzné inžinierstvo nebude počítať s tým, že nad aplikáciou bola realizovaná obfuskácia, teda nebude obsahovať postupy a metódy na jej odstránenie, môže aj takáto jednoduchá obfuskácia sťažiť spätnú analýzu programu.

5.2 Substitúcia

Substitúcie sú založené na slovníku sémanticky ekvivalentných výrazov, prípadne blokov kódu, ako bolo uvedené v sekcii 3.2. Ako prvý krok pri realizácií substitúcie je nájdenie vhodného výrazu, ktorý môže byť na základe slovníka substituovaný. Daný výraz musí okrem zhodnej operácie, prípadne počtu argumentov, vyhovovať aj svojim dátovým typom. Je taktiež nutné zohľadniť bitovú šírku jednotlivých operandov, prípadne výsledku, pretože typový systém prekladača LLVM umožňuje vykonať operáciu, iba ak sa operandy zhodujú v type a zároveň v bitovej šírke.

Tabuľka 5.4 zachytáva implementovaný slovník. Celé číslo je značené $< int >$. Pre kladné celé číslo je využité označenie $< int^+ >$, pre záporné celé číslo $< int^- >$ a číslo v plávajúcej rádovej čiarke je označené ako $< float >$. Funkcia $rand()$ vygeneruje náhodné číslo v rozsahu $< 0, 100 >$. Posledný stĺpec značí pravdepodobnosť, s akou dôjde k realizácií obfuskácie v prípade, že sa nájde vhodný výraz na jej vykonanie.

Prvé dva riadky tabuľky vkladajú do obfuskovaného programu naviac iba jednu inštrukciu, teda nárast programu je relatívne minimálny. Z pohľadu ďalších piatich inštrukcií ide o značnejší nárast programového kódu, pretože dochádza k vloženiu sekvencie inštrukcií a v prípade substitúcií, ktoré vkladajú cyklus, dochádza aj k nárastu počtu základných blokov kódu. Posledné tri substitúcie slúžia na obfuskáciu booleovských operácií, čo sa najviac prejaví vrámci podmienok cyklov a skokov.

Ako príklad kódu v LLVM IR bude slúžiť obfuskácia výrazu $x = -5$. Neobfuskovaná verzia tohoto výrazu má LLVM IR tvar 5.5.

V obfuskovanej verzii ide o nárast kódu, pričom sa zároveň vkladá iterácia, ktorá v pôvodnom kóde nebola potrebná. Celý tvar LLVM IR je v ukážke 5.6.

Z úryvku kódu je si taktiež možné všimnúť, že pôvodná inštrukcia, ktorá vkladala konštantu do premennej už v obfuskovanom kóde nie je prítomná. Na rozdiel od priechodu vkladajúceho mŕtvy kód nie je pri priechode realizujúcom substitúcie nutné zaoberať sa problémom vyhľadania vhodného miesta pre vloženie nových inštrukcií, pretože tie sú vložené priamo na miesto obfuskovanej inštrukcie.

Pôvodný výraz	**Obfuskácia**	**Pravdepodobnosť**
$x = 0$	$x = x \oplus x$	33.3%
$x = 0$	$x = x - x$	33.3%
$x =< int^+ >$	$x = 0;$ $while(x << int^+ >)$ $\{x++;\}$	80%
$x =< int^- >$	$x = 0;$ $while(x >< int^- >)$ $\{x--;\}$	80%
$x =< float >$	$x = rand(0, 100);$ $x = \frac{(x-<float>-x)}{(x-1-x)};$	80%
$x = y - z$	$x = -z;$ $x = y + x;$	80%
$x =< int > *z$	$x = 0;$ $tmp =< int >;$ $while(tmp > 0)$ $\{x = x + z; tmp--;\}$	80%
$x == 0$	$(x \&\& x) == 0$	25%
$x == 0$	$(x \parallel x) == 0$	25%
$x == 0$	$(x \oplus x) == 0$	25%

Obrázek 5.4: Implementované výrazy v priechode substitúcií.

```
store i32 -5, i32* %x, align 4
```

Obrázek 5.5: Neobfuskovaná verzia $x = -5$

5.3 Dekompozícia kódu

V rámci LLVM IR je každá funkcia rozdelená do základných blokov (basic blocks). Základný blok je sekvencia príkazov, pričom príkazy sú v tomto bloku vykonané vždy všetky [15]. Je taktiež zaručené, že príkazy sa v tomto bloku vykonajú vždy v tom poradí, v akom za sebou nasledujú v zápise programového kódu daného bloku. Každý základný blok, označovaný aj iba ako blok, má svoj vstupný a výstupný bod. V prípade LLVM je vstupným bodom ľubovoľná inštrukcia nachádzajúca sa ako prvá za návestím daného bloku. Výstupnou inštrukciou môže byť iba vybraná podmnožina inštrukcií. Medzi základné patria príkazy skoku, selekcie, alebo návratu z funkcie. Je nutné dodať, že každý základný blok má z hľadiska vnútornej reprezentácie kódu LLVM svoje vlastné unikátne návestie.

Priechod realizujúci dekompozíciu kódu má za úlohu vybrať vhodne veľké základné bloky a rozdeliť ich na menšie bloky, pričom každý takto vytvorený blok si zachováva vlastnosti základného bloku. Algoritmus, podľa ktorého tento prechod v zadnej časti prekladača LLVM pracuje je zachytený v pseudokóde 5.7. Algoritmus obsahuje konštantu s hodnotou 1.5, ktorá

```
    store i32 0, i32* %x
    br label %store_cond
  store_cond:
    %load_op1 = load i32* %x
    %0 = icmp sgt i32 %load_op1, -5
    br i1 %0, label %store_body, label %store_end
  store_body:
    %load_op2 = load i32* %x
    %obs_dec = sub i32 %load_op2, 1
    store i32 %obs_dec, i32* %x
    br label %store_cond
  store_end:
```

Obrázek 5.6: Obfuskovaná verzia $x = -5$

určuje, že na dekompozíciu budú vybrané iba tie bloky, ktoré svojou veľkosťou prevyšujú priemernú veľkosť bloku o jeden a pol násobok. Týmto sa zaistí, aby neboli bloky vzniknuté dekompizíciou príliš malé, teda aby sa bloky obsahujúce menší počet inštrukcií ďalej nedelili.

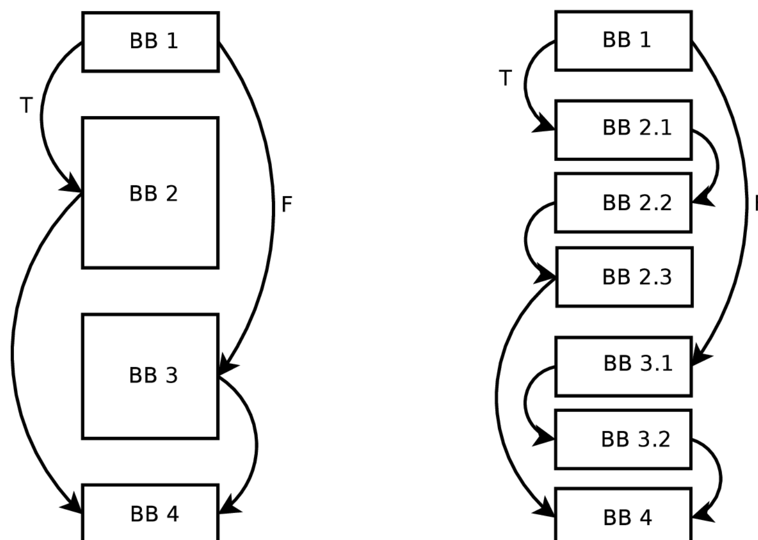
Keďže sa realizuje iba dekompozícia pôvodného bloku, tak všetky novovzniknuté bloky z dekomponovaného bloku nasledujú za sebou. Samotné dodržanie pôvodnej sémantiky programu, teda správny tok programu medzi novými blokmi, je zaistený pomocou inštrukcií nepodmieneného skoku, ktoré prepájajú jednotlivé dekomponované bloky, ako je možné vidieť na schéme 5.8 toku programu.

```
// Pseudokód algoritmu dekompozície kódu
BB = []
MIN = veľkosť najmenšieho bloku
MAX = veľkosť najväčšieho bloku
AVG = priemerná veľkosť bloku // Vrámci všetkých blokov funkcie
for ( jednotlivé bloky ) {
        if ( veľkosť bloku >= 1.5 * AVG )
                BB += blok
}
// Priemerná veľkosť blokov nezahrnutých v poli BB
CENTER = priemerná veľkosť bloku bez BB
for( jednotlivé bloky v BB ) {
        // Veľkosť žiadneho bloku neklesne pod MIN
        while ( veľkosť bloku - MIN > CENTER ) {
                // Vlastný generátor náhodných čísiel. Väčšia pravdepodobnosť generovania
                    čísiel v okolí CENTER. Generuje čísla v intervale <MIN, veľkosť bloku -
                    MIN>
                Odstráň náhodný počet inštrukcií
        }
}
```

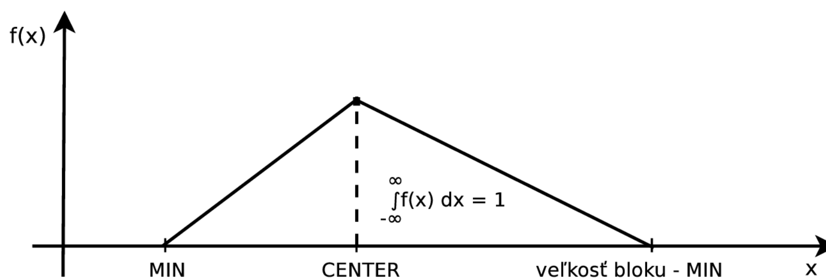
Obrázek 5.7: Algoritmus dekompozície kódu.

Veľkosť nových blokov je náhodne generovaná pomocou generátora pseoudonáhodných čísiel vytvoreného pre tento priechod. Funkcia hustoty pravdepodobnosti je znázornená na grafe 5.9, pričom graf dodržiava označenie definované v algoritme 5.7. Veľkosť nových blokov sa bude pohybovať v okolí priemernej veľkosti blokov, ktoré nebudú dekomponované.

Prepojenie novovytvorených blokov na okolité základné bloky je realizované implicitne,



Obrázek 5.8: Dekompozícia blokov kódu.



Obrázek 5.9: Funkcia hustoty pravdepodobnosti generátora pseudonáhodných čísiel.

pretože vstupný bod pôvodného bloku sa zachová v prvom dekomponovanom bloku a pôvodný výstupý bod sa zachová v poslednom dekompovovanom bloku.

Obfuskácia pomocou metódy dekompozície kódu je z pohľadu reverzného inžinierstva nenáročná na deobfuskáciu. Je to z dôvodu, že jednotlivé dekomponované bloky nasledujú za sebou, teda jednou z metód, ktorá by mohla realizovať deobfuskáciu je sekvenčné prechádzanie blokmi a v prípade, že dva bloky nasledujú za sebou a prvý obsahuje nepodmienený skok na druhý, pričom musí byť splnená podmienka, že neexistuje iný skok na tento blok, tak je možné tieto dva bloky zlúčiť. Aby sa zamedzilo takémuto postupu je vhodné tento priechod kombinovať s priechodom premiestnenia kódu, ktorý bude popísaný v nasledujúcej sekcii.

5.4 Premiestnenie kódu

Priechod realizujúci premiestnenie kódu umožňuje preskladať medzi sebou jednotlivé základné bloky vrámci jednej funkcie. Ako pri každej obfuckácií, aj v tejto obfuskácií je použitý mechanizmus generovania pseudonáhodných čísiel. Rozhodnutie o výslednom poradí blokov je pre každý beh obfuskácie úplne náhodné.

Algoritmus, podľa ktorého sa riadi priechod v zadnej časti prekladača LLVM, je za-

chytený v 5.10. V analytickej fázy algoritmu sa vytvorí reprezentácia základných blokov programu, ktorá sa následne náhodne premieša, čo určí výsledné poradie základných blokov v programovom kóde. Samotné premiestnenie sa realizuje po dvojiciach. Vyberú sa prvé dva bloky z poľa základných blokov a navzájom sa prekopírujú inštrukcie, z ktorých sa skladajú. Pri klonovaní dochádza k modifikácií inštrukcií, konktrétne odkazov na inštrukcie vrámci funkcie. Tým pádom je nutné modifikovať všetky operandy, ktoré odkazovali na pôvodnú inštruciu tak, aby odkazovali na novú, premiestnenú inštrukciu. Rovnako sa menia aj odkazy na alokované premenné vrámci blokov, čo je rovnako nutné zohľadniť.

```
// Pseudokód algoritmu premiestnenia kódu
for ( jednotlivé funkcie ) {
        Zisti počet blokov a vytvor ich reprezentáciu pomocou poľa
        Náhodne premiešaj pole
        while ( veľkosť pola > 1 ) { // Pole musí obsahovať aspoň dva prvky
                Vyber prvé dva bloky z poľa
                Premiestni ich medzi sebou
                Ak je to nutné, uprav jednotlivé alokované premenné // Zaistí správne
                    referencie na alokované premenné
        }
        Modifikuj príkazy vetvenia aby zodpovedali správnemu vykonávaniu programu
        Modifikuj PHI uzly
        Vytvor vstupný bod pre funkciu // Bude obsahovať skok na pôvodne prvý blok
}
```

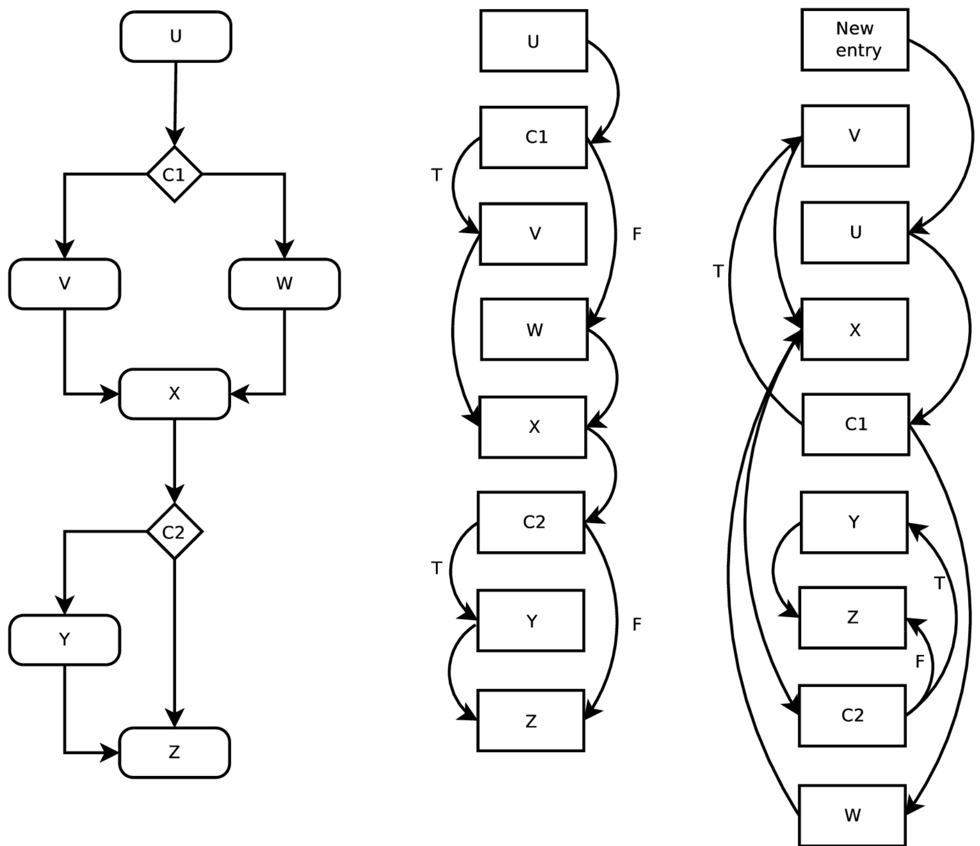
Obrázek 5.10: Algoritmus premiestnenia kódu.

Na rozdiel od dekompozície kódu nevznikajú pri premiestneniach žiadne nové základné bloky. Rovnako ani nezanikajú. Taktiež sa nevytvárajú nové príkazy podmienených alebo nepodmienených skokov. Dochádza však k modifikácií už existujúcich príkazov vetvenia a to tak, aby sa zachoval pôvodný tok riadenia programu.

Celkový počet blokov, z ktorých sa skladá funkcia, sa v závere prevádzania algoritmu zmení. Keďže sa premiestňovali jednotlivé bloky medzi sebou, tak obecne došlo aj k zmene umiestnenia bloku, ktorý tvorí vstupný bod pre danú modifikovanú funkciu. Kvôli tomuto je nutné vytvoriť nový vstupný blok, ktorý je vložený na začiatok funkcie a jeho obsahom je jediný príkaz nepodmieneného skoku na pôvodne prvý blok.

Celý tento proces je zachytený na diagramoch v 5.11. Najviac naľavo sa nachádza diagram toku pôvodného aj obfuskovaného programu (v tomto diagrame sa neberie do úvahy modifikácia vstupného bodu funkcie). Stredný graf abstraktne znázorňuje rozloženie programového kódu v LLVM IR, pričom jednotlivé šípky naznačujú skoky. V prípade, že skok je nepodmienený, neobsahuje šípka popis. Pre podmienené skoky obsahuje popis T v prípade, že sa tento skok vykoná, ak sa podmienka obsiahnutá v bloku, odkiaľ šípka smeruje, vyhodnotí ako logická hodnota $true$. Popis F, je pre komplementárnu alternatívu, teda, že výsledok vyhodnotenia danej podmienky je hodnota $false$. Posledný graf ukazuje jednu z možností, ako môže vyzerať obfuskovaná verzia tejto funkcie. Blok označený popisom $New\ entry$ značí nový vstupný blok funkcie.

Ďalším problémom, ktorý je nutné riešiť pri modifikácií príkazov vetvenia je aj modifikácia PHI (ϕ) uzlov. PHI uzol je inštrukcia, ktorá ak existuje v základnom bloku, tak tvorí jeho vstupný bod, a ktorá vracia hodnotu v závislosti na základnom bloku, z ktorého sa na ňu skákalo. Keďže bloky, ktoré obsahujú skoky na blok obsahujúci PHI uzol sa obecne môžu v kóde premiestniť, je nutné reflektovať tieto zmeny aj v tejto inštrukcii.



Obrázek 5.11: Premeniestnenie blokov kódu.

Z pohľadu deobfuskácie sa jedná o efektívnejšiu metódu ako je metóda dekompozície alebo metóda vkladania mŕtveho kódu. Pre ešte väčšiu účinnosť obfuskácie je vhodné využiť túto metódu po aplikácií metód substitúcie a/alebo dekompozície, pretože v oboch týchto metódach obecne dochádza k zvýšeniu počtu základných blokov funkcií.

5.5 Klonovanie podprogramov

Metóda klonovania podprogramov má za úlohu vytvoriť viac klonov jednej funkcie v prípade, že je táto funkcia viacnásobne volaná v pôvodnej aplikácií. Nemusí však platiť, že sa vytvorí nový klon pre každé volanie pôvodnej funkcie, ako ukazuje algoritmus 5.12, podľa ktorého pracuje implementovaný priechod v zadnej časti prekladača LLVM. V prípade, že funkcia je volaná na jedinom mieste programu, nie je obfuskácia nad touto funkciou realizovaná. Inak je náhodne vytvorený klon, alebo v prípade, že už nejaký klon existuje, môže byť volanie podprogramu obfuskované na volanie už existujúceho klonu daného podprogramu. Maximálny počet vytvorených klonov sa rovná počtu volaní danej funkcie znížený o jedna, pretože sa v obfuskovanom programe vždy zachová aj pôvodný podprogram.

Tento algoritmus má okrem vytvárania klonov funkcií ešte jednu vlastnosť. Ide o pretvorenie priamej rekurzie na rekurziu nepriamu, pretože ak je v programe využitá priama rekurzia, tak daný program obsahuje minimálne dve volania určitej rekurzívnej funkcie. Prvé volanie je mimo tejto funkcie a druhé je v jej tele. Tým pádom môže dôjsť k vytvorenie klonu, pričom jedno z volaní danej rekurzívnej funkcie sa transformuje na volanie prísluš-

```
// Pseudokód algoritmu klonovania funkcií
for ( jednotlivé inštrukcie volania funkcií ) {
      if ( rand() < 0.8 ) { // 80% pravdepodobnosť pokračovania
            // Jediné volanie funkcie nie je obfuskované
            if ( počet pôvodných volaní funkcie <= 1 )
                  continue
            // V prípade viacerých volaní jednej funkcie sa pokračuje
            if ( funkcia ešte nemá žiadny klon )
                  Naklonuj
            else if ( počet klonov funkcie >= počet pôvodných volaní tejto funkcie - 1 )
                // Dosiahol sa maximálny dovolený počet klonov
                  Náhodne vyber klon
            else
                  if ( rand() < 0.5 ) // 50% pravdepodobnosť pre obe možnosti
                        Naklonuj
         else
                        Náhodne vyber klon
      }
}
```

Obrázek 5.12: Algoritmus klonovania funkcií.

ného klonu. V tomto bode môže nastať jeden z dvoch scenárov. Prvý je, že sa transformuje volanie tejto funkcie, ktoré sa nachádza mimo danej rekurzívnej funkcie. V tomto prípade dochádza k volaniu klonu. Tento klon je klonom pôvodnej rekurzívnej funkcie, teda obsahuje kópie všetkých jej inštrukcií, z čoho vyplýva, že aj pôvodného rekurzívneho volania. Tento scénar iba vyčlení prvé rekurzívne volanie mimo pôvodnú funkciu. Zvyšok výpočtu sa zrealizuje priamou rekurziou. Zaujímavejší je druhý scenár, v ktorom je transformované rekurzívne volanie vrámci pôvodnej rekurzívnej funkcie na volanie jej klonu. V tomto prípade obsahuje pôvodná funkcia volanie klonu a klon volanie pôvodnej funkcie, čo je základnou vlastnosťou nepriamej rekurzie. Sémantika aplikácie ostáva neporušená.

Zásadným prvkom algoritmu je naklonovanie pôvodnej funkcie, ktoré sa realizuje pomocou API funkcie LLVM. Táto vlastnosť LLVM má zároveň dopad na otázky kompatibility jednotlivých verzií prekladača, ktorej vlastnosti sú priblížné v kapitole 9.

Pre názornú ukážku práce tohoto priechodu boli zvolené výňatky LLVM IR kódu, ktorý bol vygenerovaný pre jednoduchú aplikáciu obsahujúcu dve volanie tej istej funkcie. Prvá ukážka 5.13 obsahuje výňatok z neobfuskovaného programu. Samotný program sa skladá z dvoch funkcií, pričom jedna z funkcií obsahuje dvojnásobné volanie funkcie druhej. Z druhej ukážky 5.14 je zrejmé, že došlo k obfuskácií programového kódu a aplikácia sa skladá celkovo z troch funkcií, pričom jedna z nich obsahuje volania zvýšných dvoch. Správanie oboch aplikácií ostáva sémanticky ekvivalentné.

Priechod realizujúci klonovanie podprogramov je jediný z implementovaných priechodov využívajúci v rozhraní LLVM pre programovanie aplikácie, API, ako abstraktnú štruktúru na navyššej úrovni štruktúru modulu. Tá umožňuje priechodu realizovať zmeny na najvyššej úrovni, teda pridávanie a odoberanie podprogramov, prípadne globálnych objektov. Zvyšné priechody majú ako štruktúru na najvyššej úrovni štruktúru funkcie, ktorá umoňuje realizovať zmeny v rámci jednej funkcie, pričom zmeny sú postupne realizované na všetkých funkciách obsiahnutých v module.

Klonovanie podprogramov je vhodné, rovnako ako dekompozíciu kódu, kombinovať s inými priechodmi. Kombináciou sa dosiahne obfuskácia tiel klonov a zároveň aj tela pô-

```
define i32 @factorial(i32 %x) nounwind uwtable {
entry:
  ...
  %6 = load i32* %fact, align 4
  ret i32 %6
}

define i32 @main() nounwind uwtable {
entry:
  ...
  %call = call i32 @factorial(i32 5)
  store i32 %call, i32* %a, align 4
  %call1 = call i32 @factorial(i32 10)
  store i32 %call1, i32* %b, align 4
  ...
  ret i32 0
}
```

Obrázek 5.13: Neobfuskovaná verzia programu.

vodnej funkcie, čím sa sťaží deobfuskácia a analýza kódu pri reverznom inžinierstve.

5.6 Nepriehľadné predikáty

Pomocou nepriehľadných predikátov sa do programového kódu dostávajú nové bloky, ktoré majú funkciu rozhodovacích blokov. Rozhodovacie bloky majú ako svoj výstupný bod inštrukciu podmieneného skoku. Týmto je možné obfuskovať tok programu, avšak v skutočnosti nedochádza k zmene sémantiky vykonávania aplikácie.

Pri vkladaní nepriehľadných predikátov dochádza k rozšíreniu pôvodnej podmienky podmieneného skoku o nepriehľadný predikát. Naviazanie pôvodnej podmienky a nepriehľadného predikátu sa realizuje pomocou výrazov 5.3. Tie sa opierajú o axiómy Boolovej algebry, konkrétne o vlastnosti neurálnych prvkov a existenciu komplementárneho prvku.

$$
\begin{aligned}
x \;\&\&\; true &== x \\
x \parallel false &== x \\
!true &== false \\
!false &== true
\end{aligned}
\tag{5.3}
$$

V implementovam priechode pre zadnú časť prekladača LLVM sú implementované výrazy z tabuľky 5.15. Druhý stĺpec tabuľky obsahuje pravdivostnú hodnotu predikátu pre všetky vstupy, pričom výrazy sú cielené na množinu celých čísiel.

Postup pri vkladaní nepriehľadných predikátov je založený na štyroch krokoch. V prvom kroku sa nájde vhodná podmienka obsahujúca aspoň jednu celočíselnú premennú. Následne sa náhodne vyberie nepriehľadný predikát, ktorý bude použitý na obfuskáciu. Tretí krok spočíva v náhodnom rozhodnutí, či bude pre prepojenie tohoto predikátu a pôvodnej podmienky použitá operácia logického súčtu alebo logického súčinu. Po týchto rozhodovacích krokoch nasleduje samotné vloženie nepriehľadného predikátu do programového kódu.

Zároveň je zohľadnený počet premenných, ktoré budú zahrnuté do nepriehľadného predikátu. Náhodný výber predikátu, ktorý bude použitý na obfuskáciu, zohľadňuje počet pre-

```
define i32 @factorial(i32 %x) nounwind uwtable {
entry:
  ...
  %6 = load i32* %fact, align 4
  ret i32 %6
}

define i32 @main() nounwind uwtable {
entry:
  ...
  %call = call i32 @factorial_C1(i32 5)
  store i32 %call, i32* %a, align 4
  %call1 = call i32 @factorial(i32 10)
  store i32 %call1, i32* %b, align 4
  ...
  ret i32 0
}

define internal i32 @factorial_C1(i32 %x) nounwind uwtable {
entry:
  ...
  %6 = load i32* %fact, align 4
  ret i32 %6
}
```

Obrázek 5.14: Obfuskovaná verzia programu s klonmi podprogramov.

menných, ktoré sa pôvodne vyskytovali v podmienke. V prípade implementovaných predikátov je v situácií, že pôvodná podmienka pracuje s jednou premennou, k dispozícií jedenásť predikátov. Ak sú však využité dve a viac premenných, vyberá sa zo všetkých dvanástich predikátov.

Diagram 5.16 ukazuje výňatok neobfuskovaného programu. Ide o konštrukciu selekcie, presnejšie, podmienený skok. Pri obfuskácií pomocou nepriehľadných predikátov dochádza k jednému z dvoch scenárov, pričom oba sú znázornené na diagramoch v 5.17. Diagram vľavo znázorňuje vloženie predikátou pomocou operácie logického súčinu. V tomto prípade sa vložený predikát vyhodnotí vždy ako pravdivostná hodnota *true*. Diagram vpravo využíva na prepojenie pôvodnej podmienky a predikátu operáciu logického súčtu, pričom výsledkom takto vloženého predikátu je pre každý platný vstup logická hodnota $false$.

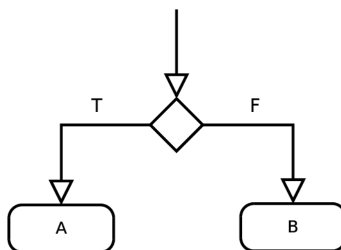
Tretí predikát v tabuľke 5.15 obsahuje logickú spojku logického súčtu. Pri obfuskácií pomocou tohoto predikátu dochádza k zanorení v hierarchii rozhodovacích blokov, pretože sa vytvoria dva nové rozhodovacie bloky, ktoré nasledujú za sebou. Ich prepojenie na existujúce bloky musí rovnako zohľadňovať pôvodnú sémantiku podmienky.

Pre porovnanie neobfuskovanej a obfuskovanej verzie inštrukcie podmieného skoku môžu slúžiť výňatky LLVM IR kódu 5.18, ktorý obsahuje jedinú inštrukciu vetvenia na základe logickej hodnoty a 5.19, ktorý znázorňuje jej obfuskovanú verziu pomocou nepriehľadného predikátu $(x^2 + x + 7)\ mod\ 81 == 0$. Prepojenie je realizované pomocou operácie logického súčtu. V bloku *opaque* sa vždy vykoná vetvenie na návestie $\%if.end$, pretože výsledkom podmienky bude pre všetky situácie logická hodnota $false$.

Algoritmus kladie podmienky aj na premenné, ktoré využíva vo vkladaných nepriehľadných predikátoch. Každá premenná musí spĺňať podmienku nielen na jej dátový typ, ale aj na maximálnu požadovanú bitovú šírku. Maximálna bitová šírka celého čísla v prekladači

Nepriehľadný predikát	Výsledok
$7 * y^2 - 1 == x^2$	false
$(x^3 - x)\ mod\ 3 == 0$	true
$(x\ mod\ 2 == 0) \parallel ((x^2 - 1)\ mod\ 8 == 0)$	true
$(x^2/2)\ mod\ 2 == 0$	true
$(x * (x + 1))\ mod\ 2 == 0$	true
$x^2 >= 0$	true
$(x^2 + x)\ mod\ 2 == 0$	true
$(x * (x + 1) * (x + 2))\ mod\ 3 == 0$	true
$(x^2 + 1)\ mod\ 7 == 0$	false
$(x^2 + x + 7)\ mod\ 81 == 0$	false
$(4 * x^2 + 4)\ mod\ 19 == 0$	false
$(x^2 * (x + 1) * (x + 1))\ mod\ 4 == 0$	true

Obrázek 5.15: Ukážka nepriehľadných predikátov.

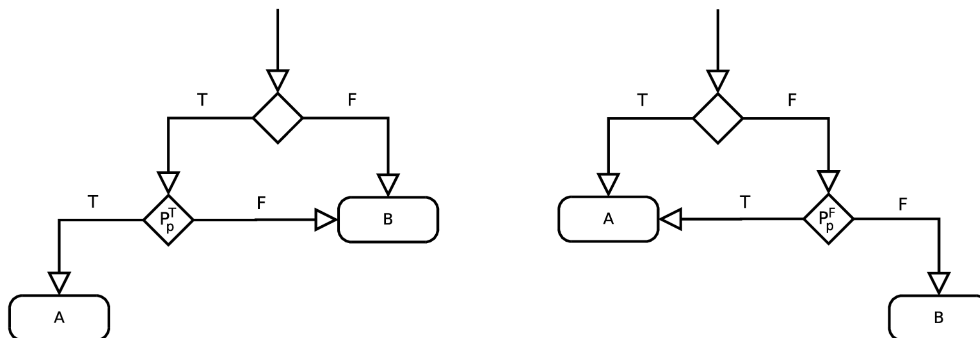


Obrázek 5.16: Vývojový diagram úseku programu bez nepriehľadných predikátov.

LLVM je 64 bitov. Keďže je potreba, aby boli výrazy zložité a náročné na deobfuskáciu, obsahujú exponenty, prípadne viacnásobné operácie násobenia. Aplikáciou týchto operácií môže obecne dôjsť k pretečeniu obsahu alokovanej premennej. Preto má každý predikát priradené celé kladné číslo reprezentujúce maximálnu povolenú bitovú šírku premennej, pomocou ktorej môže byť aplikovaný. V prípade, že premenná presiahne túto bitovú šírku, nie je možné daný nepriehľadný predikát do programu vložiť, pretože nie je možné zaručiť jeho správnu činnosť s ohľadom na pôvodnú aplikáciu.

5.7 Substitúcia pomocou genetického programovania

Prechod realizujúci substitúciu pomocou slovníka bol popísaný v 5.2. Alternatívou k tejto obfuskácií je vkladanie substitúcií metódou genetického programovania. Genetické programovanie (GP) umožňuje dynamické vytváranie výrazov sémanticky ekvivalentných k substituovaným výrazom. Z tohoto dôvodu nie je potrebný statický slovník, ale výrazy sa vyt-



Obrázek 5.17: Vývojové diagramy úseku programu s nepriehľadnými predikátmi.

```
br i1 %cmp, label %if.then, label %if.end
```

Obrázek 5.18: Neobfuskovaná verzia podmieneného skoku

várajú automaticky pomocou algoritmu. Algoritmus genetického programovania je obecný a nezávislý na cieľovej problematike. Jeho tvar je zachytený v zápise 5.20 [16].

Základné vlastnosti GP sú [16]:

- Pracuje so spustiteľnými štruktúrami, najčastejšie reprezentovanými stromami. Keďže sa jedná o stromovú štruktúru, tak jednotlivé objekty majú rôznu dĺžku.
- Obsahuje genetické operátory - reprodukcia, kríženie, mutácia, ktoré pracujú nad spustiteľnými štruktúrami.
- Vhodnosť vygenerovaného výrazu je testovaná *fitness* funkciou pre definovanú množinu vstupov.

Prácu genetického programovania je možné priblížiť na dvoch stromových reprezentáciach matematických výrazov z obrázku 5.21. Stromová štruktúra vľavo obsahuje substituovaný výraz nachádzajúci sa v pôvodnom programe. Vpravo sa nachádza jeden z vygenerovaných výrazov, ktorý je možný použiť na obfuskáciu.

Pre lepšie pochopenie GP je vhodné priblížiť prácu jednotlivých genetických operátorov. Operácia reprodukcie je najjednoduchšia z trojice implementovaných operácií a jej práca spočíva vo vybratí jedného, prípadne množiny najvhodnejších jedincov v populácií a ich okopírovanie do novej populácie. Štvorica stromov v 5.22 zobrazuje vlastnosti operácie kríženia. Ide o výber dvoch náhodných podstromov v rodičovských stromoch a ich následná výmena. Vznikajú dva synovské stromy, ktoré budú zaradené do novej populácie. Operácia mutácie je zachytená na obrázku 5.23. Mutácia náhodne vyberie podstrom mutovaného stromu a náhodne mu vygeneruje nový podstrom. Týmto sa nezmení počet jedincov v populácií, ale zmení sa charakter daného jedinca, teda hodnota jeho *fitness* funkcie. Operandy pre jednotlivé operácie sa vyberajú náhodne, avšak je vyššia pravdepodobnosť výberu vhodnejšieho jedinca populácie, teda jednica s vyšším ohodnotením pomocou testovacej *fitness* funkcie.

Tento algorimus obecne nekončí vždy nájdením požadovaného výsledku, pretože je obmedzený počtom populácií. Za výsledok sa považuje iba taký vygenerovaný výraz, ktorý ma identické správanie s pôvodným výrazom pre všetky testované vstupy.

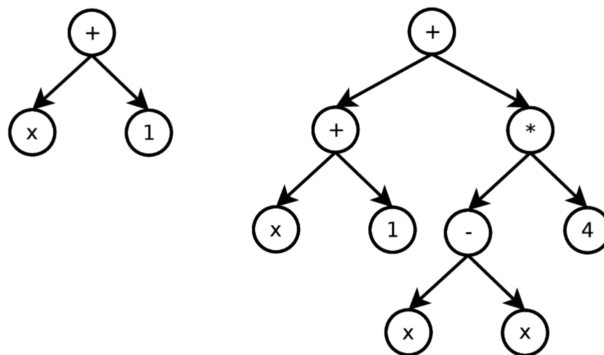
```
  br i1 %cmp, label %if.then, label %opaque
opaque:
  %opaque_load = load i32* %q
  %opaque_cast = sext i32 %opaque_load to i64
  %opaque_mul = mul i64 %opaque_cast, %opaque_cast
  %opaque_add = add i64 %opaque_mul, %opaque_cast
  %opaque_add1 = add i64 %opaque_add, 7
  %opaque_mod = srem i64 %opaque_add1, 81
  %2 = icmp eq i64 %opaque_mod, 0
  br i1 %2, label %if.then, label %if.end
```

Obrázek 5.19: Obfuskovaná verzia podmieneného skoku

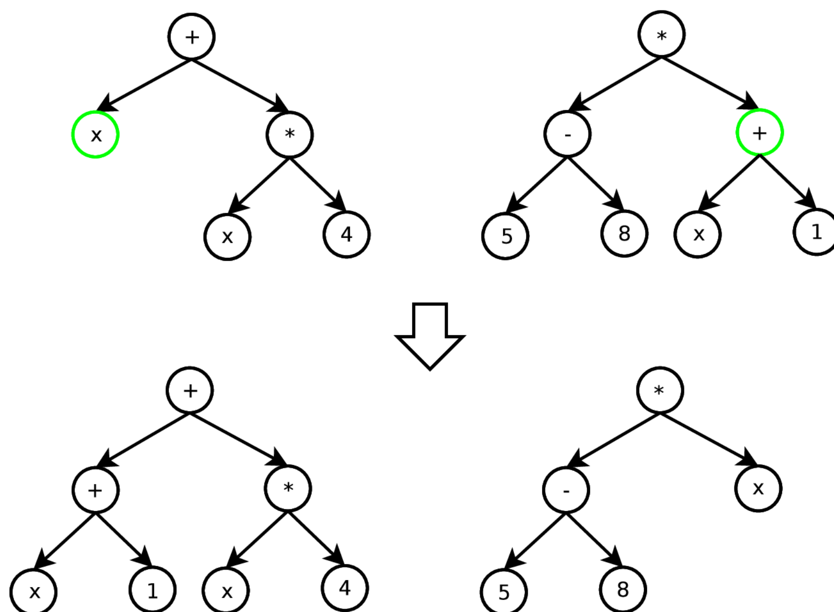
```
// Pseudokód algoritmu genetického programovania
1. Inicializácia populácie.
2. Priradenie fitness hodnoty jednotlivým jedincom.
3. while ( populácie nie je plne obsadená ) {
       Výber jedinca alebo niekoľkých jedincov selekčným algoritmom
       Realizácia genetickej operácie nad vybranými jedincami
       Vloženie výsledku k novej populácií
}
4. if ( výsledok sa nenašiel ) {
       goto 2
}
5. Vráť výsledkok
```

Obrázek 5.20: Algoritmus genetického programovania.

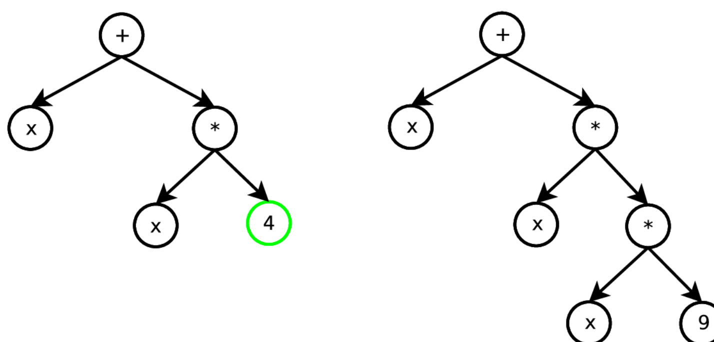
Vyššie popísané vlastnosti GP v obecnosti popisujú implementovaný obfuskačný algoritmus dynamickej substitúcie. Posledným problémom je spôsob napojenia algoritmu na LLVM IR. Ten sa deje prevodom stromovej reprezentácie do prefixovej (poľskej) notácie. Následne je táto notácia spracovaná a prevedená pomocou LLVM API do LLVM IR.



Obrázek 5.21: Substitúcia výrazu algoritmom genetického programovania.



Obrázek 5.22: Operácia kríženie.



Obrázek 5.23: Operácia mutácia.

Kapitola 6

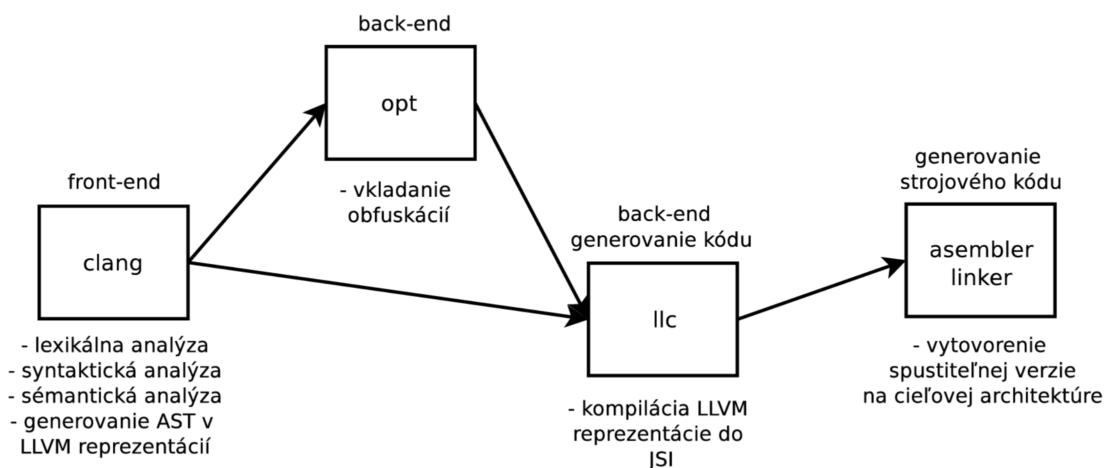
Implementácia

Implementácia obfuskačných priechodov je realizovaná v jazyku C++, pretože tento jazyk je použitý pre implementáciu prekladača LLVM. Ako bolo spomenuté už v predchádzajúcich kapitolách, obfuskácie boli rozdelené do niekoľkých priechodov, pričom každý priechod je možné aplikovať samostatne.

Schéma 6.1 zobrazuje spôsob aplikácie obfuskácií v prekladači LLVM. Ako predná časť prekladača je využitý prekladač clang, ktorý po jednotlivých analýzach vytvorí súbor obsahujúci LLVM bajtkód. Následne je možné buď priamo skompilovať tento bajtkód do jazyka symbolických inštrukcií (JSI), alebo pomocou nástroja opt vložiť do LLVM reprezentácie kódu implementované obfuskácie.

Nástroj opt je modulárny LLVM optimalizátor a analyzátor, ktorý umožňuje aplikovať obfuskačný priechod na LLVM bajtkód. Následne sa obfuskovaný bajtkód preloží do jazyka symbolických inštrukcií pomocou nástroja llc. V tomto kroku je možné zvoliť cieľovú architektúru, pre ktorú bude v ďalšom kroku vytvorený spustiteľný súbor. Z tohoto pohľadu je zrejmé, že takto implementované obfuskácie sú generické vzhľadom na cieľovú platformu, pretože k výberu cieľovej platformy dochádza až po aplikovaní obfuskácií. Samotný rozsah architektúr, na ktoré je možno aplikovať obfuskačné prechody je závislý iba na schopnosti generovať jazyk symbolických inštrukcií pre jednotlivé architektúry systémom LLVM.

V poslednom kroku sa vytvorí binárny súbor pomocou asembleru a zostavovacieho programu cieľovej architektúry.



Obrázek 6.1: Schéma vkladania obfuskácií.

Z pohľadu implementácie je taktiež nutné vyzdvihnúť vlastnosti LLVM API slúžiaceho na podporu vytvárania optimalizačných, v tomto prípade obfuskačných, priechodov v zadnej časti prekladača.

Medzi základné metódy používané pri implementácií nových priechodov patria metódy, ktorých deklarácie sú v 6.2. Ide o metódy, ktoré sa spustia pre každý modul, funkciu, prípadne základný blok spracovávaného programu. Rovnako sú implementované všetky potrebné druhy iterátorov cez jednotlivé štruktúry reprezentujúce spracovávaný program, ako sú napríklad iterátory cez operandy inštrukcií, inštrukcie, základné bloky, funkcie. Rovnako je možné použiť metódy na vyššej úrovni abstrakcie, ktorých návratové hodnoty sú výsledkom vyhľadávania v daných štruktúrach. Príkladom môže byť metóda vracajúca výstupné body základných blokov, alebo vstupné body s vlastnosťami PHI uzlov.

```
virtual bool runOnModule(Module &M) = 0;
virtual bool runOnFunction(Function &F) = 0;
virtual bool runOnBasicBlock(BasicBlock &BB) = 0;
```

Obrázek 6.2: Deklarácie niektorých metód z LLVM API.

Už spomínanou vlastnosťou LLVM je jeho typovaná reprezentácia vnútorného kódu. Pre intuitívnu prácu s jednotlivými typmi taktiež existujú metódy, ktorých príklady je možné nájsť v 6.3.

```
Type* getType ()
bool isIntegerTy ()
bool isFloatingPointTy ()
unsigned getBitWidth ()
```

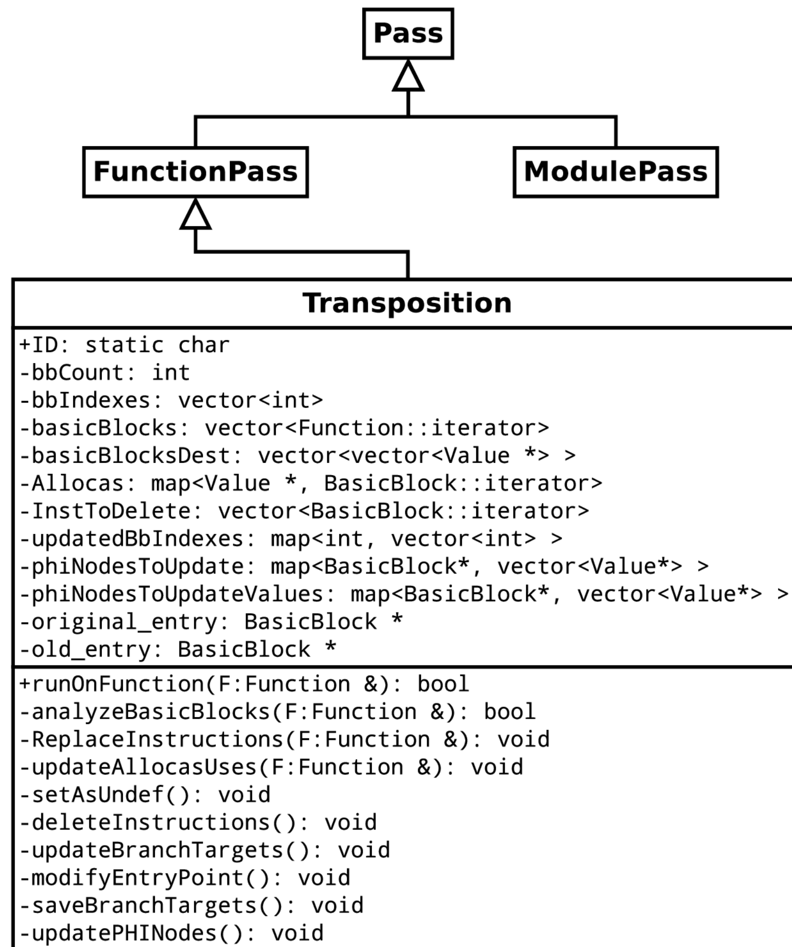
Obrázek 6.3: Deklarácie niektorých metód pre prácu s typmi z LLVM API.

Z pohľadu implementácie v zadnej časti prekladača LLVM bol najzložitejším priechodom priechod realizujúci premiestnenie základných blokov. Jeho diagram tried v notácií UML je znázornený na 6.4.

Výsledné poradie blokov sa nachádza vo vektore celých čísiel s názvom *bbIndexes*. Je to inštančná premenná, ktorá sa následne využíva vo všetkých metódach triedy *Transposition*. Jej využitie je primárne na výpočet nových cieľov skokov, prípadne na úpravu PHI uzlov, ktoré sú tiež závislé na skokoch medzi jednotlivými základnými blokmi.

Pri premiestňovaní jednotlivých inštrukcií blokov medzi sebou nastal problém v odkazoch na jednotlivé alokované premenné a v odkazoch na výsledky inštrukcií. Bolo to z dôvodu, že výsledok každej alokácie alebo výsledok inštrukcie je reprezentovaný samotným ukazateľom na danú alokáciu, prípadne inštrukciu. Pri premiestnení vzniká nový odkaz na takúto konštrukciu, teda je nutné realizovať analýzu použitia jednotlivých alokácií, prípadne výsledkov inštrukcií a pôvodné ukazatele nahradiť ukazateľmi novými. Toto majú za úlohu metódy *ReplaceInstructions(Function &F)* a *updateAllocaUses(Function &F)*.

Ďalší problém nastal pri odstraňovaní pôvodných inštrukcií, ktoré už boli premiestnené. Premiestnenie inštrukcie zanechá jej vzor na pôvodnom mieste. Tento vzor je nutné odstrániť. Samotné odstránenie sa však deje pred upravením cieľov skokov na spávne premiestnené bloky, teda analýzy, ktoré sú realizované systémom LLVM pri odstraňovaní takýchto inštrukcií môžu skončiť neúspechom. Je to z dôvodu, že inštrukcia nesmie byť z kódu od-



Obrázek 6.4: Diagram tried priechodu v zadnej časti prekladača LLVM.

stránená v prípade, že jej výsledok je využívaný inou inštrukciou. Z tohoto dôvodu boli operandy každej inštrukcie určenej na odstránenie upravené na hodnotu *llvm::UndefValue*. Táto hodnota bola v LLVM implementovaná práve pre situácie, kedy je potrebné nastaviť odkaz operandu na nešpecifikovaný bitový vzor. Tento mechanizmus zaručí neutralitu jednotlivých inštrukcií pri ich odstraňovaní.

Posledný krok, ktorý nemusí byť zrejmý je modifikácia vstupného bodu. Príslušná metóda má názov *modifyEntryPoint()*. Táto metóda najprv rozdelí prvý základný blok funkcie na dva. Prvý z nich obsahuje iba jedinú inštrukciu skoku na blok druhý, ktorý si zachová pri tomto rozdelení pôvodné inštrukcie. Následne sa modifikuje spomínaný skok tak, aby sa zachovalo pôvodné vykonávanie programu.

Diagram 6.4 taktiež zobrazuje dedičnosť vrámci tvorby priechodu pre zadnú časť prekladača LLVM. Všetky implementované priechody sú založené na tejto dedičnosti. Jedinou výnimkou je priechod realizujúci klonovanie funkcií, ktorý priamo dedí z triedy *ModulePass* tiež znázornenej na spomínanom diagrame.

Kapitola 7

Testovanie

7.1 Cieľ testovania

Stanovený cieľ testovania musí korešpondovať s definíciou obfuskácie uvedenej v 1. Teda ide o otestovanie sémantickej ekvivalencie medzi pôvodným funkčným programom a programom, nad ktorým bola realizovaná obfuskácia. Pre samotné testovanie je nutné tieto ciele bližšie špecifikovať. V priebehu testovania sa pôvodný a obfuskovaný program bude testovať na zhodnosť:

1. Výstupu
2. Návratovej hodnoty

Samotné testovanie musí prihliadať aj na generickosť obfuskácie, teda bude prebiehať na nasledujúcich architektúrach:

1. x86_64
2. x86
3. ARM

7.2 Metóda testovania

Pre samotné testovanie bolo vybraných 101 testovacích programov zapísaných v jazyku C, ktoré obsahujú rozličnú funkcionalitu a výpočetné vlastnosti tak, aby sa otestovali všetky aspekty obfuskačných priechodov. Testovacia metodika zahŕňa postupné testovanie obfuskačných priechodov od samostatných priechodov, cez všetky existujúce dvojice, trojice, až postupne po všetky permutácie bez opakovania, ktoré vznikú z množniny všetkých obfuskačných priechodov. Počet takto vzniknutých kombinácií je zachytený vo vzorci 7.1. Koštanta 6 značí počet priechodov. Premenná i nadobúda hodnoty od 1 do $spolu_priechodov - 1$.

$$\sum_{i=1}^{5} \prod_{j=0}^{i} (6 - j) = 1956 \tag{7.1}$$

7.3 Realizácia

Na testovanie bol vytvorený skript zapísaný v jazyku Bash, ktorý umožňuje realizovať vyššie popísanú metódu testovania. Testovanie pre architektúru x86_64 prebiehalo na stroji s procesorom Intel Core i5-3750K (3.4GHz), 8GB operačnej pamäte, OS Linux Mint 13. Správanie projektu na architektúre x86 bolo realizované na stroji osadenom procesorom Intel Core 2 Duo (1.4GHz), 2GB operačnej pamäte, OS Linux Mint 13. ARM architektúra bola pre účely testovania virtualizovaná na prvom stroji pomocou virtualizačného software Qemu. Na tento účel bol vybraný operačný systém Linux Debian 2.6 s architektúrou armv5tejl.

7.4 Výsledky

Jediným prípustným výsledkom testovania je zhoda vo všetkých výstupoch príslušných neobfuskovaných a obfuskovaný verziách programov. Tento výsledok bol dosiahnutý aj pri testovaním implementovaných priechodov [7.1], čím sa potvrdila platnosť podmienky stanovenej v definícií 1.

Architektúra	**Úspešných testov**
x86_64	100%
x86	100%
ARM	100%

Obrázek 7.1: Percentuálna úspešnosť testov na jednotlivých architektúrach.

Kapitola 8

Štatistiky

Kapitola 2 uvádza, že pôvodný a obfuskovaný program nemusia byť úplne ekvivalenté, teda, že obfuskovaný program môže mať dlhšiu dobu vykonávania, alebo môže spotrebovávať viac operačnej pamäte. Táto kapitola sa zameriava na porovnanie týchto vlastností obfuskácie na niekoľkých známych algoritmoch zapísaných pomocou programovacieho jazyka C. Jedná sa o nasledujúce algoritmy, ktorých zdrojové kódy v jazyku C sú uvedené v prílohe A (v zátvorkách sú uvedené ich označenia použité v nasledujúcich grafoch):

- Ackermannova funkcia (ackermann)
- Cyklický redundantný súčet (crc)
- Dijkstrov algoritmus (dijkstra)
- Fouriérova transformácia (fourier)
- Quicksort (quicksort)

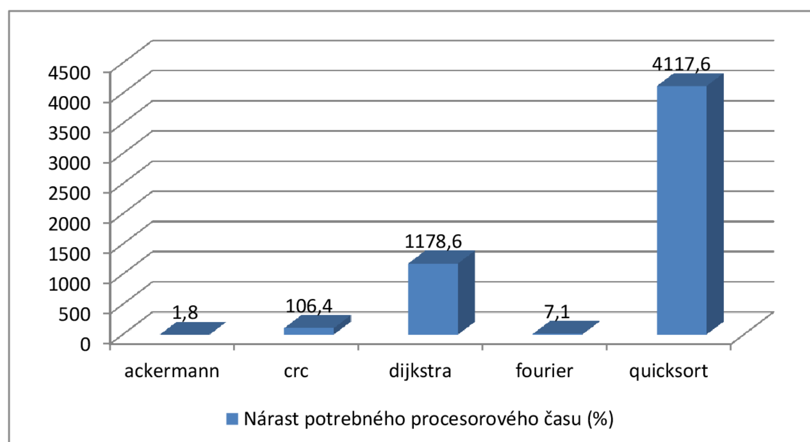
Každý z nasledujúcich grafov bude zachytávať percentuálny nárast skúmanej hodnoty v obfuskovanom programe oproti danej hodnote v pôvodnom programe. Obfuskácie boli vkladané v nasledovnom poradí: klonovanie funkcií, vkladanie mŕtveho kódu, vkladanie nepriehľadných predikátov, substitúcie, dekompozícia kódu, premiestnenie kódu.

Graf 8.1 ukazuje, že v niektorých prípadoch môže dôjsť k veľkému rozdielu medzi časom potrebným na vykonanie pôvodného programu a obfuskovaného programu. Potrebný procesorový čas sa vypočítal ako priemer časov z 10 000 behov každého programu.

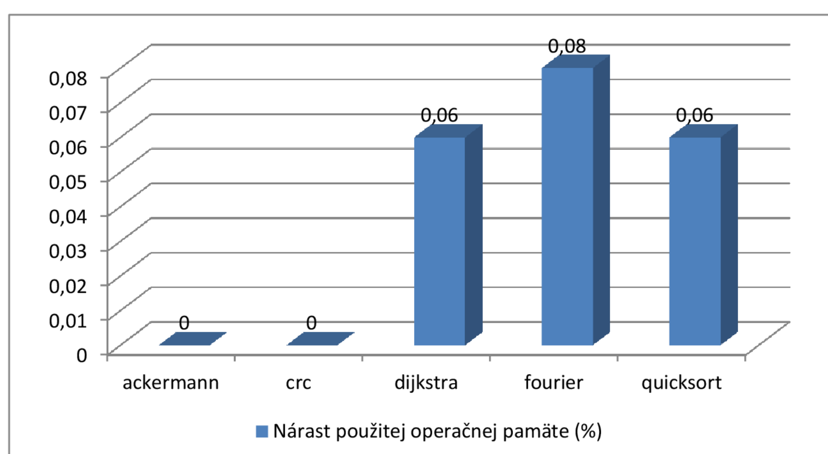
Narozdiel od procesorového času je nárast z pohľadu využívania operačnej pamäte obfuskovaného programu oproti neobfuskovanému minimálny, ako ukazuje graf 8.2. Hodnoty spotrebovanej pamäte boli získané pomocou nástroja Valgrind a jeho utility Massif.

Poslednou štatistikou je ukážka percentuálneho nárastu počtu riadkov kódu v obfuskovanom programe. Jedná sa o počet riadkov v jazyku symbolických inštrukcií, ktoré vygeneroval nástroj llc, ktorý je súčasťou zadnej časti prekladača LLVM, ako je uvedené v kapitole 6.

Celkovo je možné z grafov vyčítať, že neexistuje závislosť medzi rozsahom obfuskácie z hľadiska nových vložených inštrukcií a nárastom spotreby operačnej pamäte alebo procesorového času. V prípade ak dôjde k obfuskácií často a cyklicky sa vykonávajúcich operácií, ako to bolo v prípade obfuskácie algoritmu Quicksort, je možné aj malou zmenou kódu spôsobiť veľkú zmenu v nárokoch programu na prostredie, v ktorom beží. Naopak pri zmene

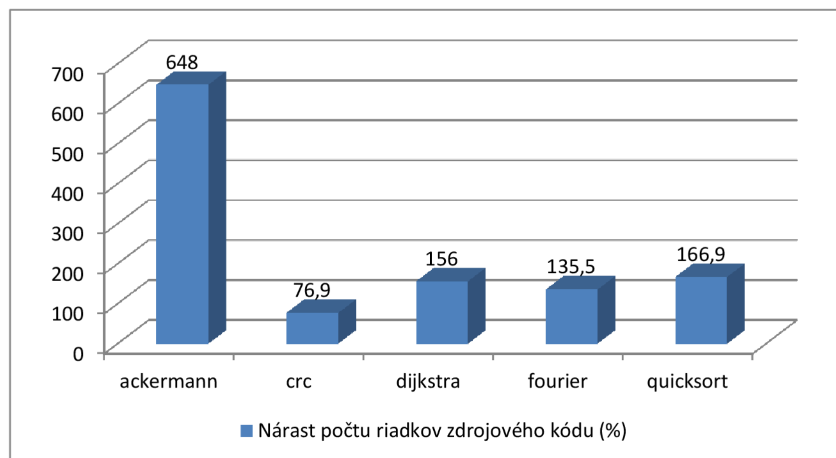


Obrázek 8.1: Stĺpcový graf zachytávajúci percentuálny nárast procesorového času.



Obrázek 8.2: Stĺpcový graf zachytávajúci percentuálny nárast spotrebovanej pamäte.

jednorázovo sa vykonávajúcich inštrukcií a inicializácií, prípadne pretvorení priamej rekurzie na nepriamu, ako to bolo v prípade Ackermannovej funkcie, nie je ani veľký nárast programového kódu zárukou veľkej zmeny nárokov aplikácie na svoje prostredie.



Obrázek 8.3: Stĺpcový graf zachytávajúci percentuálny nárast počtu riadkov kódu.

Kapitola 9

Budúci vývoj

Jednou z hlavných charakteristík obfuskácie je široká škála možností na jej rozširovanie a to ako z pohľadu nových metód obfuskácie, tak aj z pohľadu prehlbovania, zefektívňovania a doplňovania už existujúcich priechodov. Z pohľadu implemetovaných priechodov existuje niekoľko možností, ktoré sú vhodné na zváženie ohľadom budúceho vývoja.

Prvou možnosťou je rozšírenie množiny výrazov implemetovaných v priechodoch substitúcie 5.2, vkladania mŕtveho kódu 5.1 a vkladania nepriehľadných predikátov 5.6. Pri hľadaní vhodných výrazov, ktoré by mohli byť kandidátmi na implementáciu do týchto priechodov, je najvhodnejšie sa opierať o vlastnosti algebier, zväzov a teóriu čísiel.

Ďalšou možnosťou rozšírenia funkcionality je rošírenie priechodu realizujúceho vkladanie substitúcií pomocou genetického programovania 5.7 o schopnosť generovať nové základné bloky kódu s ekvivalentnou funkcionalitou ako pôvodné základné bloky, teda okrem sekvencií inštrukcií vkladajúcich matematické výrazy, by bolo možné vkladať cykly, vetvenia, prípadne iné inštrukcie ako sú napríklad volania funkcií. S tým by súviselo aj generovanie celých funkcií, ktoré by zastupovali buď základné bloky z pôvodného programu, alebo by reprezentovali klony pôvodných funkcií.

Medzi pokročilé možnosti budúceho vývoja je možné zaradiť implementáciu nových priechodov pre zadnú časť prekladača LLVM, ktoré by pri obfuskácií vložili do programového kódu šifrovaciu funkciu, čím by sa svojou funkcionalitou dostala takáto obfuskácia na úroveň obfuskácie používanej v šifrovanom, prípadne oligomorfnom alebo polymorfnom malware [17]. Samotné zaradenie do určitej skupiny by záviselo na dokonalosti obfuskačného priechodu vytvárať a vkladať šifrovacie / dešifrovacie funkcie. V prípade ešte vyšších ambícií existuje možnosť vložiť pri obfuskácií do programového kódu metamorfné jadro, ktoré by pri samotnom šírení aplikácie z jedného zdroja na druhý autonómne realizovalo obfuskáciu svojho programového kódu.

Ďalším faktom, ktorý je nutné zvážiť, je kompatibilita implementovaných priechodov s jednotlivými verziami systému LLVM. Keďže LLVM je aktívne sa vyvýjajúci projekt, vznikajú v priebehu času nové verzie, a preto by bolo vhodné, aby boli implementované obfuskácie aplikovateľné aj v budúcnosti. Vývoj obfuskačných priechodov bol zahájený v čase, keď najaktuálnejšou verziou LLVM bola verzia 3.0. V raných fázach projektu bola vydaná veria 3.1. Prechod na novú verziu bol bezproblémový a pre túto verziu bol vyvýjaný celý projekt. Počas vývoja prišlo k vydaniu daľšej stabilnej verzie LLVM s označením 3.2. Prechod na túto verziu je možný, ale je nutný zásah do zdrojových textov projektu, pretože zo strany vývojárov došlo k zmene umiestnenia hlavičkových súborov a k upraveniu niektorých vysokoúrovňových funkcií. Z pohľadu projektu k nim patrila funkcia na vytvorenie klonu podprogramu, ktorej volanie bolo rozšírené o jeden parameter. Tieto zmeny avšak neboli

kritické, pretože nedošlo k zmene vnútornej reprezentácie kódu LLVM, ktorá tvorí základ pre obfuskáciu kódu.

Ako sekundárnym faktorom pre budúci vývoj je generickosť obfuskácií vzhľadom na cieľovú platformu, pretože neustále sa vytvárajú a vznikájú nové predné časti prekladača LLVM, ktoré rozširujú prodporu LLVM pre ďalšie programovacie jazyky. Rovnako sa navrhujú a implementujú nové generátory jazyka symbolických inštrukcií pre zadnú časť prekladača LLVM, teda sa rozširuje množina architektúr, pre ktorú je možné obfuskácie realizovať.

Všetky tieto skutočnosti poukazujú na fakt, že budúci vývoj obfuskáčných priechodov pre prekladač LLVM je možný a nie je obmedzený nijakým dôležitým faktorom. Záleží už len na kreativite návrhára ako efektívne a odolné obfuskácie vytvorí.

Kapitola 10

Záver

Obfuskácia umožňuje ukrytie vnútornej štruktúry a sémantiky programu. Toto sa využíva ako na ochranu intelektuálneho vlastníctva, tak aj na účely ukrytia malware pred antivírovými nástrojmi. Existuje mnoho techník ako realizovať obfuskáciu strojového kódu. K tým najjednoduchším patrí vkladanie mŕtveho kódu. K pokročilejším technikám je možno zaradiť klonovanie podprogramov alebo nepriehľadné predikáty.

V projekte sú naimplementované priechody vkladajúce mŕtvy kód, substitúcie a nepriehľadné predikáty. Taktiež priechody realizujúce klonovanie funkcií, dekompozíciu programového kódu a premiestnenie základných blokov tvoriacich programový kód.

Jednotlivé priechody boli otestované a to nielen samostatne, ale aj vo vzájomých kombináciach. Testy preukázali správnu implementáciu obfuskačných priechodov, ktorej základným bodom bolo dodržanie ekvivalentnej funkcionality pôvodného a obfuskovaného programu. Rovnako bola zohľadnená aj generickosť obfuskácie, teda obfuskácie je možné vkladať do aplikácií cielených na rôzne architektúry.

Obfuskácia poskytuje rozsiahle možnosti budúceho vývoja. Z pohľadu projektu sa môže jednať o rozšírenie jednotlivých implementovaných množín výrazov, ale rovnako aj pridanie nových priechodov realizujúcich odlišné metódy obfuskácie. Veľkú perspektívu poskytuje aj genetické programovanie, ktorého rozšírenie môže priniesť nové techniky obfuskovania aplikácií.

Obfuskáciu je možné realizovať rôznymi spôsobmi, pričom tento dokument sa zaoberal možnosťou využitia systému LLVM, v ktorom je možné obfuskáciu realizovať pomocou samostaných priechodov v prekladači. Táto medóda taktiež dovoluje parametrizovať jednotlivé obfuskácie voľbou vhodných priechodov, čo môže byť využité pre testovanie spätných prekladačov a deobfuskátorov. Práve techniky reverzného inžinierstva umožňujú úspešne odhalovať a eliminovať obfuskáciu, čím nútia tvorcov škodlivého softvéru a aj spoločnosti, ktoré sa snažia ochrániť svoje intelektuálne vlastníctvo, vyvýjať sofistikovanejšie metódy obfuskácie.

Literatura

[1] clang: a C language family frontend for LLVM. http://clang.llvm.org/, 17-01-2013.

[2] LLVM - Frequently Asked Questions. http://llvm.org/docs/FAQ.html#what-source-languages-are-supported.

[3] LLVM Bytecode File Format. http://llvm.org/releases/1.3/docs/BytecodeFormat.html, 17-01-2013.

[4] The LLVM Compiler Infrastructure. http://llvm.org/, 17-01-2013.

[5] The LLVM Compiler Infrastructure. http://llvm.org/Features.html, 17-01-2013.

[6] LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html#introduction, 17-01-2013.

[7] LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html#type-system, 17-01-2013.

[8] LLVM Project Blog. http://blog.llvm.org/2011/11/llvm-30-type-system-rewrite.html, 17-01-2013.

[9] University of Illinois at Urbana-Champaign. http://illinois.edu/, 17-01-2013.

[10] Balakrishnan, A.; Schulze, C.: Code Obfuscation Literature Survey. http://pages.cs.wisc.edu/˜arinib/writeup.pdf, 2005.

[11] Collberg, C.; Thomborson, C.; Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, Červenec 1997.

[12] Ľubomír Karaba: *Obfuskácia zdrojového kódu*. Diplomová práce, Univerzita Komenského Fakulta Matematiky, Fyziky a Informatiky Katedra Informatiky, 2007.

[13] Kuzurin, N.; Shokurov, A.; Varnovsky, N.; aj.: On the Concept of Software Obfuscation in Computer Security. Technical report, Institute for System Programming, Moscow, Russia, Lomonosov Moscow State University, Russia.

[14] Linn, C.; Debray, S. K.: Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washingtion, DC, USA, October 27-30, 2003*, editace S. Jajodia; V. Atluri; T. Jaeger, ACM, 2003, ISBN 1-58113-738-9, s. 290–299, doi:http://doi.acm.org/10.1145/948109.948149.

[15] Meduna, A.; Lukáš, R.: Optimalizace a generování cílového programu. http://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj10-cz.pdf, 2012.

[16] Schwarz, J.; Sekanina, L.: Aplikované evoluční algoritmy. 2006, studijní opora.

[17] You, I.; Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In *BWCCA*, IEEE, 2010, s. 297–300.
URL http://dblp.uni-trier.de/db/conf/bwcca/bwcca2010.html#YouY10

Příloha A

Zdrojové kódy

A.1 Ackermannova funkcia

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

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

int main(int argc,char **argv)
{
    unsigned int res ,x ,y;
    res = ack(x = 2,y = 3);
    printf("ackerman( %d , %d ) = %d",x,y,res);
    return res;
}
```

A.2 Cyklický redundantný súčet

```c
//result must be 206

#define UPDC32(octet,crc) (crc_32_tab[((crc)^((char)octet)) & 0xff] ^ ((crc) >> 8))

static unsigned int crc_32_tab[] = { /* CRC polynomial 0xedb88320 */
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
```

```
0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

unsigned int buf[] = {5, 2, 8, 4};

unsigned int crc32buf(unsigned int* buf, unsigned int len)
{
       unsigned int oldcrc32;
      int ii, len_o;
      oldcrc32 = 0xFFFFFFFF;

      for (ii = 0; ii < 100000; ii++)
      {
       len_o = len;
       for ( ; len; --len, ++buf)
       {
            oldcrc32 = UPDC32(*buf, oldcrc32);
       }
       len = len_o;
       buf -= len;
      }

      return ~oldcrc32;

}

int
main(int argc, char *argv[])
```

```
{
	unsigned int crc = { 0 };
	unsigned int res;
	crc = crc32buf(buf, 4);

	res = (crc >> 24 & 0xFF) ^ (crc >> 16 & 0xFF) ^ (crc >> 8 & 0xFF) ^ (crc & 0xFF);
	return ( res != 206 );
}
```

A.3 Dijkstrov algoritmus

```
//must return 16
//dijkstra(0, 1) = 16. Path is: 0->17->8->1
#include <stdio.h>
#define NUM_NODES                          30
#define NONE                               256
#define BENCHMARK_RUNS                     1
#define RESULT               16

int rgnNodes[NUM_NODES];
int AdjMatrix[NUM_NODES][NUM_NODES] = {
{NONE, 32, 54, 12, 52, 56, 8, 30, 44, 94, 44, 39, 65, 19, 51, 91, 1, 5, 89, 34, 25, 58,
    20, 51, 38, 65, 30, 7, 20, 10},
{76, NONE, 65, 14, 89, 69, 4, 16, 24, 47, 7, 21, 78, 53, 17, 81, 39, 50, 22, 60, 93, 89,
    94, 30, 97, 16, 65, 43, 20, 24},
{38, 31, NONE, 40, 61, 21, 84, 51, 86, 41, 19, 21, 37, 58, 86, 100, 97, 73, 44, 67, 60,
    90, 58, 13, 31, 49, 63, 44, 73, 76},
{80, 16, 53, NONE, 94, 29, 77, 99, 16, 29, 3, 22, 71, 35, 4, 61, 6, 25, 13, 11, 30, NONE,
    27, 94, 66, 25, 64, 92, 5, 47},
{59, 7, 14, 78, NONE, 45, 54, 83, 8, 94, 12, 86, 9, 97, 42, 93, 95, 44, 70, 5, 83, 10, 40,
    36, 34, 62, 66, 71, 59, 97},
{94, 41, 3, 61, 27, NONE, 33, 35, 78, 38, 73, 14, 80, 58, 5, 99, 59, 19, 22, 40, 59, 78,
    32, 17, 47, 71, 3, 94, 39, 2},
{3, 55, 41, 76, 49, 68, NONE, 23, 67, 15, 97, 61, 13, 61, 60, 75, 33, 77, 71, 15, 39, 72,
    43, 76, 77, 59, 53, 11, 33, 88},
{68, 28, 47, 12, 82, 6, 26, NONE, 98, 75, 13, 57, 7, 8, 55, 33, 55, NONE, 76, 5, 5, 3, 15,
    3, 53, 58, 36, 34, 23, 79},
{7, 1, 46, 39, 12, 68, 41, 28, NONE, NONE, 14, 45, 91, 43, 12, 58, 17, 53, 26, 41, NONE,
    19, 92, 31, 60, 42, 1, 17, 46, 41},
{82, 97, 72, 61, 39, 48, 11, 99, 38, NONE, 27, 2, 49, 26, 59, NONE, 58, 1, 81, 59, 80, 67,
    70, 77, 46, 97, 56, 79, 27, 81},
{66, 98, 85, 82, 96, 20, 64, 73, 67, 69, NONE, 3, 23, 13, 97, 97, 66, 58, 50, 42, NONE,
    44, 57, 86, 54, 85, 82, 14, 8, 1},
{96, 13, 73, 85, 72, 18, 50, 70, 36, 24, 67, NONE, 82, 29, 51, 80, 43, 11, 35, 89, 39, 24,
    NONE, 73, 86, 44, 34, 9, 46, 34},
{60, 66, 32, 92, 65, 19, 74, 97, 32, 16, 72, 38, NONE, 97, 96, 46, 43, 88, 42, 77, 25, 9,
    34, 19, 88, 28, 56, 1, 44, 3},
{86, 28, 81, 45, 12, 37, 1, 70, 29, 64, 89, 31, 41, NONE, 20, 1, 67, 83, 73, NONE, 52, 98,
    64, 20, 78, 93, 78, 8, 17, 100},
{92, 12, 93, 12, 17, 85, 23, 7, 30, 56, 64, 34, 45, 73, NONE, 87, 20, 22, 7, 83, 59, 91,
    26, 59, 5, 79, 26, 99, 79, 32},
{19, 48, 25, 73, 77, 100, 30, 91, 99, 78, 13, 95, 98, 1, 12, NONE, 82, 91, 8, 80, 93, 22,
    61, 2, 28, 2, 66, 5, 65, 76},
{83, 51, 74, 73, 76, 42, 67, 24, 17, 44, 17, 73, 18, 49, 65, 50, NONE, 54, 7, 62, 11, 21,
    85, 32, 77, 10, 68, 94, 70, 36},
{49, 62, 53, 9, 36, 99, 53, 3, 10, 67, 82, 63, 79, 84, 45, 7, 41, NONE, 95, 89, 82, 43,
    27, 53, 5, 78, 77, 4, 69, 25},
{96, 12, 8, 98, 94, 89, 55, 38, 100, 43, 11, 68, 83, 95, 3, NONE, 39, 78, NONE, 90, 63, 8,
```

```
    37, 20, 83, 67, 1, 56, 67, 53},
{48, 14, 79, 95, 6, 70, 76, 4, 98, 98, 87, 39, 14, 81, 1, 99, 7, 33, 81, NONE, 92, 96, 16,
    15, 3, 15, 54, 30, 57, 12},
{88, 87, 4, 77, 75, 72, 69, 35, 23, 2, 35, 6, 80, 99, 15, 50, 6, 53, 61, 46, NONE, 69, 29,
    25, 80, 15, 47, 25, 34, 51},
{83, 67, 34, 49, 50, 38, 27, 33, 4, 56, 70, 60, 15, 75, 6, 33, 40, 57, 59, 46, 4, NONE,
    75, 62, 86, 100, 81, 38, 29, 17},
{75, 36, 7, 30, 18, 31, 96, 22, 12, 76, 71, 43, 50, 69, 80, 61, 78, 42, 72, 43, NONE, 13,
    NONE, 68, 30, 79, 60, 48, 31, 62},
{73, 91, 8, 82, 94, 89, 33, 57, 84, 36, 21, 31, 1, 87, 46, 9, 20, 56, 4, 82, 9, 52, 99,
    NONE, 56, 34, 8, 84, 3, 7},
{7, 75, 65, 74, 92, 64, 95, 63, 30, 57, 77, 2, 42, 11, 65, 16, 59, 7, 45, 97, 46, 66, 63,
    81, NONE, 56, 83, 66, 32, 49},
{29, 53, 97, 74, 1, 53, 83, 32, 30, 46, 52, 71, 94, 41, 42, 21, 45, 62, 85, 81, 98, 81,
    97, 73, 83, NONE, 44, 1, 85, 32},
{68, 44, 8, 95, 81, 28, 63, 85, 8, 52, 86, 35, 41, 11, 53, 94, 3, 12, 58, 71, 13, 85, 11,
    NONE, 55, 44, NONE, 87, 19, 83},
{24, 62, 63, 27, 20, 67, 51, 59, 65, 65, 90, 48, 73, 93, 66, 18, NONE, 75, 47, 63, 26, 76,
    94, 3, 59, 21, 66, NONE, 17, 64},
{59, 25, 35, 29, 81, 44, 84, 43, 24, 58, 20, 91, 45, 38, 17, 74, 100, 63, 31, 36, 3, 33,
    44, 71, 55, 50, 96, 98, NONE, 40},
{63, 52, 72, 60, 10, 71, 70, 56, 12, 59, 52, 94, 95, 68, 13, 21, 41, 94, 55, 66, 100, 25,
    48, 7, 53, 54, 99, 88, 60, NONE}};

int dijkstra(int chStart, int chEnd)
{
  int i, j, node, iCost, visited[NUM_NODES];

  for(i = 0; i < NUM_NODES; i++)
  {
    rgnNodes[i] = NONE;
        visited[i] = 0; /* the i-th element has not yet been visited */
  }

  if (chStart == chEnd)
    return 0; //Shortest path is 0 in cost. Just stay where you are.
  else
  {
    rgnNodes[chStart] = 0;

    for(j = 0; j < NUM_NODES; j++)
    {
        node = -1;
        for (i = 0; i < NUM_NODES; i++)
        if(!visited[i] && ((node == -1) || (rgnNodes[i] < rgnNodes[node])))
          node = i;

      visited[node] = 1;

        for (i = 0; i < NUM_NODES; i++)
          if ((iCost = AdjMatrix[node][i]) != NONE)
              if ((rgnNodes[i] == NONE) || (rgnNodes[i] > (iCost + rgnNodes[node])))
                {
                     rgnNodes[i] = rgnNodes[node] + iCost;
                     printf("%d ",rgnNodes[i]);
                }

    }
```

```
    printf("Shortest path from node %d to node %d is %d in cost.\n", chStart, chEnd,
        rgnNodes[chEnd]);
    return rgnNodes[chEnd];
  }
}

int main(int argc, char *argv[])
{
  int i, j, k;

  for (k = 0; k < BENCHMARK_RUNS; k++)
  {
    for (i = NUM_NODES - 1; i > 0; i--) /* for benchmarking, otherwise delete*/
    {
      j = dijkstra(0, i);
      printf("%d ",j);
    }

  }

  return !(j == RESULT); //dijkstra(0, 1);
}
```

A.4 Fouriérova transformácia

```
/*============================================================================
    fourierf.c - Don Cross <dcross@intersrv.com>
    http://www.intersrv.com/~dcross/fft.html
    Contains definitions for doing Fourier transforms
    and inverse Fourier transforms.

    This module performs operations on arrays of 'float'.
    Revision history:

1998 September 19 [Don Cross]
    Updated coding standards.
    Improved efficiency of trig calculations.
===========================================================================*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void fft_double (
    unsigned NumSamples,         /* must be a power of 2 */
    int       InverseTransform,  /* 0=forward FFT, 1=inverse FFT */
    double   *RealIn,            /* array of input's real samples */
    double   *ImaginaryIn,       /* array of input's imag samples */
    double   *RealOut,           /* array of output's reals */
    double   *ImaginaryOut );    /* array of output's imaginaries */

void fft_float (
    unsigned NumSamples,         /* must be a power of 2 */
    int       InverseTransform,  /* 0=forward FFT, 1=inverse FFT */
    float    *RealIn,            /* array of input's real samples */
    float    *ImaginaryIn,       /* array of input's imag samples */
    float    *RealOut,           /* array of output's reals */
    float    *ImaginaryOut );    /* array of output's imaginaries */
```

```
int IsPowerOfTwo ( unsigned x );
unsigned NumberOfBitsNeeded ( unsigned PowerOfTwo );
unsigned ReverseBits ( unsigned index, unsigned NumBits );

/*
**   The following function returns an "abstract frequency" of a
**   given index into a buffer with a given number of frequency samples.
**   Multiply return value by sampling rate to get frequency expressed in Hz.
*/
double Index_to_frequency ( unsigned NumSamples, unsigned Index );

#define DDC_PI  (3.14159265358979323846)

#define TRUE  1
#define FALSE 0

#define BITS_PER_WORD (sizeof(unsigned) * 8)
int IsPowerOfTwo ( unsigned x )
{
    if ( x < 2 )
        return FALSE;
    if ( x & (x-1) )      // Thanks to 'byang' for this cute trick!
        return FALSE;
    return TRUE;
}

unsigned NumberOfBitsNeeded ( unsigned PowerOfTwo )
{
    unsigned i;
    if ( PowerOfTwo < 2 )
    {
        fprintf (
            stderr,
            ">>> Error in fftmisc.c: argument %d to NumberOfBitsNeeded is too small.\n",
            PowerOfTwo );
        exit(1);
    }
    for ( i=0; ; i++ )
    {
        if ( PowerOfTwo & (1 << i) )
            return i;
    }
}

unsigned ReverseBits ( unsigned index, unsigned NumBits )
{
    unsigned i, rev;
    for ( i=rev=0; i < NumBits; i++ )
    {
        rev = (rev << 1) | (index & 1);
        index >>= 1;
    }
    return rev;
}

double Index_to_frequency ( unsigned NumSamples, unsigned Index )
{
    if ( Index >= NumSamples )
        return 0.0;
```

```
    else if ( Index <= NumSamples/2 )
        return (double)Index / (double)NumSamples;
    return -(double)(NumSamples-Index) / (double)NumSamples;
}
/*--- end of file fftmisc.c---*/

#define CHECKPOINTER(p) CheckPointer(p,#p)

static void CheckPointer ( void *p, char *name )
{
    if ( p == NULL )
    {
        fprintf ( stderr, "Error in fft_float(): %s == NULL\n", name );
        exit(1);
    }
}

void fft_float (
    unsigned NumSamples,
    int      InverseTransform,
    float    *RealIn,
    float    *ImagIn,
    float    *RealOut,
    float    *ImagOut )
{
    unsigned NumBits;  /* Number of bits needed to store indices */
    unsigned i, j, k, n;
    unsigned BlockSize, BlockEnd;

    double angle_numerator = 2.0 * DDC_PI;
    double tr, ti;    /* temp real, temp imaginary */

    if ( !IsPowerOfTwo(NumSamples) )
    {
        fprintf (
            stderr,
            "Error in fft(): NumSamples=%u is not power of two\n",
            NumSamples );

        exit(1);
    }

    if ( InverseTransform )
        angle_numerator = -angle_numerator;

    CHECKPOINTER ( RealIn );
    CHECKPOINTER ( RealOut );
    CHECKPOINTER ( ImagOut );

    NumBits = NumberOfBitsNeeded ( NumSamples );

    /*
    **  Do simultaneous data copy and bit-reversal ordering into outputs...
    */

    for ( i=0; i < NumSamples; i++ )
    {
        j = ReverseBits ( i, NumBits );
        RealOut[j] = RealIn[i];
```

```
    ImagOut[j] = (ImagIn == NULL) ? 0.0 : ImagIn[i];
}

/*
**   Do the FFT itself...
*/

BlockEnd = 1;
for ( BlockSize = 2; BlockSize <= NumSamples; BlockSize <<= 1 )
{
    double delta_angle = angle_numerator / (double)BlockSize;
    double sm2 = sin ( -2 * delta_angle );
    double sm1 = sin ( -delta_angle );
    double cm2 = cos ( -2 * delta_angle );
    double cm1 = cos ( -delta_angle );
    double w = 2 * cm1;
    double ar[3], ai[3];
    double temp;

    for ( i=0; i < NumSamples; i += BlockSize )
    {
        ar[2] = cm2;
        ar[1] = cm1;

        ai[2] = sm2;
        ai[1] = sm1;

        for ( j=i, n=0; n < BlockEnd; j++, n++ )
        {
            ar[0] = w*ar[1] - ar[2];
            ar[2] = ar[1];
            ar[1] = ar[0];

            ai[0] = w*ai[1] - ai[2];
            ai[2] = ai[1];
            ai[1] = ai[0];

            k = j + BlockEnd;
            tr = ar[0]*RealOut[k] - ai[0]*ImagOut[k];
            ti = ar[0]*ImagOut[k] + ai[0]*RealOut[k];

            RealOut[k] = RealOut[j] - tr;
            ImagOut[k] = ImagOut[j] - ti;

            RealOut[j] += tr;
            ImagOut[j] += ti;
        }
    }

    BlockEnd = BlockSize;
}

/*
**   Need to normalize if inverse transform...
*/

if ( InverseTransform )
{
    double denom = (double)NumSamples;
```

```
        for ( i=0; i < NumSamples; i++ )
        {
            RealOut[i] /= denom;
            ImagOut[i] /= denom;
        }
    }
}
/*--- end of file fourierf.c ---*/

int main(int argc, char *argv[]) {
        unsigned MAXSIZE;
        unsigned MAXWAVES;
        unsigned i,j;
        float   * RealIn,
                * ImagIn,
        * RealOutInv,
                * ImagOutInv,
        * RealOut,
            * ImagOut,
            * coeff;
        float *amp;
        int invfft=0;

        MAXSIZE=8; //atoi(argv[2]);
        MAXWAVES=32; //atoi(argv[1]);

     srand(1);

     RealIn=(float*)malloc(sizeof(float)*MAXSIZE);
     ImagIn=(float*)malloc(sizeof(float)*MAXSIZE);
     RealOut=(float*)malloc(sizeof(float)*MAXSIZE);
     ImagOut=(float*)malloc(sizeof(float)*MAXSIZE);
     coeff=(float*)malloc(sizeof(float)*MAXWAVES);
     amp=(float*)malloc(sizeof(float)*MAXWAVES);
     RealOutInv=(float*)malloc(sizeof(float)*MAXSIZE);
         ImagOutInv=(float*)malloc(sizeof(float)*MAXSIZE);

    /* Makes MAXWAVES waves of random amplitude and period */
        for(i=0;i<MAXWAVES;i++)
        {
                coeff[i] = rand()%1000;
                amp[i] = rand()%1000;
        }
     for(i=0;i<MAXSIZE;i++)
     {   /*  RealIn[i]=rand();*/
             RealIn[i]=0;
             for(j=0;j<MAXWAVES;j++)
             {     /* randomly select sin or cos */
                if (rand()%2)
                    {
                            RealIn[i]+=coeff[j]*cos(amp[j]*i);
                        }
                 else
                 {
                        RealIn[i]+=coeff[j]*sin(amp[j]*i);
                 }
        ImagIn[i]=0;
            }
```

```
    }

    /* regular*/
    fft_float (MAXSIZE,0,RealIn,ImagIn,RealOut,ImagOut);

        fft_float (MAXSIZE,1,RealOut,ImagOut,RealOutInv,ImagOutInv);

    for(i=0;i<MAXSIZE;i++)
      if( (fabs( RealIn[i] - RealOutInv[i] ) > 0.001) || (fabs( ImagIn[i] - ImagOutInv[i] )
           > 0.001) )
            exit(i+1);

        printf("RealIn: RealOutInv\n");
        for (i=0;i<MAXSIZE;i++)
        printf("%f \t %f \n", RealIn[i] ,RealOutInv[i] );
        printf("\n");

        printf("ImagIn: ImagOutInv\n");
        for (i=0;i<MAXSIZE;i++)
        printf("%f \t %f \n", ImagIn[i],ImagOutInv[i]);
        printf("\n");

    free(RealIn);
    free(ImagIn);
    free(RealOut);
    free(ImagOut);
    free(coeff);
    free(amp);
    exit(0);
}
```

A.5 Quicksort

```
#define RESULT 255

// quickSort
// This public-domain C implementation by Darel Rex Finley.
#define MAX_LEVELS 300
#define SORT_TAB_SIZE 256
int ooo1 = 0xFFFFFFFF;
int BENCHMARK_RUNS = 20;
int ooo2 = 0xFFFFFFFF;

int globalni = 5;

static int sort_tab[SORT_TAB_SIZE] = { /* note: signed int */
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
  0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
  0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
  0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
  0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
  0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
  0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
  0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
  0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
  0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
```

```
  0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
  0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
  0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
  0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
  0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
  0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
  0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
  0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
  0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
  0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
  0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
  0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
  0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
  0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
  0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
  0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
  0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
  0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
  0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
  0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
  0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
  0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
  0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
  0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
  0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

//#define LISSOM

#ifdef LISSOM

#define DPRINT_VAR(var) \
       { int v = (int)var; \
         int ignore; \
         __asm ( "DEBUG_REG %0" \
              :"=r"(ignore) \
              :"r"(v) \
       ); }

#define DPRINT_STR(str) \
       { const char* s = str; \
         int ignore; \
         __asm ( "DEBUG_STR %0" \
              :"=r"(ignore) \
              :"r"(s) \
       ); }

#else

#define DPRINT_VAR(var) {/* printf("%d\n",var);*/ }
#define DPRINT_VAR2(var) { /*printf("R3 = %d\n",var);*/ }
#define DPRINT_STR(var) { } /*printf("%s\n",var);*/ }
```

```
#endif

void quickSort(int *arr, int elements)
{
  int piv, beg[MAX_LEVELS], end[MAX_LEVELS], i = 0, L, R, swap;

  beg[0] = 0;
  end[0] = elements;
  while (i >= 0)
  {

    L = beg[i];
    R = end[i] - 1;

    if (L < R)
    {
      piv = arr[L];
      while(L < R)
      {
        while(arr[R] >= piv && L < R)
          R--;
        if (L < R)
          arr[L++] = arr[R];
        while(arr[L] <= piv && L < R)
          L++;
        if(L < R)
          arr[R--] = arr[L];
      }
      arr[L] = piv;
      beg[i + 1] = L + 1;
      end[i + 1] = end[i];
      end[i++] = L;
      if (end[i] - beg[i] > end[i - 1] - beg[i - 1])
      {
        swap = beg[i];
        beg[i] = beg[i - 1];
        beg[i - 1] = swap;
        swap = end[i];
        end[i] = end[i - 1];
        end[i - 1] = swap;
      }
    }
    else
      i--;
  }
}

int main(int argc, char **argv)
{
  int i, result = 0;

  for (i = 0; i < BENCHMARK_RUNS; i++) /* sorting of sorted array */
    quickSort(sort_tab, 256);

  for(i = 0; i < SORT_TAB_SIZE; i++)
  {
      DPRINT_VAR2(sort_tab[i]);
      result += (sort_tab[i] & 0xFF) * i;
  }
```

```
  //result should be 4161600 (3f8040h)
  DPRINT_VAR(result);

  return ! ((result >> 24 & 0xFF) ^ (result >> 16 & 0xFF) ^ (result >> 8 & 0xFF) ^ (result
      & 0xFF) == RESULT );
}
```